



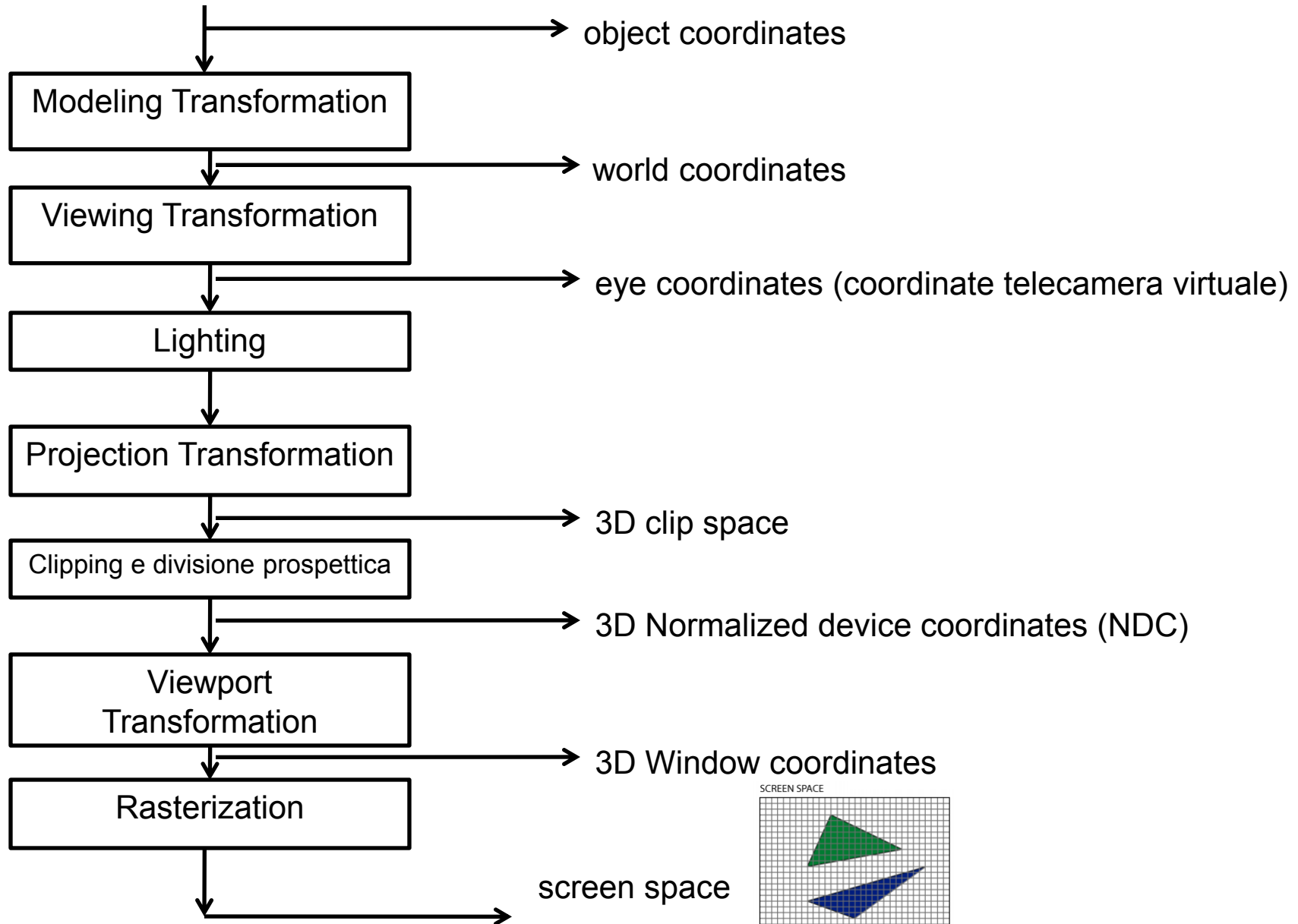
Introduzione a:

