Informatica Grafica

(II anno Corso di Laurea in Informatica)

Donato D'Ambrosio

Dipartimento di Matematica e Informatica e Centro d'Eccellenza per il Calcolo ad Alte Prestazioni
Cubo 22B, Università della Calabria, Rende 87036, Italy
mailto: d.dambrosio@unical.it
homepage: `http://www.mat.unical.it/~donato`

Anno Accademico 2014/15

Introduzione al corso

Introduzione al corso

Introduzione

- Il corso di Informatica Grafica prevede un totale di *5 crediti* di cui 2 di lezione frontale (24 ore) e d 3 di laboratorio (24 ore)
- Gli argomenti che saranno trattati nelle lezioni frontali in aula sono i seguenti:
  - Programmazione grafica tridimensionale interattiva in OpenGL
  - Algoritmi fondamentali della Computer Graphics
  - Argomenti avanzati opzionali
- La parte di laboratorio ha lo scopo di sperimentare le tecniche e gli algoritmi studiati nella parte teorica
  - Nota: le lezioni di laboratorio si terranno nel Laboratorio d'Informatica del cubo 31B
  - Coloro i quali siano muniti di computer portatile potranno svolgere le esercitazioni sul proprio pc

Testi consigliati

- R. Scateni, P. Cignoni, C. Montani, R. Scopigno. Fondamenti di grafica tridimensionale interattiva. McGraw-Hill
- Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis. The OpenGL Programming Guide - The Redbook, Liberamente scaricabile dal sito
  `http://www.opengl.org/documentation/red_book/`
- Altro materiale reperibile in rete, proposto a tempo debito durante le lezioni di teoria

Ricevimento studenti

- Il ricevimento studenti è fissato ogni giovedì alle ore 16:00 nello studio del docente, Cubo 22B, c/o Centro d'Eccellenza per il Calcolo ad Alte Prestazioni
- Ricevimenti supplementari devono essere concordati col docente, anche via email

Modalità d'esame

- Una *prova scritta*, incentrata principalmente sugli aspetti algoritmici e sui concetti chiave della grafica 3D (ad esempio le trasformazioni); la prova scritta avrà la durata di un'ora e non sarà consentito l'uso di appunti di alcun genere
- Una *prova di laboratorio* in cui sarà richiesta la scrittura di una (semplice) applicazione in OpenGL; la prova di laboratorio avrà la durata di due ore e sarà consentito l'uso di appunti e manuali
- Un progetto proposto dal docente da realizzare, da soli o in gruppo, a partire dalla seconda metà del corso; l'applicazione sviluppata dovrà essere accompagnata da un'adeguata documentazione (estratta ad esempio dai commenti con doxygen) e da una relazione scritta sul lavoro svolto

Programma del Corso (Programmazione in OpenGL)

- Introduzione alla Computer Graphics
  - Definizione di Computer Graphics
  - Le origini
  - Sistemi di grafica vettoriale e sistemi di grafica raster
  - Sistemi hardware per la grafica raster
  - Librerie grafiche
  - Esempi
- Introduzione a OpenGL
  - La libreria grafica OpenGL
  - Cosa può fare OpenGL
  - Sintassi dei comandi OpenGL
  - OpenGL come macchina a stati
  - Librerie supplementari di OpenGL: GLU e GLUT
  - Schema di un'applicazione GLUT
  - Gestione degli eventi in un'applicazione GLUT
  - Compilazione

Programma del Corso (Programmazione in OpenGL)

- Stati e Primitive Grafiche di OpenGL
  - Colore di sfondo e di disegno
  - Shading
  - Il sistema di coordinate
  - Il volume visibile
  - Aspect ratio
  - Disegno di punti, segmenti (linee) e poligoni
  - Poligoni planari e non planari
  - Specificare i vertici
  - Disegnare punti linee e poligoni
  - Costrutti all'interno di glBegin() e glEnd()
  - Gli stati di OpenGL
  - Disegno di poligoni non convessi
  - Vettori normali
  - Calcolo delle normali per poligono e per vertice

Programma del Corso (Programmazione in OpenGL)

- Trasformazioni geometriche in OpenGL
  - Composizione della scena e trasformazioni
  - Coordinate dell'osservatore (eye coordinates)
  - Trasformazioni di viewing e di modeling
  - La dualità modelview
  - La trasformazione di proiezione
  - La matematica delle trasformazioni
  - Il frustum
- Modelli di colore in OpenGL
  - I modelli RGB e HSV
  - Come funziona il monitor: risoluzione e profondità di colore
  - Uso dei colori in OpenGL
  - Modelli di illuminazione e materiali
  - Aggiungere la luce alla scena
  - Definire il tipo di materiale
  - Sorgenti e luce direzionale

Programma del Corso (Argomenti opzionali)

- Texture Mapping in OpenGL
  - Caricamento e Mappatura
  - Mipmapping
  - Texture coordinates generation
  - object linear mapping
  - eye linear mapping
  - sphere mapping
  - cube mapping
- Effetti speciali in OpenGL
  - Blending, Antialiasing e Fog
- Curve e superfici in OpenGL
  - Quadric Objects
  - Curve e superfici di Bézier e NURBS
  - Evaluators
  - Calcolo automatico delle Normali

Programma del Corso (Algoritmi)

- Algoritmi di rasterizzazione e clipping
  - Rasterizzazione
  - Clipping
  - L'approccio object-oriented vs image-oriented
  - Scan Conversion dei segmenti
    - L'algoritmo di scan conversion DDA
    - L'algoritmo di Bresenham
    - Scan Conversion del cerchio
    - Scan Conversion dei poligoni
    - Flood fill
    - Algoritmo Scan-line
  - Clipping
    - Algoritmo di Cohen-Sutherland
  - Antialiasing
- Rimozione delle superfici nascoste
  - Approcci object-space e image-space
  - Eliminazione delle back face
  - L'algoritmo z-buffer

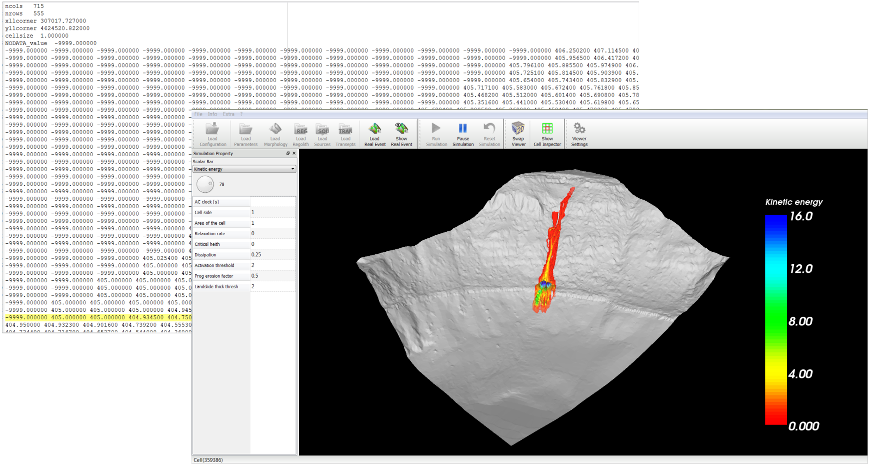
Introduzione alla Computer Graphics

Introduzione alla Computer Graphics

Cos'è e di cosa si occupa la Computer Graphics

- La Computer Graphics è una branca estremamente vasta e articolata dell'Informatica
- Si occupa dello sviluppo e dell'applicazione di tecniche di rappresentazione grafica delle informazioni atte a migliorare l'interazione tra uomo e macchina
- Ormai da qualche decennio a questa parte, i calcolatori elettronici sono in grado di produrre grandi quantità di dati, ad esempio come risultato dell'elaborazione di qualche modello di calcolo numerico; in questo caso, il risultato dell'elaborazione è generalmente di tipo numerico e un'opportuna rappresentazione grafica può in molti casi facilitarne l'interpretazione
- Si parla, in questi casi, di *Visualizzazione Scientifica*

Esempio di rappresentazione grafica di dati numerici



Origini della Computer Graphics

- Le origini della Computer Graphics si possono far risalire ai primi tentativi di utilizzare i tubi a raggi catodici (CRT - Cathode Ray Tube) come dispositivi di output grafico
- Il primo calcolatore dotato di un monitor CRT è stato realizzato nel 1950 presso l'MIT (Massachusetts Institute of Technology)
- Un CRT è un dispositivo in grado di trasformare segnali elettrici in immagini utilizzando campi elettrici per generare fasci di elettroni ad alta velocità che, opportunamente indirizzati, colpiscono uno schermo rivestito di materiale fosforescente provocando l'emissione di luce

Dispositivi di grafica vettoriale

- I primi dispositivi di output grafico entrati in commercio negli anni sessanta, e rimasti in uso fino alla metà degli anni ottanta, erano basati sul concetto di grafica vettoriale
- Il fascio di elettroni, che va a colpire il rivestimento fosforescente del CRT, può muoversi direttamente da una posizione all'altra, secondo l'ordine arbitrario dei comandi di display
- Annullando l'intensità del fascio, questo può essere spostato in una nuova posizione, senza modificare l'immagine
- A livello concettuale, questi sistemi grafici (bidimensionali) possono essere descritti con due funzioni fondamentali di drawing:

```
moveto(x, y); lineto(x, y);
```

Dispositivi di grafica raster

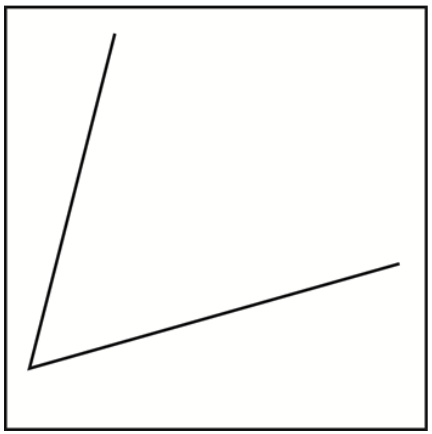
- La tecnica vettoriale è rimasta in uso fino agli anni ottanta, quando hanno cominciato a diffondersi i sistemi di grafica raster, migliori sotto il profilo delle performace e della persistenza dell'immagine a video
- Nella grafica raster, ogni immagine è rappresentata tramite una matrice di punti detti pixel (contrazione della locuzione inglese picture element), ognuno dei quali corrisponde a una piccola area dell'immagine
- Inizialmente i CRT furono adattati ai sistemi di grafica raster; tuttavia, la tecnologia LCD (Liquid Crystal Display) si è ormai imposta come standard tecnologico e ha di fatto soppiantato la tecnologia CRT

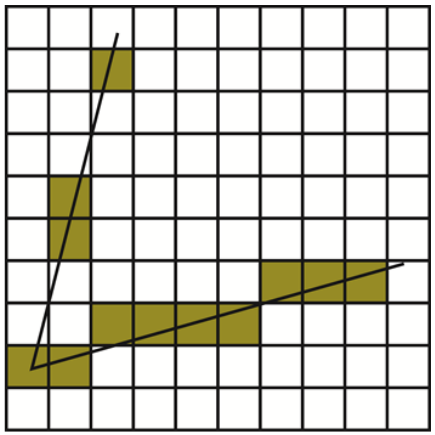
Dispositivi di grafica raster: pixmap, frame buffer e rasterizzazione

- In grafica raster l'elaborazione delle immagini è dunque basata su matrici di pixel dette pixmap (da pixel map)
- La modellazione (cioè la definizione concettuale di ciò che si vuole rappresentare) avviene sempre in un contesto di grafica vettoriale, ma per produrre l'immagine finale (rendering) è necessaria una fase ulteriore detta rasterizzazione o scan conversion

Dispositivi di grafica raster: esempio di scan conversion

- Come è facile intuire, la fase di scan conversion deve essere sufficientemente accurata al fine di ottenere matrici di pixel senza problemi di *frastagliamento* delle immagini





Dispositivi di grafica raster: algoritmi di rasterizzazione

- Proprio per evitare problemi di questo tipo, esistono numerosi algoritmi di rasterizzazione che hanno il compito di produrre matrici di pixel il più possibile accurate
- Un fattore critico di tali algoritmi è l'efficienza computazionale; in relazione alla complessità delle immagini che devono essere generate, taluni algoritmi risultano infatti troppo lenti per applicazioni pratiche
- Gli algoritmi di scan conversion saranno oggetto di lezioni successive

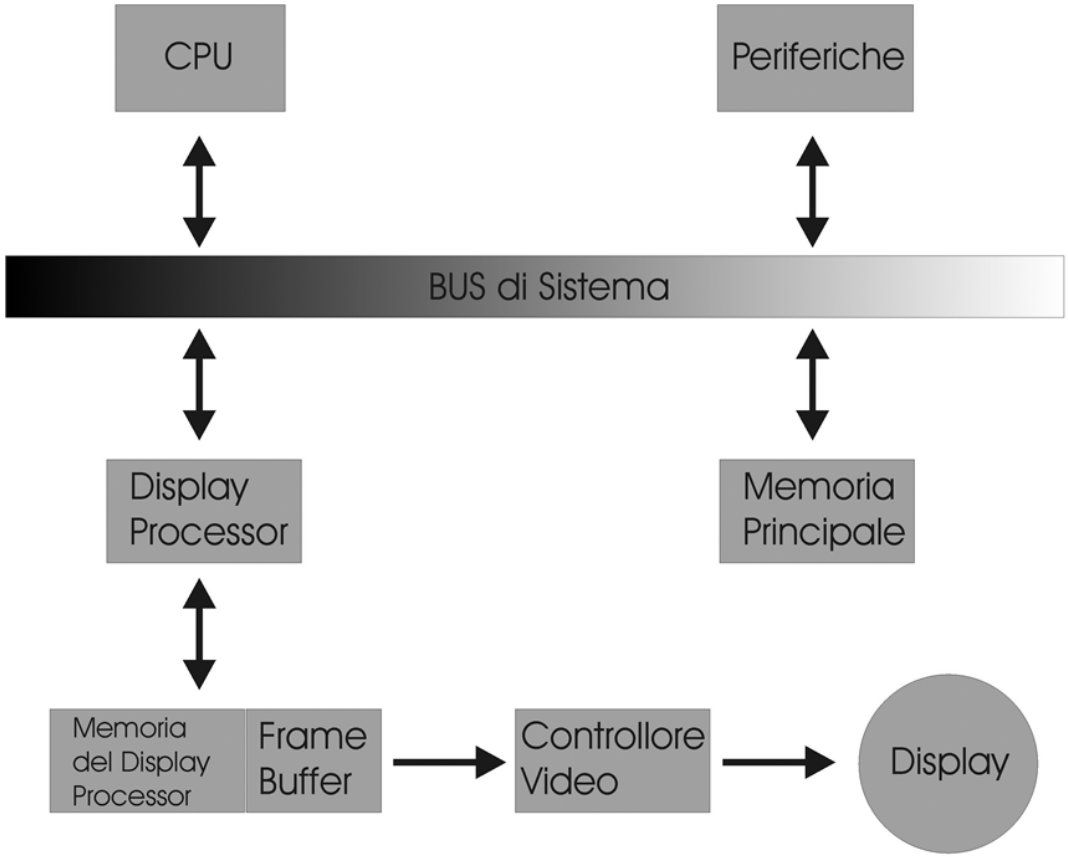
Dispositivi di grafica raster: altri algoritmi

- Nei moderni sistemi grafici, oltre agli algoritmi di rasterizzazione sono anche presenti algoritmi specifici per la rappresentazione grafica di modelli 3D
- Quando si modellano oggetti 3D, questi sono infatti collocati nel così detto volume di vista
- Se un oggetto non rientra completamente in tale volume, l'algoritmo che proietta l'oggetto sullo schermo deve tenerne conto (fase di clipping)
- Alcuni algoritmi di clipping e altri saranno oggetto di lezioni successive

Sistemi hardware di grafica raster

- L'hardware dei sistemi di grafica raster può essere organizzato secondo due diversi approcci
- Nel primo, la gestione di tutte le funzioni grafiche è affidata alla CPU
- Nel secondo, alla CPU è affiancato un display processor, che solleva la CPU dall'elaborazione grafica
- Un'ulteriore differenza deriva dalla localizzazione del frame buffer che può essere localizzato nella memoria centrale del calcolatore o (meglio) in una memoria dedicata del sottosistema grafico
- In ogni caso, un controllore video scandisce periodicamente il frame buffer (in particolare il color buffer) e lo mostra sul display a una frequenza prefissata (ad esempio 60 volte al secondo)

Sistemi hardware di grafica raster



Sistemi hardware di grafica raster

- Le attuali schede video prevedono display processors e memorie locali ad alte prestazioni
- Viste le sempre crescenti capacità di elaborazione dei display processors, questi ultimi vengono chiamati GPU (Graphic Processor Unit)
- Benché la memoria del display processor sia ormai integrata nelle schede grafiche, queste ultime devono in ogni caso accedere alla memoria centrale dell'elaboratore dove risiedono le informazioni che devono essere utilizzate per la costruzione del frame buffer
- La velocità d'accesso da parte della GPU alla memoria centrale e a quella locale, insieme alla larghezza di banda sono fattori critici per le prestazioni di un sistema grafico

Librerie grafiche e GPU

- Le prestazioni di un sistema grafico possono dipendere significativamente anche dal tipo particolare di libreria grafica utilizzata
- Le moderne GPU infatti implementano in hardware numerose primitive grafiche di alcune librerie specifiche
- Implementare in hardware una primitiva grafica (ad esempio il disegno di un segmento) vuol dire eseguire l'operazione in un solo colpo risparmiando numerosi cicli di GPU

Librerie grafiche e GPU

- Di conseguenza, maggiori sono le primitive della libreria grafica implementate in hardware, tanto più conveniente è utilizzare quella libreria
- Tra l'altro, lo sviluppo tecnologico delle GPU si sta orientando sia verso esasperazioni sempre più spinte dell'architettura di calcolo, sia verso l'implementazione di nuove primitive in hardware
- Attualmente la quasi totalità delle schede grafiche supporta le librerie OpenGL e Direct3D

Librerie grafiche e GPU: Direct3D

- Direct3D è un'API Microsoft per lo sviluppo di applicazioni grafiche 2D e 3D
- Numerose schede grafiche (es. AMD/ATI e nVIDIA) supportano Direct3D
- A scapito della portabilità, è rilasciata solo per piattaforme Windows
- È una libreria a basso livello

Librerie grafiche e GPU: OpenGL

- OpenGL (Open Graphic Library) è una libreria portabile per lo sviluppo di applicazioni grafiche interattive 2D e 3D
- È stata rilasciata per la prima volta nel 1992 come versione aperta della libreria GL (Graphic Library) della Silicon Graphics che fino a quel momento deteneva lo stato dell'arte tecnologico nel campo della Computer Graphics
- Attualmente OpenGL è disponibile per diversi sistemi operativi, tra cui:
  - Windows e Mac OS X
  - Linux e Unix (all flowers)

OpenGL (http://www.opengl.org/)

- Un consorzio indipendente, l'OpenGL Architecture Review Board, definisce le specifiche della libreria
- Fanno parte del consorzio numerose aziende interessate allo sviluppo della computer graphics, tra cui:
  - 3DLabs, Apple
  - AMD, DELL
  - IBM, Intel
  - nVIDIA, SGI
  - Oracle
- Ogni cambiamento apportato alla libreria deve essere approvato dall'ARB affichè possa far parte della release successiva

OpenGL (http://www.opengl.org/)

- L'ultima versione di OpenGL è la 4
- Non tutte le case costruttrici si sono adeguate all'ultima versione dello standard e implementano versioni precedenti nei propri drivers
- Tra i suoi punti di forza possiamo elencare i seguenti:
  - Standardizzazione, stabilità, affidabilità e portabilità
  - Language bindings per C/C++ Fortran, Ada, Python, Perl e Java
  - Maggiore semplicità d'utilizzo rispetto a Direct3D
  - Documentazione:
    - Red Book (http://www.opengl.org/documentation/red_book_1.0/)
    - Blue Book (http://www.rush3d.com/reference/opengl-bluebook-1.0/)

Alcuni software basati su OpenGL

Figura: Unreal(`http://www.unrealtournament.com`)

Alcuni software basati su OpenGL

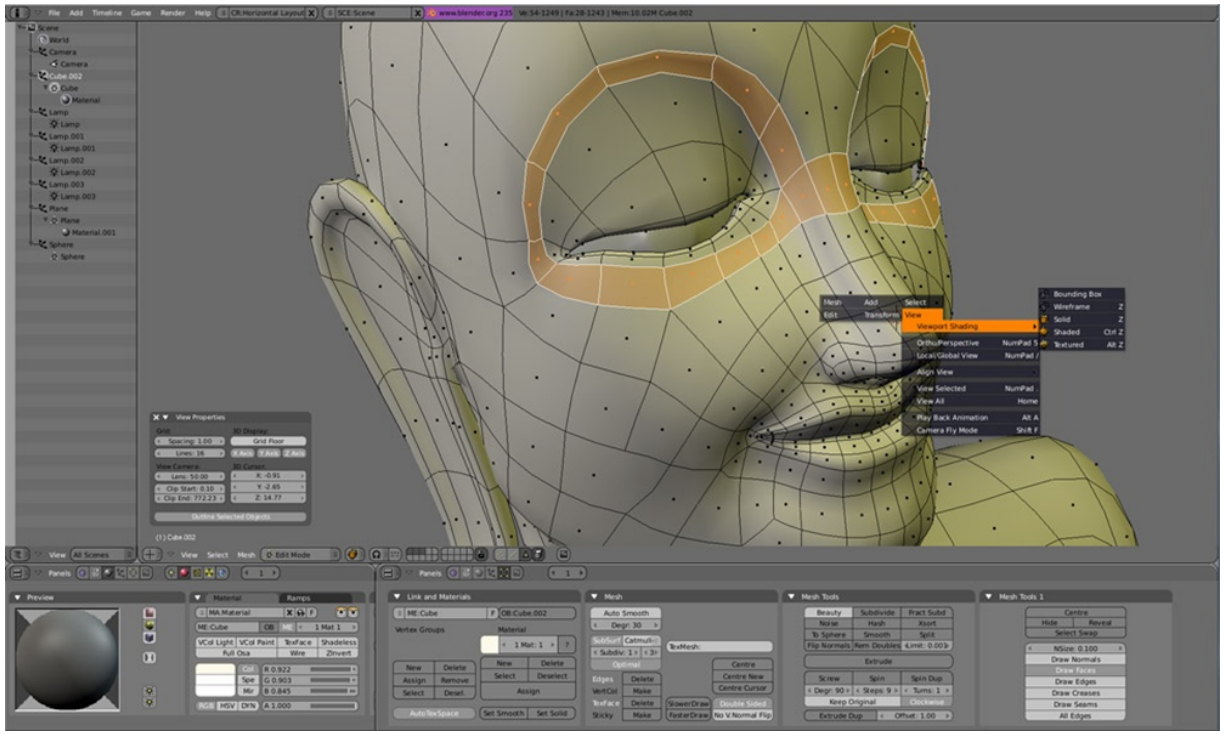


Figura: Blender (`http://www.blender.org`)

Alcuni software basati su OpenGL

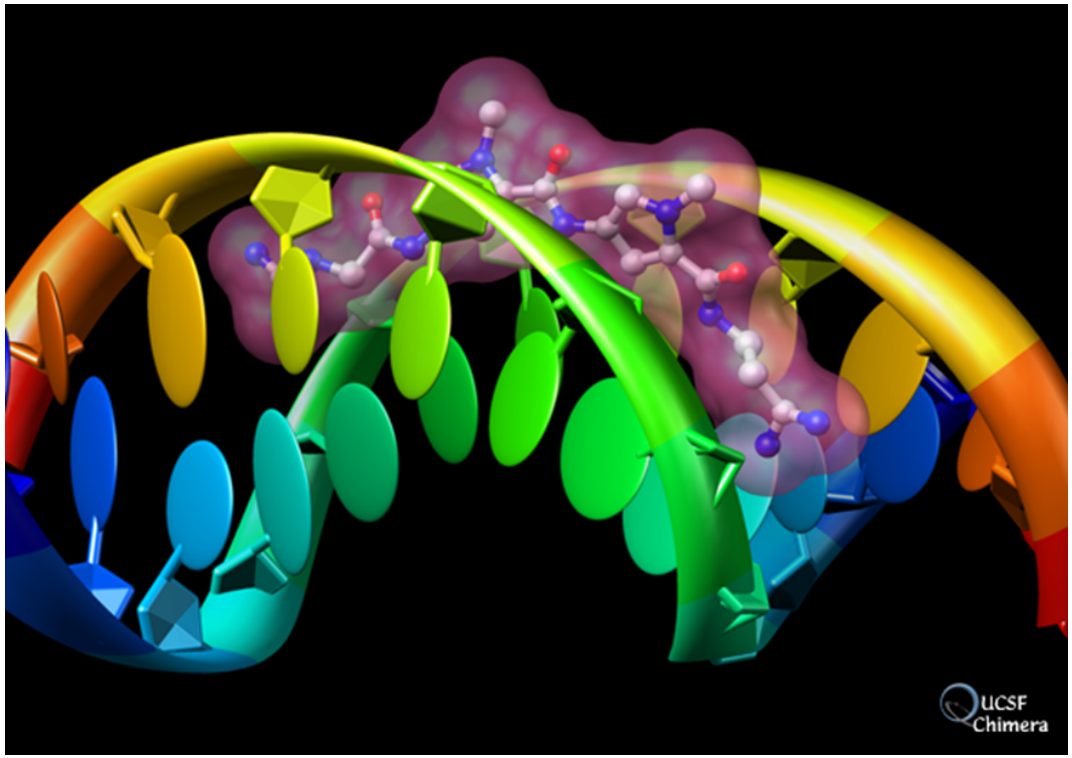


Figura: Chimera (`www.cgl.ucsf.edu/chimera`)

Alcuni software basati su OpenGL

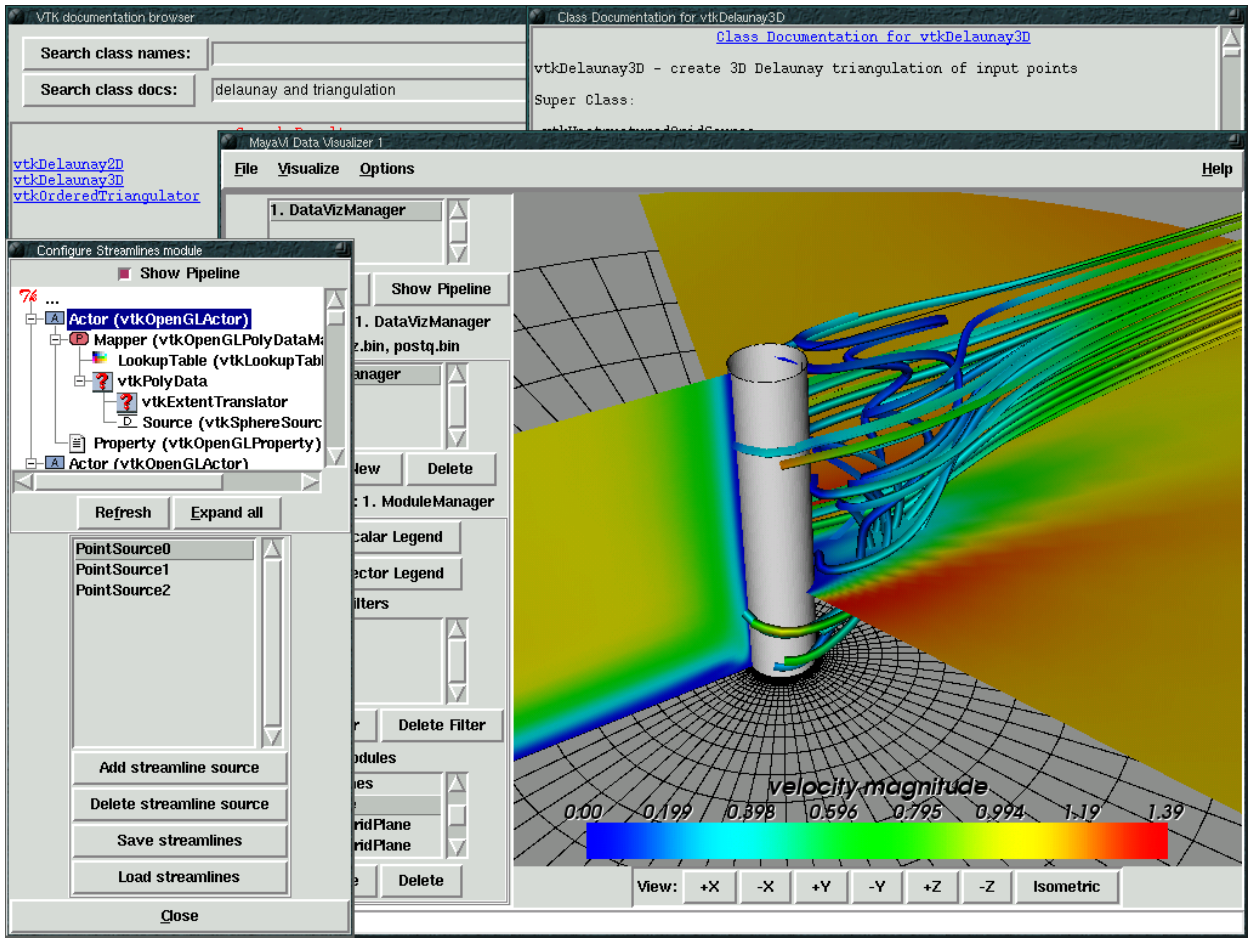


Figura: VTK-Visualization Toolkit (`http://www.vtk.org/`)

Introduzione a OpenGL

Introduzione a OpenGL

Le librerie grafiche

- Il progresso dei dispositivi hardware di output grafico ha determinato, in modo del tutto naturale, un'evoluzione delle applicazioni software e ha portato alla realizzazione di librerie grafiche di alto livello, indipendenti da qualsiasi periferica grafica di input e output e con una portabilità simile a quella dei linguaggi di programmazione di alto livello (quali FORTRAN o C)
- Il modello concettuale alla base delle prime librerie grafiche (bidimensionali) realizzate è quello ora definito modello pen plotter, con chiaro riferimento all'omonimo dispositivo output

Le librerie grafiche

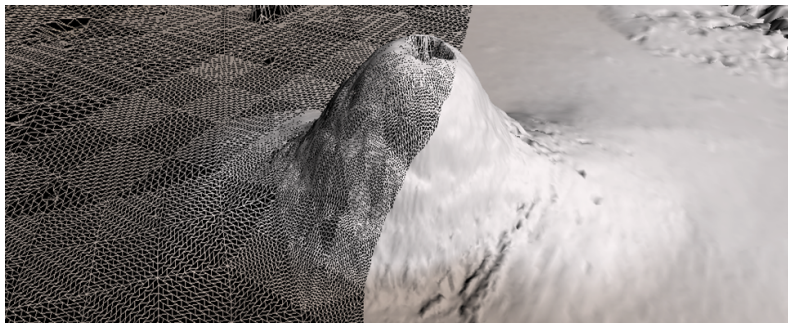
- Un pen plotter produce immagini muovendo una penna lungo due direzioni sulla carta; la penna può essere alzata e abbassata come richiesto per creare l'immagine desiderata
- Diverse librerie grafiche, come ad esempio il PostScript, creano immagini in modo simile al processo di disegnare una figura su un pezzo di carta: l'utente ha a disposizione una superficie bidimensionale, e vi muove sopra una penna
- Questi sistemi grafici possono essere descritti con due funzioni di drawing:

```
moveto(x, y); lineto(x, y);
```

- Aggiungendo la possibilità di cambiare penna per variare il colore e lo spessore delle linee, si ha un sistema grafico semplice ma completo

Le librerie grafiche

- Il modello pen plotter bidimensionale (tipico dei primi sistemi hardware vettoriali) non si estende bene ai sistemi grafici 3D; ad esempio, se vogliamo usare un modello pen plotter per produrre un'immagine di un oggetto tridimensionale su una superficie bidimensionale, è necessario proiettare sulla superficie punti dello spazio tridimensionale
- Per questo, è preferibile utilizzare un'interfaccia software che permetta di lavorare nel dominio (3D) del problema (Modeling) e di utilizzare il computer la produzione dell'immagine finale (Rendering)



La libreria grafica OpenGL

- OpenGL è un'interfaccia software per hardware grafico
- Nell'ultima versione (la 4.0) OpenGL consiste di più di 250 comandi (oltre 200 nel core di OpenGL; oltre 50 nella OpenGL Utility Library)
- Poiché uno degli obbiettivi del progetto era quello di garantire l'indipendenza dalla particolare piattaforma hardware/software, OpenGL non contiene comandi per la creazione/gestione di finestre, né per la gestione dell'input da tastiera o mouse

La libreria grafica OpenGL

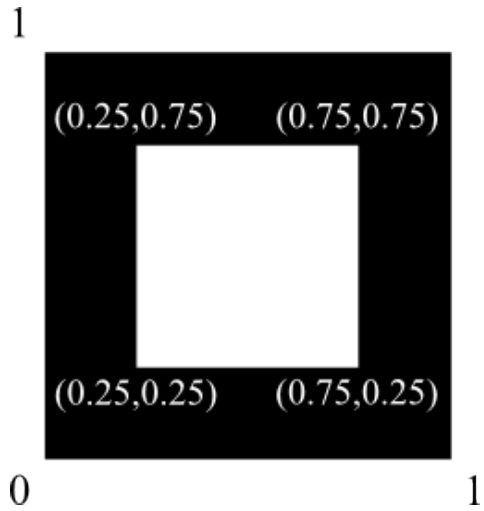
- Allo stesso modo, OpenGL non prevede funzioni per la descrizione di modelli 3D
- Con OpenGL è possibile però realizzare modelli complessi a partire di un piccolo insieme di primitive geometriche
- In realtà, alcune funzioni ad alto livello sono contenute nella OpenGL Utility Library (ad esempio le quadratiche o le curve e superfici NURBS).
- GLU è ormai parte integrante di ogni implementazione di OpenGL

Cosa può fare OpenGL

- Costruire forme 2D e 3D anche molto complesse a partire da primitive geometriche semplici come punti linee e poligoni
- Disporre oggetti nel piano 2D o nello spazio 3D
- Scegliere un punto da cui osservare la scena
- Calcolare il colore degli oggetti come risultato dell'effetto di condizioni di illuminazione, dell'applicazione di texture, dell'assegnazione esplicita da parte del programmatore
- Rasterizzazione, cioè conversione della rappresentazione matematica degli oggetti in pixel sullo schermo

Un esempio preliminare (cfr. Red Book)

```
#include <whateverYouNeed.h>

void main() {
     InitializeAWindowPlease();

     glClearColor (0.0, 0.0, 0.0, 0.0);
     glClear (GL_COLOR_BUFFER_BIT);
     glColor3f (1.0, 1.0, 1.0);
     glOrtho2D(0.0, 1.0, 0.0, 1.0);
     glBegin(GL_POLYGON);
               glVertex2f (0.25, 0.25);
               glVertex2f (0.75, 0.25);
               glVertex2f (0.75, 0.75);
               glVertex2f (0.25, 0.75);
     glEnd();
     glFlush();
     UpdateTheWindowAndCheckForEvents();
}
```



Sintassi dei comandi OpenGL

- I comandi OpenGL iniziano con il prefisso gl e continuano con il nome specifico del comando utilizzando l'iniziale maiuscola per ogni parola. Ad esempio:

```
glClearColor()
```

- Le costanti sono in maiuscolo, iniziano con il prefisso GL_ e usano il carattere underscore per separare le diverse parole. Ad esempio:

```
GL_COLOR_BUFFER_BIT
```

Sintassi dei comandi OpenGL

- I comandi OpenGL possono terminare con dei suffissi del tipo 3f, 3fv e altri. Ad esempio:

```
glColor3f(1.0,0.0,0.0);
glVertex3f(0.25,0.25,0.25);
```

- In particolare, il numero 3 indica che il comando accetta 3 argomenti, mentre la lettera f indica che gli argomenti sono di tipo float (32 bit).
- Se negli esempi precedenti fosse stata presente anche la lettera v le funzioni avrebbero accettato un array di 3 elementi di tipo float:

```
GLfloat color_array[]={1.0,0.0,0.0};
glColor3fv(color_array);
```

Sintassi dei comandi OpenGL

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

Sintassi dei comandi OpenGL

- Nell'ultima colonna della tabella precedente sono riportati i tipi OpenGL, utili per ragioni di compatibilità con implementazioni di OpenGL non scritte in linguaggio C
- L'uso dei tipi OpenGL garantisce uniformità di comportamento dell'applicazione rispetto all'implementazione OpenGL
- Ritornando alle istruzioni OpenGL, i due comandi seguenti sono equivalenti eccetto per il fatto che il primo specifica le coordinate come numeri interi, il secondo come float a 32 bit:

```
glVertex2i(1,3); glVertex2f(1.0,3.0);
```

OpenGL come macchina a stati

- OpenGL è una macchina a stati
- Ad esempio, il colore corrente (cioè attualmente in uso) è una variabile di stato: una volta settato, gli oggetti sono disegnati con quel colore fino a quando esso non viene modificato
- Altre variabili di stato determinano il tipo di proiezione corrente (ortogonale o prospettica), la posizione e le caratteristiche delle luci e dei materiali o la presenza della nebbia nella scena
- Le funzioni glEnable() e glDisable() sono utilizzate per abilitare e disabilitare alcune variabili di stato e quindi alcune funzionalità di OpenGL (es. la nebbia)
- Per default, la maggior parte sono disattivati, come ad esempio le luci o la nebbia

Gli stati di OpenGL: interrogazione stati complessi

- glIsEnabled() è utile per interrogare gli stati OpenGL del tipo on/off
- OpenGL, tuttavia, gestisce stati più complessi: ad esempio, l'istruzione glColor3f() setta 3 valori RGB del più articolato stato GL_CURRENT_COLOR; per questo esistono 5 ulteriori funzioni
  - void glGetBooleanv(GLenum pname, GLboolean *params)
  - void glGetIntegerv(GLenum pname, GLinteger *params)
  - void glGetFloatv(GLenum pname, GLfloat *params)
  - void glGetDoublev(GLenum pname, GLdouble *params)
  - void glGetPointerv(GLenum pname, GLvoid **params)
- Ad esempio lo stato del colore può essere ottenuto come segue:

```
GLfloat colorv[4];
glGetFloatv(GL_CURRENT_COLOR, colorv);
glColor3f(colorv[0]/2, colorv[1]/2, colorv[2]/2);
```

Gli stati di OpenGL: gruppi di attributi

- OpenGL implementa i gruppi di attributi, che raggruppano variabili di stato che hanno a che fare con gli stessi oggetti
- Per esempio, l'attributo GL_LINE_BIT consiste di 5 variabili di stato: (1) lo spessore della line, (2) lo stato di attivazione di GL_LINE_STIPPLE, (3) il pattern della linea, (4) il line stipple repeat counter e (5) lo stato di attivazione di GL_LINE_SMOOTH
- Le funzioni void glPushAttrib(GLbitfield mask) e void glPopAttrib() permettono di salvare e riattivare gruppi di attributi in un colpo solo
- Dalla versione 1.1 di OpenGL esiste anche un altro stack di attributi accessibile tramite le funzioni glPushClientAttrib() e glPopClientAttrib()
- Esistono 20 gruppi di attributi e 2 gruppi di attributi client; per dettagli si veda OpenGL Programming Guide IV edition

Librerie supplementari di OpenGL

- OpenGL fornisce un potente insieme di comandi di rendering a basso livello, e tutti gli oggetti (modelli) ad alto livello devono essere costruiti sulla base di queste primitive
- Inoltre, i programmi OpenGL si devono appoggiare ai sistemi a finestre dei differenti sistemi operativi
- Per fortuna esistono delle librerie esterne al nocciolo principale di OpenGL che permettono di semplificare notevolmente le operazioni di disegno e di gestione dell'interfaccia grafica

La libreria GLU

- La libreria GLU (OpenGL Utitlity Library) contiene circa 50 funzioni, costruite sulla base delle primitive OpenGL, che consentono di semplificare alcune operazioni come l'orientazione del punto di vista della scena, la tassellazione di oggetti complessi tramite poligoni e il rendering di superfici
- Le funzioni della libreria GLU iniziano col suffisso glu e seguono le convenzioni dei comandi OpenGL

La librerie per le GUI

- Per ogni sistema a finestre è stata sviluppata una libreria specifica basata su OpenGL per la creazione e gestione di interfacce grafiche utente e per la gestione dell'I/O: GLX (X Window), WGL (Windows), PGL (IBM OS/2) e AGL (Mac OS/MAC OS X)
- L'uso di queste librerie è ovviamente orientato al particolare sistema operativo e pregiudica la portabilità del software
- Queste librerie sono inoltre abbastanza complesse e generalmente sono utilizzate per applicazioni avanzate

La libreria GLUT

- La libreria GLUT (OpenGL Ultitlity Toolkit) è un toolkit indipendente dal sistema a finestre
- Implementazioni della GLUT esistono per X Window, Microsoft Windows e Carbon/Cocoa, il che rende praticamente portabile al 100% un'applicazione OpenGL con GUI basata su GLUT
- Purtroppo non offre tutte le possibilità delle API native dei window systems (es. non supporta i menu delle finestre, ma solo i menu contestuali)
- GLUT è molto semplice da usare ed è ottima per utilizzi didattici
- I comandi GLUT iniziano col suffisso glut e seguono le convenzioni dei comandi OpenGL
- Link: `http://www.xmission.com/~nate/glut.html`

Include Files

- Un'applicazione OpenGL deve almeno includere l'header file gl.h. La libreria GLU è utilizzata nella stragrande maggioranza della applicazioni OpenGL e richiede il file glu.h header file. Così ogni programma OpenGL di solito inizia con:

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

- NOTA: Microsoft Windows richiede anche l'inclusione del file windows.h prima di gl.h e glu.h

```
#include <windows.h>
```