Pipeline grafica programmabile

Shader grafici

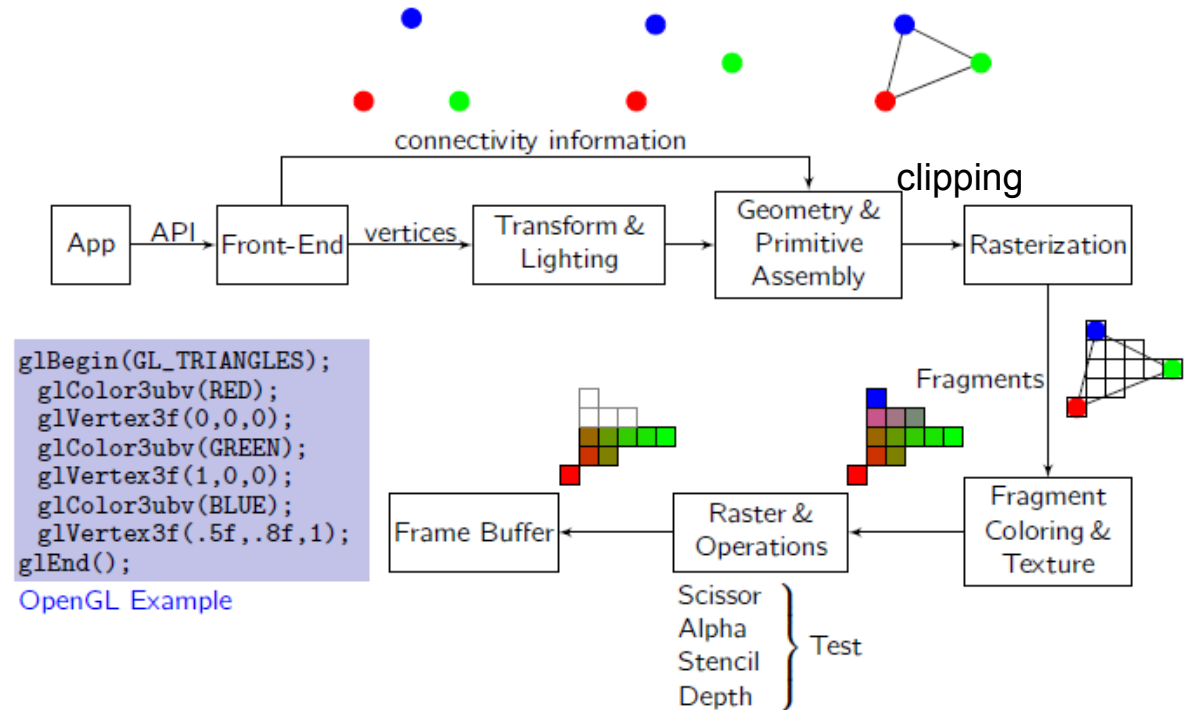
Linguaggio GLSL

Pipeline grafica fissa



Pipeline grafica fissa

Un modello più realistico

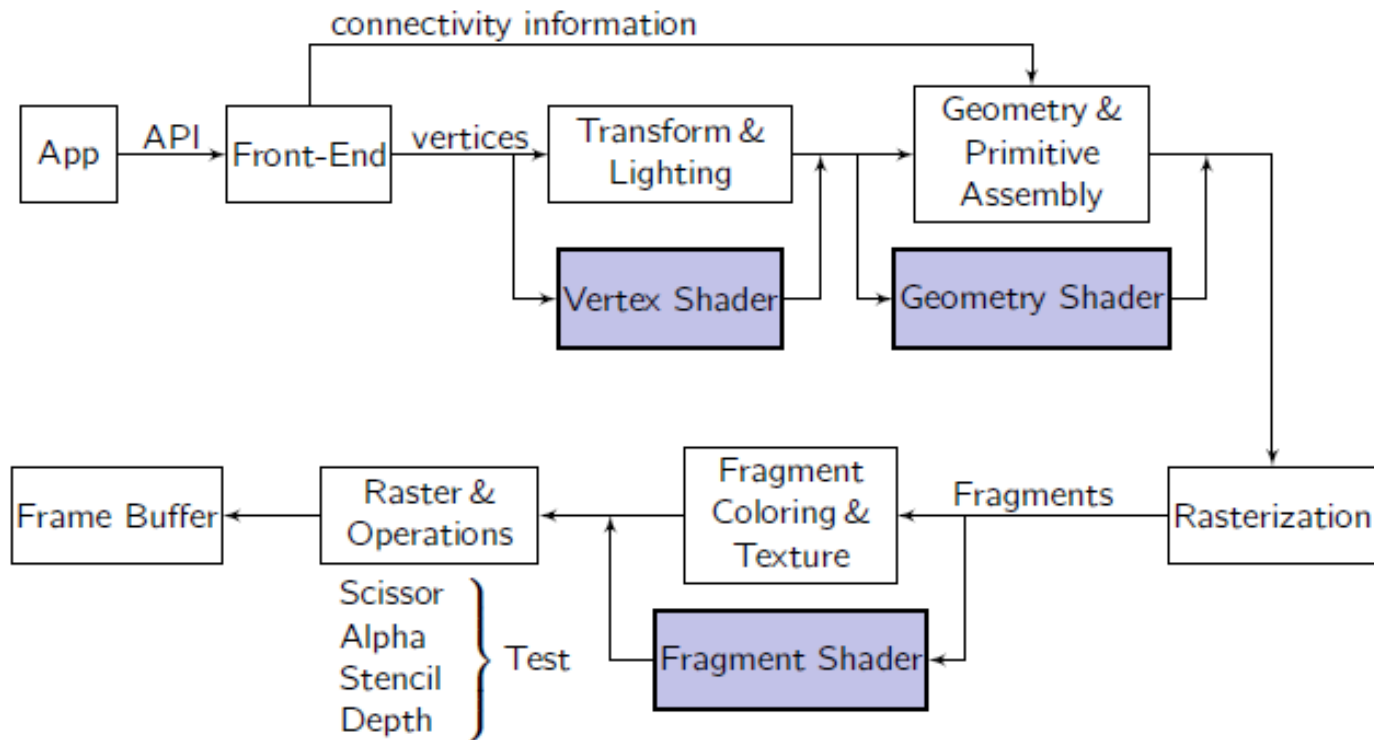


- Le trasformazioni sui vertici sono eseguite senza conoscere le informazioni di connessione, cioè le primitive a cui quei vertici appartengono.
- Le informazioni di connessione sono utilizzate successivamente, nella fase di assemblaggio.
- Il colore dei vertici, le loro normali e l'illuminazione sono calcolati all'inizio della pipeline.
- L'effettiva colorazione delle primitive avviene alla fine della pipeline, per interpolazione dei valori ai vertici delle primitive.

Pipeline grafica fissa

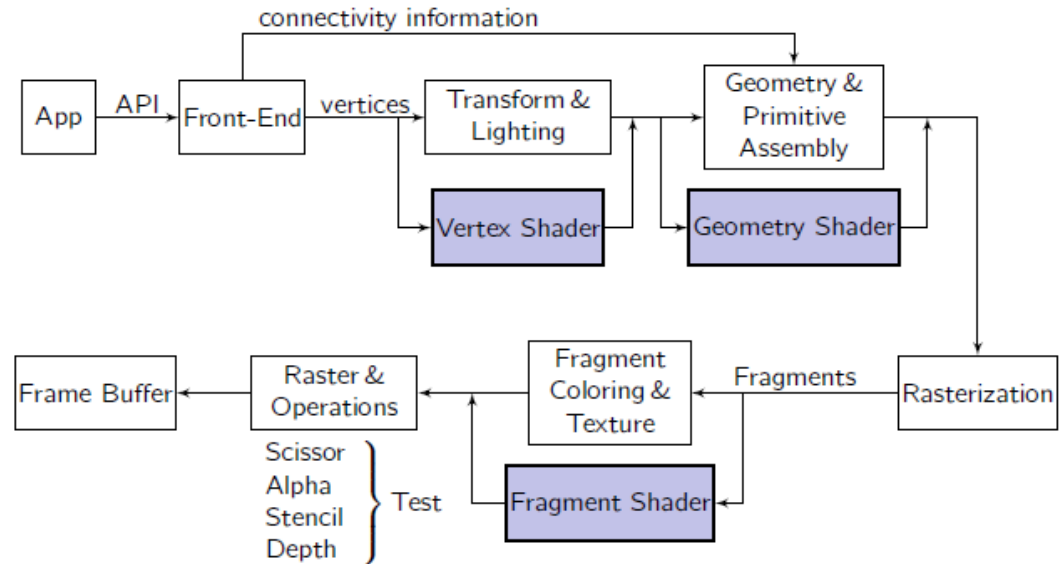
- Il modello tradizionale della pipeline grafica (fissa) impone delle limitazioni al programmatore (es: OpenGL 1.0)
- La pipeline di rendering fissa è composta da una sequenza di stadi che possono essere configurati ma non riprogrammati
- Gli algoritmi che vengono eseguiti nei vari stadi non possono essere modificati dal programmatore (es: modello di illuminazione)
- Il modello basato sulla pipeline programmabile è stato introdotto per superare queste limitazioni (OpenGL 2.0 e superiori)
- La pipeline grafica programmabile consente al programmatore di modificare il contenuto di alcuni dei più importanti stadi di elaborazione (svantaggio: **MAGGIORE COMPLESSITA': occorre imparare un nuovo linguaggio!**)