Include Files

- L'header glext.h contiene estensioni di singole case produttrici (es. nVidia) che non fanno ancora parte di gl.h ma che generalmente stanno per essere standardizzate dall'OpenGL Architecture Review Board
- Generalmente i file glext.h sono disponibili sui web sites dei produttori hardware o sul sito web di OpenGL
  `http://www.opengl.org/`
- Il file glext generalmente si include con gli apici e non con le parentesi acute:

```
#include glext.h"
```

Include Files

- Se si usano le librerie per la GUI accennate sopra è necessario includere l'header; ad esempio, nel caso di GLX si ha:

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

- Nel caso di WGL (Windows) basta includere:

```
#include <windows.h>
```

- Nel caso si utilizzi la libreria GLUT:

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

Schema di un'applicazione GLUT

Il programma principale è il classico main:

```
int main (int argc, char **argv){
    /*
        istruzioni
    */
    return 0;
}
```

All'interno del main devono essere chiamate alcune funzioni GLUT per l'inizializzazione della libreria, l'apertura della finestra, il ciclo degli eventi da tastiera e mouse e altro

Schema di un'applicazione GLUT

glutInit(int *argc, char **argv) inizializza la libreria GLUT e deve essere chiamata per prima

```
int main (int argc, char **argv){
    glutInit (argc, argv);
    /*
        istruzioni
    */
    return 0;
}
```

Schema di un'applicazione GLUT

glutInitDisplayMode(unsigned int mode) specifica se utilizzare il modo RGBA o il modo color-index; si può inoltre specificare se la finestra è single o double buffered e se sono attivi il depth buffer, ecc.

```
int main (int argc, char **argv){
    glutInit (argc, argv);
    glutInitDisplayMode ( GLUT_DOUBLE | GLUT_RGB );
    /*
        istruzioni
    */
    return 0;
}
```

La libreria GLUT

- glutInitWindowPosition(int x, int y) specifica la posizione in pixel dell'angolo in alto a sinistra della finestra rispetto al sistema di coordinate del window system
- glutInitWindowSize(int width, int height) specifica le dimensioni, in pixels, della finestra
- int glutCreateWindow(char *string) crea una finestra con un contesto OpenGL; l'argomento è la caption della finestra

Schema di un'applicazione GLUT

```
int main (int *argc, char **argv){
    glutInit (argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE, GLUT_RGB);
    glutInitWindowPosition (10, 20);
    glutInitWindowSize(800, 480) ;
    glutCreateWindow("An OpenGL window");
    /*
        istruzioni
    */
    return 0;
}
```

Schema di un'applicazione GLUT

- glutDisplayFunc(void (*func)(void)) definisce una event callback function
- Ogni volta che GLUT determina che il contenuto della finestra deve essere ridisegnato, viene eseguita la callback function registrara in glutDisplayFunc()
- Pertanto è necessario richiamare tutte le funzioni per il ridisegno della scena nella display callback function

Schema di un'applicazione GLUT

- Nota che la funzione glutPostRedisplay(void) può forzare in ogni momento l'esecuzione della display callback function
- glutReshapeFunc(void (*func)(int w, int h)) indica quale azione deve essere eseguita quando la finestra viene ridimensionata
- glutMainLoop() mostra tutte le finestre create, inizia il processo d'ascolto degli eventi, e attiva le display callback functions

Schema di un'applicazione GLUT

```
#include <GL/glut.h>
void display(){/* istruzioni */};
void reshape(int w, int h){/* istruzioni */};

int main (int *argc, char **argv){
    glutInit (argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE, GLUT_RGB);
    glutInitWindowPosition (10, 20);
    glutInitWindowSize(800, 480) ;
    glutCreateWindow("An OpenGL window");
    glutReshapeFunc(reshape);
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

Schema di un'applicazione GLUT

- glutKeyboardFunc(void (*func)(unsigned char key, int x, int y)) e glutMouseFunc(void (*func)(int button, int state, int x, int y)) determinano l'esecuzione di una funzione ogni volta che un tasto della tastiera o del mouse viene premuto o rilasciato
- glutMotionFunc(void (*func)(int x, int y)) registra una funzione da richiamare quando il mouse si muove e un tasto è contemporaneamente premuto

Installazione

- Il supporto a OpenGL (gl e glu) è offerto dai drivers della scheda video
  - In Windows è sufficiente installare Microsoft Visual C++, che contiene gli headers necessari
  - In Linux è generalmente sufficiente installare alcuni pacchetti della particolare distribuzione (nota che i pacchetti possono variare a seconda della distribuzione e della particolare versione)
- GLUT necessita di un'installazione separata sia delle librerie dinamiche che degli header
  - Per Windows trovate il file glut-3.7.6-bin.zip nei materiali di questa lezione, con le istruzioni per una corretta installazione
  - Per Linux trovate il file glut-3.7.6-src.zip con i sorgenti da ricompilare (c'è anche il progetto vc++); più semplice è individuare il pacchetto della vostra distribuzione e installare quello

Esempio di programma GLUT

- Verificate il successo della fase d'installazione eseguendo il seguente programma che mostra la versione di OpenGL installata sulla vostra macchina e altre informazioni

`OpenGLVersion.c`

`hello.c`

Compilazione

- Un programma OpenGL necessita il linkaggio in fase di compilazione di alcune librerie
- Il programma hello.c utilizza chiamate a funzioni gl e glut ed è sufficiente pertanto linkare le librerie relative a OpenGL e GLUT:

```
cc hello.c -o hello -lGL -lglut
```

- Se nel programma fossero state presenti chiamate a funzioni GLU allora:

```
cc hello.c -o hello -lGL -lGLU -lglut
```

Processi in background di un'applicazione GLUT

- glutIdleFunc(void (*func)(void)) specifica una funzione che viene eseguita mentre nessun altro evento deve essere soddisfatto
- Passare NULL come argomento implica la disattivazione del processo in background
- La Idle Function può essere utilizzata per l'esecuzione di procedure di calcolo forzando se necessario il redisplay tramite la funzione glutPostRedisplay

Esempio di programma GLUT con eventi e Idle Function

- Vedi file allegato:

  `double.c`

- Compilazione

```
cc double.c -o double -lGL -lglut
```

Materiale di riferimento sul Web

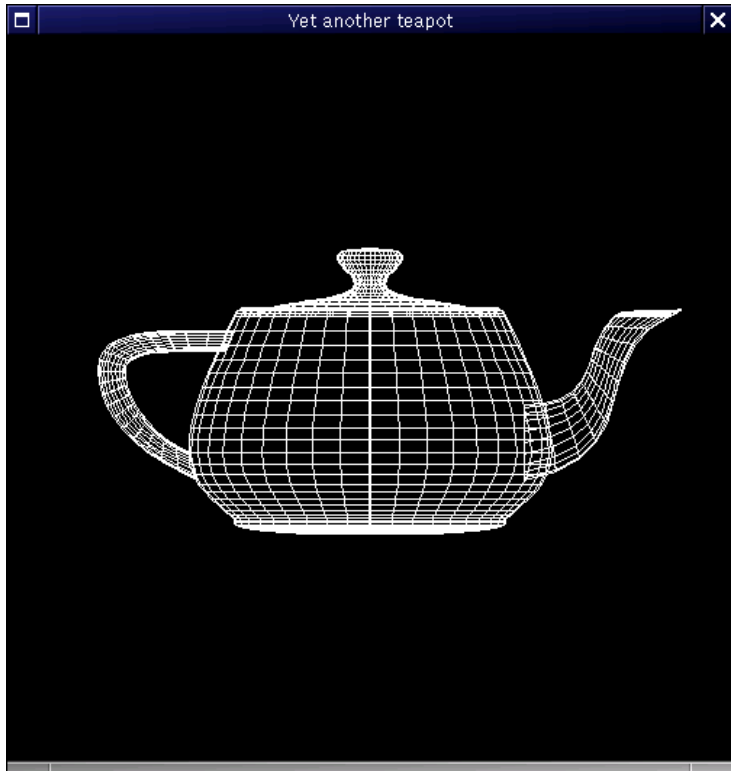
- The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3 (il pdf nei materiali di questa lezione):
  `http://www.opengl.org/resources/libraries/glut/spec3/spec3.html`
- GLUT Tutorial
  `http://www.lighthouse3d.com/opengl/glut/`

I fondamenti di OpenGL

I fondamenti di OpenGL

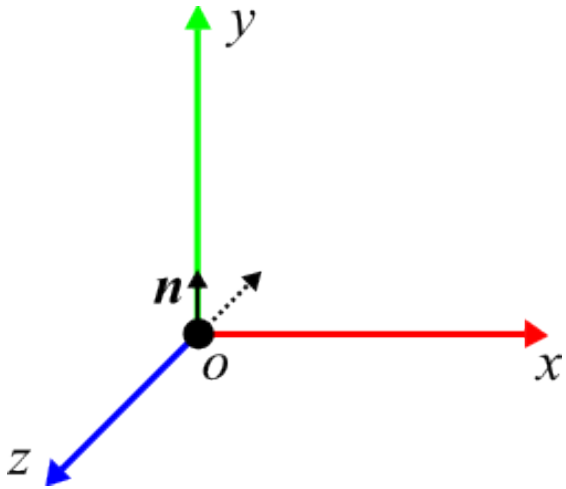
La viewport: glViewPort

- La vieport è l'area della finestra che contiene l'output grafico
- La funzione glViewPort(0,0,w,h) riserva l'intera area client della finestra come area per l'output grafico: l'angolo in basso a sinistra coincide con il punto (0,0), quello in alto a destra con il punto (w,h); w e h sono infatti il numero di colonne e di linee della finestra in pixel



Sistema di riferimento e osservatore

- La vieport mostra il rendering della modellazione geometrica che in OpenGL è sempre riferita al seguente sistema di coordinate cartesiano ortogonale



- L'osservatore è posto nell'origine, è orientato come l'asse *y* (nel senso che il vettore normale **n** all'osservatore è parallelo all'asse *y*) e guarda verso le zeta negative

Sistema di riferimento e osservatore

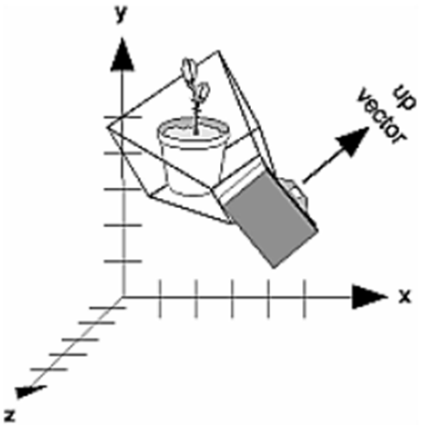
- Le impostazioni iniziali relative all'osservatore implicano che tutti gli oggetti modellati sulle z positive non risultino visibili
- La trasformazione di viewing consente di ridefinire il punto di vista dell'osservatore posizionandolo in un qualsiasi punto dello spazio 3D e specificando la direzione di vista
- Generalmente, la trasformazione di viewing è la prima a essere definita in un'applicazione OpenGL subito dopo la trasformazione di proiezione, descritta di seguito

Sistema di riferimento e osservatore

- La funzione gluLookAt permette di definire esplicitamente il punto di vista dell'osservatore

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

  - eyex, eyey, eyez specificano il punto d'applicazione
  - centerx, centery, centerz specificano il punto verso cui si guarda
  - upx, upy, upz specificano la direzione dell'up vector



Le trasformazioni di proiezione

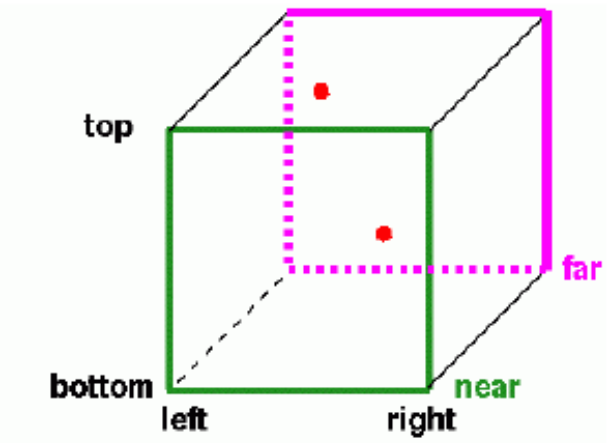
- Le trasformazioni di proiezione definiscono forma e dimensione del volume di vista (ortogonale o prospettico) e il tipo di proiezione delle primitive geometriche
- Il volume di vista ortogonale è parallelepipoidale e da luogo proiezioni per raggi perpendicolari al primo piano di clipping
- Il volume di vista prospettico è a forma di tronco di piramide e da luogo a proiezioni per raggi che convergono verso un punto di fuga che idealmente è rappresentato dalla punta della piramide
- La modalità di proiezione si attiva con la chiamata a

```
glMatrixMode(GL_PROJECTION);
```

Il volume di vista ortogonale: glOrtho

- La funzione glOrtho (glOrtho2D) definisce la porzione di spazio (piano) visibile e il sistema di coordinate cartesiano al suo interno

```
GLvoid glOrtho(GLdouble left, GLdouble right, GLdouble
    bottom, GLdouble top, GLdouble zNearClip, GLdouble
    zFarClip)
```



Il volume di vista ortogonale: proiezione

- La proiezione della geometria contenuta nel volume di vista ortogonale avviene per linee ortogonali al primo piano di clipping

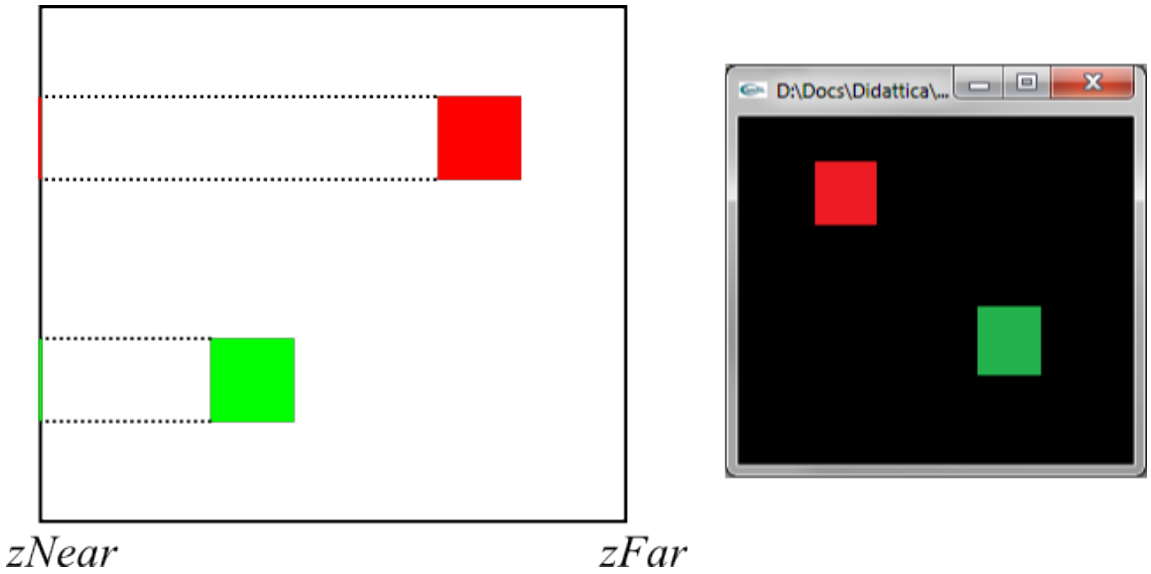


Figura: Esempio di proiezione sul primo piano di clipping nel caso di volume di vista ortogonale (vista in sezione)

Il volume di vista ortogonale: proiezione sulla viewport

- Il volume di vista definito da glOrtho viene proiettato sulla viewport; pertanto, se il volume di vista è un cubo e la viewport un quadrato, tutto funziona alla perfezione e le proporzioni sono mantenute
- Se, al contrario, la viewport non è un quadrato, la proiezione produce una distorsione

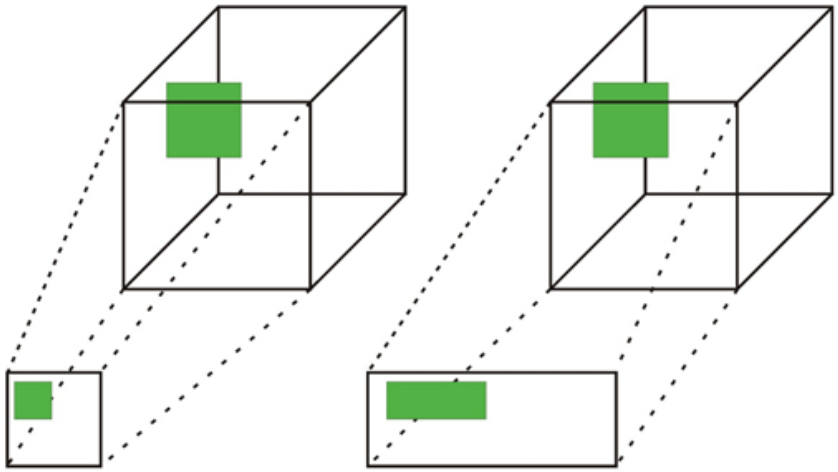


Figura: Esempio di proiezione sul primo piano di clipping nel caso di volume di vista ortogonale (vista in sezione)

Il volume di vista ortogonale: proiezione sulla viewport

- Per evitare problemi di distorsione è opportuno preservare le proporzioni del volume di vista rispetto alla viewport

```
glViewport(0, 0, w, h);
aspect_ratio = (GLfloat)w/(GLfloat)h;
if (aspect_ratio <= 1) // w <= h
    glOrtho(-100,100,-100/aspect_ratio,100/aspect_ratio
        ,1,-1);
else
    glOrtho(-100*aspect_ratio,100*aspect_ratio
        ,-100,100,1,-1);
```

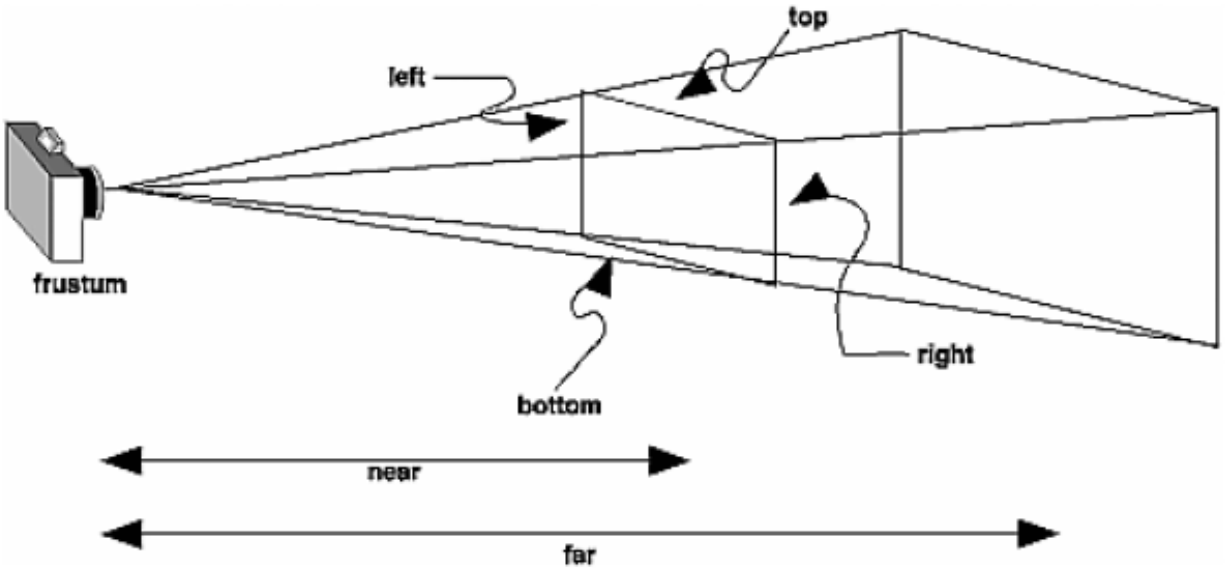
Esempio:

`GLRect.c`

Il volume di vista prospettico: glFrustum

- Il volume di vista è un tronco di piramide, detto frustum

```
void glFrustum( GLdouble left, GLdouble right, GLdouble
    bottom, GLdouble top, GLdouble near, GLdouble far )
```



Il volume di vista prospettico: proiezione

- Quando si usa un volume di vista prospettico, gli oggetti che sono più lontani dal punto di vista dell'osservatore risultano più piccoli rispetto a quelli che sono più vicini
- Infatti, la proiezione della geometria contenuta nel volume di vista prospettico avviene per linee convergenti al punto di fuoco

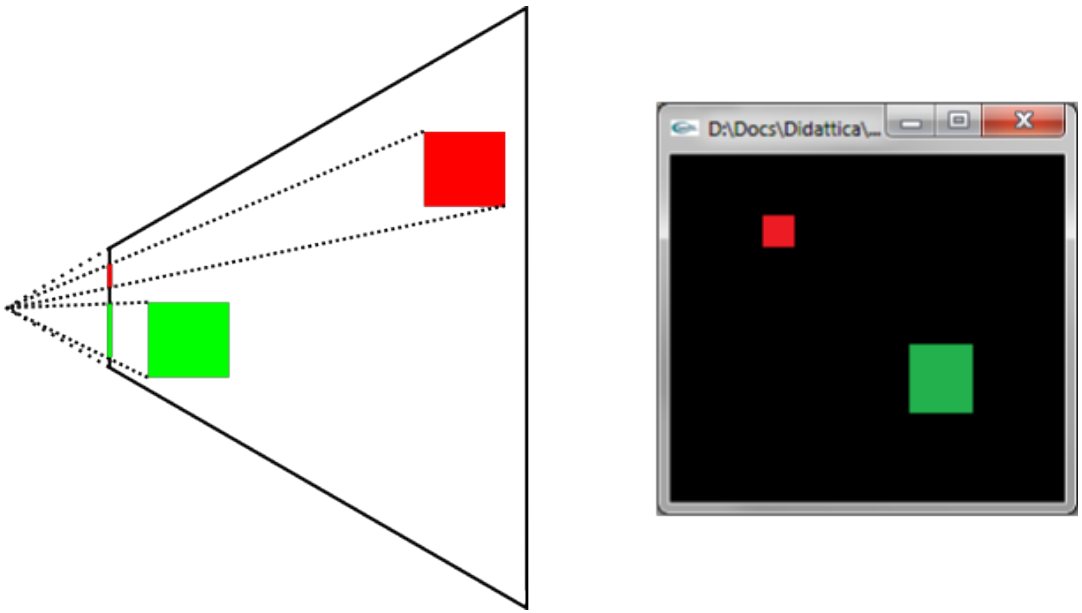
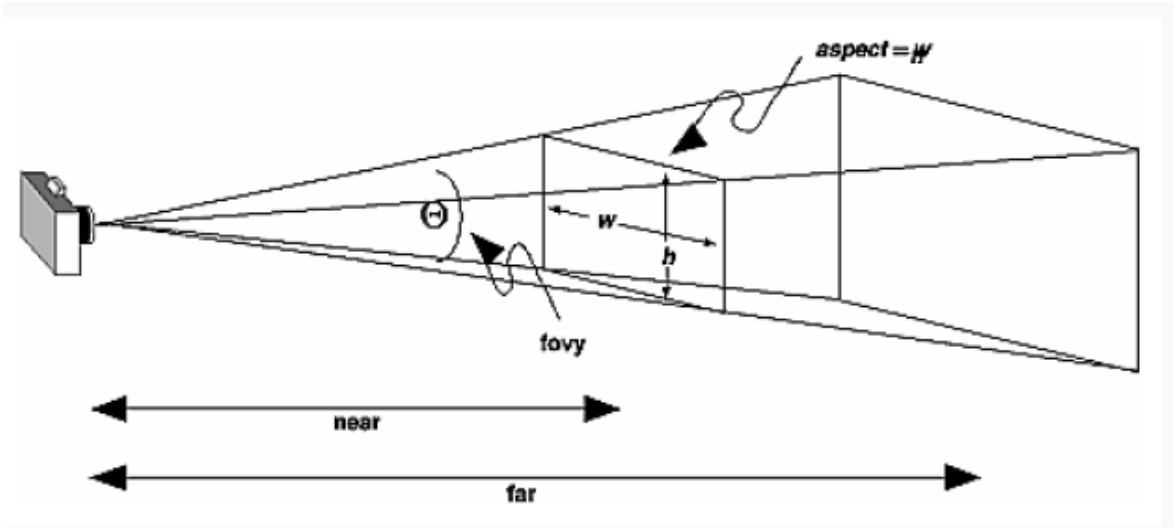


Figura: Esempio di proiezione sul primo piano di clipping nel caso di volume di vista ortogonale (vista in sezione)

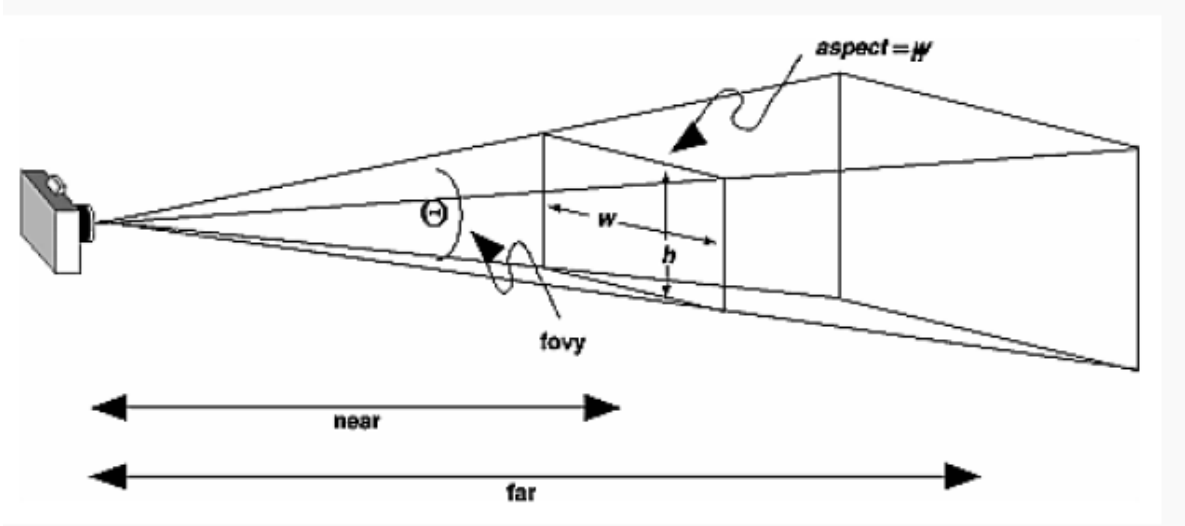
Il volume di vista prospettico: gluPerspective

- Benché sia concettualmente abbastanza semplice da comprendere, glFrustum può risultare antiintuitivo da usare; per tale motivo la libreria GLU fornisce la funzione alternativa gluPerspective per la definizione del frustum

```
void gluPerspective( GLdouble fovy, GLdouble aspect,
    GLdouble zNear, GLdouble zFar )
```



Il volume di vista prospettico: gluPerspective



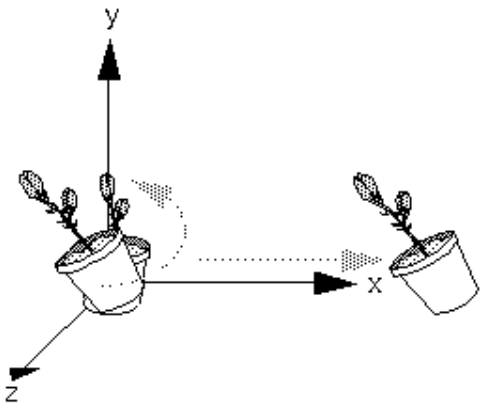
- fovy $\in [0.0, 180.0]$ è l'angolo di vista che giace nel piano $x - z$
- aspect è l'aspect ratio del frustum, (width / height); è buona norma impostarlo allo stesso aspect ratio della viewport per evitare distorsioni sulla viewport
- near e far sono le distanza dai piani di clipping

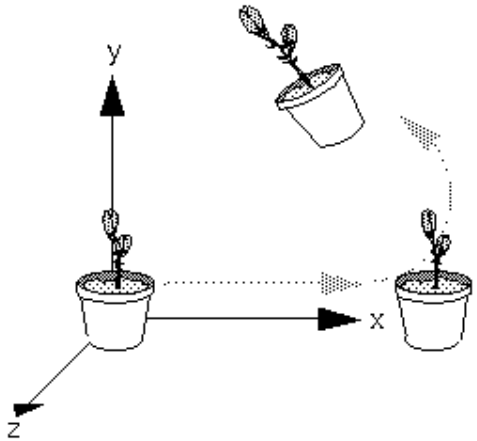
Le trasformazioni di modeling

- Una volta definiti il punto e il volume di vista è possibile occuparsi della modellazione geometrica al fine di definire i modelli che compongono la scena
- In molti casi, la definizione di un modello può risultare semplificata se viene effettuata intorno all'origine del sistema di coordinate; il modello può successivamente essere collocato in un qualsiasi punto dello spazio 3D attraverso opportune trasformazioni (traslazioni, rotazioni, trasformazioni di scala)
- Le trasformazioni di modeling consentono di posizionare nello spazio, ruotare e deformare primitive geometriche o, più in generale, modelli 3D

Le trasformazioni di modeling

- Nota che l'ordine con cui sono applicate le trasformazioni di modeling non è commutativo; infatti, in generale, se si applica prima una rotazione e poi una traslazione il risultato è diverso rispetto a quello che si ottiene applicando prima la traslazione e poi la rotazione
- Il motivo per cui gli effetti delle trasformazioni sono differenti è dovuto al fatto che ogni trasformazione è eseguita rispetto alla trasformazione precedente





La matematica delle trasformazioni

- La matematica che sottende alle trasformazioni di modeling (traslazioni, rotazioni e trasformazioni di scala) è basata sul calcolo matriciale
- Applicare una trasformazione a un vertice equivale, infatti, a moltiplicare una matrice che descrive la trasformazione, M per il vettore che descrive il vertice, v: Mv
- Innanzitutto, il vertice è trasformato in una matrice $4x1$ dove i primi 3 valori sono le coordinate x, y e z dei vertici; il quarto valore, detto coordinata w, è necessario per riportare il problema in termini di coordinate omogee
- Vedremo che questo è necessario per ricondurre le trasformazioni fondamentali al prodotto di una matrice ($4x4$) per un vertice ($4x1$)

La matematica delle trasformazioni: traslazione

- Matrice di traslazione

$$T = \begin{pmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Esempio: traslazione di una unità lungo l'asse z del vertice $v^T = (1, 0, 0, 1)$

$$T_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

risulta: $v' = T_z v^T = (1, 0, 1, 1)$

La matematica delle trasformazioni: rotazioni intorno agli assi x e y

- Rotazione intorno all'asse x

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Rotazione intorno all'asse y

$$R_y = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

La matematica delle trasformazioni: rotazioni intorno all'asse z e scaling

- Rotazione intorno all'asse z

$$R_z = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Trasformazione di scala (Scaling)

$$S = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

La matematica delle trasformazioni: ordine d'applicazione

- La condizione nella quale non è definita alcuna trasformazione di modeling equivale ad avere la *matrice identità*, *I*, come matrice di trasformazione corrente
- Ogni volta che il programmatore definisce una trasformazione, *M*, questa viene moltiplicata a destra per la matrice di trasformazione corrente; se *M* è la prima trasformazione definita dall'utente, allora la matrice di trasformazione corrente diventa *IM*
- La modellazione che segue la definizione delle trasformazioni, ad esempio la definizione di un vertice *v*, risulterà *trasformata* poiché *v* è moltiplicato a destra per la matrice di trasformazione corrente: *IMv*

La matematica delle trasformazioni: ordine d'applicazione

- Si noti quindi che, poiché le trasformazioni geometriche si ottengano tramite prodotti su matrici, l'ordine d'applicazione alla geometria è l'opposto di quello di definizione; ad esempio, se definiamo nell'ordine le trasformazioni:
  - S: trasformazione di scaling
  - T: trasformazione di traslazione
  - R: trasformazione di rotazione

  al vertice v, il vertice trasformato, v', risulterà da:

$$v \rightarrow Rv \rightarrow T(Rv) \rightarrow S(T(Rv)) = v'$$

- In altri termini, sarà applicata prima la rotazione R, al vertice così ottenuto la traslazione T e, infine, la trasformazione S

La matematica delle trasformazioni: ordine d'applicazione

- Un modo alternativo per interpretare l'effetto delle trasformazioni è quello di considerare un sistema di coordinate locali solidale al modello; così, l'ordine di applicazione della trasformazioni appare come nell'ordine naturale in cui sono definite nel codice sorgente

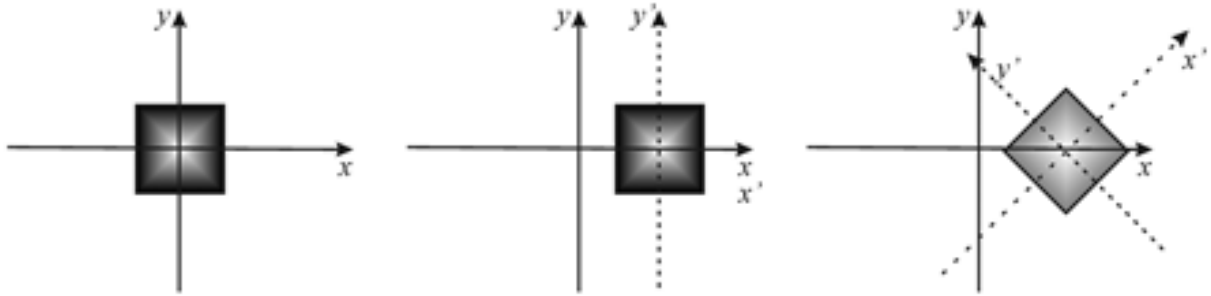


Figura: Esempio trasformazioni di modeling con sistema di riferimento locale: prima è definita la traslazione T, quindi la rotazione R

Trasformazioni di modeling in OpenGL: glLoadMatrix{fd}() e glMultMatrix{fd}()

- Se si vuole specificare esplicitamente una matrice di trasformazione e impostarla come trasformazione corrente si usa la funzione

```
void glLoadMatrix{fd}(const TYPE *M);
```

- Allo stesso modo, si usa la funzione glMultMatrix per moltiplicare la matrice corrente per la matrice passata come argomento

```
void glMultMatrix{fd}(const TYPE *M);
```

Trasformazioni di modeling in OpenGL: glLoadMatrix{fd}() e glMultMatrix{fd}()

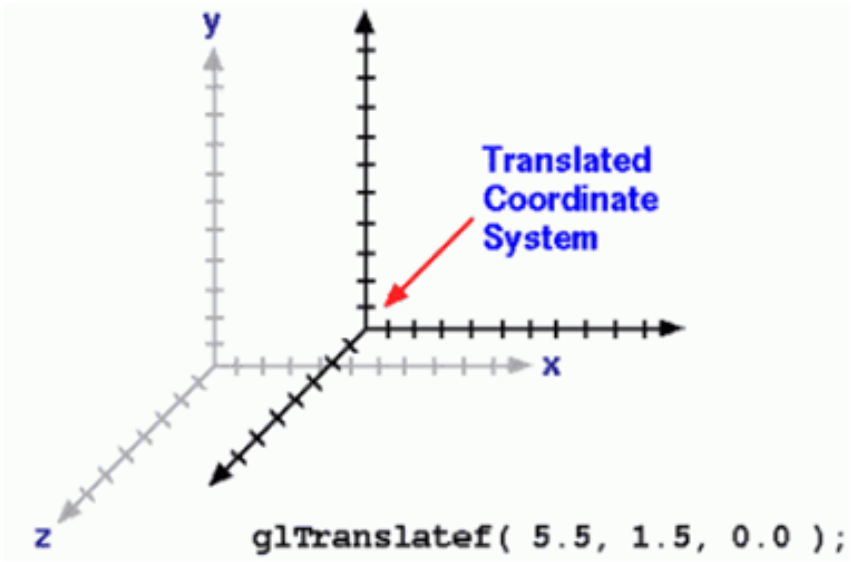
- Si noti che l'argomento per entrambi i comandi glLoadMatrix e glMultMatrix è un vettore di 16 valori $(m_1, m_2, ..., m_{16})$ che specifica una matrice M come segue:

$$M = \begin{pmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{pmatrix}$$

Trasformazioni di modeling in OpenGL: glTranslate{fd}

- La funzione glTranslate definisce una matrice di traslazione e la moltiplica per la matrice di modeling corrente

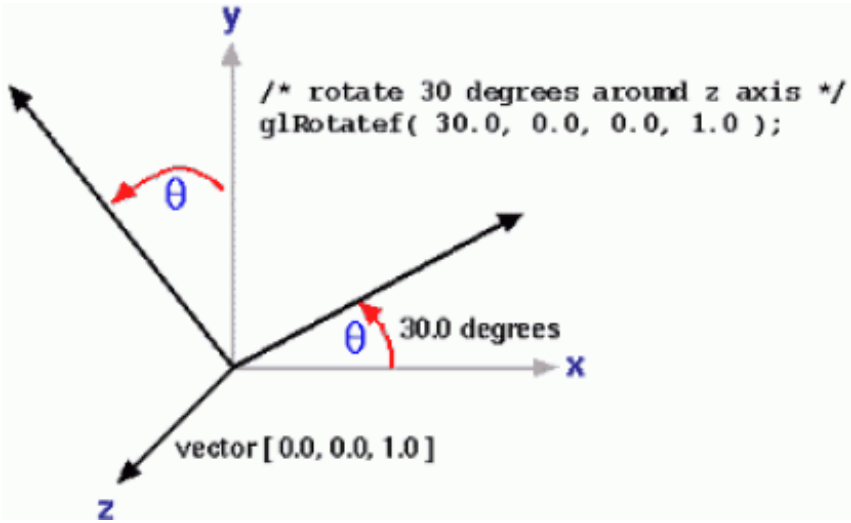
```
void glTranslated( GLdouble x, GLdouble y, GLdouble z )
void glTranslatef( GLfloat x, GLfloat y, GLfloat z )
```



Trasformazioni di modeling in OpenGL: glRotate{fd}

- La funzione glRotate definisce una matrice di rotazione e la moltiplica per la matrice di modeling corrente

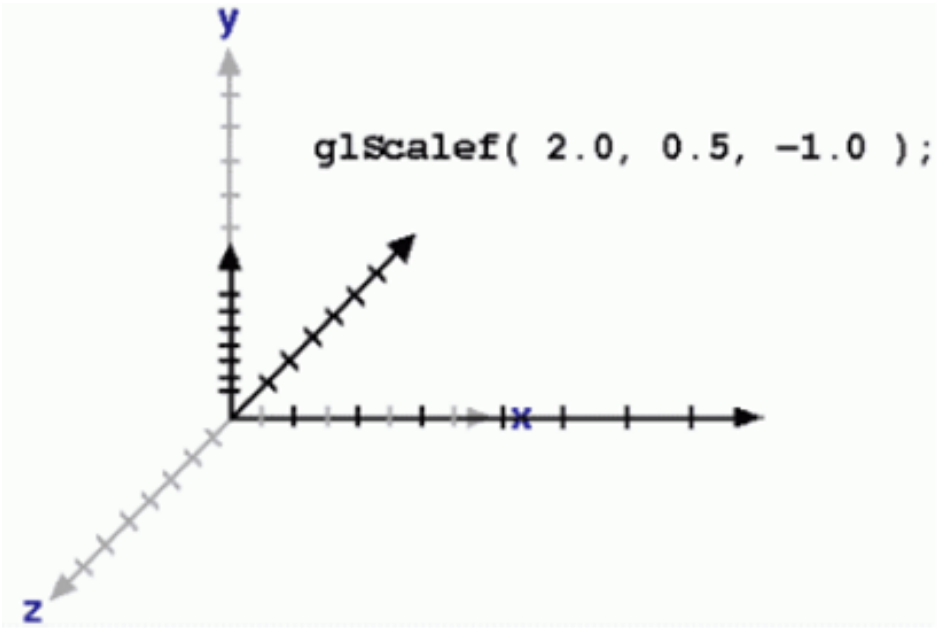
```
void glRotated( GLdouble angle, GLdouble x, GLdouble y,
    GLdouble z )
void glRotatef( GLfloat angle, GLfloat x, GLfloat y,
    GLfloat z )
```



Trasformazioni di modeling in OpenGL: glScale{fd}

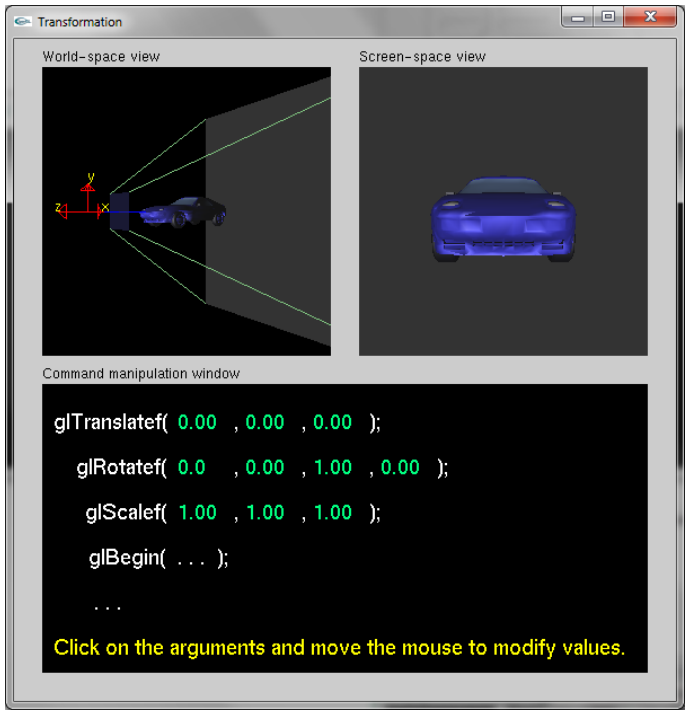
- La funzione glScale definisce una matrice di scaling e la moltiplica per la matrice di modeling corrente

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



Trasformazioni di modeling in OpenGL: Nate Robins' tutorial

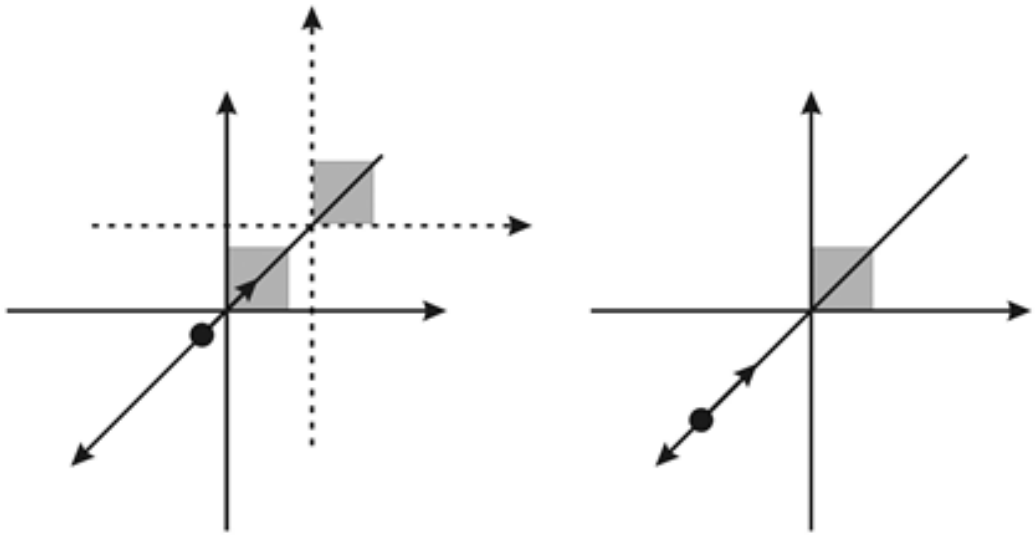
- Nate Robins ha sviluppato un tutorial per osservare l'effetto delle verie trasformazioni



`http://www.xmission.com/~nate/tutors.html`

La dualità modelview

- La distinzione tra trasformazione di viewing e modeling è puramente formale
- Si consideri una trasformazione di traslazione di 10 unità lungo l'asse *z* negativo; ebbene, questa è del tutto equivalente a una trasformazione di viewing di 10 unità lungo l'asse *z* positivo



La dualità modelview

- OpenGL non distingue tra trasformzione di viewing e di modeling; la modalità modelview di OpenGL gestisce le trasformazioni di viewing e modeling
- In modalità modelview, per una questione di comodità, è possibile specificare esplicitamente sia trasformazioni di viewing che di modeling anche se OpenGL le gestisce internamente come un unico tipo di trasformazione (di modeling)
- Per attivare la modalità modelview OpenGL ricorre alla funzione glMatrixMode