Schema semplificato



Pipeline grafica programmabile

Schema semplificato

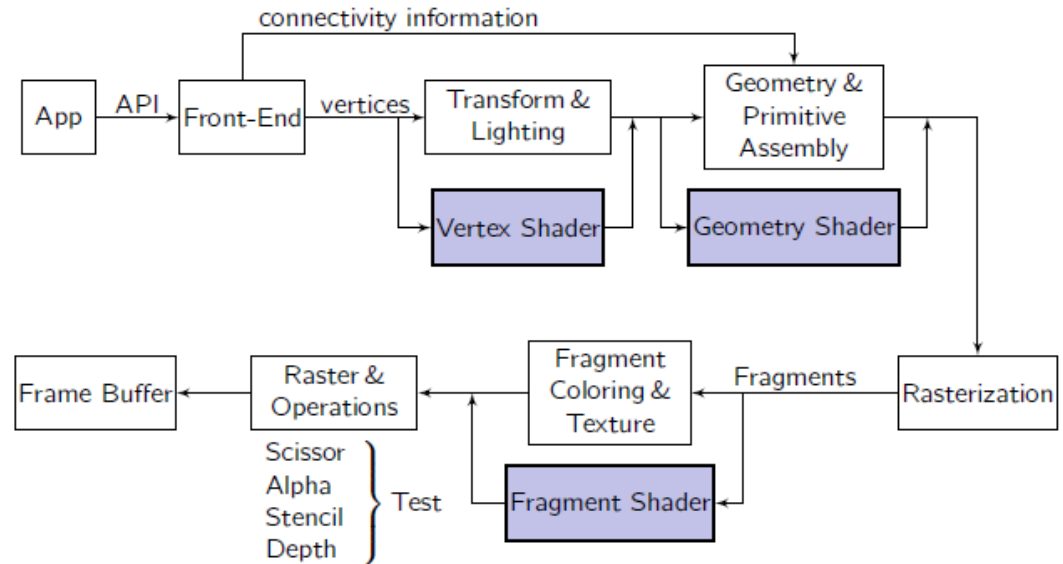


Per capire la pipeline programmabile occorre definire in modo preciso i dati su cui agisce:

- **vertice (vertex):** i vertici sono i punti che definiscono le primitive grafiche
- **frammento (fragment):** elemento generato dal rasterizzatore che potenzialmente potrebbe diventare un pixel. Ad un fragment sono associati una serie di valori necessari per generare il pixel: la locazione nel framebuffer, il colore, la profondità (Z), e altri valori. Spesso un fragment non diventa un pixel, poichè, ad esempio, fallisce il depth-test e pertanto viene filtrato.
- **pixel:** elemento visivo più piccolo del Frame Buffer (ovvero dello schermo)

Pipeline grafica programmabile

Schema semplificato

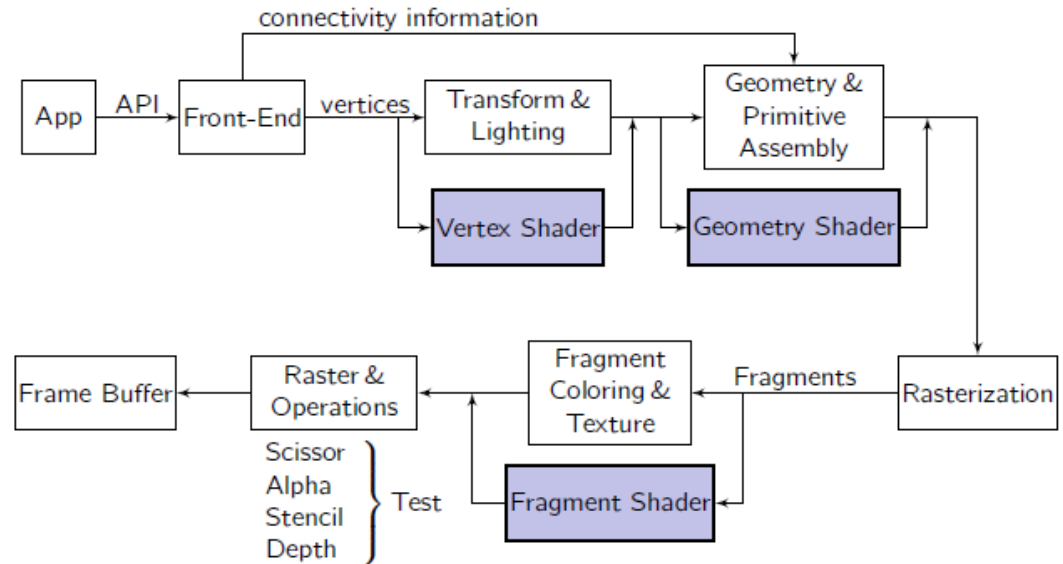


Vengono introdotti tre stadi “programmabili” che se utilizzati sostituiscono la pipeline fissa:

- **vertex shader:** operazioni sui vertici
- **geometry shader:** operazioni sulle geometrie (introdotto in OpenGL 2.1)
- **fragment shader:** operazioni sui frammenti

Pipeline grafica programmabile

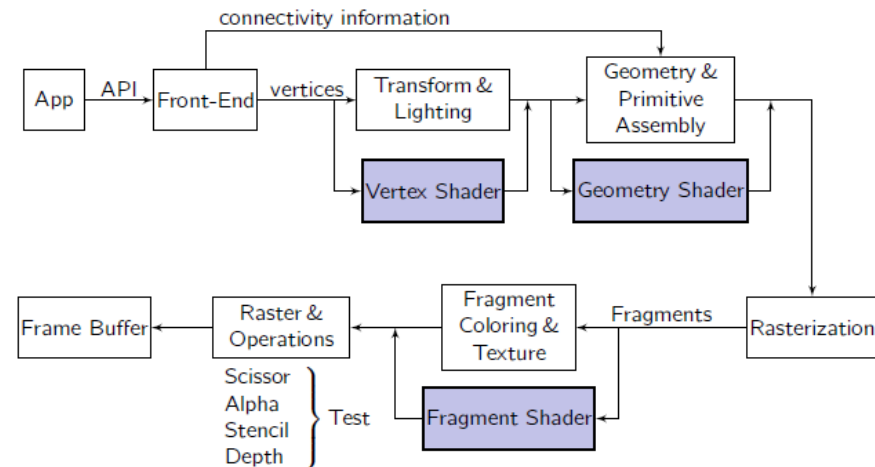
Schema semplificato



- Uno **shader** è un programma che viene eseguito sulle unità di elaborazione (shader processors) della GPU in parallelo ad altri shader.
- Uno shader è inizializzato ed eseguito una volta per ogni elemento (o gruppo di elementi) presente nel buffer di origine, e può in genere produrre un numero variabile (anche nessuno) di elementi in uscita.
- I vantaggi principali dell'uso degli shader sono l'**efficienza** e la **flessibilità**.
- Con gli shader programmabili è possibile realizzare effetti che non sarebbe possibile ottenere altrimenti. Questo accade perché gli shader possono sfruttare informazioni parzialmente elaborate a cui normalmente non si ha accesso.

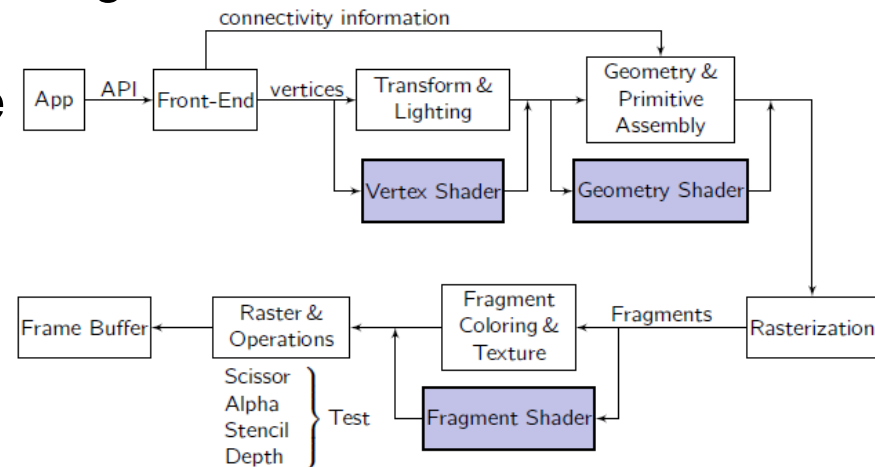
Vertex shader

- Un vertex shader sostituisce le prime operazioni della pipeline grafica fissa.
 - *Trasformazioni di modello, di vista e di proiezione dei vertici;*
 - *Illuminazione e applicazione del colore ai vertici;*
 - *Trasformazione delle normali;*
 - *Generazione delle coordinate delle texture.*
- Un vertex shader viene istanziato una volta per ogni singolo vertice in ingresso, definito dall'applicazione, e non ha alcuna informazione di connessione tra i vertici adiacenti.
- Il compito dello shader è quello di generare in output il vertice trasformato ed i relativi attributi (normale, colore).
- L'illuminazione, se calcolata per vertice come avviene nella pipeline fissa, deve essere calcolata dallo shader.



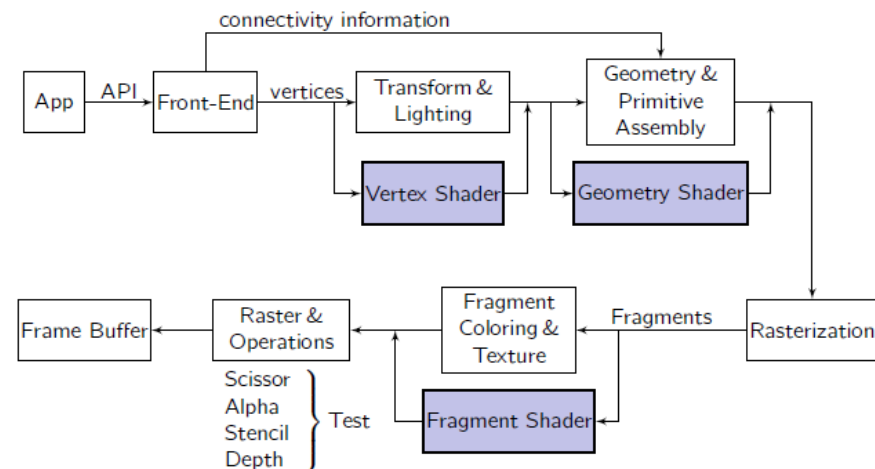
Geometry shader

- Un'istanza di un geometry shader si colloca dopo l'assemblaggio delle primitive. Riceve in ingresso una primitiva di un certo tipo (punto, linea, triangolo) e i suoi attributi e produce in uscita un qualunque numero di primitive tutte di uno stesso tipo, che possono essere di tipo diverso di quelle ricevute in ingresso.
- Può generare nuovi vertici per definire una nuova primitiva.
- Più istanze non possono mai collaborare a produrre la stessa primitiva.
- Non sostituisce nessuna parte della pipeline grafica fissa.
- Le primitive emesse sono poi processate come tutte le altre primitive specificate dall'applicazione OpenGL.



Fragment shader

- Un fragment shader riceve in input un fragment generato dal rasterizzatore e i suoi attributi quali colori, profondità etc.
- Il fragment shader esegue le ultime operazioni sui frammenti prima che essi diventino pixel. Viene eseguito indipendentemente dal fatto che il suo frammento sia poi visualizzato oppure no.
- Ha il compito di calcolare/modificare il colore di un singolo frammento ed eventualmente anche la sua profondità.
- Sostituisce le funzioni della pipeline fissa che agiscono sulla colorazione per singolo pixel (inclusi nebbia e texture).
Se queste sono necessarie, devono essere riscritte.