```
glMatrixMode(GL_MODELVIEW);
```

Caricare la matrice indentità: glLoadIdentity()

- In definitiva, in OpenGL ogni primitiva geometrica è sottoposta alle seguenti trasformazioni
  - modelview
  - Projection
  - Viewport
- Le trasformazioni di modelview e di proiezione sono gestite attraverso matrici ed è opportuno *resettarle* quando necessario, per esempio all'inizio dell'esecuzione dell'applicazione o quando si vuole annullare l'effetto di trasformazioni precedenti
- la funzione glLoadIdentity rimpiazza la matrice corrente (di modelview o di proiezione) con la matrice identità (nessuna trasformazione applicata)

```
void glLoadIdentity();
```

La pila di matrici OpenGL

- Ricaricare la matrice identità per resettare la matrice di modelview può risultare oneroso dal punto di vista computazionale e talvolta scomodo per il programmatore (che magari deve riapplicare alcune trasformazioni precedentemente definite)
- OpenGL fornisce un meccanismo alternativo: la pila di matrici
- La pila può essere adoperata sia in modalità modelview che projection; i comandi per manipolare la pila sono glPushMatrix per inserire la matrice di trasformazione corrente nella pila e glPopMatrix per recuperare l'ultima matrice inserita

```
void glPushMatrix( void );
void glPopMatrix( void );
```

La pila di matrici OpenGL

- Lo stack di matrici di modelview contiene almeno 32 matrici
- Poiché alcune implementazioni di OpenGL supportano più di 32 matrici nello stack di modelview, può essere necessario controllarne la dimensione massima attraverso la funzione

```
glGetIntegerv(GL_MAX_MODELVIEW_STACK_DEPTH, GLint *params)
```

- Lo stack delle matrici di proiezione contiene 2 matrici
- Poiché alcune implementazioni di OpenGL supportano più di 2 matrici nello stack di proiezione, può essere necessario controllarne la dimensione massima attraverso la funzione

```
glGetIntegerv(GL_MAX_PROJECTION_STACK_DEPTH, GLint *params)
```

Analisi del sorgente planet.c

```
static int year = 0, day = 0;
/* missing */

int main(int argc, char** argv){
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

Analisi del sorgente planet.c

```
void init(void){
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);
}


void keyboard (unsigned char key, int x, int y){
    switch (key) {
        case 'd':
            day = (day + 10) % 360;
            glutPostRedisplay();
            break;
        case 'y':
            /* altro */
    }
}
```

Analisi del sorgente planet.c

```
void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
```

Analisi del sorgente planet.c

```
void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);

    glPushMatrix();
        glutWireSphere(1.0, 20, 16); /* draw sun */
        glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
        glTranslatef (2.0, 0.0, 0.0);
        glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);
        glutWireSphere(0.2, 10, 8); /* draw smaller planet */
    glPopMatrix();
    glutSwapBuffers();
}
```

Analisi del sorgente planet.c

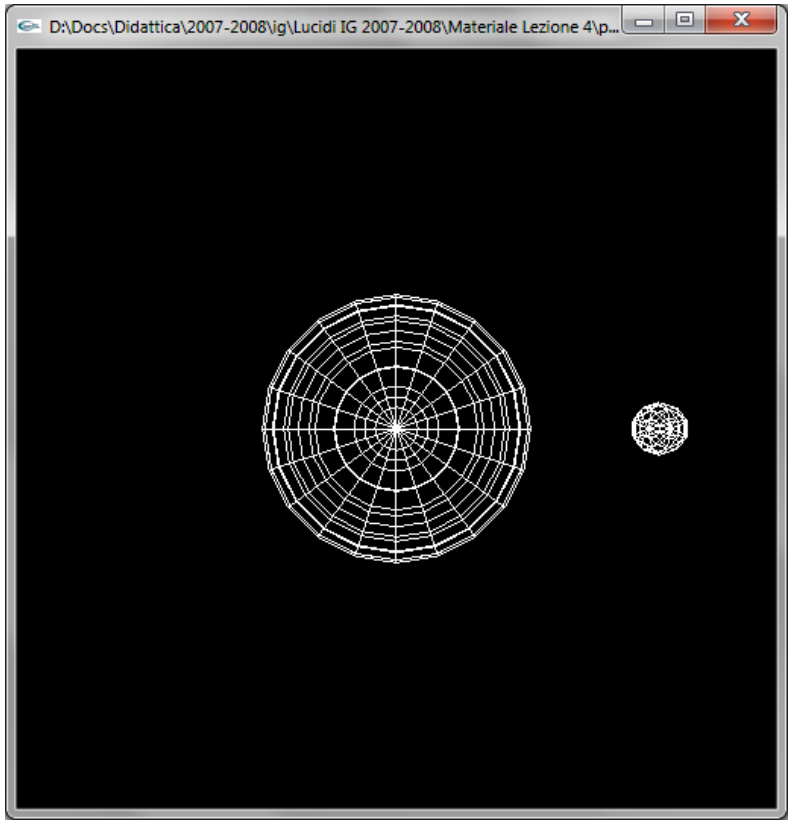


Figura: Screenshot del programma planet

Rimozione delle superfici nascoste

- Se nell'esempio precedente si prova a sostituire le chiamate a glutWireSphere con chiamate a glutSolidSphere provvedendo ad assegnare alle diverse sfere colori differenti, si noterà che non è possibile in alcun modo simulare l'effetto eclissi per cui le sfere che ruotano dietro (più in profondità rispetto) ad altre sfere continuano a essere visibili
- Quando si modellano scene attraverso poligoni ombreggiati gli oggetti nascosti da altri oggetti non devono evidentemente essere renderizzati
- L'eliminazione di oggetti (o di parti di oggetti) solidi che sono oscurati da altri oggetti è chiamata hidden-surface removal

Rimozione delle superfici nascoste

- Il modo più semplice per rimuovere le superfici nascoste è usare il depth buffer, noto anche come z-buffer, che è uno dei buffer del frame buffer
- Il depth buffer è utilizzato per associare a ogni pixel dell'immagine nel color buffer la distanza dal primo piano di clipping
- Inizialmente, il depth buffer è inizializzato usando la più grande distanza possibile (la distanza dall'ultimo piano di clipping)
- Prima che ogni pixel sia effettivamente settato nel color buffer si effettua un controllo con la distanza associata: se il nuovo valore di profondità è minore allora il relativo colore sostituisce quello attualmente impostato per il pixel; in caso contrario non accade nulla

Rimozione delle superfici nascoste

- Il depth buffering deve essere attivato attraverso il comando glEnable(GL_DEPTH_TEST)
- Inoltre, prima di modellare la scena è necessario resettare il depth buffer attraverso il comando glClear(GL_DEPTH_BUFFER_BIT)
- Infine, è necessario specificare la modalità di rimozione delle superfici nascosta a GLUT attraverso il comando glutInitDisplayMode (GLUT_DEPTH | . . .)

Screenshot del programma planet2

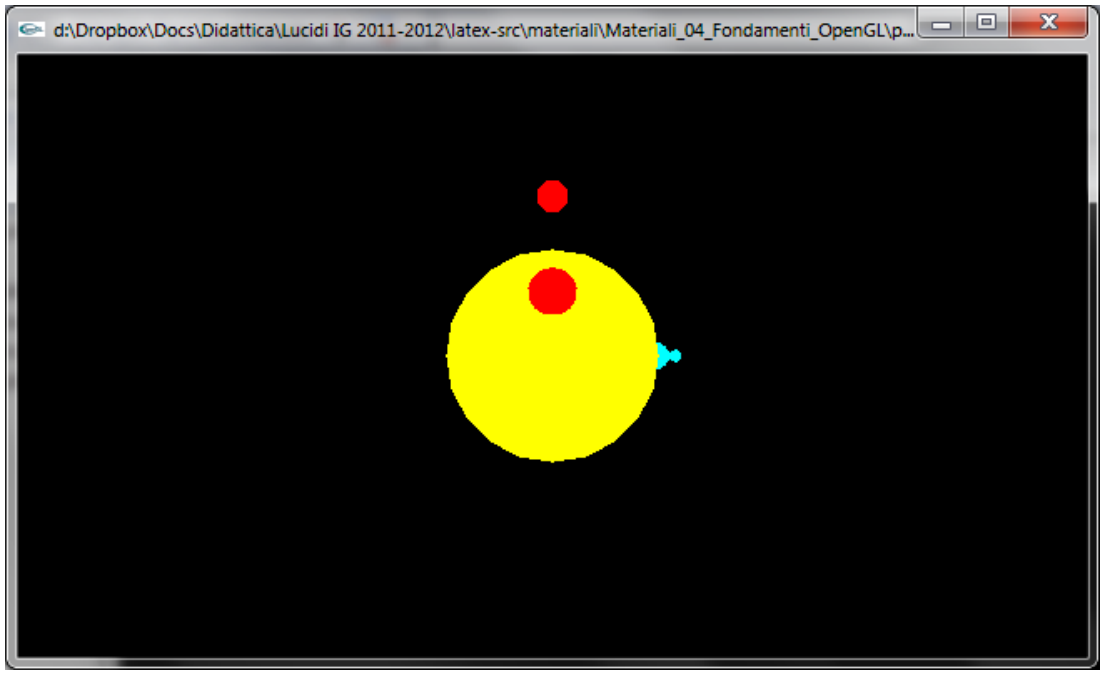
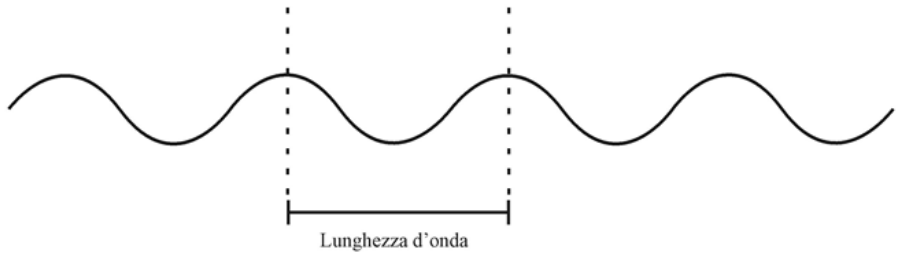


Figura: Screenshot del programma planet2 con controllo di profondità attivato

Il colore: la natura ondulatoria della luce

- Il colore è una lunghezza d'onda di luce visibile all'occhio umano
- La luce visibile di una sorgente è in realtà una miscela di differenti tipi di luce, ognuna caratterizzata dalla propria lunghezza d'onda
- La lunghezza d'onda della luce è misurata come distanza tra due picchi successivi dell'onda di luce



Il colore: la natura ondulatoria della luce

- La luce visibile varia tra 390 (violetto) e 720 nanometri (rosso); questo range è detto spettro visibile e contiene i colori dell'arcobaleno
- La luce ultravioletta è quella luce caratterizzata da lunghezza d'onda inferiore a 390 nanometri; la luce infrarossa è quella luce caratterizzata da lunghezza d'onda maggiore di 720 nanometri
- Per quanto riguarda i rimanenti colori, essi sono combinazioni a varie intesità dei colori dello spettro visibile
- In particolare, il colore nero rappresenta l'assenza di luce, mentre il bianco è la combinazione di tutti i colori dello spettro alla massima intensità

Il colore: la natura particellare della luce

- Se si pensa alla luce nella sua natura particellare anziché ondulatoria, essa è costituita da piccolissime particelle, i fotoni
- Quando i fotoni incidono sulla superficie di un oggetto, una parte viene generalmente riflessa e una parte trattenuta
- L'occhio umano viene colpito dai fotoni riflessi (si tratta di miliardi di fotoni) che sono *messi a fuoco* sulla retina
- La retina contiene miliardi di cellule (i coni) che, *eccitate* dai fotoni, trasmettono segnali elettrici al cervello che li interpreta come informazioni di luce e colore

Il colore: la natura particellare della luce

- Maggiore è il numero di fotoni che colpisce la retina, più intensa risulta l'eccitazione dei coni e, di conseguenza, la percezione della luminosità
- In realtà esistono tre differenti tipologie di cellule coni, ognuna particolarmente sensibile a una ben precisa lunghezza d'onda:
  - Una reagisce principalmente alla luce rossa (red cones)
  - Una reagisce principalmente alla luce verde (green cones)
  - Una reagisce principalmente alla luce blu (blue cones)
- Una luce caratterizzata da una lunghezza d'onda prossima al rosso eccita principalmente i coni rossi e il cervello percepirà il colore rosso; una combinazione di differenti lunghezze d'onda produrrà l'eccitazione di più tipologie di coni e il cervello percepirà una *miscela* dei tre colori fondamentali

Il colore: il modello RGB

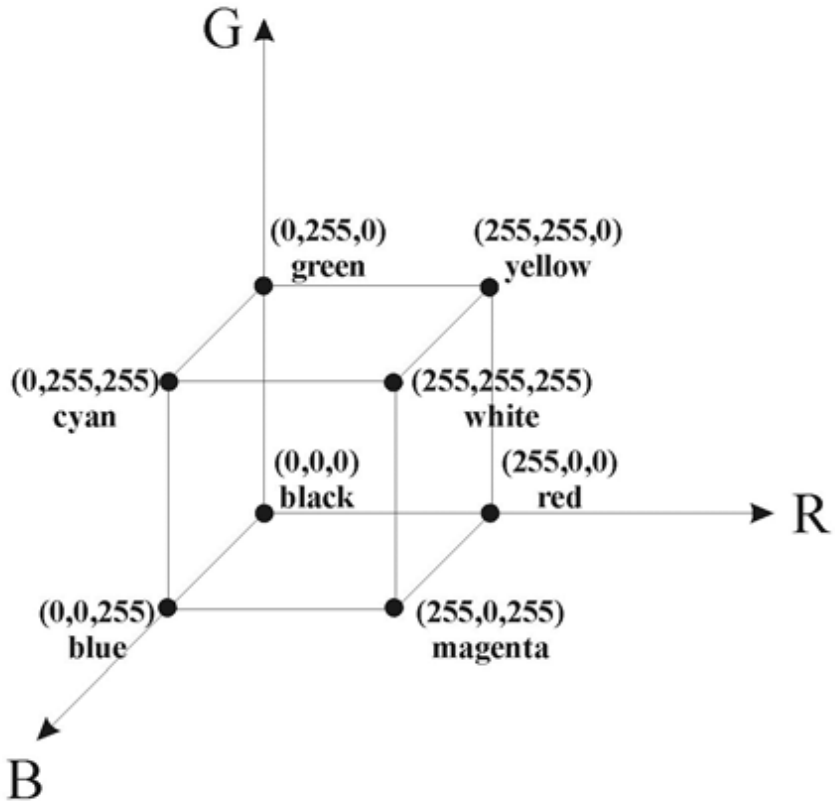
- Visto come funziona l'occhio umano e come il cervello interpreta le informazioni da esso trasmesso, risulta ovvio comprendere il modo in cui i calcolatori elettronici producono le immagini
- Ogni monitor è progettato per produrre tre tipi di luce Rossa, Green e Blu (modello RGB) con diverse intensità
  - Il cannone a elettroni di un CRT produce l'eccitazione dei fosfori posizionati sul retro del pannello video: ogni punto dello schermo (pixel) è composto da tre fosfori differenti, ognuno dei quali produce luce rossa, verde o blu, con differenti intensità
  - Similmente, i cristalli liquidi di un LCD sono progettati in modo da riprodurre i colori fondamentali attraverso l'attivazione di segnali elettrici nella matrice posta sul retro del pannello

Il colore: il modello RGB

- OpenGL adotta il modello di colore RGB a 24 bit
- La profondità di colore a 24bit consente di maneggiare 256 intensità per ognuno dei tre colori fondamentali
- Per motivi di performance, tuttavia, le schede grafiche supportano la modalità a 32bit di cui 24bit sono usati per i colori fondamentali, mentre i rimanenti 8 per il canale alpha (che vedremo in seguito)

Uso dei colori in OpenGL

- Poiché tutti i colori visibili sono specificati da terne di tre valori RGB, possiamo rappresentarli tutti attraverso il così detto RGB colorspace



Uso dei colori in OpenGL

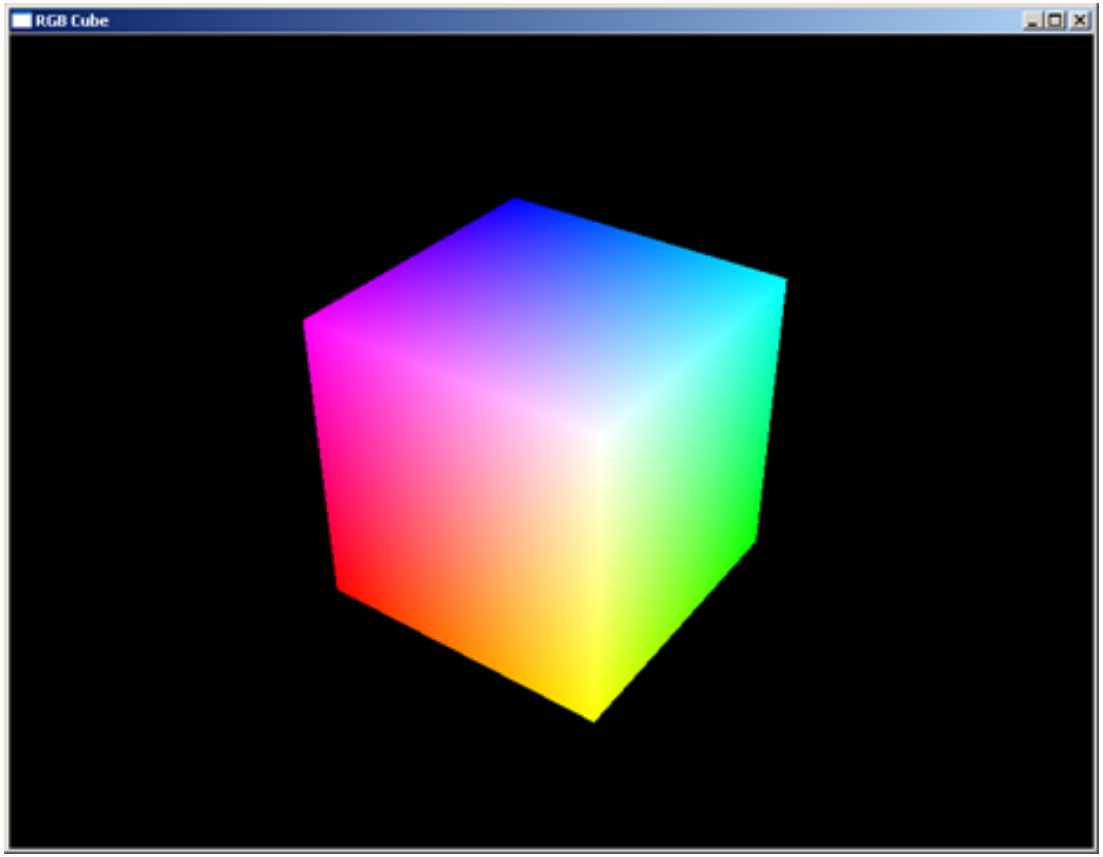


Figura: Screenshot del programma ccube.c (nei materiali della lezione)

Colore di sfondo

- OpenGL prevede una funzione per la definizione e l'applicazione del colore di sfondo

```
glClearColor (R, G, B, A)
glClear (GL_COLOR_BUFFER_BIT)
```

- In alternativa sarebbe possibile disegnare un rettangolo che copra l'intera area di disegno
- Questa seconda soluzione pone alcuni problemi:
  - è meno efficiente
  - può creare problemi in caso di applicazioni 3D

Impostazione del colore di disegno

- La funzione che setta il colore di disegno è glColor* (o, in forma vettoriale, glColor*v); Ad esempio:

```
glColor3f (0.0, 0.0, 0.0) /* nero */
glColor3f (1.0, 0.0, 0.0) /* rosso */
glColor3f (0.0, 1.0, 0.0) /* verde */
glColor3f (1.0, 1.0, 0.0) /* giallo */
glColor3f (0.0, 0.0, 1.0) /* blu */
glColor3f (1.0, 0.0, 1.0) /* magenta */
glColor3f (0.0, 1.0, 1.0) /* ciano */
glColor3f (1.0, 1.0, 1.0) /* bianco */
```

Impostazione del colore di disegno: shading

- Primitive di un solo colore sono dette FLAT Shaded; primitive con colori che sfumano gradualmente sono dette SMOOTH Shaded
- La chiamata alla funzione glShadeModel(GL_SMOOTH) attiva lo smooth shading, quella a glShadeModel(GL_FLAT), il flat shading; ad esempio:

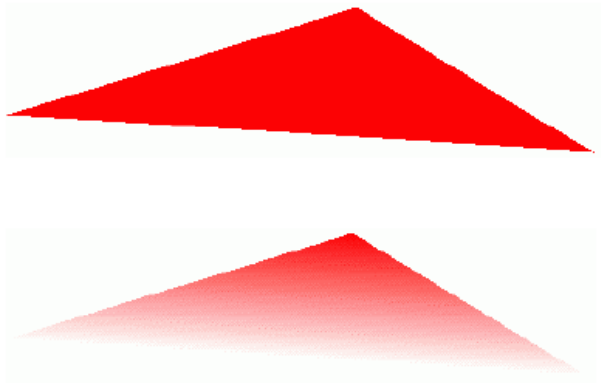


Figura: Esempio di flat e smooth shading

Modellazione di punti, segmenti e poligoni

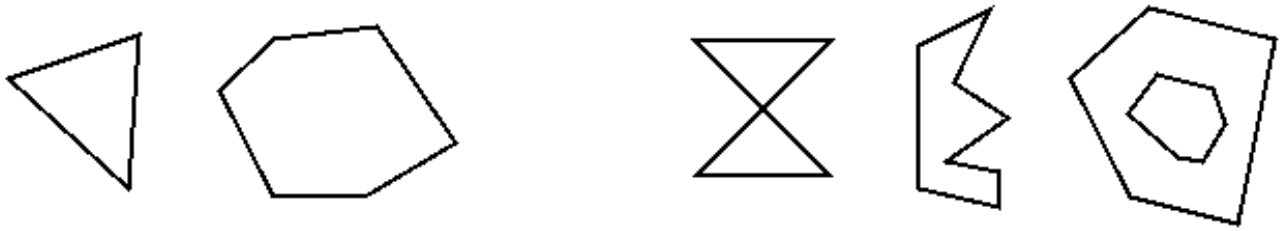
- Un punto è rappresentato da un insieme, chiamato vertice, di numeri floating-point
- Un vertice è definito da una tripla (x,y,z), anche se è possibile definire vertici 2D
  - Generalmente un punto ha la dimensione di un pixel, ma la sua grandezza può essere variata attraverso opportune funzioni OpenGL

  Esempio:

  `POINTS.C`
- Un segmento è rappresentato da una coppia di vertici
  - Generalmente un segmento ha lo spessore di un pixel, ma può essere variato

Modellazione di punti, segmenti e poligoni

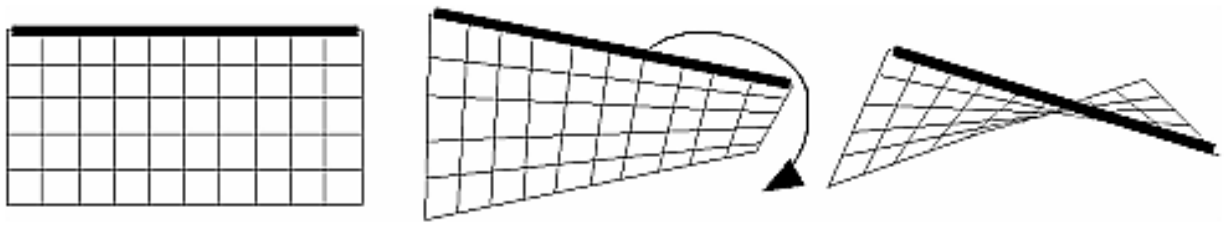
- I poligoni sono aree chiuse delimitate da segmenti
- Un poligono è determinato dai vertici dei segmenti che lo definiscono
- OpenGL impone alcune restrizioni sui poligoni
  - I lati non possono intersecarsi (poligoni semplici)
  - Devono essere convessi



- Poligoni *non validi* possono comunque essere rappresentati attraverso più poligoni *validi*

Poligoni planari e non planari

- Il triangolo è sempre un poligono planare visto che per tre punti (i vertici del triangolo) passa un solo piano mentre tutti gli altri poligoni (rettangoli, esagoni, ecc.) possono essere non planari
- La non planarità dei poligoni può creare alcuni problemi nel caso di applicazioni 3D con rotazioni, scaling, ecc. e produrre poligoni non validi, come nella figura sotto, che possono creare problemi agli algoritmi di rasterizzazione



Rettangoli

- Poiché i rettangoli sono molto comuni in applicazioni grafiche, OpenGL fornisce una funzione per il loro disegno:

```
glRect*(x1,y1,x2,y2)
```

- Esiste anche la versione vettoriale glRect*v.
- Il rettangolo viene sempre disegnato sul piano $z = 0$ ma può essere successivamente ruotato, traslato e scalato

Specificare i vertici

- In OpenGL ogni oggetto geometrico è, in definitiva, definito dai suoi vertici
- La funzione glVertex* definisce un vertice; ad esempio

```
glVertex2s(2,4);
glVertex3d(0.0, 0.0, 3.1415926535898);
glVertex3f(1.0, 7.1, 10.23);
```

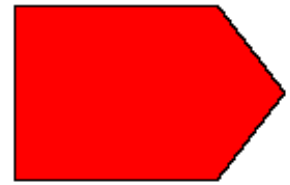
- Esiste anche la forma vettoriale glVertex*v, più efficiente su alcune macchine

```
GLfloat fv[3] = {5.0, 6.2, 8.9};
glVertex3fv(fv);
```

Il blocco glBegin() - glEnd()

- Per disegnare linee o poligoni a partire dai vertici è necessario definire questi ultimi all'interno delle chiamate a glBegin() e glEnd(); l'argomento di glBegin() definisce il tipo di oggetto che si vuole disegnare

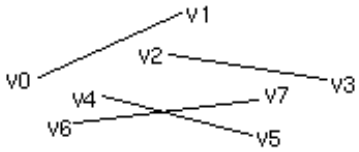
```
glBegin(GL_POLYGON);
    glColor3f(1.0, 0.0, 0.0);
    glVertex2f(0.0, 0.0);
    glVertex2f(0.0, 3.0);
    glVertex2f(4.0, 3.0);
    glVertex2f(6.0, 1.5);
    glVertex2f(4.0, 0.0);
glEnd();
```



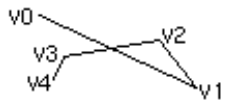
GL_POLYGON

Argomenti di glBegin()

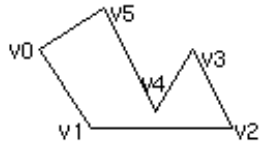
GL_POINTS



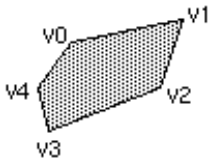
GL_LINES



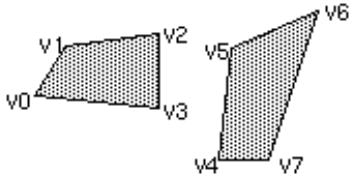
GL_LINE_STRIP



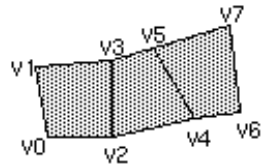
GL_LINE_LOOP



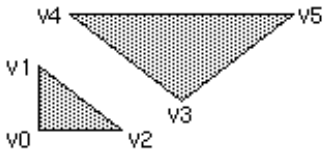
GL_POLYGON



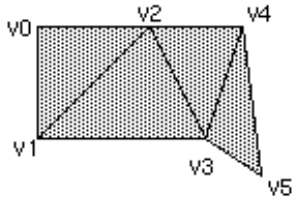
GL_QUADS



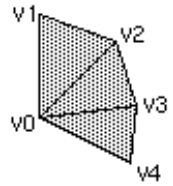
GL_QUAD_STRIP



GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

Proprietà dei vertici

- All'interno di glBegin() e glEnd() è possibile specificare ulteriori informazioni:
  - Colore del vertice
  - Vettore normale
  - Coordinata di texture
  - Ogni combinazione delle precedenti
  - altro

Command	Purpose of Command
glVertex*()	set vertex coordinates
glColor*()	set current color
glIndex*()	set current color index
glNormal*()	set normal vector coordinates
glTexCoord*()	set texture coordinates
glEdgeFlag*()	control drawing of edges
glMaterial*()	set material properties
glArrayElement()	extract vertex array data
glEvalCoord*(), glEvalPoint*()	generate coordinates
glCallList(), glCallLists()	execute display list(s)

Costrutti all'interno di glBegin() e glEnd()

```
//...
GLint circle_points = 100;
glBegin(GL_LINE_LOOP);
    for (i = 0; i < circle_points; i++) {
        angle = 2*M_PI*i/circle_points;
        glVertex2f(cos(angle), sin(angle));
    }
glEnd();
//...
```

- Nota che esistono alternative più efficienti per l'esempio precedente

Punti, segmenti e poligoni: dimensione dei punti

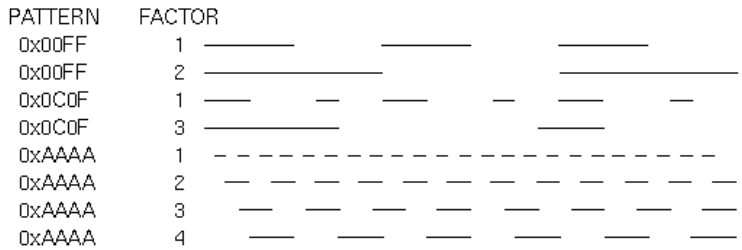
- Di default, un punto è disegnato come singolo pixel sullo schermo, una linea è disegnata continua e un poligono è disegnato pieno
- La funzione glPointSize(GLfloat size) setta la dimensione in pixel per i punti; il valore di default è ovviamente 1
- Così, la chiamata a glPointSize(2.1) indica che i punti devono essere disegnati usando 2.1 pixel in orizzontale e 2.1 in verticale
  - Se non è attivo l'antialiasing il valore 2.1 viene arrotondato al valore più vicino (2), per un totale di 4 pixel per punto
  - Se è attivo l'antialiasing l'arrotondamento non avviene e il valore è utilizzato per l'algoritmo di smoothing (che vedremo in seguito)

Punti, segmenti e poligoni: dimensione e pattern dei segmenti

- Come per i punti, anche per le linee è possibile specificare uno spessore diverso da 1.0 (valore di dafault) attraverso l'istruzione glLineWidth(GLfloat width)
- È inoltre possibile specificare anche un pattern per ottenere linee a puntini, tratteggiate, ecc., attraverso la funzione glLineStipple(GLint factor, GLushort pattern)
- Il line stippling deve inoltre essere attivato tramite glEnable(GL_LINE_STIPPLE)

Punti, segmenti e poligoni: dimensione e pattern dei segmenti

- Nella funzione glLineStipple(GLint factor, GLushort pattern):
  - pattern è una sequenza di 16 bit (punti) dove 0 vuol dire acceso, 1 spento; in genere è dato in forma esadecimale
  - factor è un fattore di stretching del pattern



- Ad esempio, il seguente codice attiva la modalità di stippling con pattern 0xAAAA e factor 1

```
glLineStipple(1, 0xAAAA);
glEnable(GL_LINE_STIPPLE);
```

Esempio

- Vedi allegato

`lines.c`

- Compilazione

```
cc line.c -o triangle -lGL -lglut
```

Punti, segmenti e poligoni: poligoni

- I poligoni sono generalmente disegnati pieni, ma è ovviamente possibile modificare le impostazioni di default
- Un poligono ha 2 facce (front e back) che possono essere disegnati sia nello stesso modo che in modo differente (per esempio il back può essere disegnato come linee)
- Per default, entrambe le facce sono disegnate nello stesso modo

Punti, segmenti e poligoni: poligoni

- void glPolygonMode(GLenum face, GLenum mode) specifica il modo di disegno per la parte front e back del poligono
  - face può essere GL_FRONT, GL_BACK o GL_FRONT_AND_BACK
  - mode può essere GL_POINT, GL_LINE o GL_FILL
- Per esempio, il seguente codice specifica il modo fill per il front e il modo line (detto anche wirefame) per il back

```
glPolygonMode(GL_FRONT, GL_FILL);
glPolygonMode(GL_BACK, GL_LINE);
```

- Per convenzione la parte front del poligono è quella per la quale la sequenza dei vertici che la definiscono sono in senso anti-orario (counterclockwise)

Punti, segmenti e poligoni: poligoni

- La funzione glFrontFace(GLenum mode) consente di cambiare la convenzione: mode può essere GL_CCW (counterclockwise) o GL_CW (clockwise)
- È opportuno costruire superfici orientabili (es. una sfera o un toro) con poligoni che abbiano la stessa orientazione; così, nel caso di una sfera, ad esempio, è possibile istruire OpenGL in modo che non disegni la parte back dei poligoni oppure, se il punto di vista è all'interno della sfera, si può istruire OpenGL in modo che non disegni la parte front
- La funzione void glCullFace(GLenum mode) determina la faccia dei poligoni che devono essere ignorati in fase di disegno; mode può essere GL_FRONT, GL_BACK o GL_FRONT_AND_BACK
- Nota che la funzione di culling deve essere attivata tramite la chiamata glEnable(GL_CULL_FACE);

Esempio

- Vedi allegato

  `Triangle.c`

- Compilazione

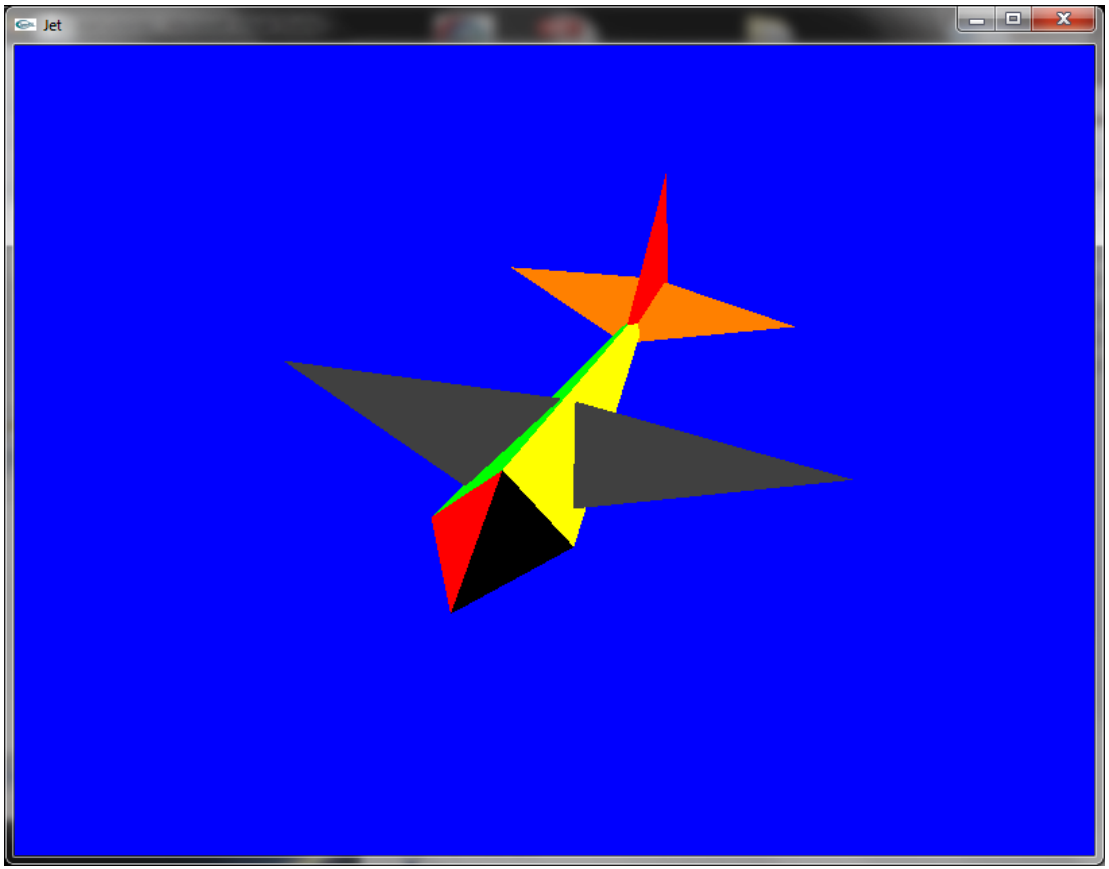
```
cc Triangle.c -o Triangle -lGL -lglut
```

Illuminazione

Illuminazione

Modelli d'illuminazione

- L'uso dei colori puro e semplice, come nell'esempio sotto, rende difficile realizzare modelli realistici



Modelli d'illuminazione

- Ciò che manca all'esempio precedente è un modello di illuminazione
- Ciò che rende realistico un modello 3D è infatti in gran parte il modo in cui la luce incide sul modello stesso
- OpenGL prevede tre differenti tipi di luci che possono essere aggiunte alla scena:
  - La luce ambiente (ambient light)
  - La luce diffusa (diffuse light)
  - La luce speculare (specular light)

Modelli d'illuminazione: Ambient Light

Caratteristiche della luce ambiente

- Non proviene da alcuna direzione
- Ha una sorgente, ma i raggi sono riflessi sulla scena e diventano a-direzionali
- Gli oggetti risultano illuminati nello stesso modo indipendentemente dall'orientazione delle facce (poligoni) nello spazio

- La scena precedente può essere considerata come illuminata da luce ambiente, benché non sia stato definito alcun modello di illuminazione

Modelli d'illuminazione: Diffuse Light

Caratteristiche della luce diffusa

- Proviene da una direzione specifica
- È riflessa in ogni direzione
- Il livello di luminosità dipende dall'angolo di incidenza sulla superficie
- Il massimo di luminosità si ottiene quando la luce incide perpendicolarmente

- La luce diffusa è fondamentale per conferire realismo alla scena nelle applicazioni grafiche

Modelli d'illuminazione: Specular Light

Caratteristiche della luce speculare

- Proviene da una direzione specifica
- È riflessa in modo speculare
- Il livello di luminosità dipende dall'angolo di incidenza sulla superficie
- Il massimo di luminosità si ottiene quando la luce incide perpendicolarmente

- Una forte luce speculare tende a generare un riflesso sulla superficie su cui incide (specular highlight)

Modelli d'illuminazione: le componenti delle sorgenti

- In generale, un modello d'illuminazione risulta dalla combinazione di piu tipi di luci
- In tal senso una luce in una scena è detta essere definita dalle tre componenti
  - Ambient
  - Diffuse
  - Specular
- Come per la definizione del colore, ogni componente della luce è definita da una quadrupla RGBA, in cui, tuttavia, la componente A è ignorata