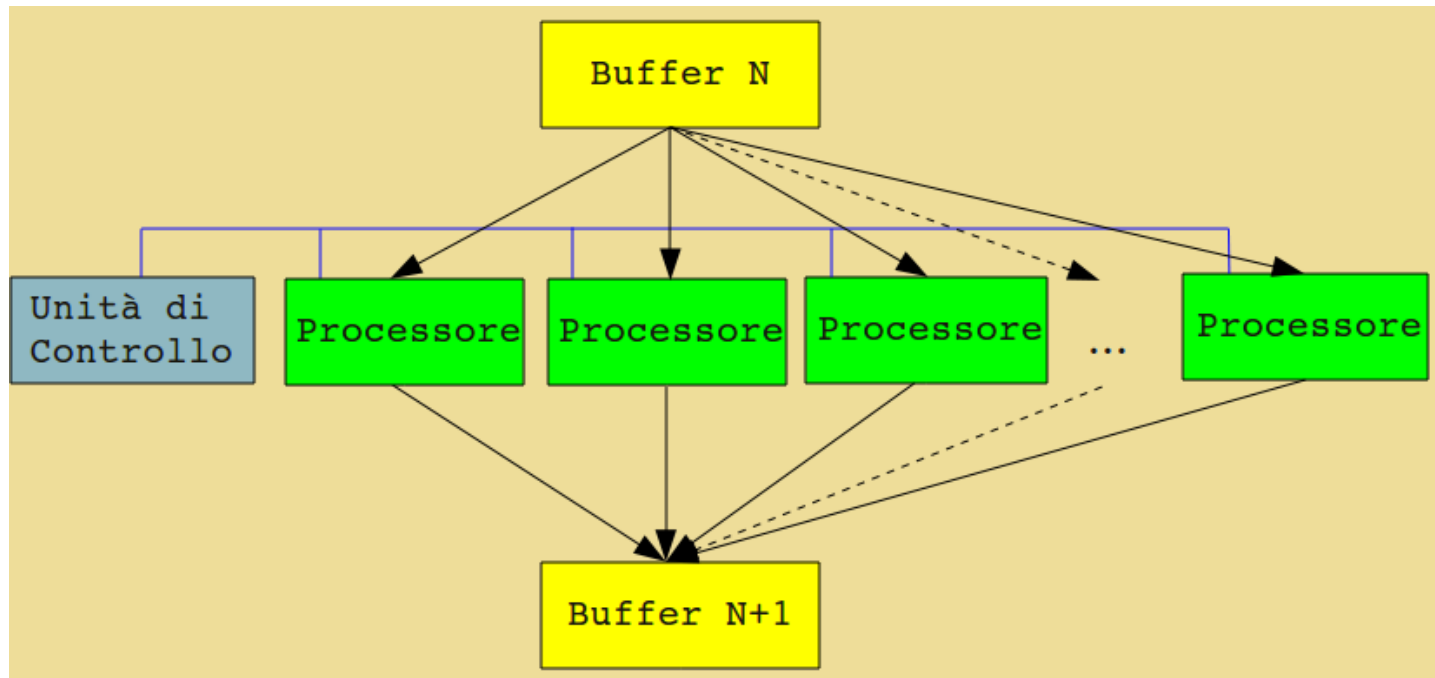


Programmazione di shader grafici

- La programmazione degli shader grafici può avvenire
 - con linguaggi di “basso livello”
 - con linguaggi di “alto livello”
- I linguaggi di “basso livello” sono simili all'assembler (poco pratici e dipendenti dall'hardware)
- I linguaggi di “alto livello” sono simili al “C”. **Cg** (Computer Graphics, Nvidia), **HLSL** (High Level Shading Language, DirectX Microsoft), **GLSL** (OpenGL Shading Language)
- Il codice degli shaders viene solitamente richiamato nel contesto di una applicazione grafica OpenGL che fornisce delle funzioni che permettono di “caricare” (compilare e linkare) il codice degli shaders sulla GPU a tempo di esecuzione (runtime).

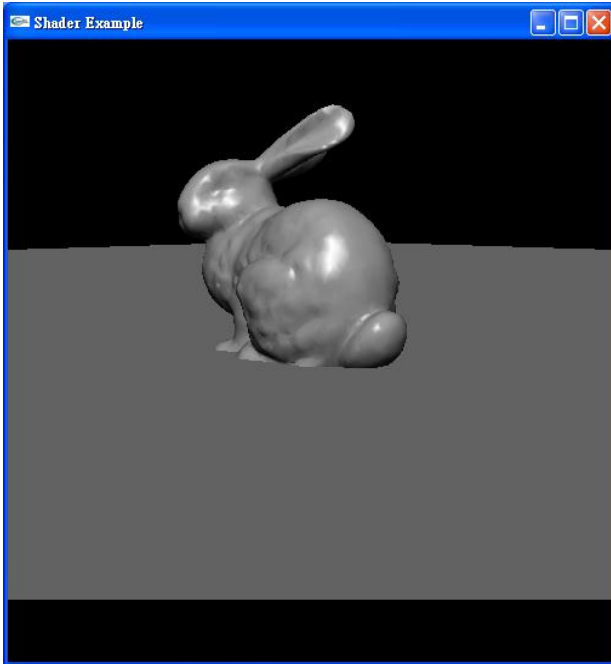
Parallelismo

- Gli shader, quando installati, sostituiscono gli stadi corrispondenti della pipeline fissa. L'elaborazione mediante pipeline fissa viene ripristinata quando gli shader vengono rimossi.
- Gli shader di uno stesso tipo vengono eseguiti in parallelo sulla GPU su dati diversi.

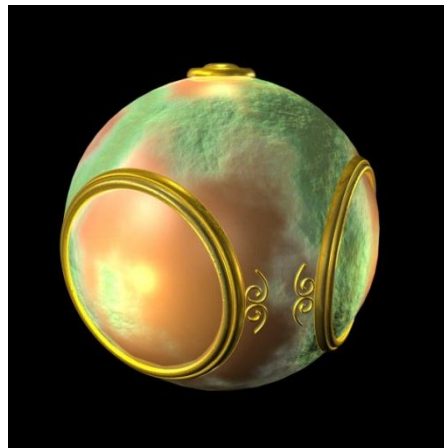
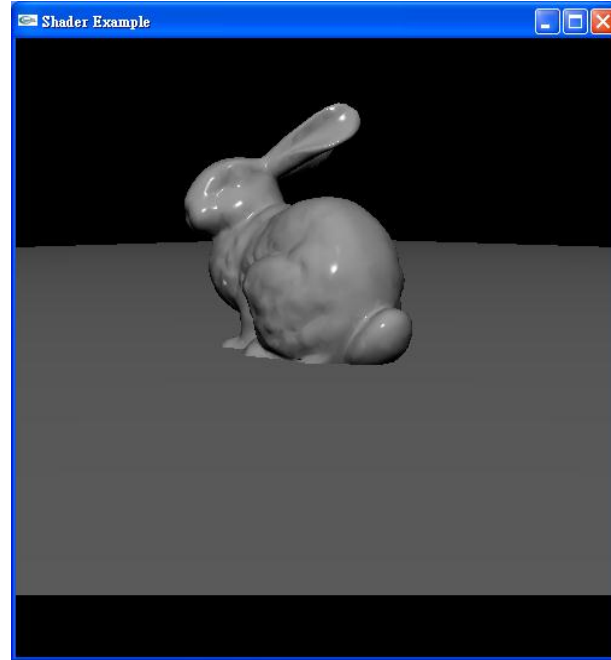


Applicazioni

OpenGL
Gouraud
Shading
(pipeline
fissa)