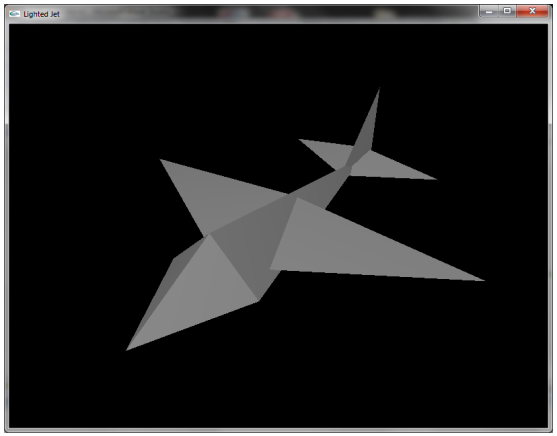
Modelli d'illuminazione: esempi

- Esempio: laser rosso

Type	R	G	B	A
Ambient	0.05	0.00	0.00	1.00
Diffuse	0.10	0.00	0.00	1.00
Specular	0.99	0.00	0.00	1.00

- Un altro esempio

Type	R	G	B	A
Ambient	0.20	0.20	0.20	1.00
Diffuse	0.70	0.70	0.70	1.00
Specular	0.20	0.20	0.20	1.00



Materiali

- Come abbiamo avuto modo di osservare in precedenza, gli oggetti del mondo reale non hanno di per se un colore, ma ne assumono uno in relazione alle proprie caratteristiche fisiche (come riflettono la luce) e al tipo di luce che li illumina
- Tuttavia, per convenzione, si definisce colore naturale di un oggetto il colore che l'oggetto assume quando è illuminato da una luce bianca, cioè una luce per cui (R,G,B,A) = (1, 1, 1, 1)
- Quando si utilizzano le luci non è più necessario specificare esplicitamente il colore degli oggetti nella scena, ma piuttosto il comportamento dell'oggetto rispetto alla luce; in altri termini, quello che si fa è specificare le caratteristiche del materiale di cui è composto l'oggetto

Materiali

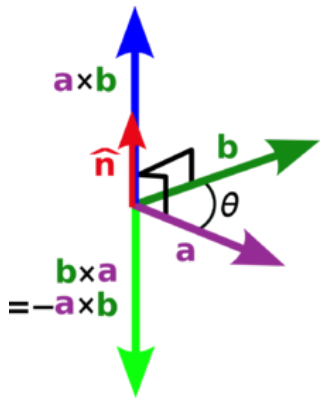
- In particolare è necessario definire le caratteristiche di riflessione della luce (sia di tipo ambient che diffuse che specular)
- Ad esempio, invece di dire che un poligono è rosso, diciamo che il poligono è composto da un materiale che riflette prevalentemente luce rossa (quindi anche nel caso sia colpito da una luce bianca che contiene una componente di luce rossa)
- Un materiale può, ad esempio, essere molto luminoso e riflettere bene la luce, oppure assorbire tutta la luce speculare

Effetti della luce ambiente

- Supponiamo di avere una sorgente di luce definita dalla terna (trascuriamo la componente A)
  $L_A = (R, G, B) = (0.5, 0.5, 0.5)$
- Se la luce L_A illumina un oggetto con la proprietà di riflettere la luce ambiente (componente di colore del materiale)
  $R_A = (0.5, 1.0, 0.5)$, allora, il colore risultante sarà:
  $C = L_A R_A = (0.25, 0.5, 0.25)$
- Così, la componente di colore del materiale determina la percentuale di luce che viene riflessa e che è quindi visibile

Effetti della luce diffusa e speculare

- Le luci diffuse e speculari sono direzionali e il livello massimo di illuminazione si ha quando la luce incide perpendicolarmente sulla superficie: la luminosità dipende quindi dall'angolo d'incidenza
- OpenGL calcola l'angolo d'incidenza grazie ai vettori normali; il vettore normale può essere calcolato sfruttando la definizione di prodotto vettoriale



Sorgenti di luce: calcolo delle normali

Prodotto vettoriale

Dati due vettori in a e b, il prodotto vettoriale è il vettore ortogonale sia ad a che a b tale che

$$a \times b = \mathbf{n}|a||b| \sin\theta$$

Calcolo del prodotto vettoriale in forma matriciale

$$a \times b = det \begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{pmatrix} =$$

$$= (a_2 b_3 - a_3 b2)\mathbf{i} + (a_3 b_1 - a_1 b3)\mathbf{j} + (a_1 b_2 - a_2 b1)\mathbf{k}$$

Sorgenti di luce: calcolo delle normali

Esempio: calcolo della normale da tre vertici

Siano $v_1 = (1,1,0)$; $v_2 = (2,1,0)$ e $v_3 = (1,3,0)$ tre vertici nello spazio, allora
$a = v_2 - v_1 = (1,0,0)$ e $b = v_3 - v_1 = (0,2,0)$, da cui

$$a \times b = det \begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 1 & 0 & 0 \\ 0 & 2 & 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 2 \end{pmatrix}$$

da cui si ottiene

$$\mathbf{n} = \frac{a \times b}{|a \times b|} = (0,0,1)^T$$

Sorgenti di luce: le normali

- La funzione glNormal permette di definire il vettore normale

```
void glNormal3{bsidf}(TYPE nx, TYPE ny, TYPE nz);
void glNormal3{bsidf}v(const TYPE *v);
```

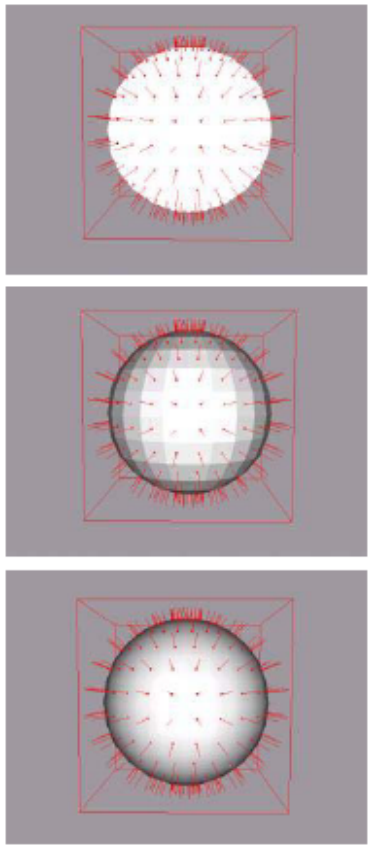
- Le successive chiamate a glVertex impostano il vertice con la normale precedentemente definita

```
glBegin(GL_POLYGON);
    glNormal3fv(n0);
    glVertex3fv(v0);
    glVertex3fv(v1);
    glVertex3fv(v2);
    glVertex3fv(v3);
glEnd();
```

```
glBegin(GL_POLYGON);
    glNormal3fv(n0); glVertex3fv(v0);
    glNormal3fv(n1); glVertex3fv(v1);
    glNormal3fv(n2); glVertex3fv(v2);
    glNormal3fv(n3); glVertex3fv(v3);
    glNormal3fv(n4); glVertex3fv(v4);
glEnd();
```

Sorgenti di luce: le normali

- OpenGL permette di specificare un vettore normale per ogni poligono o per ogni vertice
- Nell'esempio al lato osserviamo una sfera in cui lo shading è effettuato senza tener conto delle normali (in alto), tenendo conto di una normale per poligono (al centro) e tenendo conto di una normale per ogni vertice (in basso)
- È possibile calcolare le normali ai vertici mediando (componente per componente) le normali ai poligoni



Sorgenti di luce: le normali

- Dato un poligono (es. un triangolo) nello spazio, esistono due vettori perpendicolari che puntano in direzione opposta: per convenzione, il vettore normale è quello che punta verso l'esterno ed è applicato alla parte front
- Il vettore normale rimane di modulo unitario se si applicano trasformazioni di rotazione o traslazione; negli altri casi è possibile usare le funzioni

```
glEnable(GL_NORMALIZE)        // funzione generale
glEnable(GL_RESCALE_NORMAL) /* funzione specifica per le
    operazioni di scaling uniforme (uguale su x, y e z)*/
```

Aggiungere la luce alla scena

- In OpenGL, la chiamata alla funzione glEnable(GL_LIGHTING) abilita l'uso delle luci e dei materiali per il calcolo del colore dei vertici della scena
- Successivamente è necessario specificare il modello di illuminazione attraverso la funzione glLightModel oppure definire una (o più) sorgenti d'illuminazione attraverso la funzione glLight
- Il seguente esempio definisce l'intensità e il colore di un modello d'illuminazione con la sola componente ambiente

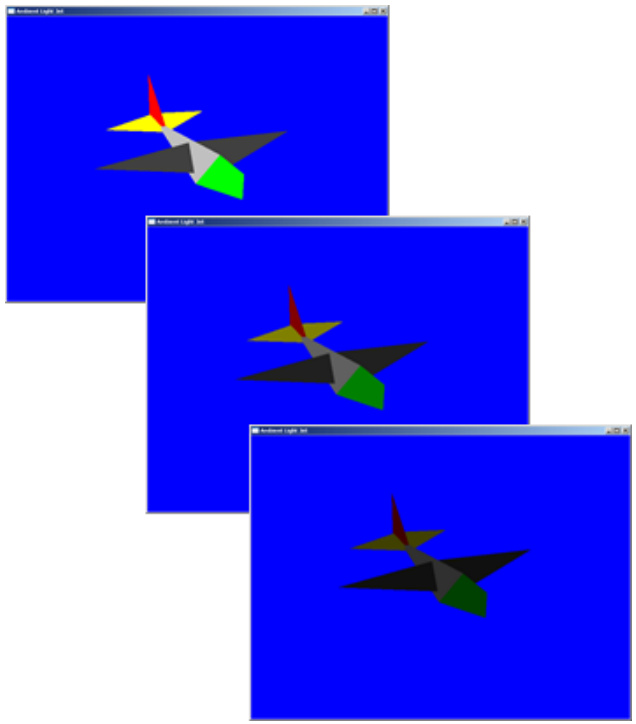
```
GLfloat ambientLight[] = {1.0, 1.0, 1.0, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientLight);
```

Un esempio di illuminazione

```
GLfloat ambientLight[] = { 1.0f
    , 1.0f, 1.0f, 1.0f };
```

```
GLfloat ambientLight[] = { 0.5f
    , 0.5f, 0.5f, 1.0f };
```

```
GLfloat ambientLight[] = { 0.25
    f, 0.25f, 0.25f, 1.0f };
```



Sorgenti di luce

- Per aggiungere realismo alla scena è necessario aggiungere una sorgente di luce
- Oltre all'intensità e al colore, le sorgenti di luce hanno le seguenti proprietà:
  - Hanno un punto di applicazione nello spazio
    - Questo può essere infinitamente lontano dalla scena (es. il sole)
    - In prossimità della scena (es. una lampada da scrivania)
  - Hanno una direzione di illuminazione che può:
    - Illuminare in una direzione particolare
    - Illuminare in ogni direzione
- OpenGL supporta almeno 8 differenti sorgenti di luce (GL_LIGHT0, . . . , GL_LIGHT7), posizionabili ovunque nella scena

Sorgenti di luce

- La funzione glLightfv setta il tipo di luce; ad esempio:

```
GLfloat ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
```

  specifica che la GL_LIGHT0 ha una componente ambient definita dal vettore ambientLight

- Allo stesso modo è possibile aggiungere una componente diffusa alla sorgente con una chiamata supplementare a glLightfv

```
GLfloat diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
```

- La sorgente deve essere attivata tramite la chiamata a glEnable(GL_LIGHT0)

Definire il tipo di materiale: glMaterial

- Una volta definiti i modelli o le sorgenti d'illuminazione, è necessario specificare le proprietà dei materiali
- Esistono due modi per definire le proprietà di riflessione dei materiali
- Il primo utilizza la funzione glMaterial, che deve essere chiamata prima della specificazione dei poligoni; ad esempio

```
GLfloat gray[ ] = {0.75, 0.75, 0.75, 1.0};
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, gray);
drawTriangle();
```

Definire il tipo di materiale: color traking

- Il secondo metodo, detto color traking, consente di utilizzare glColor al posto di glMaterial.
- Per abilitare il color traking si utilizza la chiamata alle funzioni:

```
glEnabe(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
```

Settaggio di una sorgente di luce (SetupRc)

```
// Light values and coordinates
GLfloat  ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
GLfloat  diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };

// Enable lighting
glEnable(GL_LIGHTING);

// Setup and enable light 0
glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
glEnable(GL_LIGHT0);

// Enable color tracking
glEnable(GL_COLOR_MATERIAL);
// Set Material properties to follow glColor values
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
```

Settaggio di una sorgente di luce (ChangeSize)

- È ovviamente necessario specificare la locazione della sorgente; a tal scopo, si fa riferimento ancora alla funzione glLightfv

```
GLfloat lightPos[] = { -50.f, 50.0f, 100.0f, 1.0f };
glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
```

- L'ultima componente di lightPos definisce se la luce è posta all'infinito lungo la direzione specificata dalle prime 3 componenti o è esattamente in quel punto
  - 0.0 indica una sorgente infinitamente lontana
  - 1.0 indica una sorgente vicina, nella posizione specificata dai primi tre valori dell'array.

Un altro esempio di illuminazione

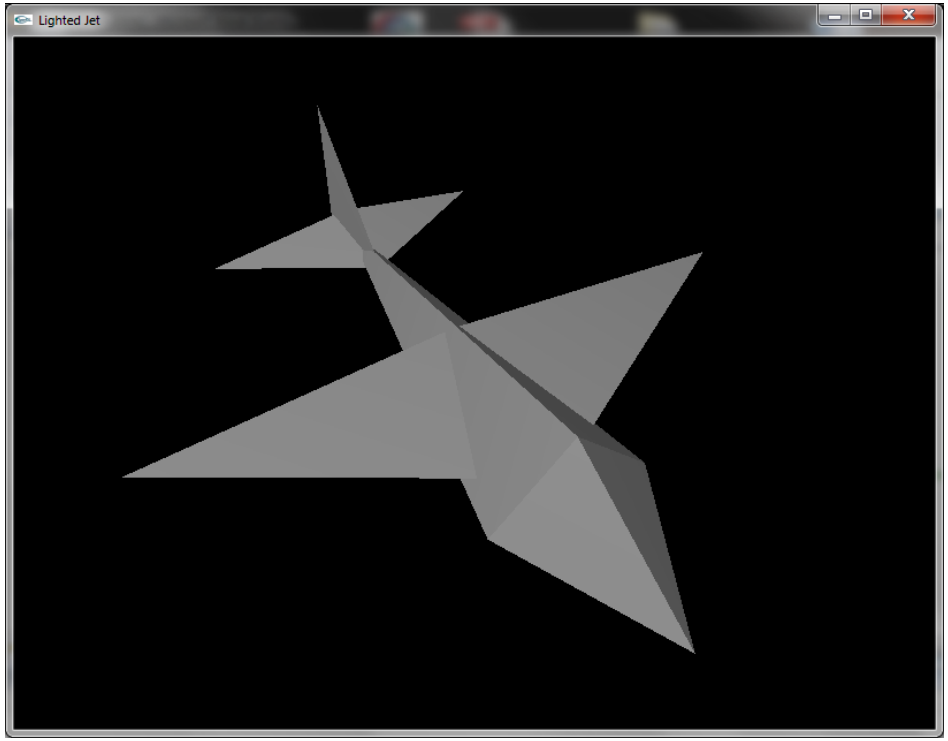


Figura: ScreenShot del programma LitJet.c (progetto *jetlight (light source)* nei materiali della lezione)

La componente speculare

- La luce ambiente e la luce diffusa sono sufficienti a descrivere oggetti opachi (che non riflettono luce)
- Questo, però, non è il caso di un jet con pareti metalliche che invece riflettono la luce
- Aggiungere una componente speculare a una sorgente di luce è molto semplice
  - Si definisce un array con l'intesità e il colore della componente speculare
  - Si abilita la componente tramite glLightfv

```
GLfloat  specularLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
glLightfv(GL_LIGHT0,GL_SPECULAR,specularLight);
```

La componente speculare

- Infine, è possibile controllare le dimensioni dello specular highlight con glMaterial (l'ultimo parametro può variare tra 1 e 128 e determina la dimensione dell'highlight - 128 indica un highlight piccolo, 1 un highlight molto grande)

```
GLfloat high_shininess[] = { 1.0 };
glMaterialfv(GL_FRONT, GL_SHININESS, high_shininess);

// oppure è possibile considerare la variante con
    glMateriali
GLint low_shininess = 128;
glMateriali(GL_FRONT, GL_SHININESS, low_shininess);
```

Spot light

- Le spot light sono luci direzionali come quelle generate da una lampada da scrivania che emettono un cono di luce
- Una spot light è semplicemente una sorgente di luce direzionale alla quale è applicato un angolo di cut off; ad esempio:

```
//...
glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
//spot direction is (0,0,-1) by default
glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,spotDir);

// Cut off angle is 50 degrees
glLightf(GL_LIGHT0,GL_SPOT_CUTOFF,50.0f);

// Enable this light in particular
glEnable(GL_LIGHT0);
```

Esempio di spot light

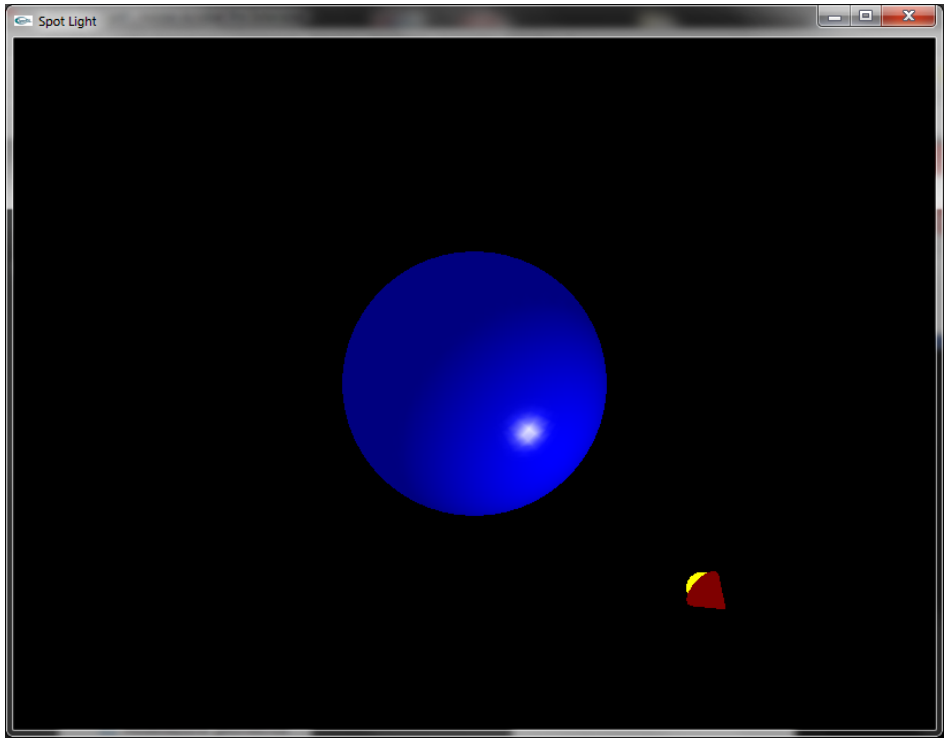


Figura: Screenshot del programma spot (nei materiali della lezione)

Emission

Specificando un colore per GL_EMISSION è possibile far apparire un oggetto emettere luce di quel colore

```
GLfloat mat_emission[] = {0.3, 0.2, 0.2, 0.0};
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
```

- Gli oggetti appaiono leggermente incandescenti per cui è utile per simulare luci accese come lampade o lampioni
- Si noti, tuttavia, che un oggetto con la proprietà di emissione non emette luce; se si desidera ottenere l'effettiva emissione di luce è necessario definire una sorgente di luce nella stessa posizione dell'oggetto dotato di emisssività

Effetti Speciali

Effetti Speciali

Effetti Speciali

- Nella lezione precedente abbiamo visto come l'applicazione delle luci alla scena può produrre un effetto molto più realistico rispetto a quello che si ottiene impostando direttamente il colore del materiale
- Tuttavia, le luci da sole non sono sufficienti a modellare altre caratteristiche del mondo reale, come ad esempio le trasparenze o la nebbia

Blending

- Abbiamo visto che OpenGL conserva le informazioni di colore nel color buffer e (se è attivato lo z-buffer) quelle di profondità nel depth buffer
- Se lo z-buffer è attivo un vertice può rimpiazzare un vertice precedentemente salvato nel color buffer in base al valore di z
- Questo non succede più se è attivata, con una chiamata a glEnable(GL_BLEND), la modalità di blending per la simulazione delle trasparenze

Blending

- Quando è attivo il blending, il nuovo colore è *mescolato* con il colore attualmente presente nel color buffer
- Il colore attualmente presente nel color buffer è chiamato destination color ed è della forma (R,G,B,A)
- Il colore entrante (che sostituisce o si combina col destination color) è chiamato source color ed è anch'esso della forma (R,G,B,A).

Blending

- Come il destination e source color siano combinati quando è attivo il blending dipende da una funzione, detta equazione di blending
- Siano
  - C_f il colore finale dopo il blending
  - C_s il source color
  - C_d il destination color
  - S il fattore di blending per il source color
  - D il fattore di blending per il destination color

  allora

$$C_f = (C_s * S) + (C_d * D)$$

Blending

- I fattori di blending S e D sono settati attraverso la funzione glBlendFunc(GLenum S, GLenum D)

Constant	Relevant Factor	Computed Blend Factor
GL_ZERO	source or destination	(0, 0, 0, 0)
GL_ONE	source or destination	(1, 1, 1, 1)
GL_DST_COLOR	source	(Rd, Gd, Bd, Ad)
GL_SRC_COLOR	destination	(Rs, Gs, Bs, As)
GL_ONE_MINUS_DST_COLOR	source	(1, 1, 1, 1)-(Rd, Gd, Bd, Ad)
GL_ONE_MINUS_SRC_COLOR	destination	(1, 1, 1, 1)-(Rs, Gs, Bs, As)
GL_SRC_ALPHA	source or destination	(As, As, As, As)
GL_ONE_MINUS_SRC_ALPHA	source or destination	(1, 1, 1, 1)-(As, As, As, As)
GL_DST_ALPHA	source or destination	(Ad, Ad, Ad, Ad)
GL_ONE_MINUS_DST_ALPHA	source or destination	(1, 1, 1, 1)-(Ad, Ad, Ad, Ad)
GL_SRC_ALPHA_SATURATE	source	(f, f, f, 1); f=min(As, 1-Ad)

Blending

- Un esempio abbastanza comune di blending è il seguente

```
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

- Supponiamo di avere il colore rosso già salvato nel color buffer, C_d=(1,0,0,0), e un colore sorgente C_s=(0,0,1,0.5); essendo S=0.5 e D=1-0.5=0.5, si ha

$$C_f = (C_s * S) + (C_d * D) =$$

$$= (0, 0, 0.5, 0) + (0.5, 0, 0, 0.5) = (0.5, 0, 0.5, 0)$$

- Esempio: alpha.c nei materiali della lezione

Equazioni di Blending

- L'equazione $C_f = (Cs * S) + (Cd * D)$ è quella di default, ma OpenGL ne fornisce delle altre
- È possibile cambiare funzione attraverso glBlendEquation(GLenum mode) dove mode può essere
  - GL_FUNC_ADD ($C_f = (C_s * S) + (C_d * D)$)
  - GL_FUNC_SUBTRACT ($C_f = (C_s * S) - (C_d * D)$)
  - GL_FUNC_REVERSE_SUBTRACT ($C_f = (C_d * S) - (C_s * D)$)
  - GL_MIN ($min(Cs, Cd)$)
  - GL_MAX ($max(Cs, Cd)$)

Blending

- Il blending può essere utilizzato anche per l'effetto di riflessione di un oggetto su una superficie

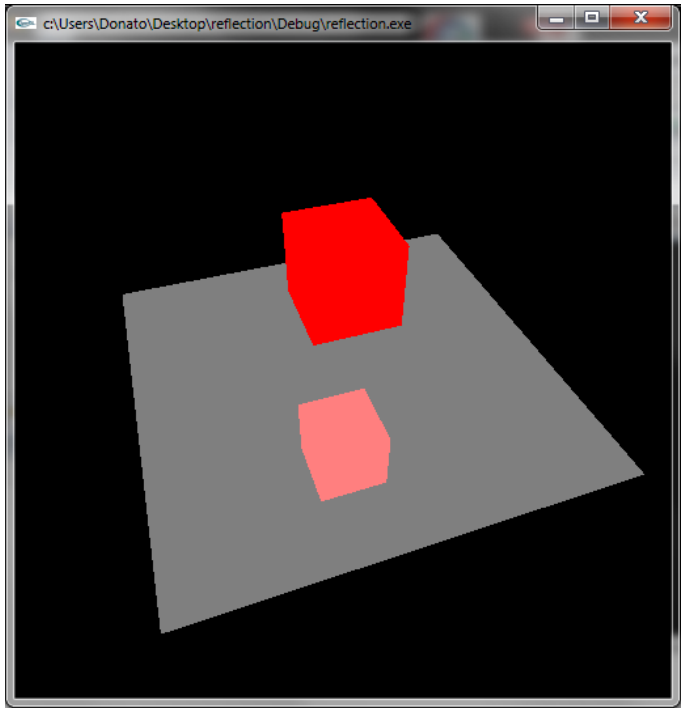


Figura: ScreenShot del programma reflection.c (nei materiali della lezione)

Fog

- Un altro effetto speciale supportato da OpenGL è la nebbia
- L'effetto nebbia è sempre basato sul blending: OpenGL mescola il colore assegnato alla nebbia con la geometria della scena come ultima operazione di rendering
- La quantità di colore di nebbia mescolato con la geometria varia in funzione della distanza tra la camera e la geometria stessa
- Questo semplice trucco da l'impressione della presenza della nebbia nella scena che, tra l'altro, può rendere maggiormente percepibile la profondità

Fog

- Attivare e disattivare l'effetto nebbia è semplice e si fa con le solite glEnable(GL_FOG) e glDisable(GL_FOG)
- Ovviamente l'effetto nebbia può essere parametrizzato attraverso le funzioni
  - void glFogi(GLenum pname, GLint param);
  - void glFogf(GLenum pname, GLfloat param);
  - void glFogiv(GLenum pname, GLint* param);
  - void glFogfv(GLenum pname, GLfloat* param);

  Ad esempio

```
GLfloat fog_color[ ] = {0.1, 0.1, 0.1, 1.0};
glFogfv(GL_FOG_COLOR, fog_color);
```

- Nota che se il fog_color differisce dal colore di sfondo l'effetto nebbia risulta poco naturale

Fog

- Le seguenti funzioni permettono di specificare il range di distanza rispetto alla posizione della camera per l'applicazione dell'effetto

```
glFogf(GL_FOG_START, 5.0);
glFogf(GL_FOG_END, 35.0);
```

- La transizione tra START e END è controllata da un'equazione di fog; ad esempio

```
glFogi(GL_FOG_MODE, GL_LINEAR);
```

Fog

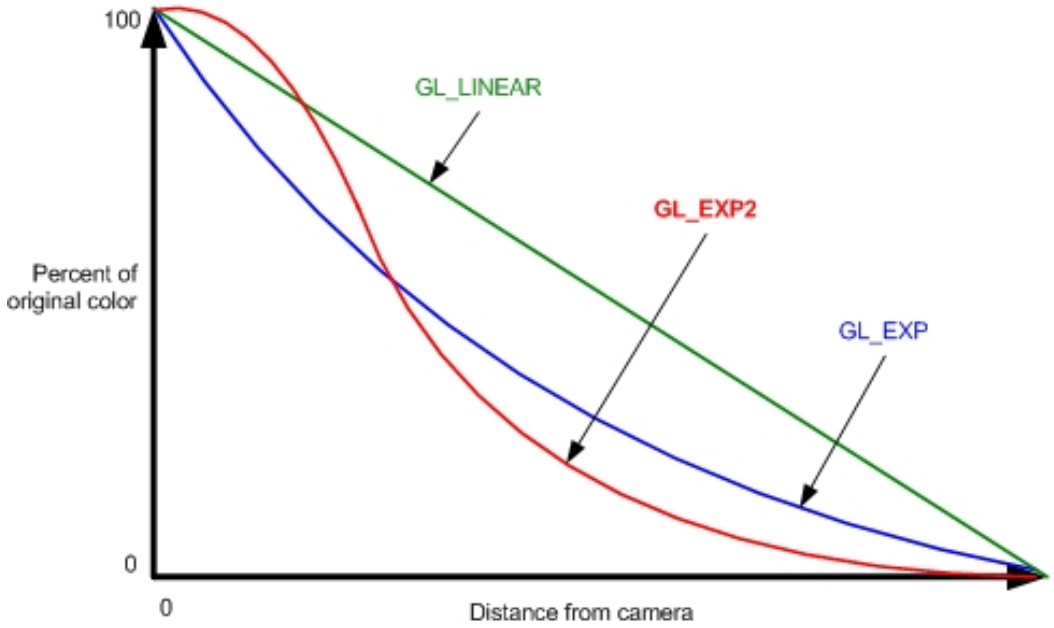
- L'equazione di fog calcola un valore $f \in [0, 1]$, detto fattore di fog, in relazione al range [Sart, End]. In altri termini

$$f : [Sart, End] \rightarrow [0, 1]$$

- I fog mode supportati sono i seguenti
  - GL_LINEAR $(f = (end - c)/(end - start))$
  - GL_EXP $(f = exp(-d * c))$
  - GL_EXP2 $\left(f = exp((-d * c))^2\right)$

  dove
  - c è la distanza del punto dalla camera
  - d è il fattore di fog che si imposta con la funzione glFogf(GL_FOG_DENSITY, 0.5);

Fog


100
0
0
Percent of original color
Distance from camera
GL_LINEAR
GL_EXP2
GL_EXP

Fog

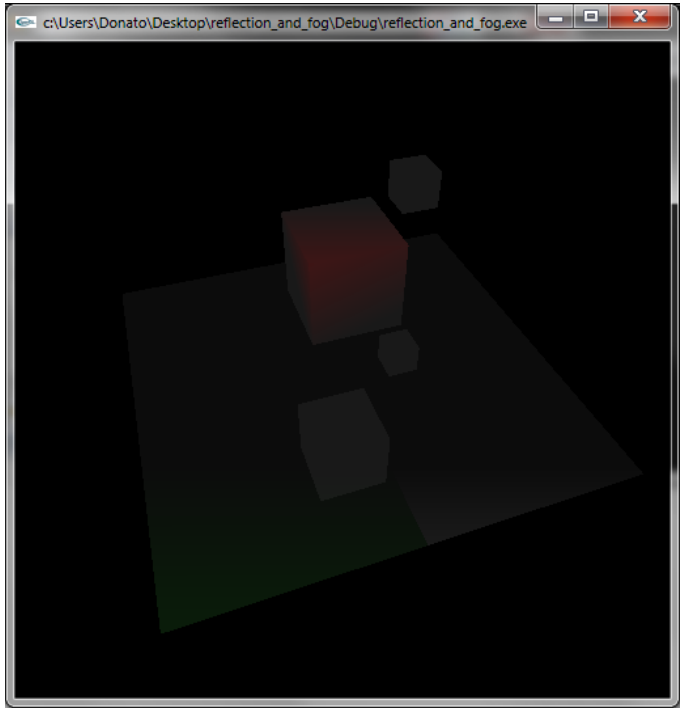


Figura: ScreenShot del programma reflection_and_fog.c (nei materiali della lezione)

Display Lists

Display Lists

Display Lists

- Una *display list*, letteralmente *elenco di visualizzazione*, è un modo per definire e memorizzare una sequenza di comandi OpenGL che possono essere richiamati in qualsiasi momento
- É possibile mischiare liberamente l'uso delle display list con la specificazione di comandi in *immediate mode* all'interno di un programma (gli esempi di programmazione che abbiamo visto fino a ora hanno usato la modalità immediata)
- Le display list possono migliorare le prestazioni dell'applicazione poiché memorizzano solo i comandi OpenGL, ad esempio le chiamate a glColor* e a glVertex*, in modo che quando sono invocate non vengono eseguite le istruzioni che sono state necessarie per determinarne i parametri

Esempio: modellare un triciclo con le display list

- Si consideri il problema di modellare un triciclo in cui le due ruote posteriori hanno le stesse dimensioni ma in posizioni differenti mentre la ruota anteriore è più grande ed è anche in una posizione diversa
- Un modo efficace per modellare le ruote del triciclo sarebbe quello di conservare la geometria di una ruota in una diaplay list ed eseguire la lista tre volte
- Ovviamente si dovrà impostare la matrice di modelview correttamente ogni volta prima di eseguire la lista per calcolare dimensione e posizione delle ruote

Esempio: modellare un *toro* con le display list

- Si supponga di voler disegnare un toro e osservarlo da diverse angolazioni: il modo più efficiente è memorizzare il toro in una display list per poi eseguirla (dopo aver cambiato la matrice di modelview) ogni volta che si desidera modificare la vista

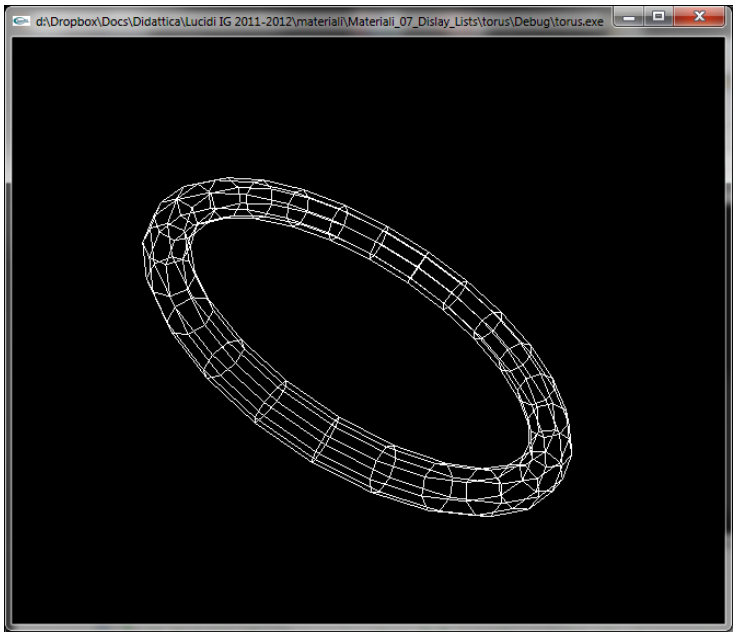


Figura: ScreenShot del programma torus.c

Esempio: sorgente del programma torus.c

```
//...
GLuint theTorus;

/* Draw a torus */
static void torus(int numc, int numt)
{
  int i, j, k;
  double s, t, x, y, z, twopi;
  twopi = 2 * (double)M_PI;
  for (i = 0; i < numc; i++) {
    glBegin(GL_QUAD_STRIP);
      for (j = 0; j <= numt; j++) {
        for (k = 1; k >= 0; k--) {
          s = (i + k) % numc + 0.5;
          t = j % numt;

          //...
```

Esempio: sorgente del programma torus.c

```
            //...

            x = (1+.1*cos(s*twopi/numc))*cos(t*twopi/numt);
            y = (1+.1*cos(s*twopi/numc))*sin(t*twopi/numt);
            z = .1 * sin(s * twopi / numc);
            glVertex3f(x, y, z);
          }
        }
      glEnd();
    }
}
```

Esempio: sorgente del programma torus.c

```
/* Create display list with Torus and initialize state */
static void init(void)
{
    theTorus = glGenLists (1);
    glNewList(theTorus, GL_COMPILE);
        torus(8, 25);
    glEndList();
    //...
}
/* Clear window and draw torus */
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glCallList(theTorus);
    glFlush();
}
```

Esempio: sorgente del programma torus.c

- La funzione init() crea una lista; si noti che la funzione che modella il toro è racchiuso tra glNewList() e glEndList()
- L'argomento *theTorus* per glNewList() è un indice intero generato da glGenLists() che identifica in modo univoco la display list
- La funzione display() semplicemente cancella lo schermo e chiama glCallList() per eseguire i comandi nell'elenco di visualizzazione
- Se avessimo operato in immediate mode display() avrebbe dovuto rieseguire la funzione torus (8 , 25) per ridefinire i comandi OpenGL necessari per rappresentare il toro

Considerazioni e vincoli nell'uso delle display list

- Una display list contiene solo comandi OpenGL/GLU, glCallList() compreso (display list innestate)
- I parametri dei comandi OpenGL vengono copiati nella display list insieme ai comandi stessi quando la display list viene creata
  - Tutti i calcoli trigonometrici necessari per creare il toro sono eseguiti solo la prima volta per definire i parametri dei comandi OpenGL
- I parametri, una volata inseriti in una display list, non possono essere cambiati; inoltre, non è possibile aggiungere o rimuovere comandi da una display list
- In altri termini, le display list non possono essere modificate; è possibile comunque eliminare l'intera display list e crearne una nuova

Display Lists: comandi principali

- GLuint glGenLists(GLsizei range);
  - Allocates *range* number of contiguous, previously unallocated display-list indices. The integer returned is the index that marks the beginning of a contiguous block of empty display-list indices. The returned indices are all marked as empty and used, so subsequent calls to **glGenLists()** don't return these indices until they're deleted. Zero is returned if the requested number of indices isn't available, or if *range* is zero.
- void glCallList(GLuint list);
  - This routine executes the display list specified by *list*. The commands in the display list are executed in the order they were saved, just as if they were issued without using a display list. If *list* hasn't been defined, nothing happens.

Display Lists: comandi principali

- void glNewList (GLuint list, GLenum mode);
  - Specifies the start of a display list. OpenGL routines that are called subsequently (until **glEndList()** is called to end the display list) are stored in a display list, except for a few restricted OpenGL routines that can't be stored. (Those restricted routines are executed immediately, during the creation of the display list.) *list* is a nonzero positive integer that uniquely identifies the display list. The possible values for *mode* are GL_COMPILE and GL_COMPILE_AND_EXECUTE. Use GL_COMPILE if you don't want the OpenGL commands executed as they're placed in the display list; to cause the commands to be executed immediately as well as placed in the display list for later use, specify GL_COMPILE_AND_EXECUTE.
- void glEndList (void);
  - Marks the end of a display list.

Display Lists: comandi principali

- GLboolean glIsList (GLuint list);
  - Returns GL_TRUE if list is already used for a display list and GL_FALSE otherwise.
- void glDeleteLists (GLuint list, GLsizei range);
  - Deletes range display lists, starting at the index specified by list. An attempt to delete a list that has never been created is ignored.

Eliminazione di una display list

Una display list può essere eliminata esplicitamente con la funzione glDeleteLists() oppure implicitamente ridefinendone un'altra con con lo stesso indice

Display list multiple

OpenGL fornisce un meccanismo efficiente per eseguire diplay list in successione: si mettano gli indici in un array e si chiama glCallLists()

Rappresentazione di una superficie topografica

Ogni riga della matrice dei punti quotati, rappresentata attraverso una triangle strip, può essere compilata in una display list indipendente; le display list così create possono essere eseguite in un solo colpo attraverso la funzione glCallLists()

Rappresentazione di un font

È possibile far in modo che ogni indice corrisponda al valore ASCII di un carattere utilizzando glListBase() per specificare l'indice iniziale (quello corrispondente alla prima lettera, ad es. la A) prima di chiamare glCallLists()

Display Lists: comandi principali

- void glListBase (GLunint base);
  - Specifies the offset that's added to the display-list indices in glCallLists() to obtain the final display list indices. The default display list base is 0.
- void glCallLists (GLsizei n, GLenum type, const GLvoid *lists);
  - Executes *n* display lists. The indices of the lists to be executed are computed by adding the offset indicated by the current display list base (specified with **glListBase()**) to the signed integer values in the array pointed to by *lists*.
    The *type* parameter indicates the data type of the *values* in lists. It can be set to GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, or GL_FLOAT.

Esempio: sorgente del programma stroke.c

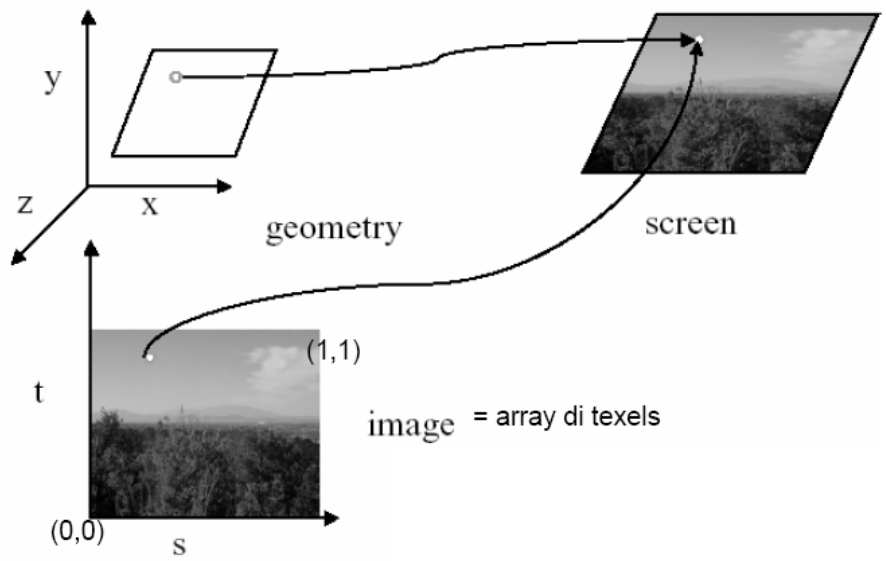
```
void initStrokedFont(void){
    GLuint base = glGenLists(128);
    glListBase(base);
    glNewList(base+'A', GL_COMPILE); drawLetter(Adata);
        glEndList();
    // ...
    glNewList(base+'Z', GL_COMPILE); drawLetter(Zdata);
        glEndList();
    glNewList(base+' ', GL_COMPILE); /* space character */
        glTranslatef(8.0, 0.0, 0.0); glEndList();
}

void printStrokedString(GLbyte *s){
    GLint len = strlen(s);
    glCallLists(len, GL_BYTE, s);
}
```

Textures

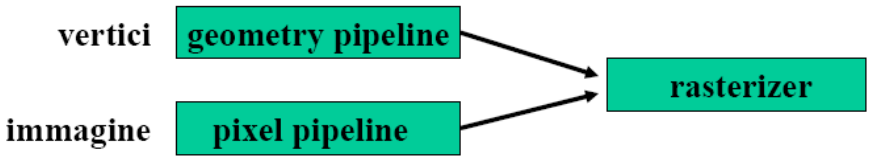
Textures

Texture Mapping

OpenGL consente di mappare un'immagine, detta texture, su primitive geometriche



Immagini e oggetti geometrici seguono due pipeline separate che si uniscono poi nella fase di rasterizzazione



Applicare le texture

Applicare una texture consiste in:

- Specificare la texture
  - Leggere o generare l'immagine[1]
  - Assegnare l'immagine alla *texture memory*
- Specificare i parametri della texture
  - Filtering, wrapping
- Assegnare le coordinate di texture ai vertici

[1] OpenGL non è in grado di leggere i formati immagine tipo PNG, JPEG, GIF o TGA; Il file immagine deve essere letto e decodificato per ottenere l'informazione colore

Caricamento della texture

- Gli elementi di una texture sono detti texel (e non pixel)
- La modalità di texture si abilita/disabilita con
  glEnable(GL_TEXTURE_2D)
  glDisable(GL_TEXTURE_2D)
  (nota che OpenGL supporta anche texture 1D e 3D)
- Il primo passo è leggere il file che contiene la texture
- Successivamente si deve caricare in memoria la texture, attraverso la funzione
  glTexImage2D
  (glTexImage1D e glTexImage3D per texture 1D e 3D)

Caricamento della texture (glTexImage2D)