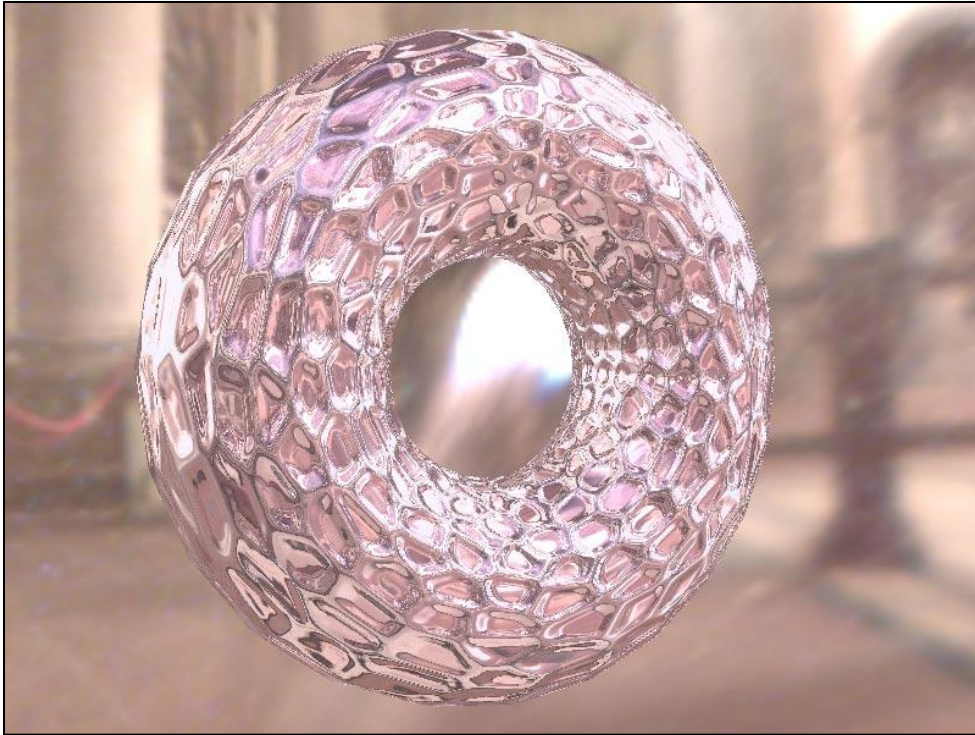


GLSL Phong
Shading

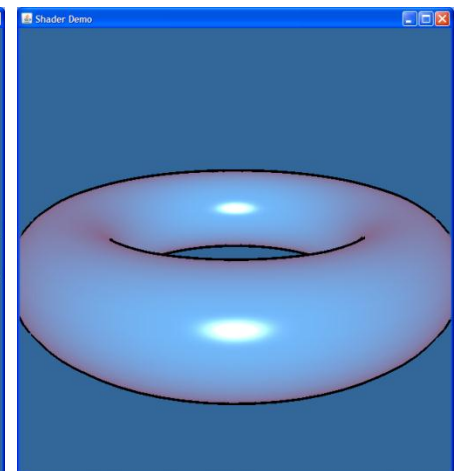
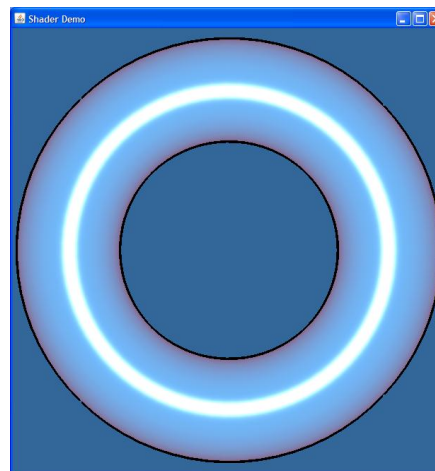
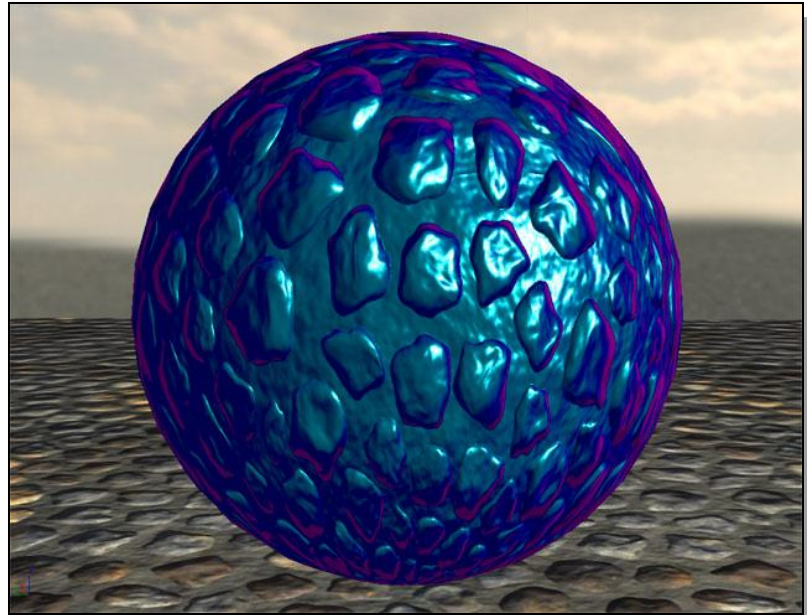


bump mapping

Applicazioni



Applicazioni

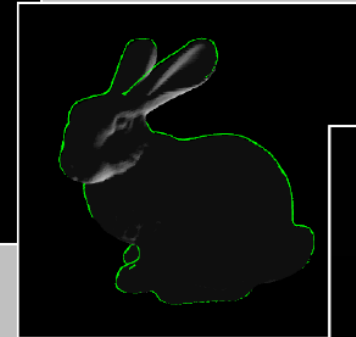


Applicazioni

Example: Shrinking Triangles



Example: Bunny Silhouettes



Argomenti che affrontiamo

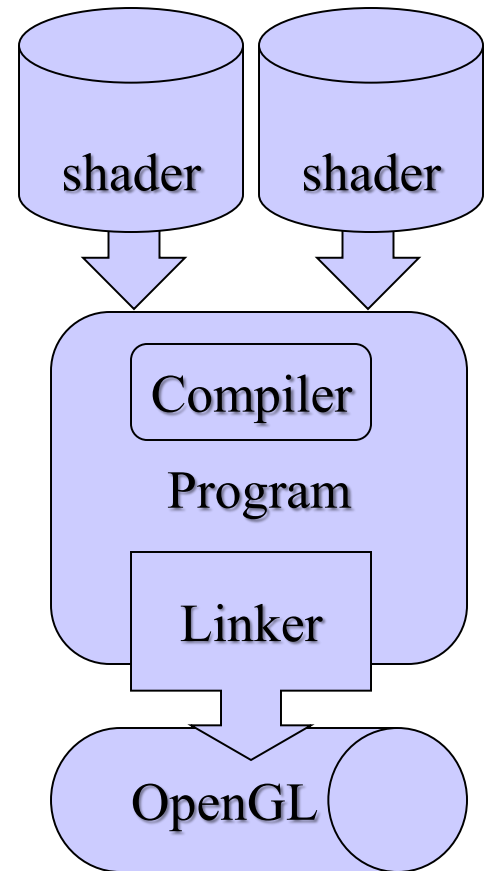
- 1) Come si collegano gli shader ad una applicazione OpenGL?
 - *program*
- 2) Come si programmano gli shader?
 - Introduzione al linguaggio GLSL
- 3) Esempi

Program

- Gli shader si collegano ad una applicazione OpenGL mediante i “*program*”
- Un *program* è una raccolta di shader compatibili tra loro.
- In ogni istante può essere attivo nella GPU un solo *program* e tutti gli shader che contiene sono attivi contemporaneamente.
- È possibile attivare un *program*, cambiare il *program* in uso o ripristinare la pipeline fissa in qualunque momento, eccetto durante la definizione di una primitiva.
- Se all'interno di un *program* non è definito nessuno shader di un certo tipo, quella parte è sostituita con lo shader di default della pipeline fissa.
- Più shader dello stesso tipo possono essere assegnati allo stesso *program*. In questo caso, gli shader sono uniti (fase di "linking") in modo simile ai file oggetto generati dai compilatori nei programmi per la CPU. Questo significa che deve essere sempre definito uno shader principale, che viene invocato per primo. È questo shader che può chiamare le funzioni che sono definite negli altri shader, che sono trattati come semplici librerie.