```
void glTexImage2D(
  GLenum target,
  GLint level,
  GLint internalformat,
  GLsizei width,
  GLsizei height,
  GLint border,
  GLenum format,
  GLenum type,
  const GLvoid *pixels
);
```

target deve essere GL_TEXTURE_2D (o GL_TEXTURE_1D o GL_TEXTURE_3D se si usano texture a 1 o 3 dimesioni)

Caricamento della texture (glTexImage2D)

- level specifica il livello di dettaglio (LOD) della texture; per ora usiamo il valore 0 (no mippapping)
- internalformat è il numero di componenti di colore (1-4) dei texel; per ora usiamo GL_RGB, cioè 3
- width e height sono le dimensioni della texture in pixel; devono essere della forma 2^m (o $2^m + 2$ in caso border valga 1); se le dimensioni non sono potenza di due usare la funzione void gluScaleImage
- border specifica lo spessore del bordo della texture e può essere 0 (nessun bordo) o 1; per ora usiamo 0
- format specifica il formato dei pixel (es. GL_BGR_EXT, GL_BGRA_EXT, GL_LUMINANCE)
- type specifica il tipo di dato del pixel; per ora usiamo GL_UNSIGNED_BYTE
- pixels è il puntatore all'immagine in memoria

Caricamento della texture (glTexImage2D)

```
#include "tga.h" // per caricare la texture in formato tga
//...
GLubyte *pBytes;
GLint iWidth, iHeight, iComponents;
GLenum eFormat;
//...
// Load texture
pBytes = gltLoadTGA("Stone.tga", &iWidth, &iHeight, &
    iComponents, &eFormat);
glTexImage2D(GL_TEXTURE_2D, 0, iComponents, iWidth, iHeight, 0,
     eFormat, GL_UNSIGNED_BYTE, pBytes);
free(pBytes);
//...
glEnable(GL_TEXTURE_2D)
```

Mappatura della texture

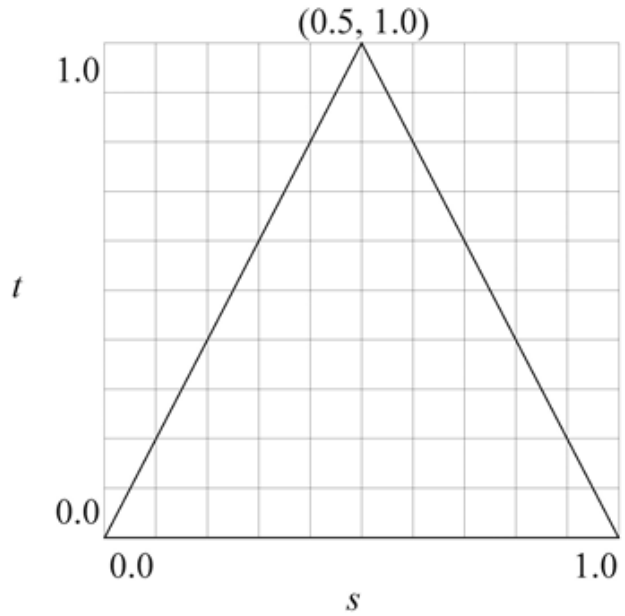
- Una volta caricata la texture in memoria è necessario specificare come questa deve essere applicata alla geometria
- Per fare questo si specificano le texture coordinates ai vertici
- Tipicamente le texture coordinates sono specificate come valori floating point nell'intervallo [0, 1]
- Similmente alle coordinate dei vertici, che sono della forma (x, y, z, w), le coordinate di texture sono della forma (s, t, r, q). Nota che come nel caso di w, anche q non è generalmente utilizzata

Mappatura della texture

- Le coordinate di texture sono applicate ai vertici attraverso la funzione
  void glTexCoord2f (GLfloat s, GLfloat t);
- glTexCoord2f deve essere invocata prima della specificazione del vertice
- La specifica delle coordinate di texture può produrre una distorsione dell'immagine originaria

Mappatura della texture

```
glColor3f(1.0, 1.0, 1.0);
//...
//draw a textured triangle
glNormal3fv(vNormal);
glTexCoord2f(0.5f, 1.0f);
glVertex3fv(vCorners[0]);
glTexCoord2f(0.0f, 0.0f);
glVertex3fv(vCorners[1]);
glTexCoord2f(1.0f, 0.0f);
glVertex3fv(vCorners[2]);
//...
```



Texture environment

Come OpenGL combina il colore della primitiva e quello della texture è controllato dal texture environment mode attraverso le funzione glTexEnvf, glTexEnvi, glTexEnvfv, glTexEnviv

```
void glTexEnvi(
  GLenum target,      // GL_TEXTURE_ENV
  GLenum pname,       // GL_TEXTURE_ENV_MODE
  GLint param         // GL_MODULATE, GL_DECAL,
                      // GL_BLEND o GL_REPLACE
);
```

Texture environment

- GL_MODULATE moltiplica il colore di texel con quello della geometria dopo aver applicato le luci
- GL_ADD somma i colori (e tronca a 1.0)
- GL_REPLACE rimpiazza il colore della geometria con quello del texel
- GL_DECAL si comporta come GL_REPLACE se la texture non ha la componente alpha, altrimenti miscela colore di texture e geometria
- GL_BLEND la texture è blendata con un colore di blendig costante che è necessario specificare ad esempio così

```
GLfloat fColor[4] = { 1.0f, 0.0f, 0.0f, 0.0f };
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_BLEND);
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, fColor);
```

Texture environment

```
#include "tga.h"
//...
GLubyte *pBytes;
GLint iWidth, iHeight, iComponents;
GLenum eFormat;
//...

// Load texture
pBytes = gltLoadTGA("Stone.tga", &iWidth, &iHeight, &
    iComponents, &eFormat);
glTexImage2D(GL_TEXTURE_2D, 0, iComponents, iWidth, iHeight, 0,
     eFormat, GL_UNSIGNED_BYTE, pBytes);
free(pBytes);
//...

glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
glEnable(GL_TEXTURE_2D);
```

Texture environment

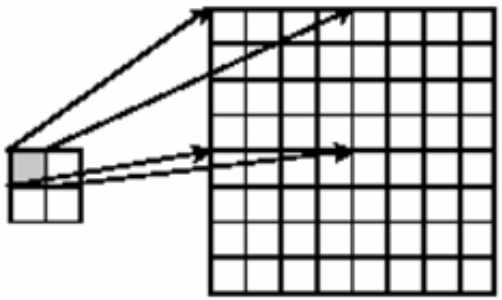
Alcuni parametri influenzano il comportamento del mapping e sono settati dalle funzioni glTexParameterf, glTexParameteri, glTexParameterfv, glTexParameteriv:

```
void glTexParameteri(
  GLenum target,      // GL_TEXTURE_2D
  GLenum type,        // GL_TEXTURE_MIN_FILTER, ...
  GLint mode          // GL_LINEAR, GL_NEAREST, ...
);
```

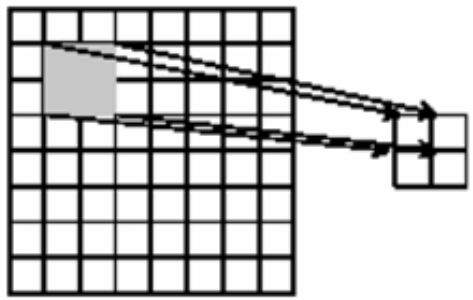
Texture parameters

glTexParameteri(GL_TEXTURE_2D, type, mode);

- Type: GL_TEXTURE_MIN_FILTER o GL_TEXTURE_MAG_FILTER.
- Mode: GL_NEAREST, GL_LINEAR, o modalità speciali per mipmapping



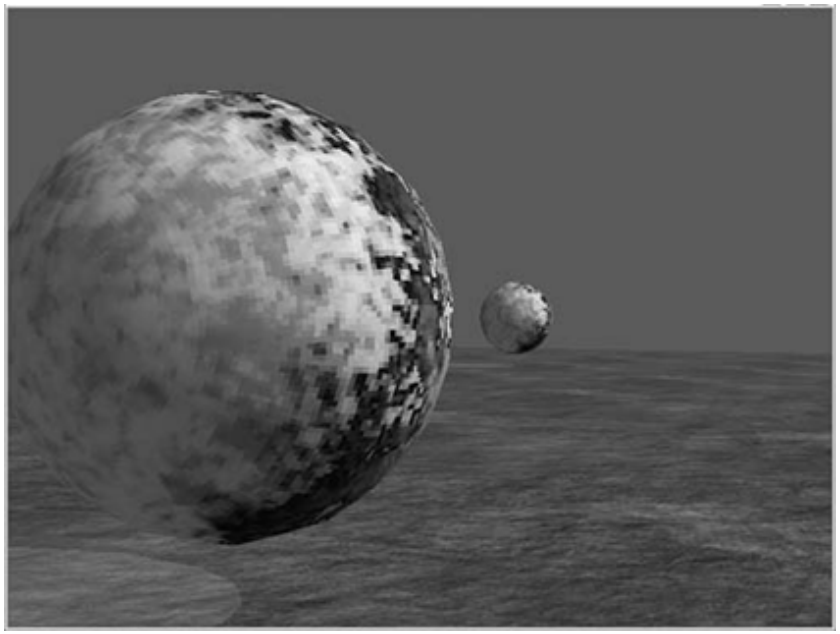
Magnification:
il texel è più grande di un pixel
Soluzione: interpolazione



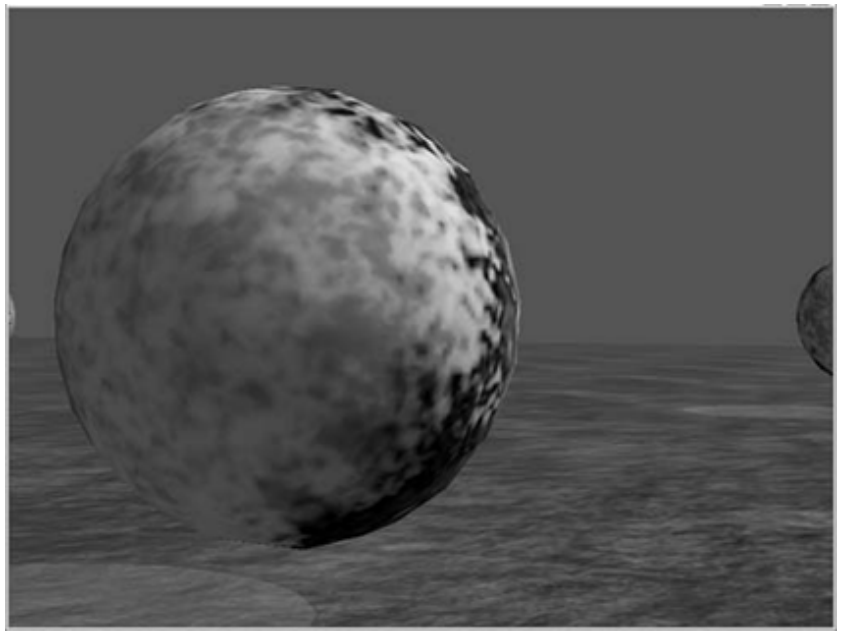
Minification:
il texel è più piccolo di un pixel
Soluzione: media

Filtering

Nearest neighbor



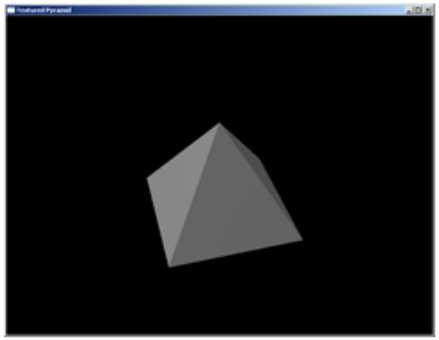
Linear filtering

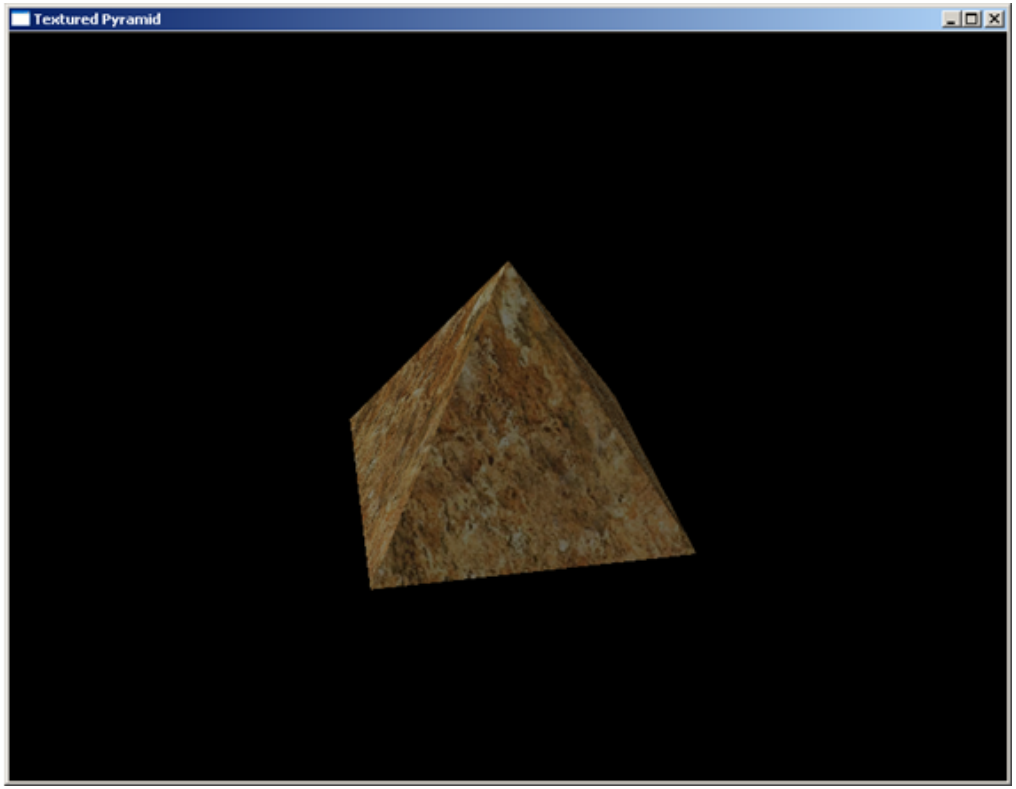


Filtering

```
#include "tga.h"
//...
GLubyte *pBytes;
GLint iWidth, iHeight, iComponents;
GLenum eFormat;

// Load texture
pBytes = gltLoadTGA("Stone.tga", &iWidth, &iHeight, &
    iComponents, &eFormat);
glTexImage2D(GL_TEXTURE_2D, 0, iComponents, iWidth, iHeight, 0,
     eFormat, GL_UNSIGNED_BYTE, pBytes);
free(pBytes);
//...
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
glEnable(GL_TEXTURE_2D);
```

Esempio (Pyramid.c)





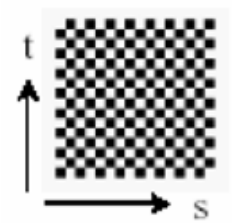
Wrap Mode

- Può succedere che la texture sia troppo piccola per ricoprire l'oggetto desiderato e si voglia ripetere più volte la texture sull'oggetto
- Questo si può fare sempre con la funzione glTexParameteri; ad es.

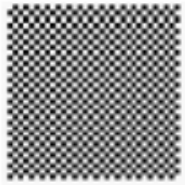
```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
```

  e specificando valori al difuori dell'intervallo [0,1] per le coordinate di texture

Esempio (PyramidWrap.c)



texture

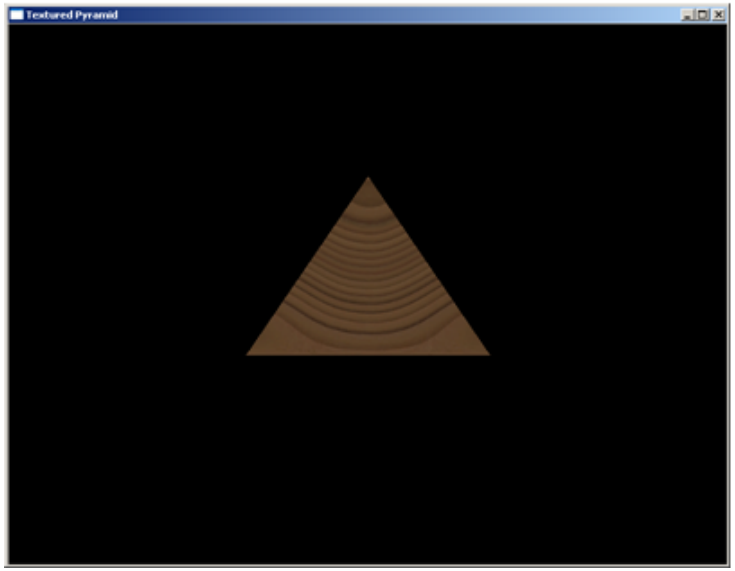


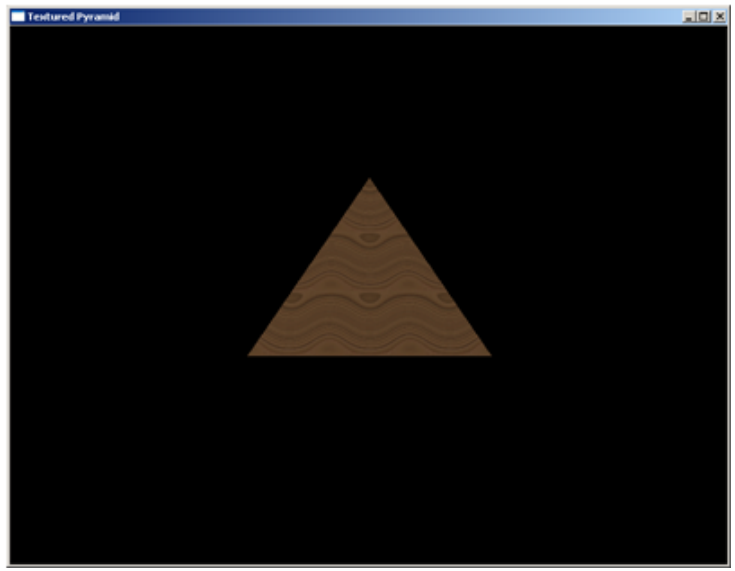
GL_REPEAT

> [0,1]

GL_CLAMP

Valori >1.0 set 1.0
Valori < 0.0 set 0.0





Texture Objects

- I texture objects permettono di caricare in un colpo solo più texture e settare i parametri di texture per ognuna
- L'uso dei texture objects semplifica l'applicazione e produce un beneficio in termini di performance
- La seguente funzione alloca *n* texture objects:

```
void glGenTexture (GLsizei n, GLuint *textures);
```

dove textures è un puntatore a un array che contiene gli identificatori delle texture

Texture Objects

- La funzione glBindTexture effettua il texture binding

```
void glBindTexture(GLenum target, GLuint texture);
```

dove target è ad es. GL_TEXTURE_2D, mentre texture è un oggetto texture

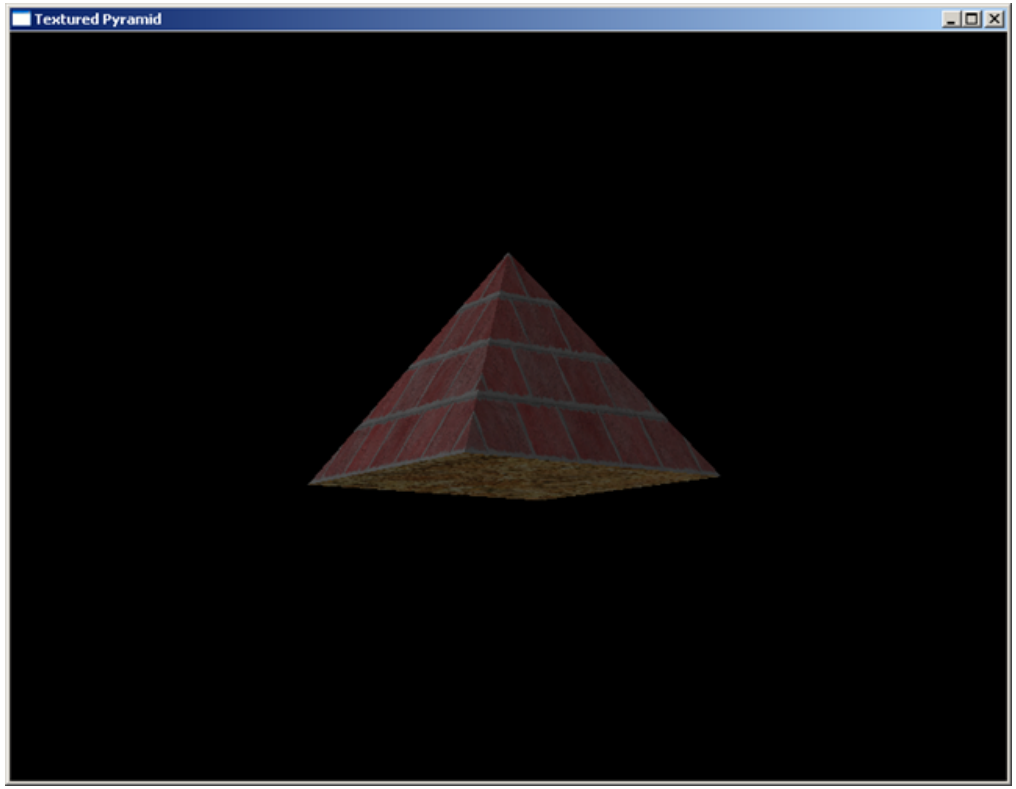
- Effettuato il binding, tutte le chiamate a glParameteri saranno riferite (collegare) all'oggetto texture corrente
- La funzione glDeleteTextures elimina oggetti texture

```
void glDeleteTextures(GLsizei n, GLuint *textures);
```

Infine, la funzione glIsTexture restituisce GL_TRUE se texture è la texture corrente, GL_FALSE altrimenti

```
GLboolean glIsTexture(GLuint texture);
```

Esempi (Pyramid3)



Secondary specular color

- In alcune implementazioni di OpenGL è possibile applicare l'effetto di luce riflessa dopo l'applicazione della texture per un effetto di maggiore luminosità
- La funzione glLightModeli applica la luce riflessa dopo la texture mapping

```
glLightModeli(GL_LIGHT_MODEL_CONTROL,
      GL_SEPARATE_SPECULAR_COLOR);
```

- Per tornare alla modalità di default si usa

```
glLightModeli(GL_LIGHT_MODEL_CONTROL,  GL_COLOR_SINGLE);
```

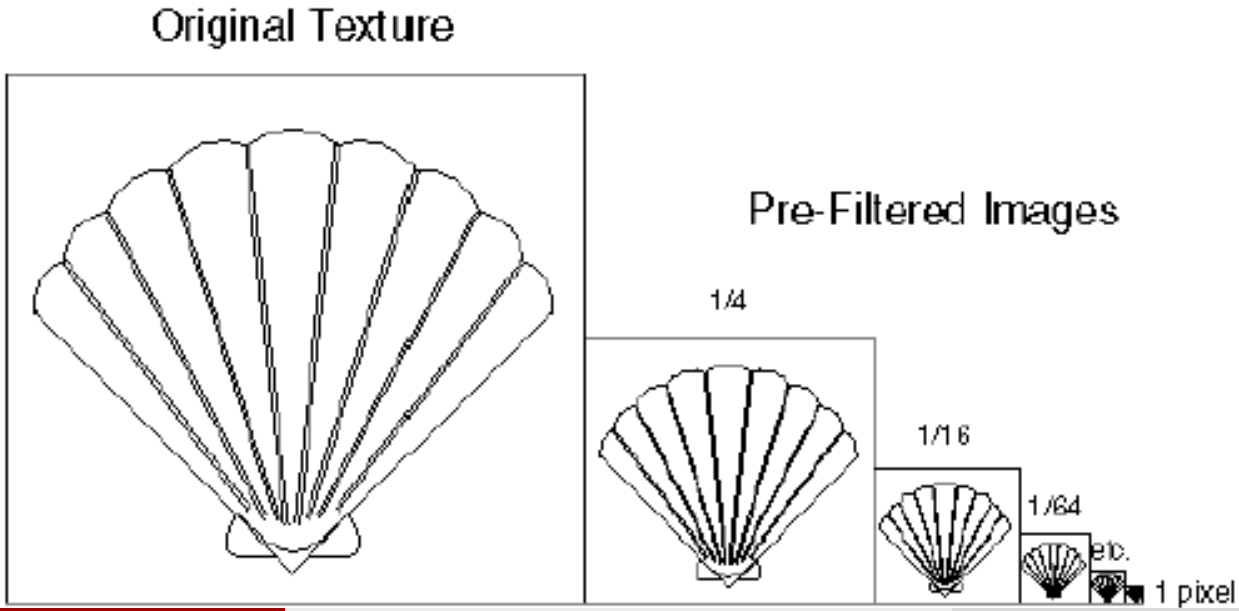
Mipmapping

- Mip sta per multum in parvo, cioè molto con poco
- È una tecnica che permette di risolvere due problemi che si presentano solitamente con le texture:
  - Incrementa le performance
  - Limina l'effetto di scintillazione che si verifica quando la geometria su cui è mappata la texture è troppo piccola rispetto alla dimensione della texture stessa
- L'unico svantaggio è un lieve incremento di memoria per lo storage della texture

Mipmapping

Una mipmaps è in realtà costituita da più immagini che ne definiscono il livello

- Al livello 0 troviamo la texture originaria
- Al livello 1 la stessa textue ma con dimensioni dimezzate sia lungo s che t
- Al livello 2 troviamo la texture di livello 1 dimezzata, e così via
- È OpenGL a decidere quale livello applicare



Mipmapping

- Innanzitutto si specifica il livello di dettaglio in glTexImage2D
- Il mipmapping aggiunge due ulteriori filtri di rendering ai precedenti due (GL_NEAREST, GL_LINEAR) da attivare in glTexParameteri, e cioè:
  GL_NEAREST_MIPMAP_NEAREST
  GL_NEAREST_MIPMAP_LINEAR
  GL_LINEAR_MIPMAP_NEAREST
  GL_LINEAR_MIPMAP_LINEAR

Mipmapping

- La funzione gluScaleImage può essere utilizzata per generare i LOD a partire da una texture standard
- La generazione dei LOD può essere eseguita anche automaticamente da OpenGL tramite la seguente funzione

```
int gluBuild2DMipmaps(
  GLenum target, // GL_TEXTURE_2D
  GLint components, // The number of color components in
      the texture
  GLint width,
  GLint height,
  GLenum format,  // The format of the pixel data
  GLenum type, // The data type for data (es. GL_BYTE)
  const void *data // A pointer to the image data in memory
      .
);
```

Selezione

Selezione

Grafica Interattiva

- La Selezione è una potente funzionalità di OpenGL che permette di fare click in una determinata posizione sulla viewport e determinare quali oggetti sono sotto il puntatore del mouse
- Questo è possibile poiché viene creato un volume di vista locale e circoscritto attraverso la funzione gluPickMatrix e poi vengono individuati quali oggetti rientrano al suo interno
- L'atto di selezionare un oggetto sullo schermo è detto picking
- Il feedback è un'altra modalità di OpenGL (che non trattiamo) che permette di ottenere informazioni da OpenGL su come il vertici sono trasformati e illuminati; è possibile utilizzare queste informazioni per trasmettere i risultati di rendering su una rete, inviarli ad un plotter o aggiungere altri elementi grafici

Selezione

- In modalità di selezione le primitive che ricadono all'interno del volume di vista, che normalmente appaiono nel frame buffer, producono invece dei record di selezione (hit records) nel selection buffer
- È necessario impostare questo buffer in anticipo e assegnare nomi a primitive o gruppi di primitive (oggetti o modelli) in modo che possano essere identificati nel buffer di selezione; in tal modo è possibile analizzare il buffer di selezione per determinare quali oggetti intersecato il volume di vista
- Tipicamente si modifica il volume di visualizzazione creandone uno locale relativamente alla posizione del mouse sulla viewport prima di entrare in modalità di selezione e si riesegue il codice di disegno di modo che i nomi degli oggetti che ricadono in quel volume siano inviati al selection buffer

Selezione: attribuire i nomi

- I nomi degli oggetti non sono altro che numeri interi senza segno, motivo per cui il selection buffer, a differenza di altri buffer di OpenGL, è semplicemente un array di valori interi
- In modalità selezione è possibile lavorare con selezioni singole o multiple; in entrambi i casi viene utilizzato un name stack
- Dopo aver inizializzato il name stack, è possibile sostituire il nome attualmente in cima alla pila (selezione singola) oppure effettuare operazioni di push in modo da avere più nomi nello stack (selezione multipla)
- Quando si verifica un hit, tutti i nomi attualmente nel name stack vengono aggiunti alla fine del selection buffer, così, se necessario, un singolo hit può restituire più di un nome

Selezione: attribuire i nomi (vedi sorgente Planets.c)

```
// Define object names
#define SUN     1
#define MERCURY 2
#define VENUS   3
//...

// Initialize the names stack; glPushName pushes 0 on
// the stack to put at least one entry on the stack
glInitNames();
glPushName(0);

// Name and draw the Sun; glLoadName names the object and
// replaces the current name on top of the name stack
glColor3f(1.0f, 1.0f, 0.0f);
glLoadName(SUN);
DrawSphere(15.0f);
```

Selezione: attribuire i nomi (vedi sorgente Planets.c)

```
// Name and draw Mercury
glColor3f(0.5f, 0.0f, 0.0f);
glPushMatrix();
    glTranslatef(24.0f, 0.0f, 0.0f);
    glLoadName(MERCURY);
    DrawSphere(2.0f);
glPopMatrix();

// Name and draw Venus
glColor3f(0.5f, 0.5f, 1.0f);
glPushMatrix();
    glTranslatef(60.0f, 0.0f, 0.0f);
    glLoadName(VENUS);
    DrawSphere(4.0f);
glPopMatrix();

//...
```

Lavorare in Selection mode

- Come accennato in precedenza, OpenGL può funzionare in tre diverse modalità di rendering: GL_RENDER, GL_SELECTION e GL_FEEDBACK
- La modalità predefinita è GL_RENDER; per utilizzare la selezione, dobbiamo cambiare la modalità di rendering:

```
glRenderMode (GL_SELECTION);
```

- Generalmente, si utilizza la stessa funzione di rendering sia in modalità GL_RENDER che in modalità GL_SELECTION

Lavorare in Selection mode: il selection buffer

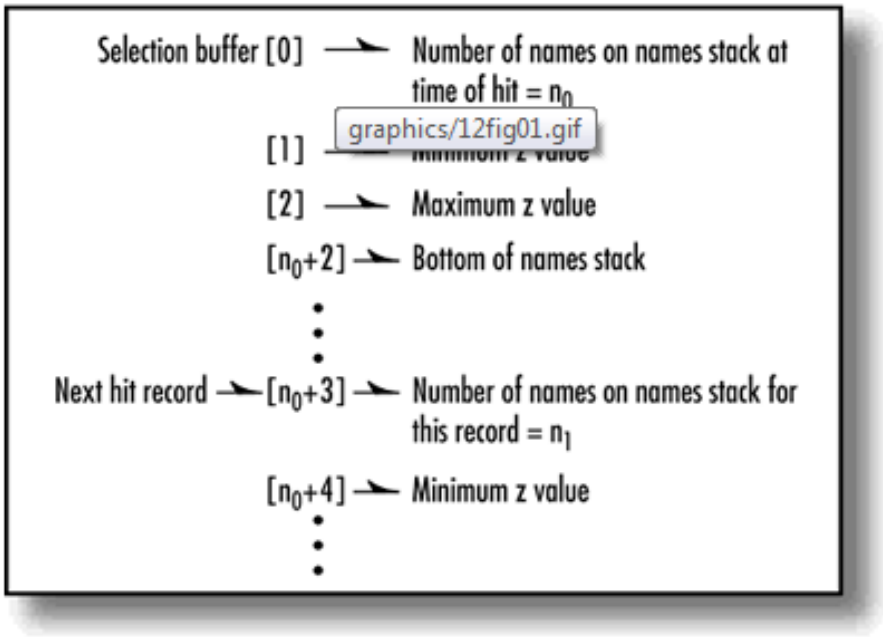
- Il codice seguente verifica se è stato effettuato in clic del tasto sinistro del mouse e passa le coordinate del mouse a ProcessSelection, che si occupa della gestione del click

```
// Process the mouse click
void MouseCallback(int button, int state, int x, int y)
{
    if(button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        ProcessSelection(x, y);
}
```

Lavorare in Selection mode: il selection buffer

Ogni hit record nel selection buffer contiene:

- Il numero di nomi attualmente nel name stack (1 nell'esempio)
- Max e min z dell'oggetto, utile in caso di selezione multipla
- Il primo nome contenuto nel names stack (bottom element)
- Se più nomi compaiono nel names stack, essi seguono il quarto elemento



Lavorare in Selection mode: il selection buffer (vedi sorgente Planets.c)

```
#define BUFFER_LENGTH 64

// Process the selection, which is triggered by a right mouse
    click at (xPos, yPos)
void ProcessSelection(int xPos, int yPos) {
    GLfloat fAspect;

    // Define the selection buffer
    static GLuint selectBuff[BUFFER_LENGTH];

    // Hit counter and viewport storage
    GLint hits, viewport[4];

    // Setup selection buffer
    glSelectBuffer(BUFFER_LENGTH, selectBuff);
```

Lavorare in Selection mode: il selection buffer (vedi sorgente Planets.c)

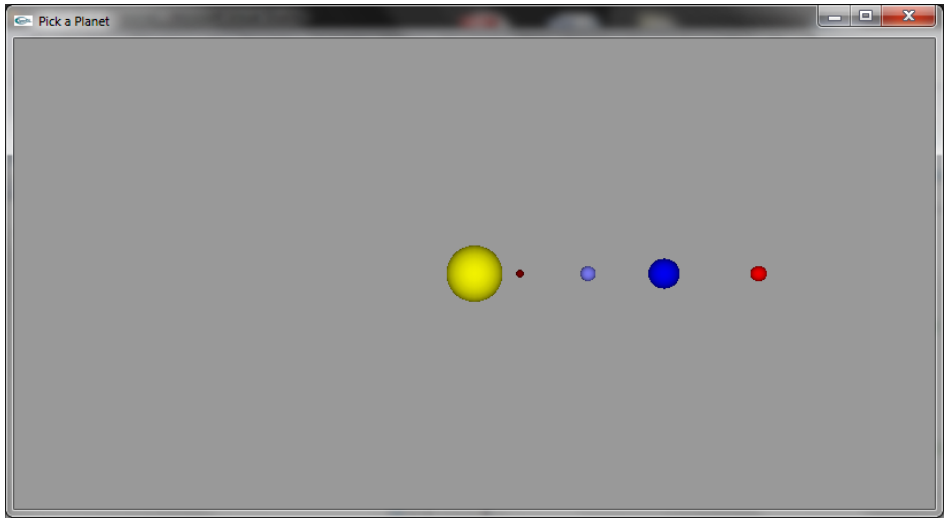
```
// Get the viewport
glGetIntegerv(GL_VIEWPORT, viewport);

// Switch to projection and save the matrix
glMatrixMode(GL_PROJECTION);
glPushMatrix();

// Change render mode
glRenderMode(GL_SELECT);

// Establish new clipping volume to be unit cube around
// mouse cursor point (xPos, yPos) and extending two pixels
// in the vertical and horizontal direction
glLoadIdentity();
gluPickMatrix(xPos, viewport[3] - yPos, 2,2, viewport);
```

Lavorare in Selection mode: il selection buffer (vedi sorgente Planets.c)

```
// Apply perspective matrix
fAspect = (float)viewport[2] / (float)viewport[3];
gluPerspective(45.0f, fAspect, 1.0, 425.0);

// Draw the scene
RenderScene();

// Collect the hits
hits = glRenderMode(GL_RENDER);

// If a single hit occurred, display the info.
if(hits == 1)
    ProcessPlanet(selectBuff[3]);
```

Lavorare in Selection mode: il selection buffer (vedi sorgente Planets.c)

```
    // Restore the projection matrix
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();

    // Go back to modelview for normal rendering
    glMatrixMode(GL_MODELVIEW);
}
```



Lavorare in Selection mode: il selection buffer

- La funzione gluPickMatrix è una comoda utility che crea una matrice che descrive il nuovo volume di visualizzazione:

```
void gluPickMatrix(GLdouble x, GLdouble y, GLdouble width,
    GLdouble height, GLint viewport[4]);
```

- I parametri x e y definiscono il centro del volume di picking in OpenGL window coordinates (si usano in genere le coordinate del mouse)
- I parametri width e height specificano le dimensioni del volume di vista in window pixels
- L'array viewport contiene le coordinate della vieport corrente; si possono ottenere facilmente queste informazioni chiamando la funzione glGetIntegerv(GL_VIEWPORT, viewport);

Lavorare in Selection mode: il selection buffer

- Per utilizzare gluPickMatrix, si deve prima salvare l'attuale matrice di proiezione, quindi chiamare glLoadIdentity per creare volume di vista unitario
- gluPickMatrix sposta questo volume si visualizzazione nella posizione corretta
- Infine, è necessario applicare eventuali ulteriori proiezioni prospettiche applicate nella scena originale, altrimenti non sarà possibile ottenere una mappatura corretta
- La funzione ProcessPlanet scrive il nome del pianeta clickato sulla window caption sfruttando glutSetWindowTitle

Algoritmi

Algoritmi

Stadi Principali

- La visualizzazione di un modello grafico passa attraverso quattro stadi principali, che costituiscono la cosiddetta pipeline grafica:
  - Modellazione
  - Elaborazione geometrica
  - Rasterizzazione
  - Display
- Il risultato della modellazione è un insieme di vertici che specificano la scena
- Il compito dell'elaborazione geometrica è quello di determinare quali siano gli oggetti visibili e di assegnare il colore a tali oggetti; in questa fase vengono applicati: normalizzazione, clipping, rimozione delle linee nascoste ed ombreggiatura

Stadi Principali

- Dopo che è stata effettuata la proiezione, si lavora con oggetti (vertici) bidimensionali che sono quindi sottoposti alla rasterizzazione o scan conversion per ottenere la proiezione della scena 3D nel piano 2D
- Infine, il processo di trasferimento dell'immagine dal frame-buffer allo schermo è svolto automaticamente dall'hardware grafico

Rasterizzazione

- Nei sistemi di grafica raster le primitive geometriche, descritte in termini dei loro vertici sul piano cartesiano, sono approssimate da matrici di pixel
- Con una matrice abbastanza densa è possibile rappresentare praticamente qualsiasi oggetto geometrico, anche complesso: punti, linee, cerchi, ellissi, etc.
- La procedura con cui le figure di tipo continuo si rappresentano come insieme di pixel discreti è chiamata scan conversion o rasterizzazione

Clipping

- Il clipping è il processo che consiste nell'eliminazione delle parti degli oggetti al di fuori del volume di vista
- Ci sono diversi modi per effettuare il clipping
- La tecnica più ovvia è quella di effettuare il clipping della primitiva prima della scan conversion, calcolandone le intersezioni analitiche con i bordi del rettangolo di visualizzazione; i punti di intersezione sono quindi usati per definire nuovi vertici per la primitiva
- Il vantaggio di eseguire il clipping prima della scan conversion risiede ovviamente nel fatto che la scan conversion deve trattare solo con la nuova versione delle primitive, non con quella originale, che potrebbe essere molto più estesa

Algoritmi di Clipping e Rasterizzazione

- I display raster invocano gli algoritmi di clipping e scan conversion ogni volta che un'immagine viene creata o modificata
- Questi algoritmi non solo devono creare immagini visualizzabili in modo soddisfacente, ma devono eseguire queste operazioni il più velocemente possibile
- Come vedremo, gli algoritmi di scan conversion usano metodi incrementali per minimizzare il numero di calcoli da eseguire durante ogni iterazione; inoltre essi usano l'aritmetica intera e non quella in virgola mobile

Approcci image-oriented e object-oriented

- La libreria grafica prende in input un insieme di vertici che specificano gli oggetti geometrici e produce in output i pixel nel frame buffer
- Per produrre un'immagine la libreria grafica deve considerare ciascun pixel, e per produrre un'immagine corretta, deve processare ogni primitiva geometrica e ogni sorgente luminosa
- La produzione delle immagini finali è perseguita generalmente con uno dei seguenti approcci:
  - image-oriented
  - object-oriented

L'approccio image-oriented

- L'approccio image-oriented è basato su un ciclo della forma

```
for (each_pixel)
    assign_a_color(pixel);
```

- Pertanto, la variabile fondamentale è il pixel
- Con questo approccio è necessario determinare quali primitive geometriche coinvolgono il pixel e, in base a esse determinare il colore per il disegno
- Nota: la quantità di memoria fisica richiesta per l'elaborazione del singolo pixel può essere inferiore alle dimensioni del frame buffer; tuttavia, a ogni passo, è richiesta l'elaborazione di tutte le primitive

L'approccio object-oriented

- L'approccio object-oriented è basato su un ciclo della forma

```
for (each_object)
    render(object);
```

- I vertici sono processati da una serie di moduli (funzioni) che li trasformano, li ombreggiano e determinano se sono visibili
- Ogni primitiva geometrica che emerge dal processing geometrico può influenzare ciascun pixel del frame buffer; quindi la memoria richiesta a ogni iterazione non può, in generale, essere minore di quella richiesta per l'intero frame buffer
- Comunque, grazie al fatto che le memorie grafiche sono diventate più generose ed economiche che in passato, queste richieste sono diventate accettabili
- Le librerie grafiche più diffuse propendono ormai per l'approccio object-oriented

Scan Conversion dei segmenti

- In questa sezione descriveremo alcuni algoritmi per la scan conversion dei segmenti
- Faremo l'ipotesi di aver già effettuato il clipping delle primitive, e supporremo che gli oggetti siano già stati proiettati sul piano bidimensionale
- Trascureremo, inoltre, l'effetto della rimozione delle linee nascoste, su cui torneremo più avanti

Scan Conversion dei segmenti

- Supponiamo che il frame buffer sia una matrice $n \times m$ di pixel, con origine nell'angolo in basso a sinistra.
- I pixel possono essere accesi attraverso una singola istruzione della forma
- L'approccio object-oriented è basato su un ciclo della forma

```
writePixel(int x, int y, int color);
```

- Il frame buffer è inerentemente discreto; per questo motivo possiamo usare i numeri interi per individuare le locazioni dei pixel al suo interno

Scan Conversion dei segmenti

- I pixel possono essere visualizzati secondo diverse forme e dimensioni
- Per il momento supponiamo che ciascuno sia visualizzato come un quadrato centrato nella locazione associata al pixel stesso, e di lato pari alla distanza tra due pixel
- Infine, faremo l'ipotesi che un processo concorrente legga il contenuto del frame buffer e visualizzi l'immagine sul display alla frequenza desiderata
- Questa assunzione ci permette di trattare la scan conversion indipendentemente dalla visualizzazione del contenuto del frame buffer

L'algoritmo di scan conversion DDA

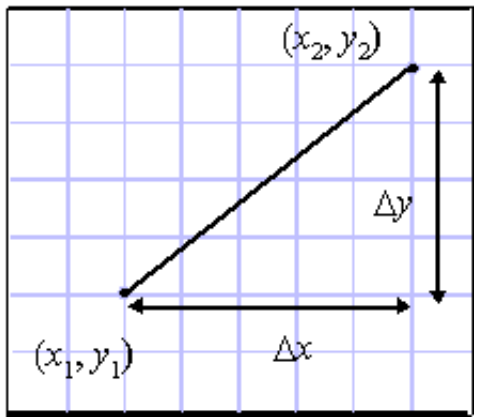
- Supponiamo di avere un segmento lineare definito dagli estremi (x_1, y_1) e (x_2, y_2)
- Assumeremo che questi valori siano già stati arrotondati in modo da corrispondere esattamente alla locazione di due pixel
- Si osservi che questa assunzione non è necessaria per derivare l'algoritmo; infatti, si può utilizzare una rappresentazione in virgola mobile per gli estremi e fare i calcoli usando l'aritmetica in virgola mobile, con il vantaggio di ottenere una scan conversion più accurata

L'algoritmo di scan conversion DDA

- La pendenza del segmento è definita da:

$$m = \Delta y / \Delta x$$

- Il più semplice algoritmo di scan conversion per i segmenti consiste nel calcolare m, incrementare x di 1 a partire dal punto più a sinistra e nel calcolare $y_i = mx_i + h$, dove h è l'ordinata del punto dove la retta incrocia l'asse y, per ciascuno degli x_i



L'algoritmo di scan conversion DDA

- Questa strategia è tuttavia inefficiente, perché ciascuna iterazione richiede una moltiplicazione e una somma in virgola mobile più un arrotondamento
- Si può eliminare la moltiplicazione usando una *tecnica incrementale* che consiste nel calcolare un punto della retta sulla base del punto precedente
- L'algoritmo che si ottiene prende il nome di algoritmo DDA [2]

[2] Il termine DDA deriva dall'inglese Digital Differential Analyzer), un dispositivo meccanico che risolve le equazioni differenziali applicando un metodo numerico: dato che una retta di pendenza m soddisfa l'equazione differenziale $dy/dx = m$, generare un segmento è equivalente a risolvere numericamente una semplice equazione differenziale

L'algoritmo di scan conversion DDA

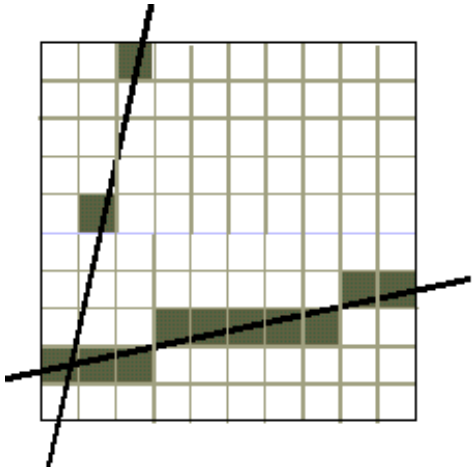
- Assumiamo che $0 \leq m \leq 1$ (gli altri valori di m possono essere trattati in modo simmetrico)
- L'algoritmo DDA consiste nel tracciare un pixel per ciascun valore di x compreso tra x_1 e x_2 tramite il comando writePixel()
- Per ogni variazione di x pari a Δx, la corrispondente variazione di y deve essere pari a $\Delta y = m\Delta x$
- Muovendosi da x_1 a x_2, x viene incrementato di 1 a ogni iterazione, ossia $\Delta x = 1$, e dunque dobbiamo incrementare y di $\Delta y = m$

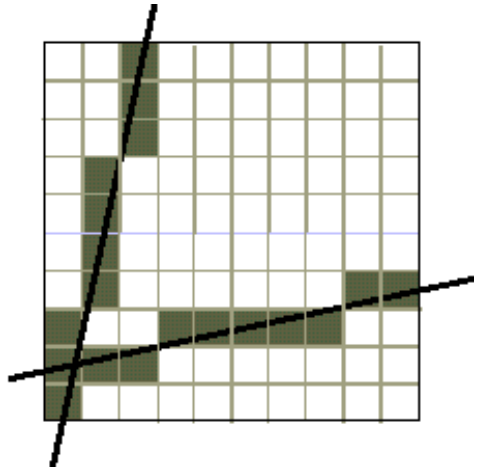
L'algoritmo di scan conversion DDA

```
int x;
float dy, dx, y, m;

dy = y2 - y1;
dx = x2 - x1;
m = dy/ dx;
y = y1;
for (x = x1, x <= x2, x++)
{
writePixel(x, round(y), line_color);
y += m;
}
```

L'algoritmo di scan conversion DDA

- La ragione per cui abbiamo limitato la pendenza massima a 1 può essere compresa osservando la figura sotto a sinistra
- L'algoritmo DDA è infatti della forma: per ogni x, cerca il miglior y, e per pendenze $> 45°$ si può perdere qualche pixel
- Per pendenze maggiori di 1 possiamo allora scambiare x e y, in modo tale che l'algoritmo diventi della forma: per ogni y, cerca il miglior x (vedi figura sotto a destra)





L'algoritmo di scan conversion di Bresenham

- L'algoritmo DDA è abbastanza efficiente e può essere facilmente implementato
- Tuttavia esso richiede un'addizione in virgola mobile per ogni pixel generato
- L'algoritmo di Bresenham per la scan conversion dei segmenti evita invece i calcoli in virgola mobile, e per questo motivo è diventato l'algoritmo di riferimento per la scan conversion dei segmenti
- Supponiamo, esattamente come nell'algoritmo DDA, che il segmento abbia estremi nei punti di coordinate (x_1, y_1) e (x_2, y_2) e che la pendenza m sia compresa tra 0 e 1

L'algoritmo di scan conversion di Bresenham

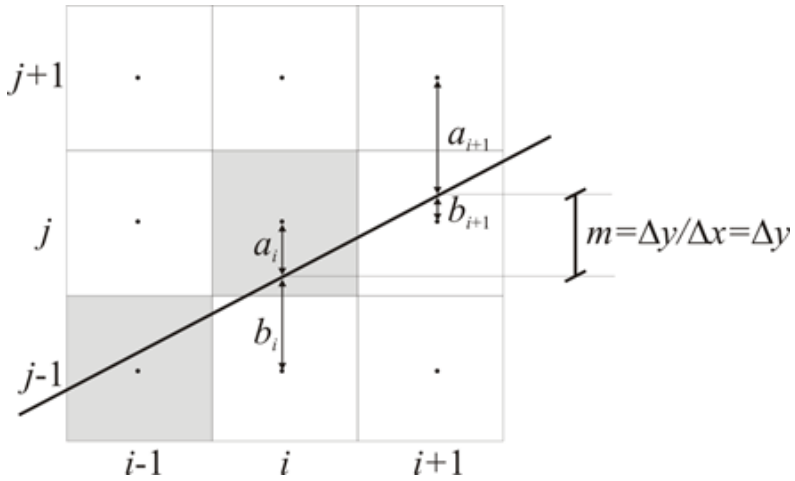
- Supponiamo di essere ad un passo intermedio del procedimento di rasterizzazione del segmento e di aver acceso il pixel in posizione (i, j)
- Per $x = i$ la retta $y = mx + h$, a cui appartiene il segmento, deve passare attraverso il pixel centrato su (i, j) (stiamo assumendo che i centri dei pixel siano posti nei punti intermedi tra due interi); in caso contrario, l'operazione di arrotondamento non avrebbe generato questo pixel
- Se andiamo avanti a $x = i + 1$, la condizione sulla pendenza indica che dobbiamo accendere uno tra due pixel possibili: o il pixel in posizione $(i + 1, j)$ o il pixel in posizione $(i + 1, j + 1)$

L'algoritmo di scan conversion di Bresenham

- Avendo ridotto la nostra scelta a soli due pixel, possiamo riformulare il problema in termini di una variabile decisionale

$$d = b - a$$

dove a e b rappresentano le distanze tra la retta e i due pixel candidati, di ascissa $x = i + 1$, e di ordinata $y = j + 1$ e $y = j$, rispettivamente (si veda la figura sotto)



L'algoritmo di scan conversion di Bresenham

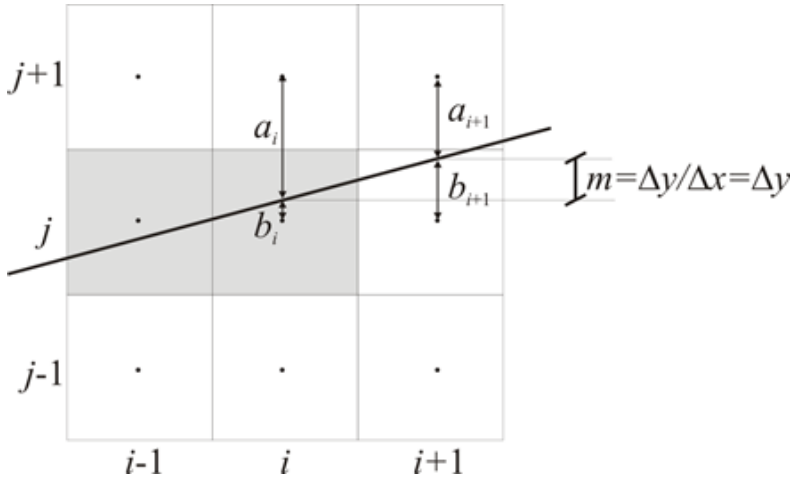
- Se la variabile di decisione ha valore negativo, la retta passa più vicino al pixel più in basso, così la scelta migliore, che garantisce la migliore approssimazione per il segmento, è quella rappresentata dal pixel in posizione $(i + 1, j)$; altrimenti, si sceglie il pixel in posizione $(i + 1, j + 1)$
- Si potrebbe calcolare la variabile decisionale d semplicemente usando l'equazione della retta $y = mx + h$, ma questo richiederebbe calcoli in virgola mobile
- Per ottenere il vantaggio computazionale dell'algoritmo di Bresenham, occorrono due ulteriori passi

L'algoritmo di scan conversion di Bresenham

- Per prima cosa si usa una tecnica incrementale per calcolare d che consente di eliminare i calcoli in virgola mobile
- Supponiamo che d_i sia il valore assunto da d in corrispondenza di $x = i$; vogliamo calcolare d_{i+1} incrementalmente, a partire da d_i
- Ci sono due situazioni da considerare che dipendono dal fatto che al passo precedente non sia stata incrementata l'ordinata y (CASO 1; $d_i \leq 0$), o che invece lo sia stato (CASO 2; $d_i > 0$)

L'algoritmo di scan conversion di Bresenham

CASO 1 ($d_i \leq 0$)



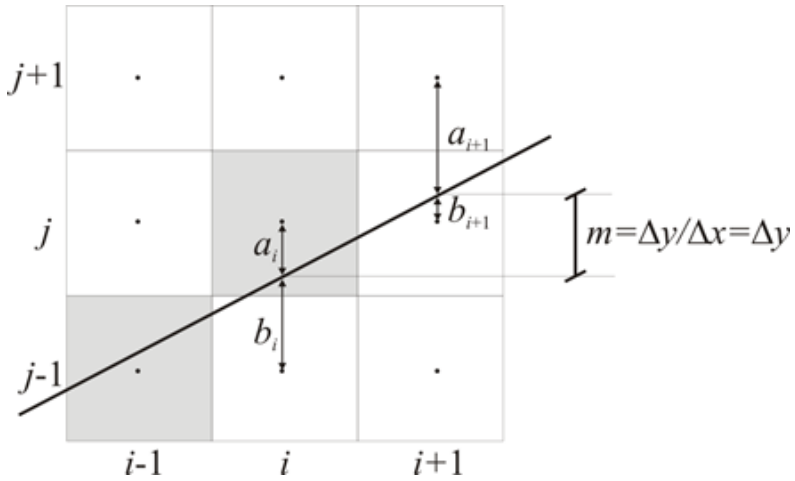
$$b_{i+1} = b_i + m$$

$$a_{i+1} = a_i - m$$

$$d_{i+1} = b_{i+1} - a_{i+1} = b_i - a_i + 2m = d_i + 2m$$

L'algoritmo di scan conversion di Bresenham

CASO 2 ($d_i > 0$)



$$1 + b_{i+1} = b_i + m \Rightarrow b_{i+1} = b_i + m - 1$$

$$a_{i+1} = 1 - b_{i+1} = 1 - (b_i + m - 1) = 1 - (1 - a_i + m - 1) = a_i - m + 1$$

$$d_{i+1} = b_{i+1} - a_{i+1} = b_i - a_i + 2m - 2 = d_i + 2m - 2 = d_i + 2(m - 1)$$

L'algoritmo di scan conversion di Bresenham

- Ricapitolando, abbiamo:

$$d_{i+1} = \begin{cases} d_i + 2m & \text{se } d \leq 0 \\ d_i + 2(m-1) & \text{se } d > 0 \end{cases}$$

- Se moltiplichiamo d_{i+1} per $\Delta x = x_2 - x_1$ non variamo il segno (in realtà è proprio il segno che ci interessa) e otteniamo (si noti che qui $\Delta y = y_2 - y_1$)

$$d_{i+1}\Delta x = \begin{cases} d_i\Delta x + 2m\Delta x = d_i\Delta x + 2\Delta y & \text{se } d \leq 0 \\ d_i\Delta x + 2(m-1)\Delta x = d_i\Delta x + 2(\Delta y - \Delta x) & \text{se } d > 0 \end{cases}$$

- Infine, imponiamo $d_i = d_i\Delta x$ (è lecito perché ci interessa solo il segno) per cui abbiamo:

$$d_{i+1} = d_i + \begin{cases} 2\Delta y & \text{se } d \leq 0 \\ 2(\Delta y - \Delta x) & \text{se } d > 0 \end{cases}$$

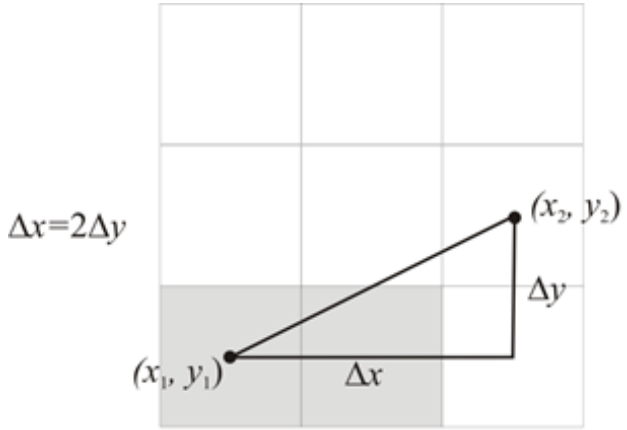
La viewport: glViewPort

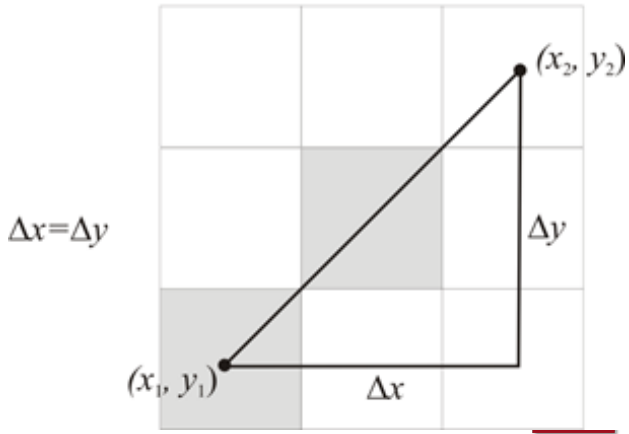
Per quanto riguarda l'inizializzazione di d, questa può essere:

$$d = 2(y_2 - y_1) - (x_2 - x_1)$$

In tal modo

- $d \leq 0$ quando l'angolo è < di 22.5° e viene correttamente scelto il pixel a Est
- $d > 0$ quando la pendenza è > di 22.5° e viene correttamente scelto il pixel a Nord Est





L'algoritmo di scan conversion di Bresenham

Il calcolo di ogni pixel successivo richiede dunque solo un'addizione e un test di segno; riassumendo:

- a ogni passo l'algoritmo di Bresenham sceglie tra due pixel sulla base del segno della variabile di decisione d
- Il segno della variabile d dipende dal valore di d al passo precedente e non richiede calcoli in virgola mobile
- è più efficiente dell'algoritmo DDA e rappresenta lo standard per la scan conversion dei segmenti

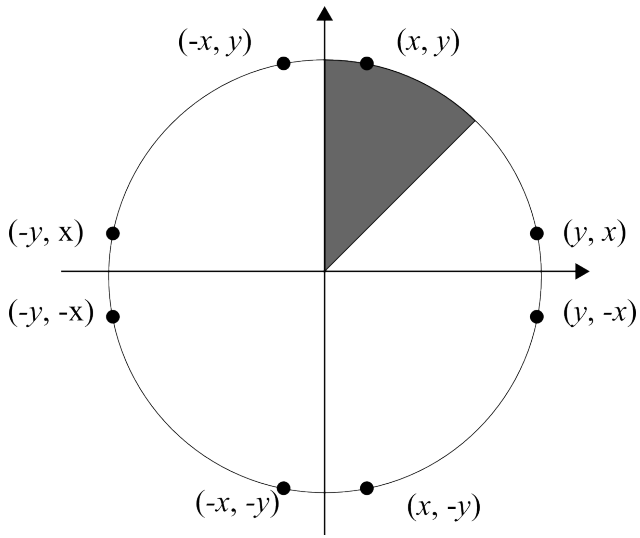
L'algoritmo di scan conversion di Bresenham

```
dx = x2 - x1; dy = y2 - y1;
d = dy * 2 - dx;
incrE = 2 * dy; incrNE = 2 * (dy - dx);
x = x1; y = y1; writePixel(x, y, value);
while (x < x2) {
    if (d <= 0) {
        d += incrE;
        x++;
    }
    else
    {
        d += incrNE;
        x++;
        y++;
    }
    writePixel(x, y, value);
}
```

L'algoritmo di Bresenham per le circonferenze

Osserviamo che, essendo il cerchio una figura simmetrica, dato un punto di coordinate (x, y), gli altri sette si trovano riflettendo, scambiando e invertendo di segno le coordinate:

$$P1 = (x, y) \qquad P2 = (y, x)$$
$$P3 = (-y, x) \qquad P4 = (-x, y)$$
$$P5 = (-x, -y) \qquad P6 = (-y, -x)$$
$$P7 = (y, -x) \qquad P8 = (x, -y)$$



L'algoritmo di Bresenham per le circonferenze

- Si considera l'equazione della circonferenza in forma implicita

$$F(x, y) = x^2 + y^2 - R^2$$

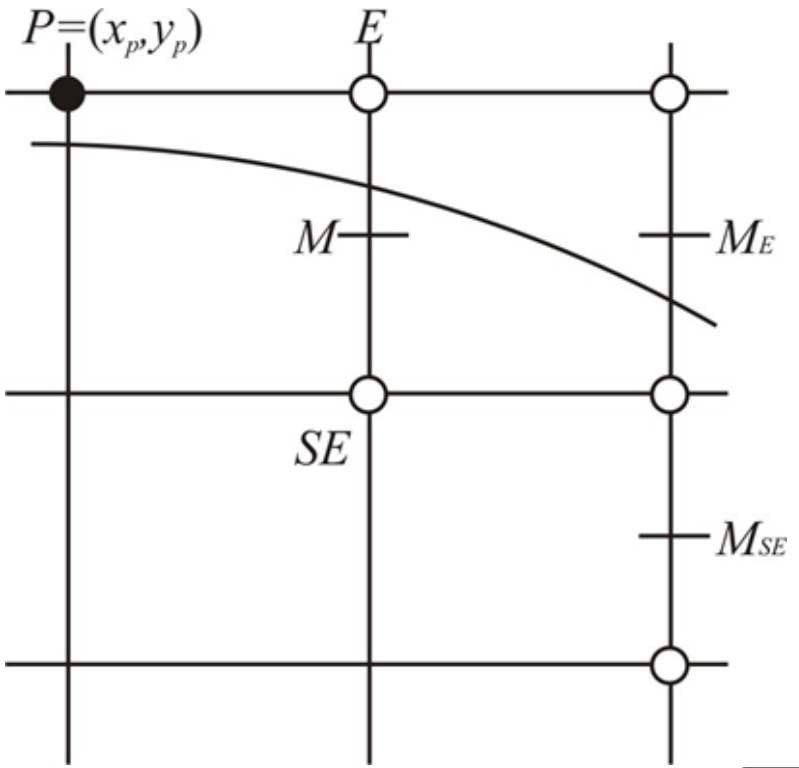
  - $F(x, y) = 0$ se (x, y) appartiene alla circonferenza
  - $F(x, y) < 0$ se (x, y) è interno alla circonferenza
  - $F(x, y) > 0$ se (x, y) è esterno alla circonferenza

- e si considera l'ottante che parte da 90 gradi e procede in senso orario verso 45 gradi

L'algoritmo di Bresenham per le circonferenze

Se P è l'ultimo pixel acceso, la scelta del prossimo pixel da accendere si restringe a E oppure SE

La scelta dipende da M: se è interno si accende E, se è esterno si accende SE



L'algoritmo di Bresenham per le circonferenze

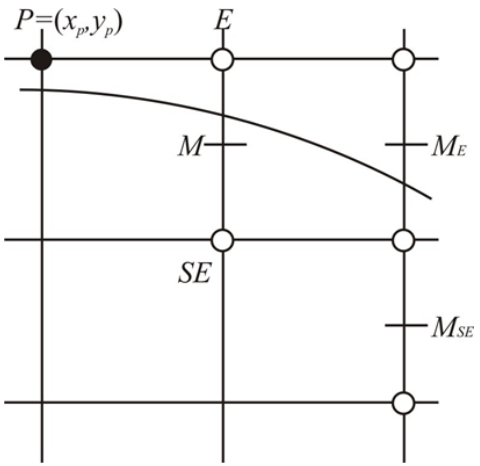
- Si può definire la variabile di decisione d come

$$d_M = F(x_p + 1, y_p - 1/2) = x_p^2 + y_p^2 + 2x_p - y_p - R^2 + 5/4$$

- Se $d_M \leq 0$ viene scelto il pixel E e il nuovo valore di d diventa

$$d_{M_E} = F(x_p + 2, y_p - 1/2) = x_p^2 + y_p^2 + 4x_p - y_p - R^2 + 17/4$$

- Per cui si ha:



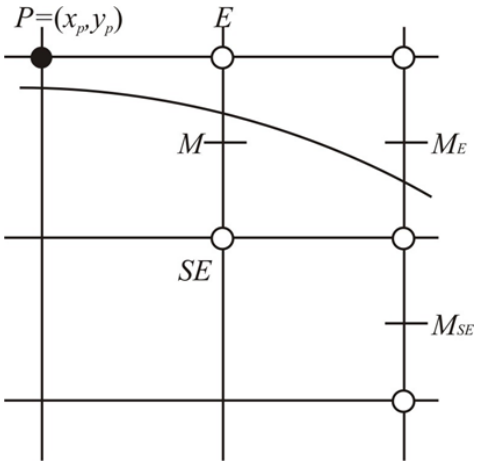
$$d_{M_E} - d_M = 2x_p + 3$$
$$d_{M_E} = d_M + 2x_p + 3$$

L'algoritmo di Bresenham per le circonferenze

- Se $d_M > 0$ viene scelto il pixel SE e il nuovo valore di d diventa

$$d_{M_{SE}} = F(x_p + 2, y_p - 3/2) = x_p^2 + y_p^2 + 4x_p - 3y_p - R^2 + 25/4$$

- Per cui



$$d_{M_{SE}} - d_M = 2x_p - 2y_p + 5$$
$$d_{M_{SE}} = d_M + 2x_p - 2y_p + 5$$

L'algoritmo di Bresenham per le circonferenze

- d può essere inizializzata in $(x_p, y_p) = (0, R)$

$$d = F(1, R - 1/2) = 5/4 - R$$

- Purtroppo il valore iniziale di d non è intero
- Tuttavia, si può sostituire a d l'espressione $h + 1/4$ per cui si ha:

$$h + 1/4 = 5/4 - R \Rightarrow h = 1 - R$$

- e sostituire il confronto $d = 0$ con $h = -1/4$
- Tuttavia, poiché h è inizializzata a un valore intero ed è incrementata di valori interi, il confronto $h = -1/4$ può essere finalmente sostituito dal confronto $h = -1$

L'algoritmo di Bresenham per le circonferenze

```
int x, y; float d;
/* inizializzazione */
x = 0;
y = R;
d= 1-R;
CirclePoints(x, y, line_color);
while (y > x) {
    if (d <= -1) {/* scelta di E */
        d += 2*x + 3;
        x++;
    }
    else { /* scelta di SE */
        d += 2*(x - y) + 5;
        x++;
        y--;
    }
    CirclePoints(x, y, line_color);
}
```

Scan Conversion dei poligoni

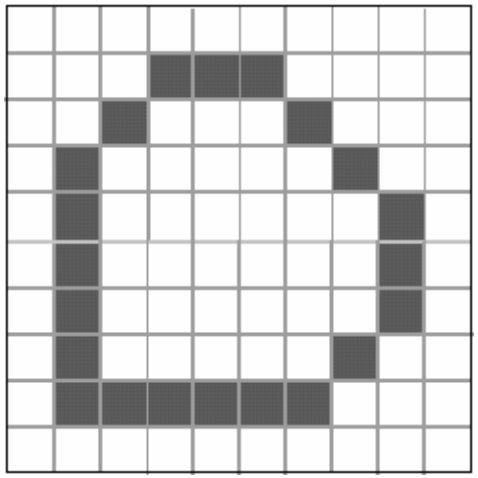
- Uno dei principali vantaggi introdotti dalla grafica raster è stato quello di permettere il riempimento delle aree interne dei poligoni
- A differenza della scan conversion dei segmenti, dove un singolo algoritmo ha un ruolo dominante sugli altri, per i poligoni sono stati sviluppati e applicati diversi metodi, e la scelta di un metodo in particolare dipende fortemente dall'architettura su cui si effettua l'implementazione

Scan Conversion dei poligoni

- Supponiamo di voler riempire il poligono con un unico colore (scelta che ci consente di semplificare la trattazione); la regola di base per effettuare il riempimento del poligono è la seguente: se un punto è all'interno del poligono, coloralo usando il colore di riempimento prescelto
- Questo algoritmo concettuale mostra come il problema di riempimento di un poligono sia in effetti un problema di ordinamento, basato sulla divisione dei pixel del frame buffer in due categorie
  - quelli che sono all'interno del poligono
  - quelli che sono all'esterno

Scan Conversion dei poligoni: l'algoritmo Flood fill

- Possiamo visualizzare un poligono solo tramite i suoi lati, senza colorare la regione interna, applicando l'algoritmo di Bresenham per effettuare la scan conversion dei lati
- Supponiamo di avere solo due colori a disposizione: un colore di sfondo (bianco) e un colore di primo piano, o di drawing (grigio)
- Possiamo usare il colore di drawing per la scan conversion dei lati ottenendo, ad esempio, il frame buffer illustrato in figura



Scan Conversion dei poligoni: l'algoritmo Flood fill

- Per effettuare il riempimento del poligono s'individua innanzitutto un punto di coordinate (x, y) all'interno del poligono:
- A tal fine si può considerare il vertice di ordinata minima (x_{min}, y_{min}) e una retta del tipo $y = k$ (detta linea di scansione, o scan-line)
  1. s'inizializza $k = y_{min} + 1$
  2. si valutano le intersezioni con gli spigoli del poligono;
  3. se si hanno due intersezioni (x_1, k) e (x_2, k) tali che $|x_1 - x_2| > 1$, allora si pone
     $$(x, y) = (x_1 + 1, k)$$
  4. in caso contrario, si pone $k = k + 1$ e si itera dal punto 2.
- Individuato il punto di coordinate (x, y) interno al poligono, allora si considerano ricorsivamente i punti vicini e, nel caso in cui non appartengano ai lati del poligono (colore diverso dal colore degli spigoli), li si colora di grigio (colore di drawing prescelto)

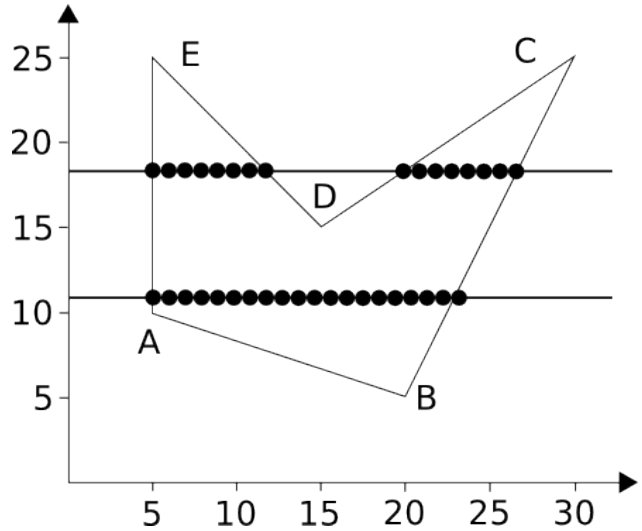
Scan Conversion dei poligoni: l'algoritmo Flood fill

- L'algoritmo può essere espresso nel seguente modo, in cui assumiamo che ci sia una funzione readPixel che restituisce il colore di un pixel