Program

1. Creare uno *shader object* con **glCreateShader**.
2. Caricare il codice dello shader, contenuto in un file di testo, con **glShaderSource**.
3. Compilare lo shader con **glCompileShader**.
4. Creare un *program object* con **glCreateProgram**.
5. Un program è una raccolta di shader, perciò bisogna indicare quali shader ne fanno parte. Eseguire il bind degli shader al program con **glAttachShader**.
6. Eseguire il link degli shader caricati nel *program* con **glLinkProgram**.
7. Attivare il program con **glUseProgram** prima di disegnare la scena.



Linguaggio GLSL

- OpenGL adotta un meccanismo ad estensioni. Le nuove features di OpenGL vengono prima proposte, poi inserite come estensioni e solo di seguito entrano a far parte dello standard.
- Gli esempi che vediamo richiedono l'uso della libreria GLEW (OpenGL Extension Wrangler). GLEW fornisce un unico header che include l'header originale di OpenGL (gl.h), GLU e i nomi delle funzioni per tutte le estensioni disponibili.
- Utilizzeremo Microsoft Visual Studio perché gli shader non sono supportati nel sistema operativo Linux virtuale
- Gli esempi che vedremo sono basati sulle specifiche di GLSL 1.2

Linguaggio GLSL

La sintassi di GLSL è molto simile al C con alcune differenze:

- Non esistono i puntatori;
- Non esistono i caratteri e le stringhe;
- Ci sono tipi predefiniti di dati vettoriali.
- **tipi di dati scalari: int, float, bool**
 - non esiste il typecast automatico delle variabili
float tre = 3.0; // valido **float tre** = 3; // non valido
- **vettori:**
 - float: **vec2, vec3, vec4**
 - int: **ivec** and bool: **bvec**
 - costruttori in stile C++
 - **vec3 a = vec3(1.0, 2.0, 3.0)**
 - **vec2 b = vec2(a)**
- **matrici (solo float) : mat2, mat3, mat4**
 - memorizzate per colonne
 - **mat2 m = mat2(1.0, 2.0, 3.0, 4.0);**
 - Accesso con parentesi quadre **m[column][row]**

Linguaggio GLSL

- **array di variabili:**
`float a[4];` // array di 4 float
`vec2 b[2];` // array di due vettori di 2 float
- **Costruttori**
`vec4 v = vec4(1.0,2.0,3.0,4.0);` // crea e inizializza il vettore con i valori 1, 2, 3, 4
`vec3 reali = vec3(inter);` // converte tre interi in tre reali
`mat2 matrice2 = mat2(2.0,1.0,0.0,3.0);`
- **Accesso agli elementi di un vettore**
`vec4 vettore = vec4(-1.0,0.0,1.0,3.0);`
`float secondo = vettore[1];` // seleziona il secondo elemento del vettore
- **Scope delle variabili:** le variabili possono essere locali o globali
- **Strutture dati**
`struct Luce {`
 `vec4 colore;`
 `vec3 posizione; };`

`Luce luce;`
`luce.colore = vec4(0.0,0.0,1.0,1.0);`
`vec3 pos = luce.posizione;`

Linguaggio GLSL

- **Operazioni di Swizzling**

vec2 vecxy = v.xy; // seleziona i primi due elementi

vec3 color = v.rgb; // seleziona i primi tre elementi

vec3 reversecolor;

reversecolor = v.bgr; // seleziona i primi tre elementi, ma in ordine inverso

vec4 riordina = v.barg; // seleziona tutti i componenti, ma in ordine diverso dall'originale (blue, alpha, red, green)

- **Funzioni, prototipi e operatori**

Lo shader principale (per un certo tipo e un certo program) è lo shader che definisce la procedura **void main()**

Per ogni tipo di shader e per ogni *program* deve essere definita una sola funzione `main()`. Questa funzione è quella che viene chiamata dalla GPU quando lo shader comincia la sua esecuzione.

- Il tipo di dato di ritorno può essere uno qualunque dei tipi di dato fondamentale (anche i vettori e le matrici), una struttura oppure `void` per indicare nessun valore di ritorno.

```
vec4 GetX() {  
    float w = 1.0;  
    return vec4(1.0,0.0,0.0,w);  
}
```

Linguaggio GLSL

- **Passaggio di parametri.**

sono state definite tre parole chiave:

- **in** indica che il parametro è un parametro di ingresso alla funzione
- **out** indica che il parametro è un parametro di uscita (deve essere scritto)
- **inout** indica che il parametro è sia un parametro di ingresso sia un parametro di uscita (corrisponde ad un passaggio per riferimento)

```
int Dividi(in int a,in int b,out int resto) {  
    resto = a % b;  
    return a / b;  
}
```

- I prototipi delle funzioni in GLSL sono dichiarati allo stesso modo del C. I prototipi servono soprattutto per usare funzioni che sono definite altrove.

- **OPERATORI**

GLSL consente di usare i normali operatori aritmetici e booleani anche su operandi costituiti da un vettore di elementi.

Linguaggio GLSL

- OPERATORI

`vec4 piu = a + b; // somma di a e b`

`vec4 c = a * b; // moltiplica tutte le componenti di a per b`

`vec4 d = a / b; // divide tutte le componenti di a per b`

`vec3 xyz = // ...`

`mat3 m = // ...`

`mat3 v = // ...`

`mat3 mv = v * m; // matrix * matrix`

`mat3 xyz2 = mv * xyz; // matrix * vector`

`mat3 xyz3 = xyz * mv; // vector * matrix`

Linguaggio GLSL

- **ALCUNE DELLE NUMEROSE FUNZIONI PREDEFINITE**

`float dot(in vec4 a,in vec4 b); // prodotto scalare (dot product)`

`vec3 cross(in vec3 a,in vec3 b); // prodotto vettoriale (cross product)`

`vec3 normalize(in vec3 a); // estrae il versore dal vettore,`

`float abs(in float a); // valore assoluto del parametro.`

`float sign(in float a); // -1.0 se  $a < 0.0$ , 0.0 se  $a = 0$ , 1.0 se  $a > 0.0$`

`vec4 clamp(in vec4 a, in float min, in float max); // ritorna a, ma se contiene valori
> max, essi sono posti = a max, se sono < min, vengono posti = a min`