```
floodFill(int x, int y)
{
    if (readPixel(x, y) == WHITE){
        writePixel(x, y, GRAY);
        floodFill(x-1, y);
        floodFill(x+1, y);
        floodFill(x, y-1);
        floodFill(x, y+1);
    }
}
```

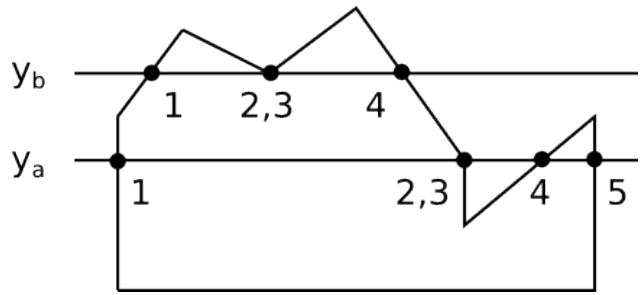
Scan Conversion dei poligoni: l'algoritmo scan-line

- L'algoritmo scan-line è un algoritmo generale in quanto funziona per qualsiasi tipo di poligono: convessi, concavi, intrecciati, con o senza buchi
- L'algoritmo scan-line fa uso delle linee di scansione del frame buffer per individuare i pixel interni al poligono



Scan Conversion dei poligoni: l'algoritmo Scan-line

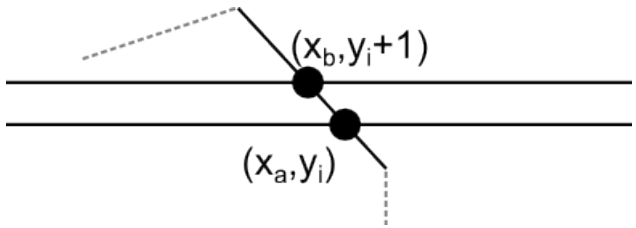
- Per ogni linea di scansione vengono individuate le intersezioni con gli spigoli del poligono e i pixel tra coppie di intersezioni dispari-pari (1-2, 3-4, ecc.) sono accesi



- Si noti che la scan-line di ordinata y_a dà luogo a 5 intersezione: in tal caso le intersezioni numero 2 e 3 vanno considerate come unica poiché comuni a due spigoli consecutivi monotonicamente crescenti (o decrescenti)
- Al contrario, le intersezioni 2 e 3 della scan-line di ordinata y_b vanno mantenute poiché il vertice comune rappresenta un minimo (o un massimo) locale

Scan Conversion dei poligoni: l'algoritmo Scan-line

- La rasterizzazione del poligono procede dal basso verso l'alto e da sinistra verso destra
- Così, ogni scan-line differisce in y dalla precedente di una sola unità



- Per tale motivol, il coefficiente angolare dello spigolo è dato da

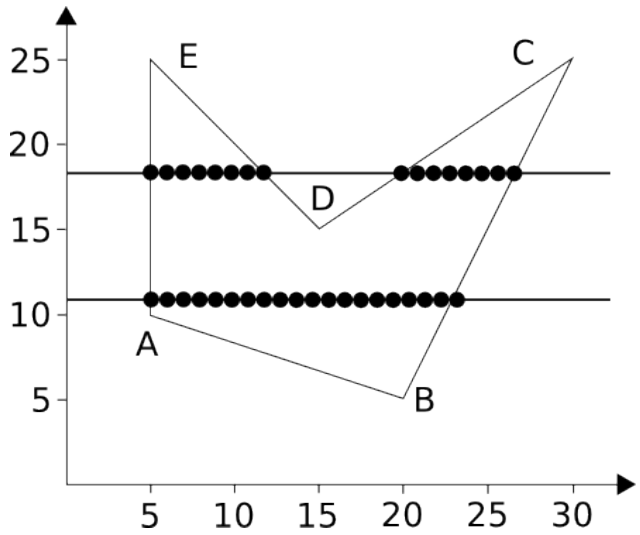
$$m = \frac{y_{i+1} - y_i}{x_b - x_a} = \frac{1}{x_b - x_a}$$

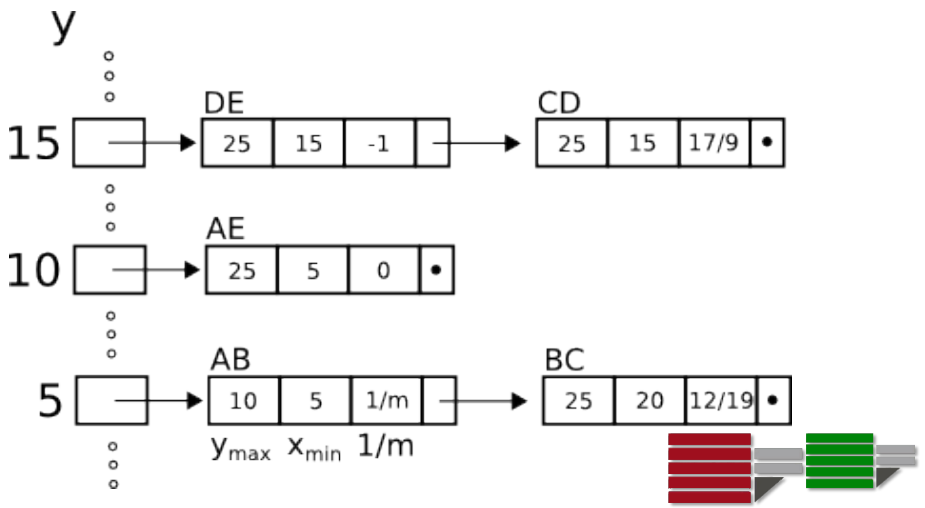
da cui deriva che

$$x_b = x_a + \frac{1}{m}$$

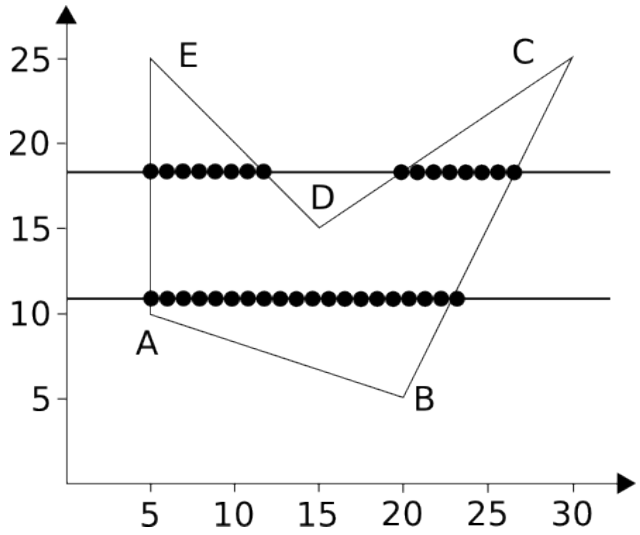
Scan Conversion dei poligoni: l'algoritmo Scan-line

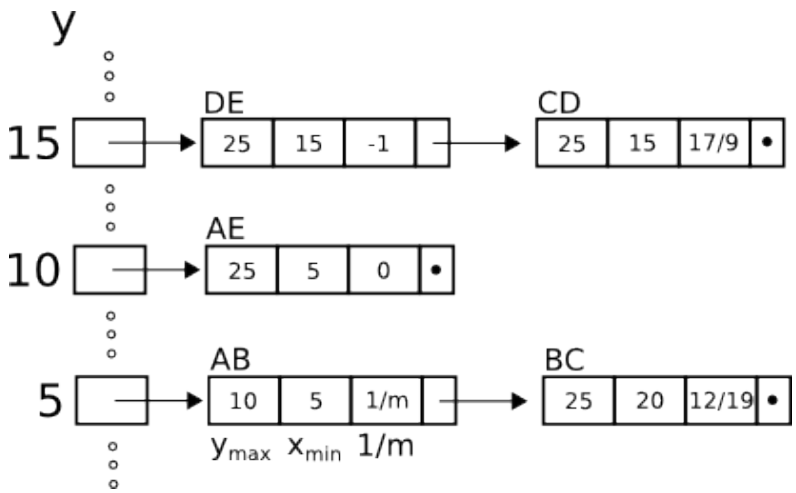
- Pertanto, una volta calcolata la prima intersezione (di ordinata minima), la relazione $x_b = x_a + 1/m$ permette di calcolare le direttamente le successive incrementando y di 1 e x di $1/m$
- L'algoritmo utilizza una tabella di spigoli (ET), un vettore di puntatori avente tante entrate quante sono le linee di scansione del frame buffer





Scan Conversion dei poligoni: l'algoritmo Scan-line

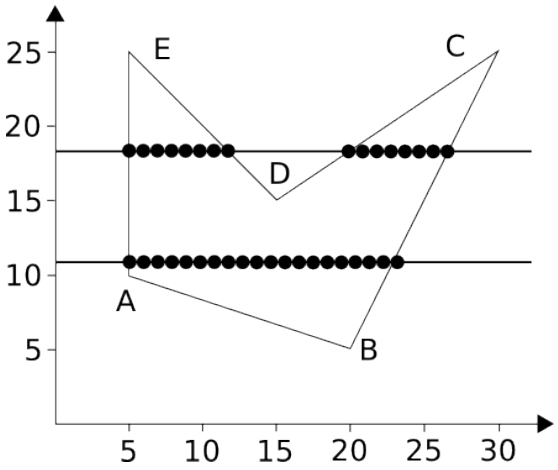


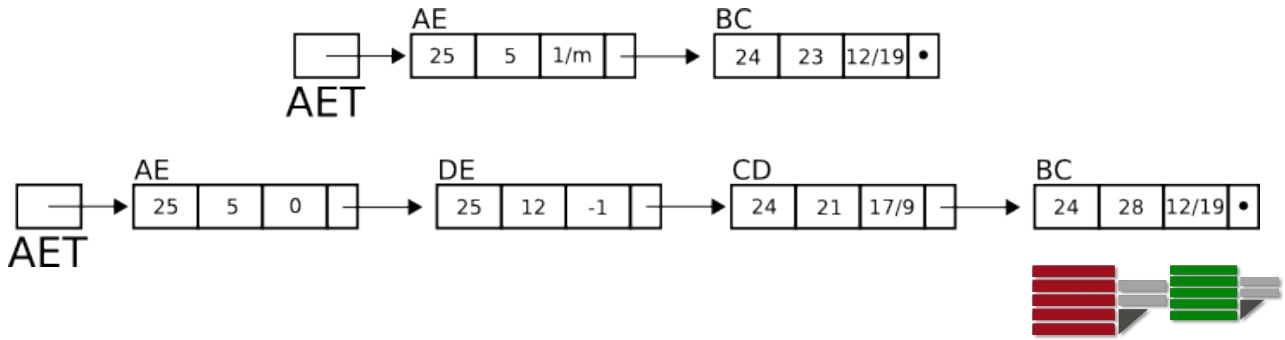


- La *j*-esima posizione della ET, se non nulla, punta alla lista degli spigoli aventi ordinata minima *j*
- All'interno di ogni posizione, gli spigoli sono ordinati per ascissa minima (x_{min}) crescente

Scan Conversion dei poligoni: l'algoritmo Scan-line

- L'ET è affiancata dalla tabella degli spigoli attivi (AET), cioè gli spigoli interessati dalla linea di scansione corrente
- Per ogni spigolo rappresentato, tuttavia, il secondo campo vale l'ascissa corrente e non l'ascissa minima
- La figura sotto mostra il contenuto dell'AET in due differenti momenti dell'elaborazione: in corrispondenza delle scan-line $y = 11$ e $y = 18$





Scan Conversion dei poligoni: l'algoritmo Scan-line

L'algoritmo procede quindi nel seguente modo:

1. imposta la scan-line corrente alla più piccola ordinata y per cui la corrispondente posizione in ET non sia vuota ($y = 5$ nel nostro esempio);
2. inizializza AET alla struttura vuota
3. ripeti i seguenti passi fino a che ET e AET non siano vuote:
   - sposta in AET gli spigoli relativi all'ordinata y;
   - accendi i pixel corrispondenti agli intervalli di ascisse di posizione dispari-pari (1-2, 3-4, ecc.);
   - rimuovi da AET gli spigoli per cui risulti $y = y_{max}$ (che non saranno interessati dalla prossima scan-line);
   - incrementa y di 1, in caso di spigoli non verticali, x di $1/m$.

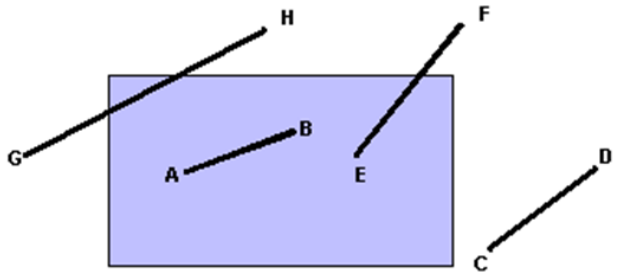
Clipping

- Un altro compito fondamentale dei sistemi di grafica raster è il clipping, ovvero il processo di determinare quali primitive, o parti di primitive, sono interne al volume di vista
- Diremo che le primitive che cadono entro il volume di visualizzazione sono accettate mentre le rimanenti sono eliminate o rifiutate
- Le primitive che cadono solo parzialmente entro il volume di visualizzazione devono essere tagliate in modo che le parti che cadono al di fuori del volume di vista siano rimosse

Clipping

- Il clipping può essere eseguito prima della scan conversion, oppure nel corso della stessa scan conversion
- Per i pacchetti grafici che operano in aritmetica in virgola mobile, la soluzione migliore è quella di effettuare prima il clipping nel sistema di coordinate in virgola mobile, e quindi la scan conversion delle primitive già ridotte
- Nella libreria OpenGL si effettua il clipping delle primitive rispetto ad un volume tridimensionale prima di effettuare la scan conversion

Clipping



- Solo il segmento AB apparirà integralmente sullo schermo, mentre l'intero segmento CD sarà eliminato; i segmenti EF e GH dovranno invece essere accorciati prima di essere visualizzati
- Tutta l'informazione necessaria per il clipping può essere determinata calcolando le intersezioni tra le rette cui appartengono i segmenti e i bordi della finestra di clipping
- È preferibile ridurre il calcolo delle intersezioni poiché ogni intersezione richiede calcoli in virgola mobile

Algoritmo di Cohen-Sutherland

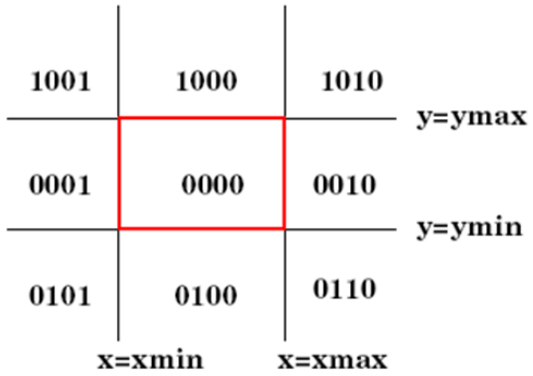
- Nell'algoritmo di Cohen-Sutherland la maggior parte di moltiplicazioni e divisioni in virgola mobile è sostituita da una combinazione di sottrazioni in virgola mobile e operazioni su bit
- I lati della finestra di clipping vengono estesi idealmente all'infinito, dividendo il piano in nove regioni
- A ogni regione si assegna un codice binario $b_0 b_1 b_2 b_3$ composto di quattro bit e chiamato outcode

Algoritmo di Cohen-Sutherland

Si pone:

$$b_0 = \begin{cases} 1 & \text{se } y > y_{max} \\ 0 & \text{se } y \leq y_{max} \end{cases}$$

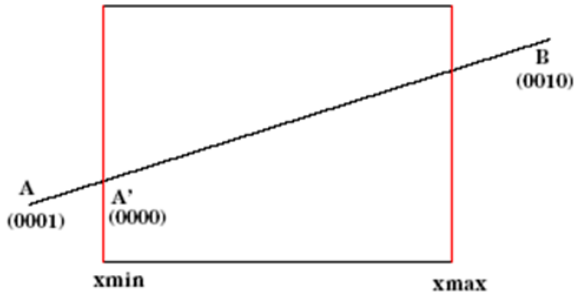
- Analogamente si pone b_1 è uguale a 1 se $y < y_{min}$
- b_2 e b_3 sono invece determinati dalle relazioni tra x e i bordi sinistro e destro della finestra di visualizzazione
- Si noti che la regione che corrisponde alla finestra di visualizzazione è caratterizzata dall'outcode 0000



Algoritmo di Cohen-Sutherland

- CASO 1: $O_1 = O_2 = 0000$
  Entrambi gli outcode sono 0000; il segmento è INTERNO
- CASO 2: $O_1 \& O_2 \neq 0000$
  I due punti estremi sono entrambi sotto, sopra, a sinistra o a destra della window; il segmento è ESTERNO
- CASO 3: $O_1 \& O_2 = 0000$
  Gli estremi sono esterni, ma la linea non può essere rifiutata perché potrebbe intersecare la window
- CASO 4: $O1 \neq 0000$; $O2 = 0000$, o viceversa
  Si considera fra i due estremi quello con outcode non nullo; si determina l'intersezione fra il segmento dato e il lato della window (corrispondente al bit non nullo dell'outcode); si aggiorna l'estremo considerato con l'intersezione trovata

Algoritmo di Cohen-Sutherland



- Il segmento AB non è né interno né esterno
- sia A l'estremo con outcode non nullo; calcolo A', intersezione tra AB e $X = Xw_{min}$ (retta associata al bit 4); il segmento risultante è $A'B$
- Dagli outcode di A' e B risulta che $A'B$ non è né interno né esterno
- B è l'unico estremo con outcode non nullo; calcolo B' intersezione tra $A'B$ e $X = Xw_{max}$ (retta associata al bit 3); il segmento è $A'B'$ che è interno

Algoritmo di Cohen-Sutherland

- I controlli sugli outcode richiedono solo operazioni Booleane e le intersezioni vengono calcolate solo se necessario
- L'algoritmo di Liang-Barsky per il clipping dei segmenti è più efficiente ma non sarà trattato per questioni di tempo così come gli algoritmi di clipping di poligoni (es. Sutherland-Hodgman)

Antialiasing

- La scan conversion dei segmenti e dei poligoni può apparire disturbata da alcuni difetti di visualizzazione
- Questi difetti si verificano tutte le volte che cerchiamo di passare dalla rappresentazione continua di un oggetto, che ha una risoluzione infinita, alla sua approssimazione discreta, che ha invece una risoluzione limitata
- A questo fenomeno si dà il nome di aliasing; l'applicazione di tecniche che riducono e eliminano l'aliasing sono dette di antialiasing

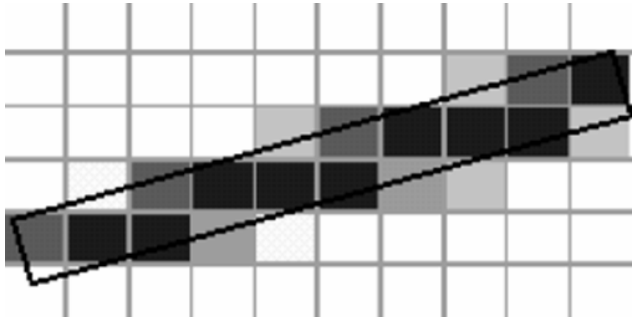
Antialiasing

- Gli errori sono causati da tre problemi collegati alla natura discreta del frame buffer, laddove il singolo pixel non sia indistinguibile a occhio nudo
  1. in un frame buffer di dimensione n x m, il numero di pixel è fissato e segmenti tra loro diversi potrebbero essere approssimati dallo stesso pattern di pixel
  2. le locazioni dei pixel sono fissate su una griglia uniforme
  3. i pixel hanno dimensione e forma fissata

Antialiasing

- Nonostante le rette siano entità unidimensionali, le rette ottenute con la scan conversion devono avere un'ampiezza per poter essere visibili
- Una primitiva, una volta rasterizzata, occupa un'area finita dello schermo; possiamo quindi pensare al nostro segmento come ad un rettangolo con un certo spessore che copre una parte della griglia di pixel
- Possiamo allora osservare che un segmento non dovrebbe accendere un solo pixel di una colonna usando il colore nero ma, piuttosto, dovrebbe visualizzare più di un pixel per riga o colonna, con intensità opportuna di colore

Antialiasing



- Allora, per le linee di ampiezza pari a un pixel, solo quelle verticali e orizzontali dovrebbero influenzare esattamente un pixel per colonna e riga, rispettivamente
- Per le linee di pendenza diversa, più di un pixel deve essere visualizzato in una riga o una colonna

Antialiasing: Unweighted Area Sampling

- Il metodo *unweighted area sampling (UAS)*, molto efficiente per valutare approssimare la percentuale di area del pixel coperta dal segmento, consiste nell'ipotizzare una risoluzione del frame buffer almeno doppia di quella reale, il chè porta ad avere ogni pixer suddiviso in almeno 4 sub-pixel, utili al fine di poter differenziare le intensità in almeno 4 differneti tonalità
- Il metodo UAS è caratterizzato da tre proprietà di base:
  1. l'intensità di ciascun pixel decresce al crescere della distanza tra il centro del pixel e il bordo esterno del segmento;
  2. i pixel che non sono intersecati dalla linea restano bianchi;
  3. porzioni di pixel di eguale superficie danno luogo a intensità uguali indipendentemente dalla loro distanza dal centro del pixel

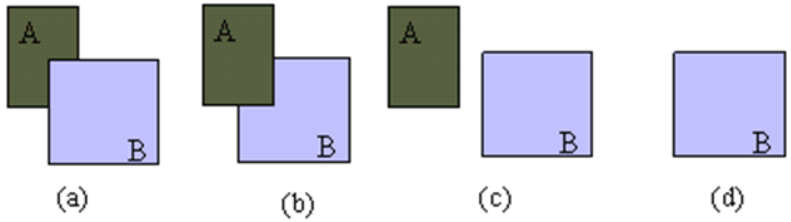
Antialiasing: Weighted Area Sampling

- Il metodo *Weighted area sampling (WAS)*, migliora il metodo UAS visto che il precedente punto 3 implica che porzioni di pixel contribuiscono in egual misura indipendentemente dalla loro posizione
- Il metodo WAS attribuisce un peso a ogni porzione del pixel
- La funzione peso è di solito un prisma a base quadrata o, per comodità, un cono con base circoscritta al pixel
- il peso è dato dal volume relativo alla proiezione dell'area intersecata dal segmento

Rimozione delle Superfici Nascoste

- Una volta che a un insieme di vertici sono state applicate le trasformazioni geometriche richieste e gli oggetti definiti da questi vertici sono stati sottoposti al clipping, prima di eseguire la scan conversion, si deve risolvere il problema della rimozione delle superfici nascoste per determinare se un oggetto è visibile all'osservatore, oppure rimane oscurato da altri oggetti
- Gli algoritmi per la rimozione delle superfici nascoste si possono dividere in due classi:
  - gli algoritmi object-space determinano, per ogni oggetto, quali parti dell'oggetto non sono oscurate da altri oggetti nella scena
  - gli algoritmi image-space determinano, per ogni pixel, quale è l'oggetto più vicino all'osservatore
- La libreria OpenGL utilizza un particolare algoritmo, l'algoritmo z-buffer, che appartiene alla seconda classe

Approccio object-space



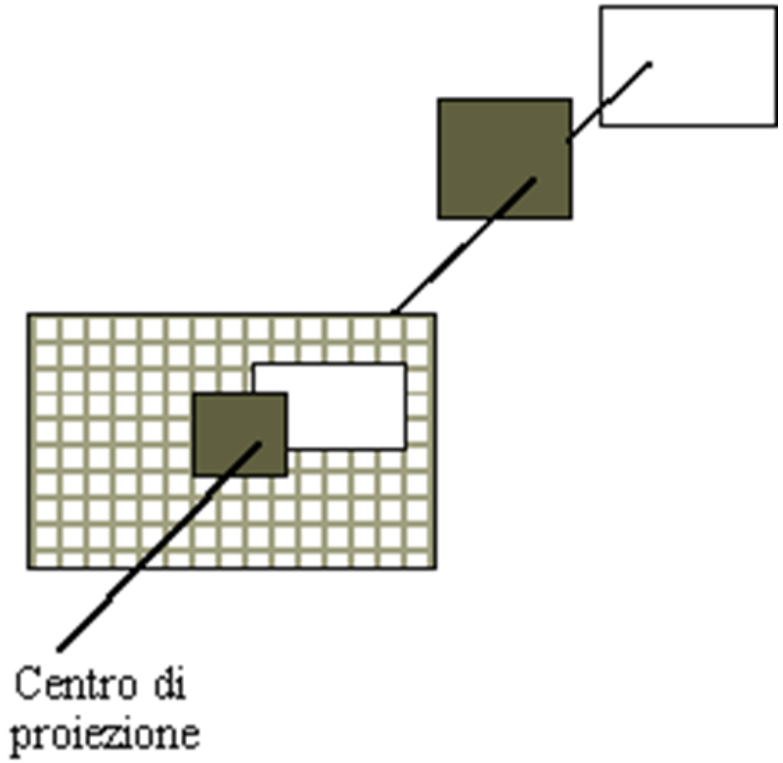
- Data una scena tridimensionale composta da k poligoni piatti e opachi, si può derivare un generico algoritmo di tipo object-space considerando gli oggetti a coppie: data una coppia di poligoni, ad esempio A e B, ci sono quattro casi da considerare:
  - A oscura B: visualizzeremo solo A
  - B oscura A: visualizzeremo solo B
  - A e B sono completamente visibili: visualizzeremo sia A che B
  - A e B si oscurano parzialmente l'un l'altro: dobbiamo calcolare le parti visibili di ciascun poligono

Approccio object-space

- Dal punto di vista della complessità, la determinazione del caso da esaminare e il calcolo della parte visibile di un poligono sono considerate come una singola operazione; si procede quindi induttivamente
- Si prende uno dei k poligoni e lo si confronta con tutti i restanti $k-1$; in questo modo si determina quale parte del poligono sarà visibile
- Questo processo è ripetuto con gli altri poligoni: a ogni passo si confronta il poligono in esame con tutti i poligoni rimanenti, fino a quando rimangono solo due poligoni
- La complessità di questo approccio risulta quindi di ordine $O(k^2)$
- È dunque chiaro che l'approccio object-space è consigliabile solo quando gli oggetti nella scena sono relativamente pochi

Approccio image-space

- Per ogni pixel, si considera un raggio che parte dal centro di proiezione e passa per quel pixel; il raggio è intersecato con ciascuno dei piani determinati dai k poligoni per determinare per quali piani il raggio attraversa un poligono
- Infine, si determina quale intersezione è più vicina al centro di proiezione e si colora il pixel in esame usando la gradazione di colore del poligono nel punto di intersezione



Approccio image-space

- L'operazione fondamentale dell'approccio image-space è il calcolo delle intersezioni dei raggi di proiezione con i poligoni
- Per un display n x m questa operazione deve essere eseguita $n \cdot m \cdot k$ volte e la complessità risulta di ordine O(k)
- Naturalmente, per aumentare l'accuratezza delle immagini visualizzate, si possono considerare anche più di un raggio per pixel

Eliminazione delle back face

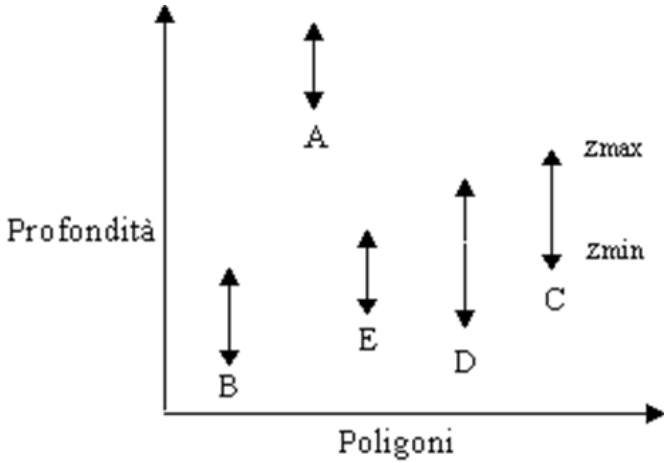
- Per ridurre il carico di lavoro richiesto per la rimozione delle superfici nascoste è opportuno eliminare prima tutti i poligoni il cui vettore normale è orientato verso il semispazio opposto all'osservatore, ossia i poligoni che definiscono la parte posteriore della superficie di un oggetto solido, non visibile all'osservatore
- Se indichiamo con α l'angolo tra la normale e l'osservatore, il poligono in esame definisce la parte anteriore di un oggetto se e solo se -90° < α < 90° o, equivalentemente, cos $\alpha > 0$
- In OpenGL, l'eliminazione delle back faces viene eseguita dalla funzione glCullFace

L'algoritmo depth sort

- L'algoritmo depth sort (di ordinamento in profondità) è una implementazione dell'approccio object-space
- Consideriamo una scena composta da poligoni planari (estensioni a casi più complessi sono possibili): l'algoritmo depth sort è in realtà una variante di un algoritmo ancora più semplice, chiamato algoritmo del pittore che, consiste nel dipingere i poligoni dal più lontano al più vicino, dipingendo man mano sopra le parti dei poligono più lontani non visibili all'osservatore
- L'idea alla base dell'algoritmo depth sort è proprio quella di rasterizzare gli elementi in modo inverso rispetto all'ordine di profondità in modo che gli elementi più lontani siano progressivamente oscurati da quelli più vicini

L'algoritmo depth sort

- I problemi da risolvere per implementare questo approccio riguardano l'ordinamento in profondità dei poligoni



- Più precisamente, si considera l'estensione nella direzione z di ogni poligono, come illustrato in figura

L'algoritmo depth sort

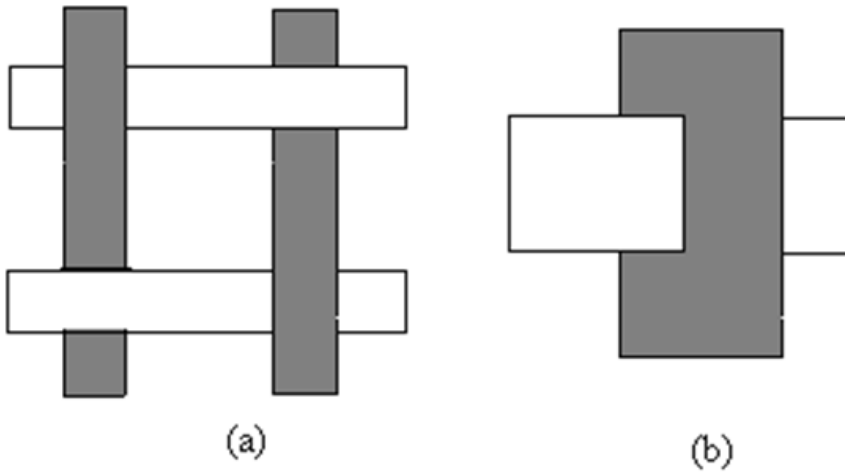
- Se la profondità minima di ogni poligono è maggiore della profondità massima del poligono situato sul retro, possiamo dipingere i poligoni partendo da quello più in profondità (è il caso del poligono A nell'esempio in figura, che è situato dietro a tutti gli altri poligoni e può essere dipinto per primo)
- Gli altri poligoni, tuttavia, non possono essere dipinti basandosi solo sulla loro estensione lungo z: se le estensioni lungo z di due poligoni si sovrappongono, dobbiamo determinare un ordine per dipingerli individualmente che permetta di ottenere l'immagine corretta
- L'algoritmo depth sort esegue a questo scopo una serie di test

L'algoritmo depth sort

- Consideriamo ad esempio una coppia di poligoni le cui estensioni z si sovrappongono: il test più semplice consiste nel controllare le estensioni lungo x e lungo y
  - Se non c'è sovrapposizione in almeno una delle due direzioni, allora sicuramente nessuno dei due poligoni può oscurare l'altro, ed essi possono essere dipinti in un ordine qualsiasi
- Rimangono da considerare due situazioni problematiche, per cui non esiste un ordine corretto di rappresentazione

L'algoritmo depth sort

- La prima si verifica quando tre o più poligoni si sovrappongono ciclicamente (a)
- La seconda situazione si verifica invece quando un poligono penetra nell'altro (b)



L'algoritmo depth sort

- In questi casi, è necessario derivare i dettagli delle intersezioni, spezzare i poligoni in corrispondenza dei segmenti di intersezione e provare a cercare un ordine corretto di rappresentazione del nuovo insieme di poligoni
- L'analisi delle prestazioni dell'algoritmo depth sort è difficile poiché i particolari delle singole applicazioni determinano quanto spesso si possano verificare i casi difficili da trattare
- In ogni caso, la complessità risulta più che lineare rispetto al numero di poligoni, poiché è necessario eseguire un algoritmo di sorting delle profondità

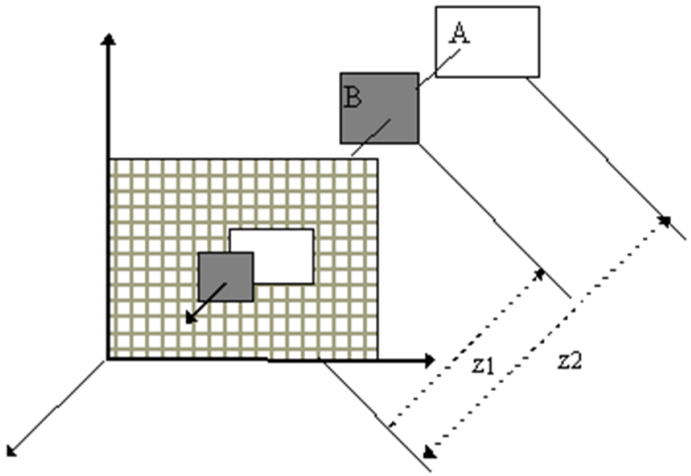
L'algoritmo z-buffer

- L'algoritmo z-buffer è un algoritmo di tipo image-space, basato su una logica molto semplice, e facile da implementare
- Lavora in stretto accoppiamento con l'algoritmo di scan conversion e necessita, oltre alla memoria di display (frame buffer), anche di un'area di memoria in cui memorizzare le informazioni di profondità relative ad ogni pixel
- Quest'area addizionale di memoria è chiamata z-buffer
- Nonostante l'algoritmo z-buffer sia di tipo image-space, i cicli al suo interno sono eseguiti rispetto ai poligoni e non rispetto ai pixel (approccio ibrido)

L'algoritmo z-buffer

- Supponiamo di dovere visualizzare due poligoni
- Nel corso dell'esecuzione della scan conversion, è possibile calcolare il colore da associare ad ogni punto di intersezione tra un raggio diretto dal centro di proiezione a un pixel, applicando un modello di shading
- Contemporaneamente, si può controllare se il punto di intersezione è visibile o meno, secondo la regola che stabilisce che il punto è visibile se è il punto di intersezione più vicino al centro di proiezione

L'algoritmo z-buffer



- Quindi, se si esegue la scan conversion del poligono B, il suo colore apparirà sullo schermo poiché la distanza z_1 è minore della distanza z_2 relativa al poligono A
- Al contrario, se si esegue la scan conversion del poligono A, il pixel che corrisponde al punto di intersezione non apparirà sul display

L'algoritmo z-buffer

- Poiché si procede poligono per poligono, non è possibile disporre di tutte le informazioni relative agli altri poligoni; tuttavia, si possono memorizzare e aggiornare le informazioni relative alla profondità che si rendono via via disponibili nel corso della scan conversion
- Supponiamo di avere a disposizione una memoria, lo z-buffer, con la stessa risoluzione del frame buffer e con una profondità consistente con la risoluzione che si vuole ottenere per le distanze

L'algoritmo z-buffer

- Ad esempio, se abbiamo un display 1280 x 1024 e utilizziamo l'aritmetica in virgola mobile per i calcoli di profondità, possiamo utilizzare uno z-buffer 1280 x 1024, con elementi a 32 bit
- Ogni elemento è inizializzato al valore della distanza massima dal centro di proiezione
- Il frame buffer è inizializzato al colore di sfondo; a ogni istante, nel corso della scan conversion, ogni locazione dello z-buffer contiene la distanza, lungo il raggio corrispondente a quella locazione, del punto di intersezione più vicino tra quelli relativi a tutti i poligoni incontrati fino a quel momento

L'algoritmo z-buffer

La computazione procede nel modo seguente

- Si effettua la scan conversion, poligono per poligono
- Per ciascun punto si calcola la distanza dal centro di proiezione e la si confronta con il valore memorizzato nello z-buffer in corrispondenza di quel pixel
- Se la distanza è maggiore della distanza memorizzata nello z-buffer, significa che abbiamo già incontrato un poligono più vicino all'osservatore, e il punto in esame non sarà quindi visibile
- Se invece la distanza è minore di quella presente nello z-buffer, significa che abbiamo individuato un poligono più vicino all'osservatore, e dobbiamo quindi aggiornare la distanza nello z-buffer e memorizzare il colore calcolato per il punto in esame nella locazione corrispondente nel frame buffer

Interazione luce-materia

Interazione luce-materia

Interazione luce-materia

Modello d'Illuminazione

È una formulazione matematica dell'equazione del trasporto dell'energia luminosa (Equazione d'Illuminazione)

- In Computer Graphics, un modello d'illuminazione descrive come un punto di una superficie è illuminato in funzione di:
  - posizione nello spazio
  - posizione delle sorgenti (dirette e indirette)
  - posizione dell'osservatore
  - caratteristiche dei materiali

Interazione luce-materia

Un modello d'illuminazione della radiazione luminosa naturale è fondamentale per il rendering fotorealistico

Modello d'Illuminazione

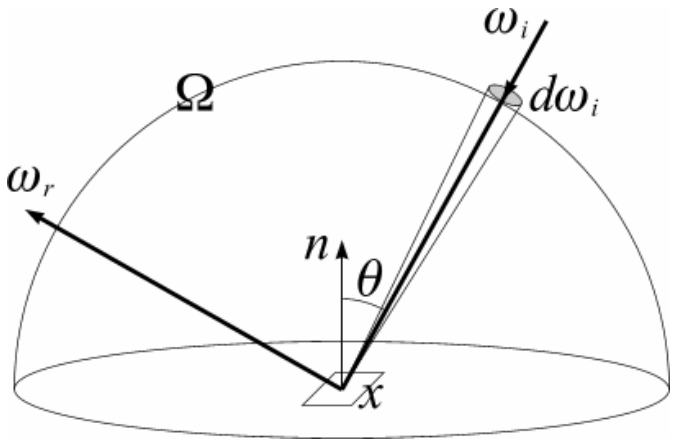
Al fine del calcolo dell'equazione d'Illuminazione, ci si riferisce ai termini:

- lighting per il calcolo della quantità di radiazione luminosa incidente
- shading calcolo del colore risultante

Equazione della Radianza

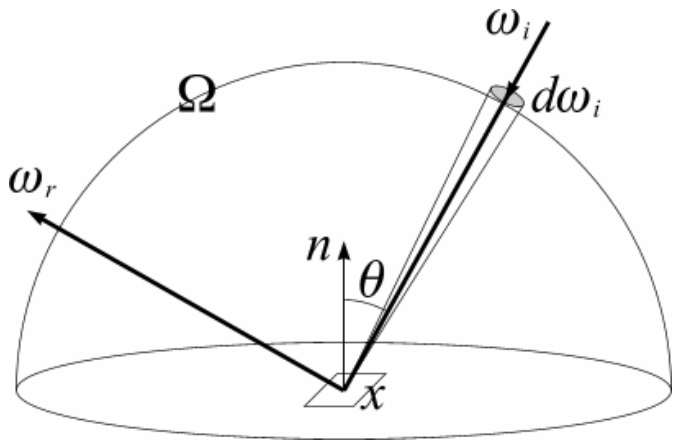
$$L_o(\omega, \overrightarrow{\omega_r}) = L_e(\omega, \overrightarrow{\omega_r}) + L_r(x, \Omega)$$

$$L_o(\omega, \overrightarrow{\omega_r}) = L_e(\omega, \overrightarrow{\omega_r}) + \int_\Omega L_i(x, \overrightarrow{\omega_i})(\overrightarrow{\omega_i} \cdot \overrightarrow{n}) f_r(x, \overrightarrow{\omega_r}, \overrightarrow{\omega_r}) d\overrightarrow{\omega_i}$$



Equazione della Radianza

$$L_o(\omega, \overrightarrow{\omega_r}) = L_e(\omega, \overrightarrow{\omega_r}) + L_r(x, \Omega)$$

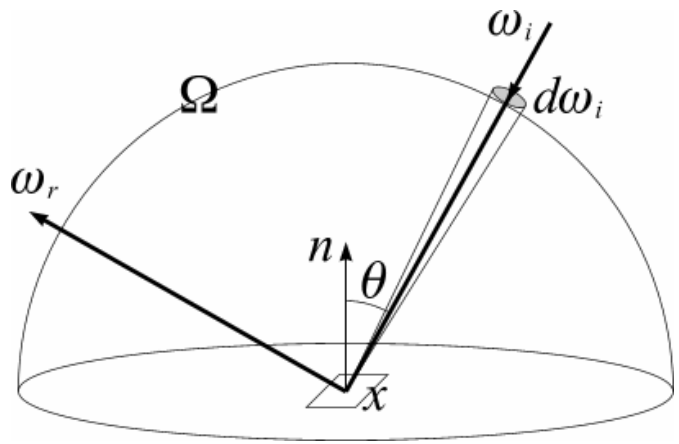


Da una posizione x e direzione $\overrightarrow{\omega_r}$ date, l'ammontare di luce uscente L_o è la somma della luce emessa L_e e della luce riflessa L_r

Equazione della Radianza

L_r è la somma dei raggi incidenti L_i provenienti da tutte le direzioni $\overrightarrow{\omega_i}$ per cosθ per la formula di riflessione della superficie f_r

$$L_r = \int_{\Omega} L_i(x, \overrightarrow{\omega_i})(\overrightarrow{\omega_i} \cdot \overrightarrow{n}) f_r(x, \overrightarrow{\omega_r}, \overrightarrow{\omega_r}) d\overrightarrow{\omega_i}$$

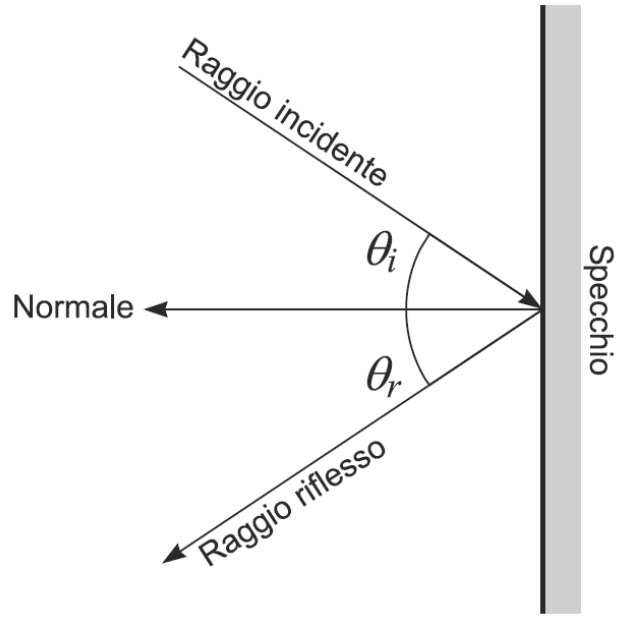


Il modello di Phong

- L'equazione della radianza è troppo onerosa dal punto di vista computazionale per applicazioni interattiva
- OpenGL utilizza il modello semplificato di Phong dove:
  - le sorgenti sono puntiformi
  - non si tiene conto delle inter-riflessioni
  - il calcolo dell'equazione è svolto in maniera locale senza tener conto delle possibili occlusioni che potrebbero causare ombre portate
  - la funzione di riflessione f_r è approssimata con due costanti che permettono di caratterizzare il materiale della superficie riflettente
- Così, l'unico fenomeno fisico modellato è la riflessione (speculare e diffusa)

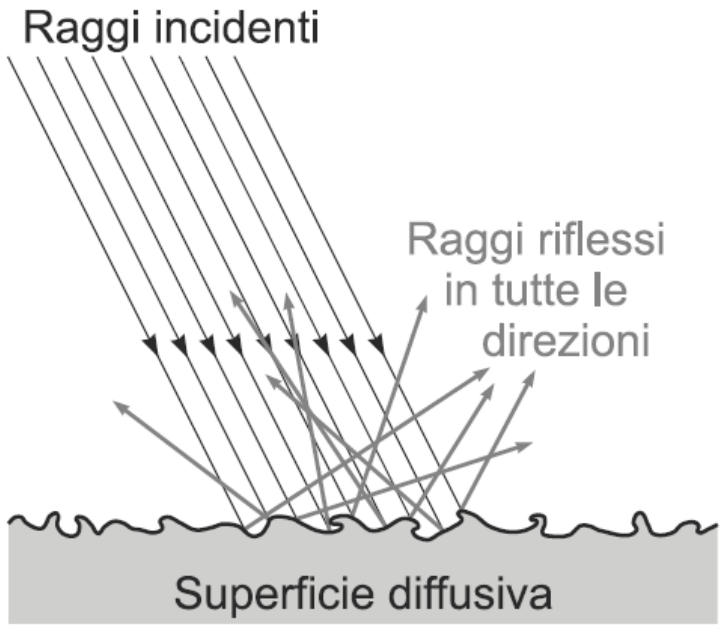
Il modello di Phong: riflessione speculare

- L'angolo che forma il raggio riflesso con la normale della superficie è uguale all'angolo che forma il raggio incidente con la stessa normale ($\theta_i = \theta_r$)
- Così, l'intensità luminosa dipende dalla posizione dell'osservatore



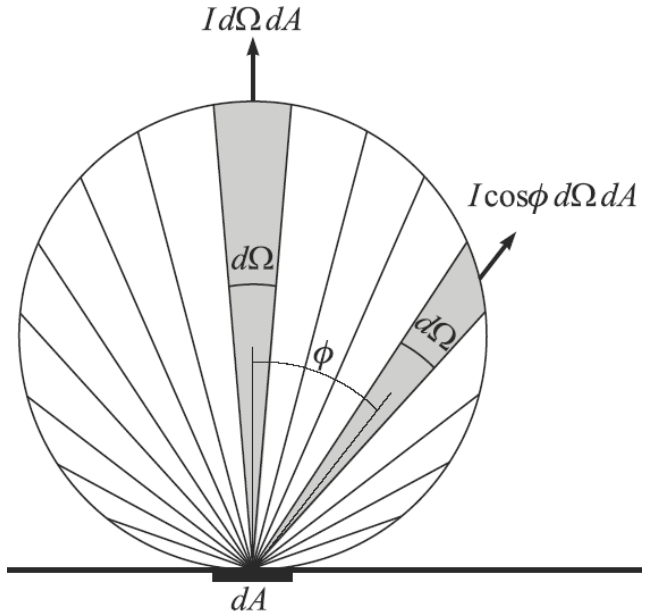
Il modello di Phong: riflessione diffusa

- I raggi vengono riflessi in modo uniforme verso tutte le direzioni
- L'intensità luminosa non dipende dalla posizione dell'osservatore ed è proporzionale al coseno dell'angolo compreso tra la direzione del raggio e la normale alla superficie



Il modello di Phong: riflessione diffusa

- Infatti, la quantità di luce riflessa lungo una data direzione è proporzionale al coseno dell'angolo compreso tra la normale alla superficie e la direzione di osservazione (ϕ in figura)
- Poiché la porzione di superficie visibile è $dA/\cos\phi$, allora la quantità di luce riflessa è costante e vale $Id\Omega dA$



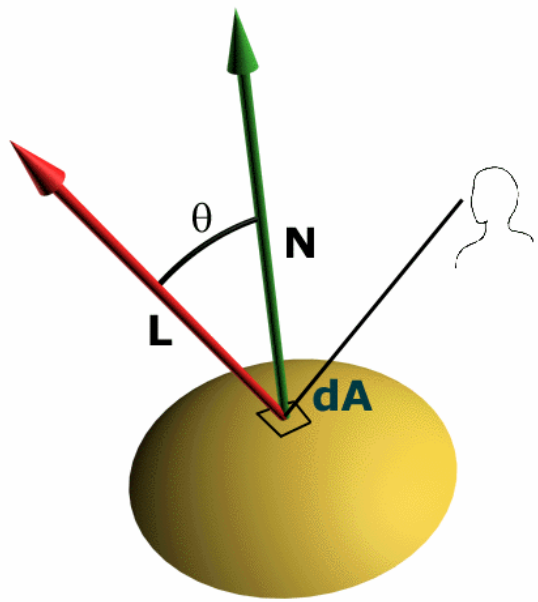
Il modello di Phong: modellizzazione della riflessione diffusa

Equazione della riflessione diffusa nel modello di Phong

$$I_{diff} = I_p k_d \cos\theta = I_p k_d (\vec{N} \cdot \vec{L})$$

Dipende solo da:

- l'orientamento della superficie, $\vec{N}$, cioè la sua normale
- la direzione della luce $\vec{L}$
- la funzione di riflessione della superficie, approssimata con la costante k_d



Il modello di Phong: modellizzazione della riflessione speculare

- Empiricamente, la riflessione speculare si manifesta come una macchia d'intensa luminosità che si sposta sulla superficie dell'oggetto in funzione degli spostamenti dell'osservatore
- Questo comportamento è modellato attraverso un cono assumendo che la quantità di luce riflessa diminuisca esponenzialmente all'allontanarsi dall'asse del cono
- La massima riflessione si ha per $\alpha = 0$ e decade rapidamente secondo la legge

$$(\cos\alpha)^n$$

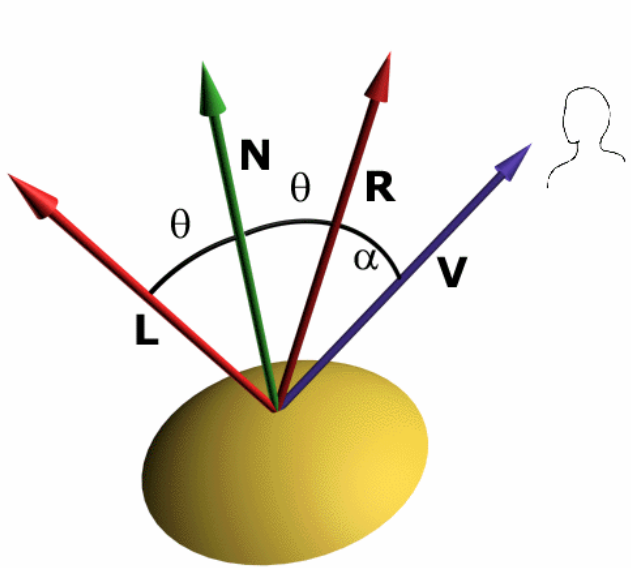
dove $n \in [1, 128]$ è detto *specular reflection exponent*

Il modello di Phong: modellizzazione della riflessione speculare

Equazione della riflessione speculare nel modello di Phong

$$I_{spec} = I_p k_s \cos^n \alpha = I_p k_s (\vec{R} \cdot \vec{V})^n$$

dove k_s è la costante che approssima la funzione di riflessione speculare della superficie



Il modello di Phong: modellizzazione della componente ambiente

Modella le inter-riflessioni di cui non tengono conto i modelli di luce diffusa e speculare

Equazione della luce ambiente nel modello di Phong

$$I_{amb} = I_a k_a$$

dove k_a è una costante caratteristica del materiale

Il modello di Phong: formulazione completa

Modella le inter-riflessioni di cui non tengono conto i modelli di luce diffusa e speculare

Formulazione completa del modello di Phong

$$I = I_a k_a + \sum_p I_p \left(k_d (\vec{N} \cdot \vec{L}) + k_s (\vec{R} \cdot \vec{V})^n \right)$$

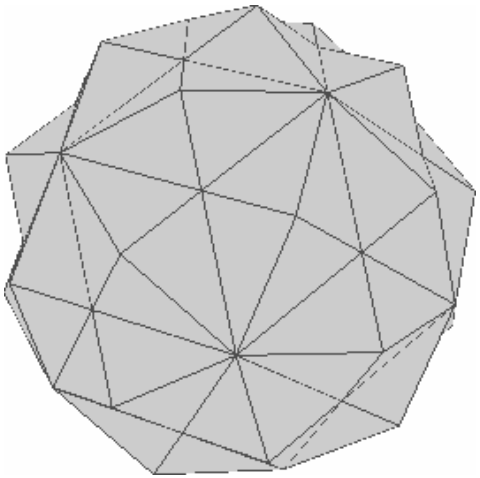
Nella sua formulazione più completa, il modello di Phong prevede anche la possibilità di tener conto dell'attenuazione dell'illuminazione all'aumentare della distanza; in tal caso si introduce un fattore di attenuazione che è inversamente proporzionale alla distanza

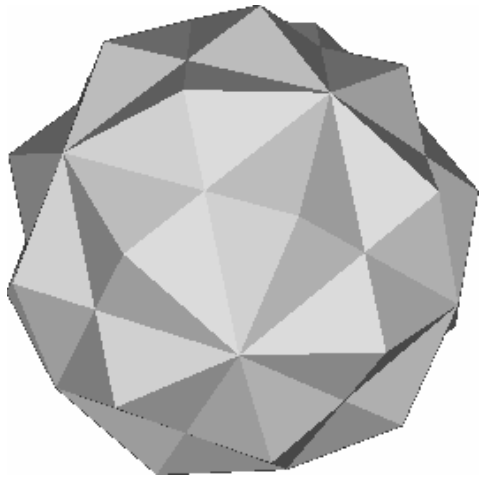
Tecniche di shading

- Il calcolo del colore da attribuire a ogni pixel dell'immagine finale si ottiene attraverso l'applicazione di una tecnica di shading
- La tecnica di shading più precisa consiste nel calcolare l'equazione del modello d'illuminazione per ogni pixel dell'immagine finale
- Nelle applicazioni in real-time, per questioni di efficienza computazionale, è necessario ricorrere a soluzioni approssimate

Shading constante

- Lo shading costante, o flat shading, è il modello più semplice possibile dove il modello d'illuminazione è applicato una solo volta per ogni primitiva geometrica (triangolo o poligono)
- Il valore d'illuminazione calcolato è poi esteso a tutta la superficie della primitiva



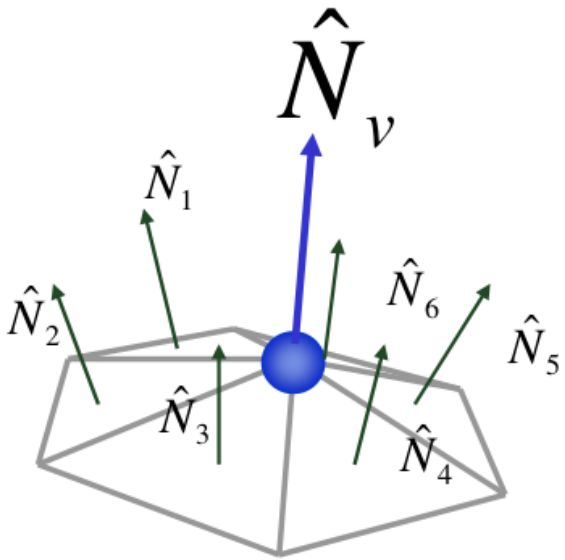


Gouraud shading

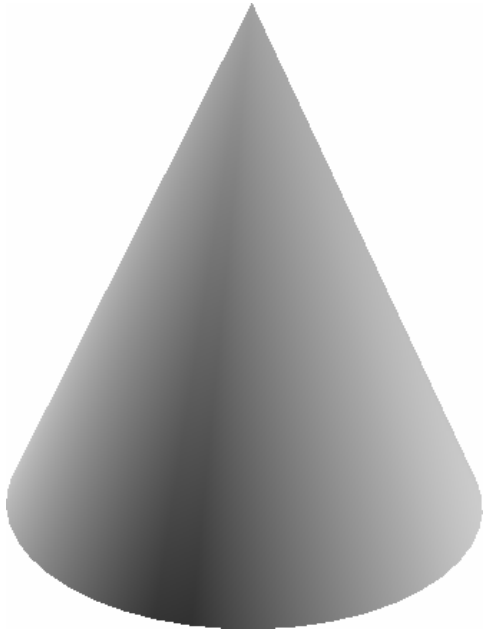
- Il Gouraud shading sfrutta la linearità dello spazio colore RGB: il colore intermedio tra due colori può essere derivato per interpolazione lineare, una per ogni componente R, G e B
- Si calcola quindi l'equazione d'illuminazione per ogni vertice della primitiva geometrica e si ottengono i colori degli spigoli e dei pixel interni per interpolazione
- Inoltre, il Gouraud shading prevede che nel caso in cui la mesh di triangoli rappresenti una superficie curva, non si utilizza la normale al triangolo ma alla superficie

Gouraud shading

- Come normale si può utilizzare la normale alla superficie (se nota) oppure la media delle normali dei triangoli che condividono un dato vertice

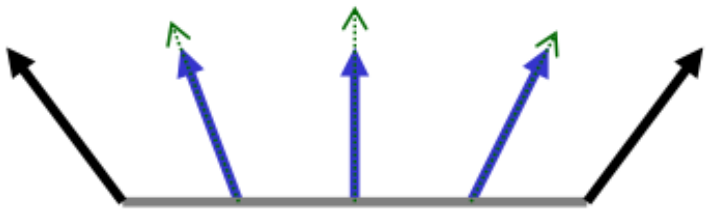


Gouraud shading

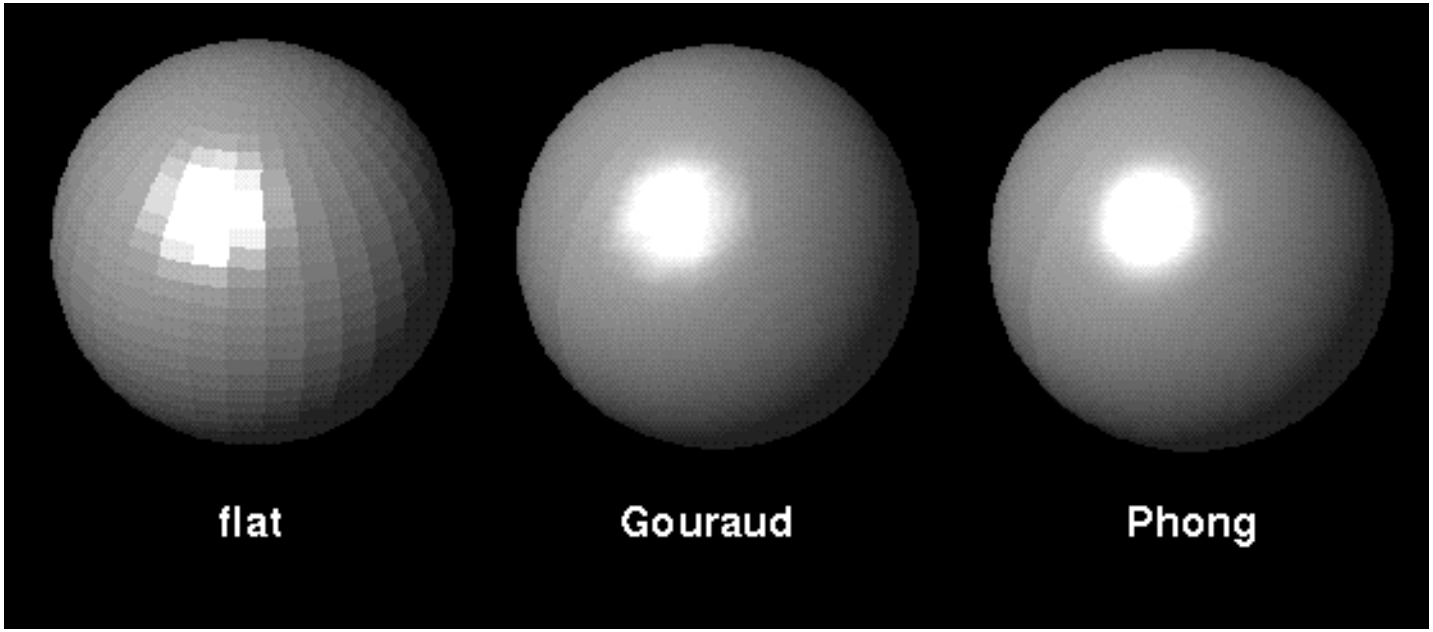


Phong shading

- Il Phong shading garantisce risultati migliori del Gouraud shading nei casi di superfici con alto coefficiente di riflessione speculare
- il Phong shading utilizza un metodo basato sull'interpolazione delle normali
- L'equazione d'illuminazione si calcola in tutti i punti interni al triangolo usando, pixel per pixel, il vettore normale calcolato interpolando linearmente e normalizzando le normali nei vertici



Confronto tra diversi modelli di shading



Rendering Globale

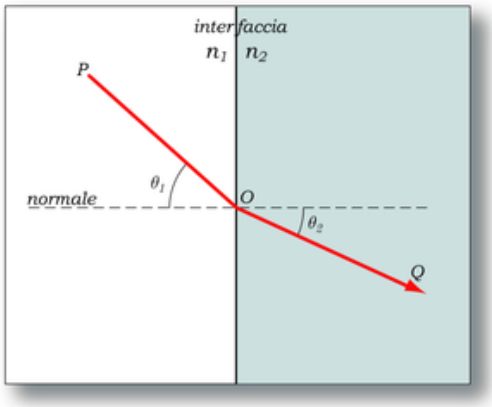
Rendering Globale

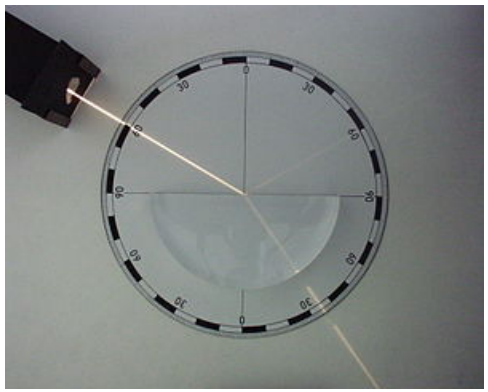
Modelli d'illuminazione globali

- Il modello di Phong è di tipo locale
- I modelli locali forniscono una rappresentazione grossolana del modello fisico
- Non sono infatti in grado di modellare esplicitamente il comportamento di rifrazione (o trasparenza), tutti gli effetti di riflessione e le ombre portate
- Per modellare esplicitamente tali fenomeni è necessario far riferimento a modelli d'illuminazione globali

Rifrazione e semi-trasparenza

- Il comportamento di un materiale semitrasparente può essere modellato definendo il suo coefficiente di Fresnel (un coefficiente del 40% indica che il 40% viene riflesso mentre il 60% è trasmesso all'interno del materiale)
- La legge di Snell modella poi l'angolo tra la direzione d'ingresso e di uscita su una superficie di discontinuità come $n_1 sin\theta_1 = n_2 sin\theta_2$, essendo n_1 ed n_2 gli indici di rifrazione dei due mezzi (es: 1 per l'aria, 1,33 per l'acqua, 1,5 per il vetro)



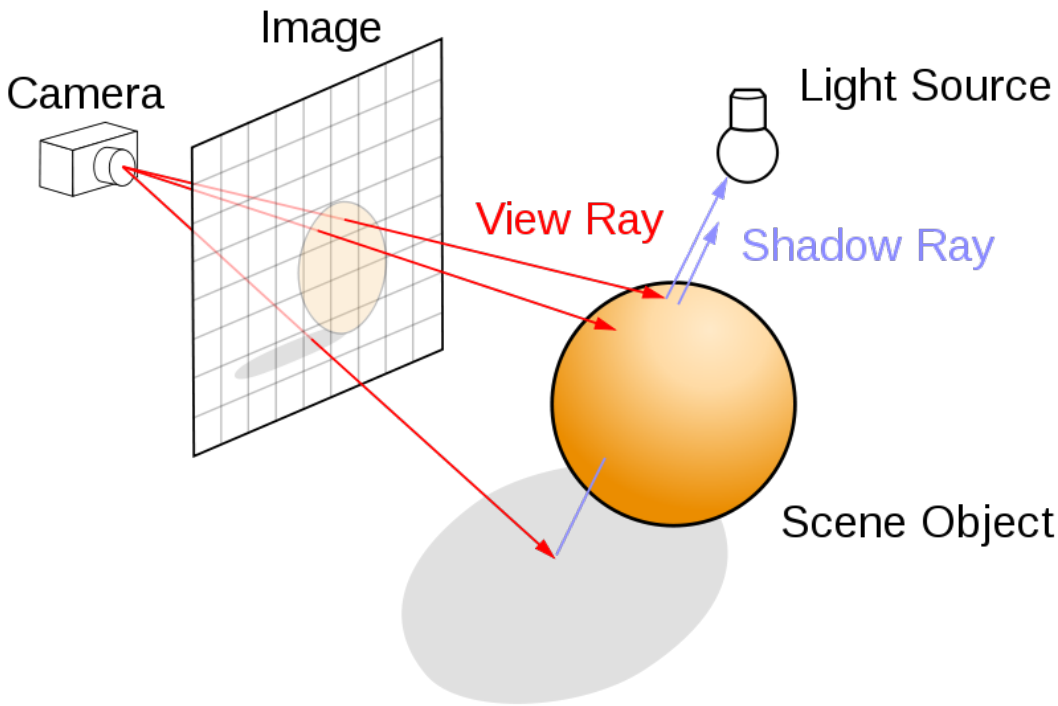


Ray-tracing

- Si basa sull'osservazione che solo un sottoinsieme dei raggi luminosi prodotti dalle sorgenti giunge all'osservatore
- I raggi possono raggiungere l'osservatore anche indirettamente (per riflessione e/o rifrazione)
- L'idea alla base del ray-tracing è seguire la traiettoria, in senso inverso, dei raggi che colpiscono l'osservatore, dall'osservatore verso la sorgente

Ray-tracing

- Data la posizione dell'osservatore, a ogni pixel del piano immagine corrisponderà un raggio che possiamo seguire a ritroso dall'osservatore alla scena, detto raggio primario o view ray

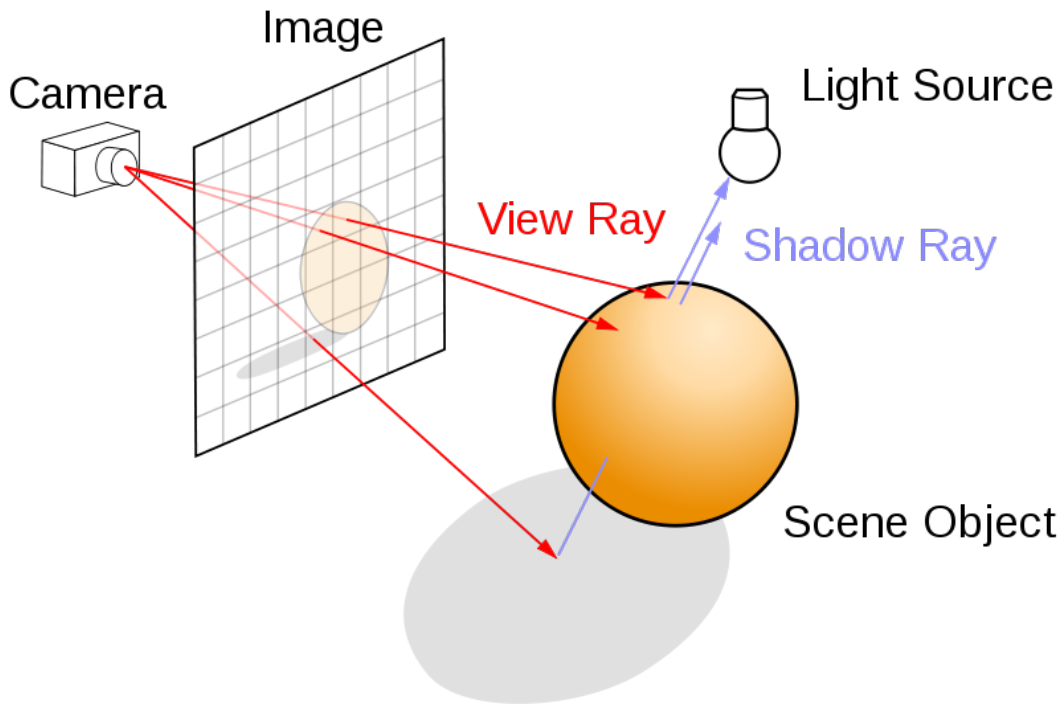


Ray-tracing

- Ciascun raggio primario può:
  1. perdersi all'infinito
  2. intersecare un oggetto presente nella scena
- Nel primo caso, ai pixel corrispondenti ai raggi andati a vuoto verrà assegnato un colore di sfondo
- In caso contrario si stabilisce quale sia l'intersezione del raggio primario con l'oggetto più vicino nella scena e si valuta se quel punto si trovi in una zona d'ombra

Ray-tracing

- Per valutare se un punto è in una zona d'ombra, si traccia un raggio d'ombra verso ogni sorgente nella scena

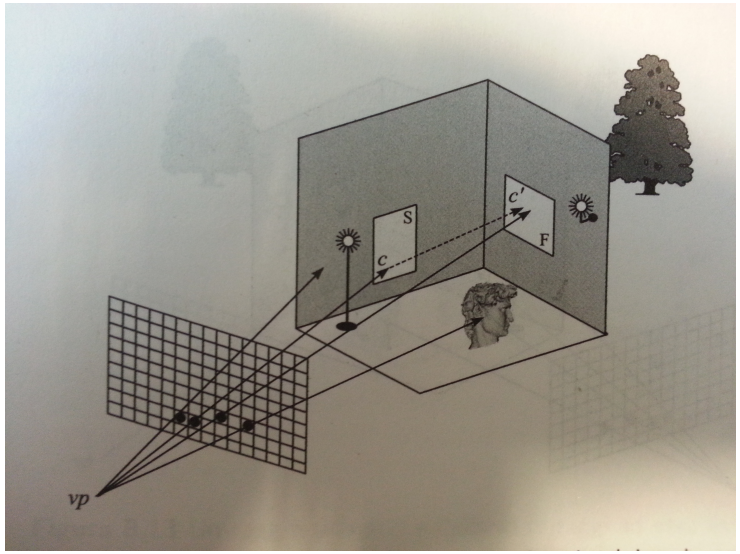


Ray-tracing

- Se il raggio ombra interseca una superficie terza, allora il pixel colpito dal raggio primario ricade in una zona d'ombra
- Così, nel calcolo dello shading si tiene conto solo dei *contributi diretti* delle sorgenti
- Il tracciamento dei raggi primari e dei raggi ombra permette quindi di modellare le ombre portate

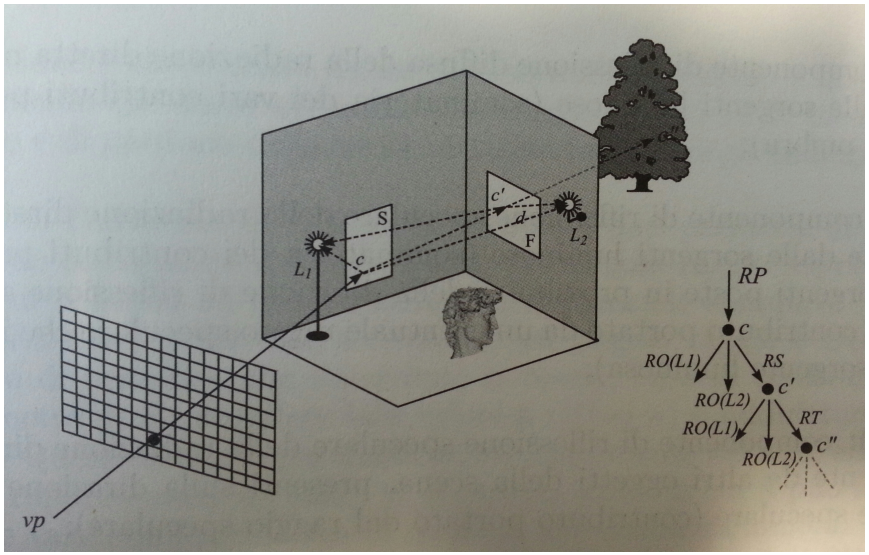
Ray-tracing

- Se un raggio primario intercetta un punto c di una superficie S caratterizzata da riflessività speculare, è necessario tracciare un altro raggio, detto raggio di riflessione speculare, che segue appunto la direzione di riflessione speculare
- Se il raggio speculare colpisce un altro oggetto nella scena in un punto c', allora il colore in c' contribuirà a definire il colore in c



Ray-tracing

- Se un raggio primario intercetta un punto c' di una superficie F caratterizzata da un comportamento rifrattivo o semi-trasparente, è necessario tracciare un altro raggio, detto raggio di di trasparenza, che segue appunto la direzione di rifrazione
- Come nel caso precedente, se il raggio di di trasparenza colpisce un altro oggetto nella scena in un punto c'', allora il colore in c'' contribuirà a definire il colore in c'

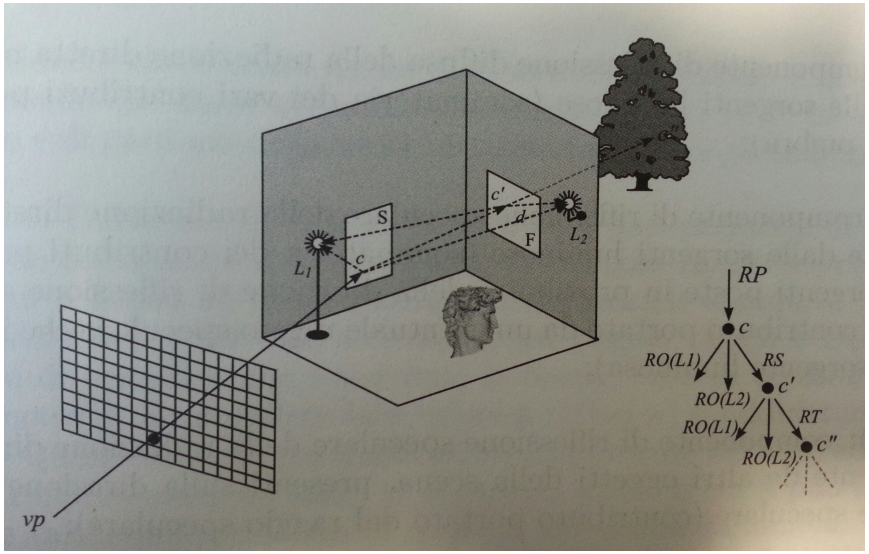


Ray-tracing

- Dunque, per ogni intersezione dovuta a un raggio primario, il ray-tracing genera una serie di raggi secondari:
  - ombra
  - speculari
  - trasparenza
- Gli ultimi due tipi di raggi, speculare e trasparenza, individuano quale sia la superficie che interagisce con il punto intercettato dal raggio primario per calcolarne il colore

Ray-tracing

- Per calcolare il colore nel punto intercettato dal raggio primario, si istanzia ricorsivamente il processo ray-tracing
- Così, per ogni raggio primario, il ray-tracing genera un albero di raggi, la cui profondità dipende dalla complessità della scena e dall'accuratezza richiesta



Ray-tracing

I criteri di terminazione del processo di ricorsione sono:

- profondità massima di ricorsione: l'osservatore non è in grado di percepire la mancanza di riflessioni speculari superiori a tre (a eccezione di scene speciali con specchi contrapposti)
- soglia minima di contributo: non si tracciano raggi il cui contributo sia inferiore a una soglia prestabilita; per valutare l'entità del contributo, si può stimare sommando la percentuale di trasparenza o riflessione dei raggi calcolati precedentemente

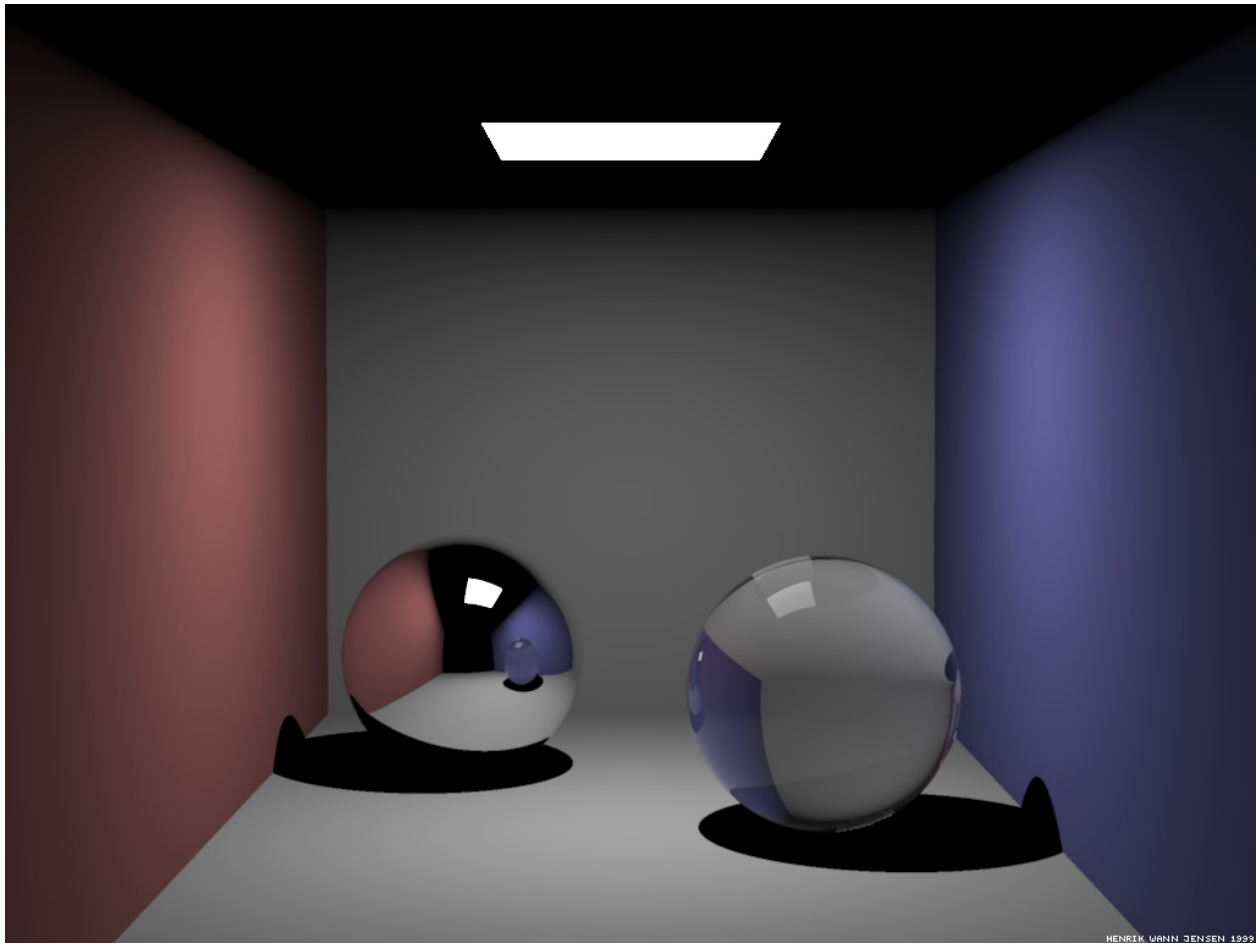
Ray-tracing

Concluso il processo di ricorsione, è possibile calcolare l'equazione d'illuminazione:

$$I(\lambda) = k_a I_a(\lambda) + k_{dr} I_{dr}(\lambda) + k_{sr_l} I_{sr_l}(\lambda) + k_{sr_{obj}} I_{sr_{obj}}(\lambda) + k_{st} I_{st}(\lambda)$$

- I_a: componente ambiente, che permette di approssimare gli effetti di riflessione diffusiva non considerati dal ray-tracing
- I_{dr}: componente di riflessione della radiazione diretta proveniente dalle sorgenti
- I_{sr_l}: componente di riflessione speculare della radiazione diretta proveniente dalle sorgenti
- $I_{sr_{obj}}$: componente di riflessione speculare della radiazione diretta proveniente da altri oggetti della scena
- I_{st}: componente di trasmissione, accumulato sui vari raggi trasparenza
- k_a, k_{dr}, k_{sr_l}, $k_{sr_{obj}}$, k_{st}: coefficienti dei materiali

Ray-tracing: esempio



Ray-tracing: esempio

Ray-tracing: esempio

Ray-tracing: esempio

Radiosity

- A parte l'elevato costo computazionale, dovuto al calcolo delle intersezioni raggio-oggetto, il ray-tracing ha come limite fondamentale il fatto che modella esclusivamente gli effetti della *luce diretta*
- Questo porta a risultati innaturali (illuminazione e ombre troppo nette) poiché nella realtà la radiazione luminosa che raggiunge una superficie è il risultato della somma delle componenti dirette e indirette (fotoni riflessi da altre superfici presenti nella scena)
- Il metodo radiosity permette di ottenere un'illuminazione morbida valutando (esclusivamente) la componete diffusa
- Gli eventuali effetti di rifrazione e riflessione si ottengono applicando successivamente il metodo ray-tracing

Radiosity

- Nel metodo radiosity ogni porzione di superficie nella scena è essa stessa un illuminante
- Se nella scena sono presenti n entità $p_1, p_2, ..., p_n$ (es. gli n triangoli che tassellano gli oggetti nella scena), valutare la radiosity significa stimare, per ogni p_i:
  - le k componenti che riflettono radiazione luminosa su p_i
  - l'intensità eventualmente riflessa
- Definiti i fattori di riflessione r_i per gli elementi che compongono la scena, il primo passo dell'algoritmo radiosity è individuare i così detti fattori di forma che legano le varie coppie di elementi nella scena

Radiosity

Il fattore di forma relativo a due elementi (patch) p_i p_j è definito come:

$$f_{ij} = \frac{cos\Theta_i cos\Theta_j}{\pi d^2} da_j h_{ij}$$

dove:

- da_j è l'area della patch p_j
- d è la distanza tra le patch
- Θ_i e Θ_j sono gli angoli tra le normali alle patch e il segmento che le congiunge
- h_{ij} è l'indice di visibilità che vale 1 se le patch sono visibili, zero altrimenti

Radiosity

Il problema si riduce al calcolo della radiosity, b_i per ogni patch nella scena. Detta e_i la radiosità emessa della patch i, vale l'equazione (radiosity equation):

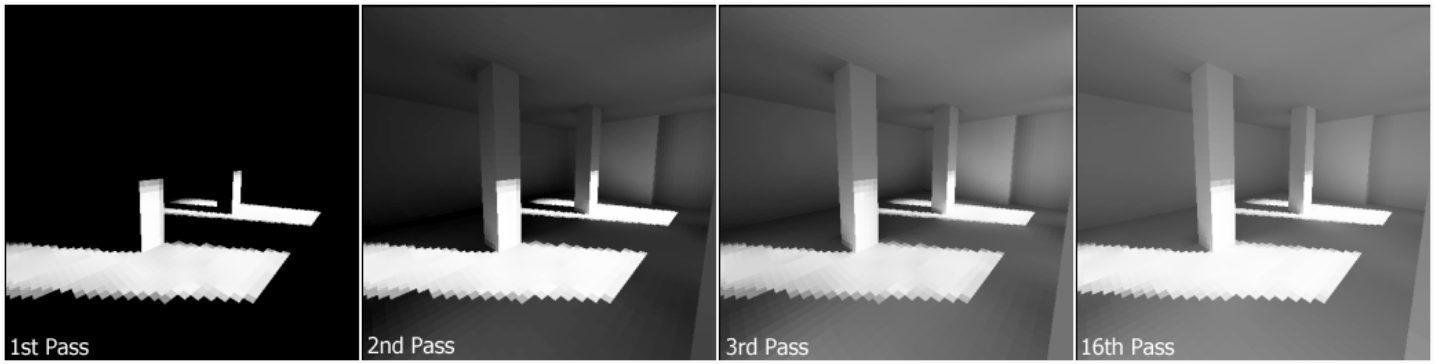
$$b_i = e_i + r_i \sum_{j=1}^{n} b_j f_{ij}$$

il che dà luogo a un sistema lineare nelle n incognite b_1, ..., b_n ():

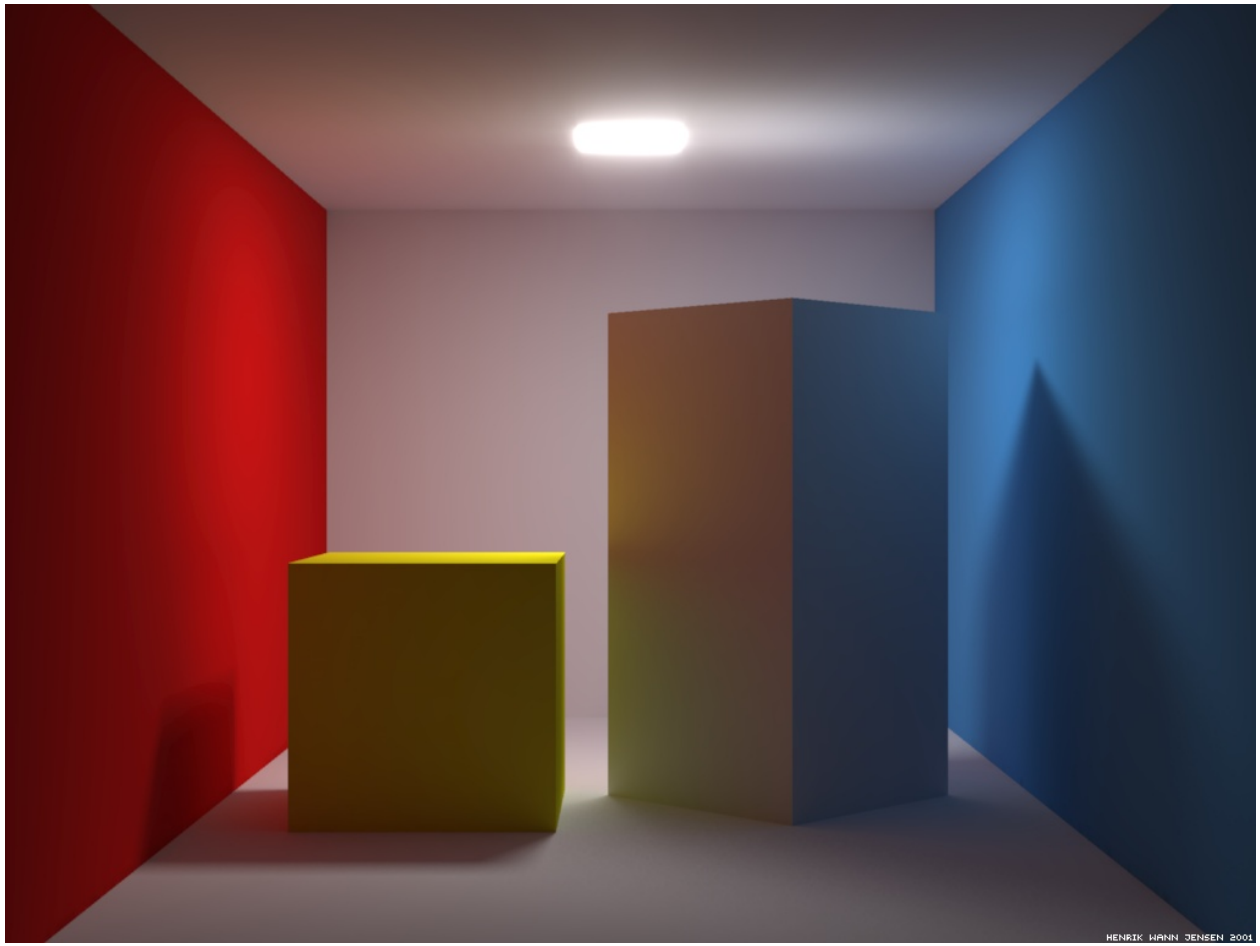
$$\begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{pmatrix} = \begin{pmatrix} 1 - r_1 f_{11} & -r_1 f_{12} & \dots & -r_1 f_{1n} \\ -r_2 f_{21} & 1 - r_2 f_{22} & \dots & -r_2 f_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -r_n f_{n1} & -r_n f_{n2} & \dots & 1 - r_n f_{nn} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Radiosity

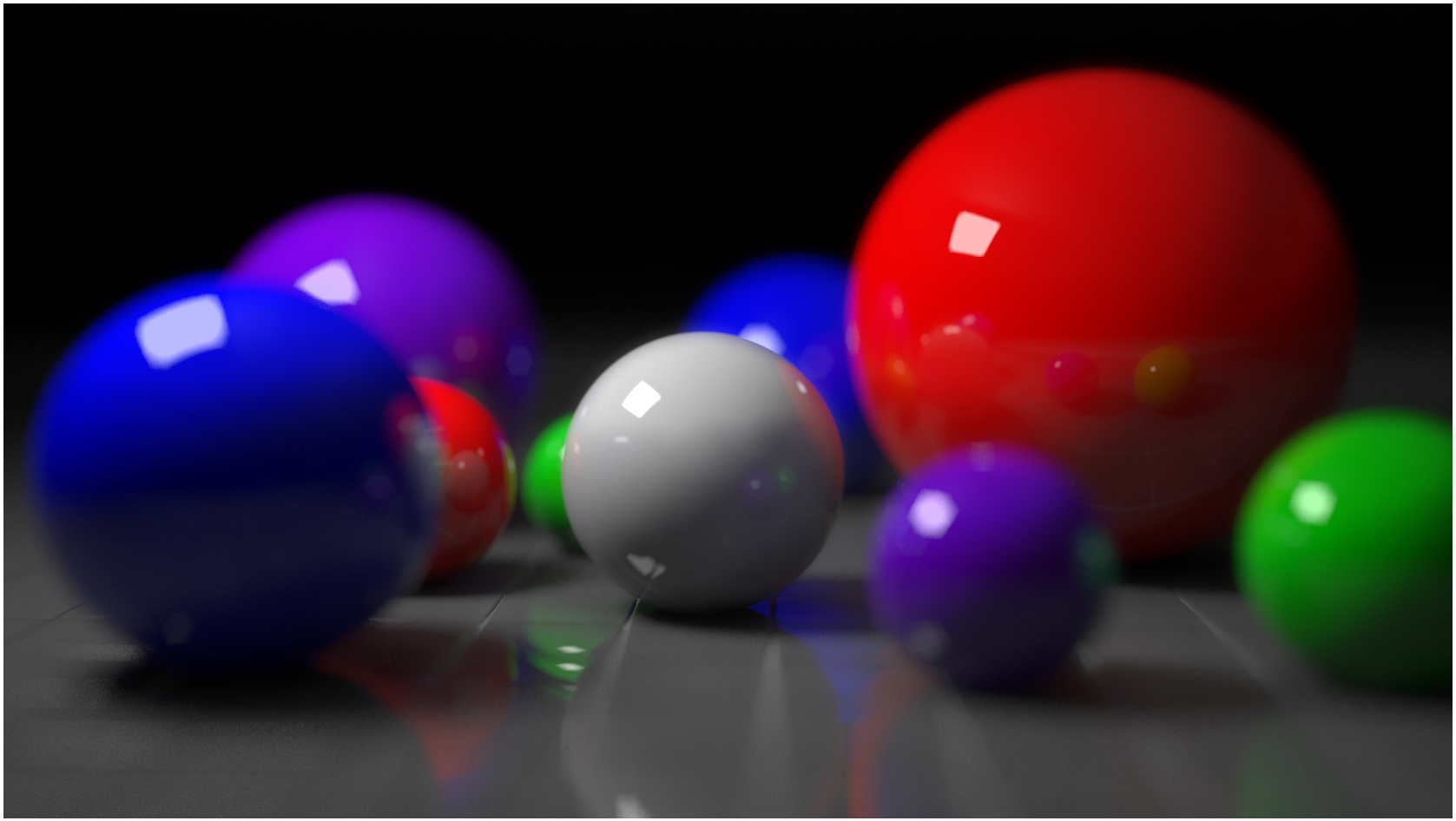
- Una volta valutati i fattori di forma, l'algoritmo considera per prime le patch che rappresentano le sorgenti luminose
- La luce emessa da queste patch è quindi trasmessa per riflessioni successive in funzione dei rispettivi fattori di forma
- Il processo viene iterato fino al raggiungimento di una condizione di equilibrio che si valuta confrontando i valori di radianza delle varie patch in due passi consecutivi dell'algoritmo



Radiosity: esempio



Radiosity: esempio



Photon tracing

- Un metodo alternativo all'algoritmo radiosity per il calcolo della componente diffusa è costituito dall'algoritmo photon tracing
- Il photon tracing è un metodo particellare che calcola esaustivamente come i fotoni emessi dalle sorgenti luminose presenti nella scena si distribuiscano colpendo le superfici ed essendo da queste riflesse
- Il tracciamento della particella termina dopo un numero prestabilito di iterazioni oppure quando la percentuale di assorbimento supera una soglia prestabilita

Photon tracing: esempio