`float sin(in float x); // seno (radianti)`

`float cos(in float x); // coseno (radianti)`

`float tan(in float x); // tangente (radianti)`

`float sqrt(in float x); // radice quadrata`

`float pow(in float x, in float y); // x elevato alla y`

`float ceil(in float x); // arrotondamento per eccesso float`

`floor(in float x); // arrotondamento per difetto`

`vec3 reflect(in vec3 ray, in vec3 normal); // riflette il vettore ray rispetto a normal`

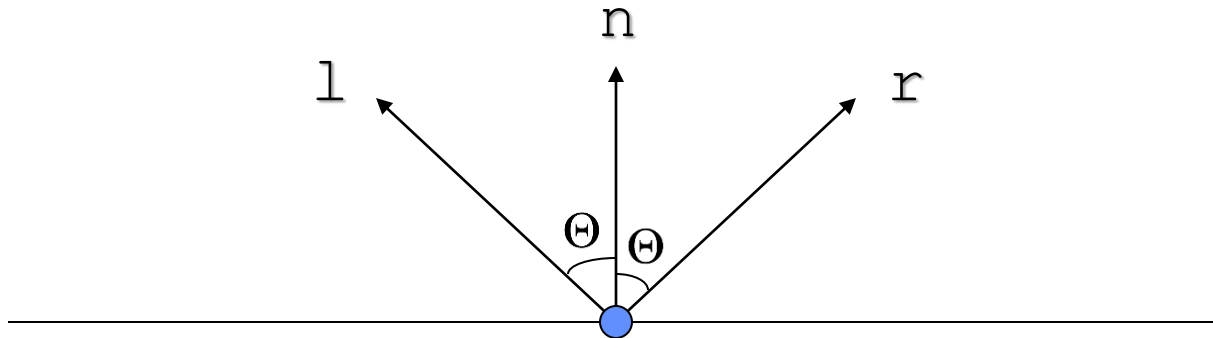
`vec3 mix(in vec3 x, in vec3 y,in float a); // uguale a:  $x * (1.0 - a) + y * a$`

Linguaggio GLSL

- ALCUNE DELLE NUMEROSE FUNZIONI PREDEFINITE

`reflect(-l, n)`

- Dati l e n , calcola r . angolo incidente = angolo riflesso



Linguaggio GLSL

- **Controllo di flusso (while, if, for).**

GLSL ha in comune con il C tutti i costrutti di controllo eccetto lo switch-case, che non è presente.

- **ALTRI QUALIFICATORI MOLTO IMPORTANTI**

- **attribute**
- **uniform**
- **varying**

Linguaggio GLSL

- **ATTRIBUTE**

- Variabili globali che cambiano per vertice.
- Sola-lettura, usati solo nei vertex shaders.
- Il valore viene passato allo shader dall'applicazione OpenGL (vedi più avanti).
- Esempi:
 - Built-in (variabili di stato OpenGL)
 - **gl_Vertex**
 - **gl_Color**
 - **gl_Normal**
 - **gl_SecondaryColor**
 - **gl_MultiTexCoordn**
 - **gl_FogCoord**
 - Definiti dall'utente
 - **attribute float temperature;**
 - **attribute vec3 velocity;**

Linguaggio GLSL

- **UNIFORM**
 - **Variabili globali costanti per primitiva (es: triangolo)**
 - Possono essere usate sia in vertex che in fragment shaders
 - Sola lettura
 - Il valore viene passato allo shader dall'applicazione OpenGL
 - **Built-in uniforms**
 - **uniform mat4 gl_ModelViewMatrix;**
 - **uniform mat4 gl_ProjectionMatrix;**
 - **uniform mat4 gl_ModelViewProjectionMatrix;**
 - **uniform mat4 gl_TextureMatrix[n];**
 - **uniform gl_LightSourceParameters
gl_LightSource[gl_MaxLights];**

gl_LightSourceParameters è una struttura dati che contiene i parametri delle luci

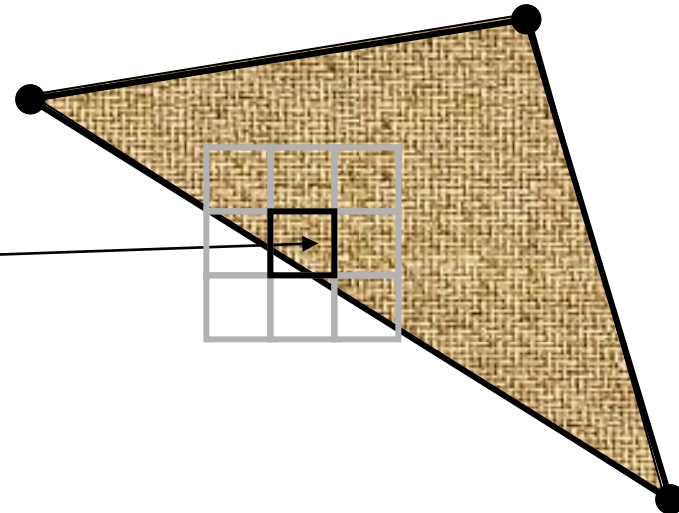
gl_LightSource[i].position posizione della luce in **eye-coordinates**

Linguaggio GLSL

- **UNIFORM**
 - Definite dall'utente
 - `uniform float myCurrentTime;`
 - `uniform vec4 myAmbient;`
- **VARYING**
 - Usate per passare dati dal vertex shader al fragment shader che vengono automaticamente interpolati (con correzione prospettica)
 - Possono essere scritte nel vertex shader
 - Sola lettura nel fragment shader.
 - Vengono automaticamente interpolate dal rasterizzatore!!!!!!!!!!
 - Built in
 - colori dei vertici: `varying vec4 gl_FrontColor; // vertex`
 - `varying vec4 gl_BackColor; // vertex`
 - `varying vec4 gl_Color; // fragment`
 - Definite dall'utente
 - Esempio: `varying vec3 normal` illuminazione per-pixel

Linguaggio GLSL

- **VARYING** – è la chiave fondamentale per capire la potenza degli shader grafici
- **Gli shader consentono di elaborare il singolo frammento di una primitiva**



Dopo l'elaborazione dei singoli vertici di ogni primitiva, le primitive vengono rasterizzate. Nella fase di rasterizzazione il colore dei vertici viene interpolato. I valori interpolati vengono passati ai fragment shaders che li possono ulteriormente elaborare.

Linguaggio GLSL

- **Passaggio di variabili attribute/uniform dall'applicazione OpenGL agli shader**
- Le variabili attribute/uniform sono definite negli shader
- Il linker crea una tabella delle variabili
- L'applicazione OpenGL può richiedere un riferimento alle variabili definite negli shader tramite l'indice della tabella del linker, per esempio:

GLint i = glGetAttribLocation(P, "myAttrib");

dove "P" rappresenta il nome del *program*, e "myAttrib" il nome della variabile

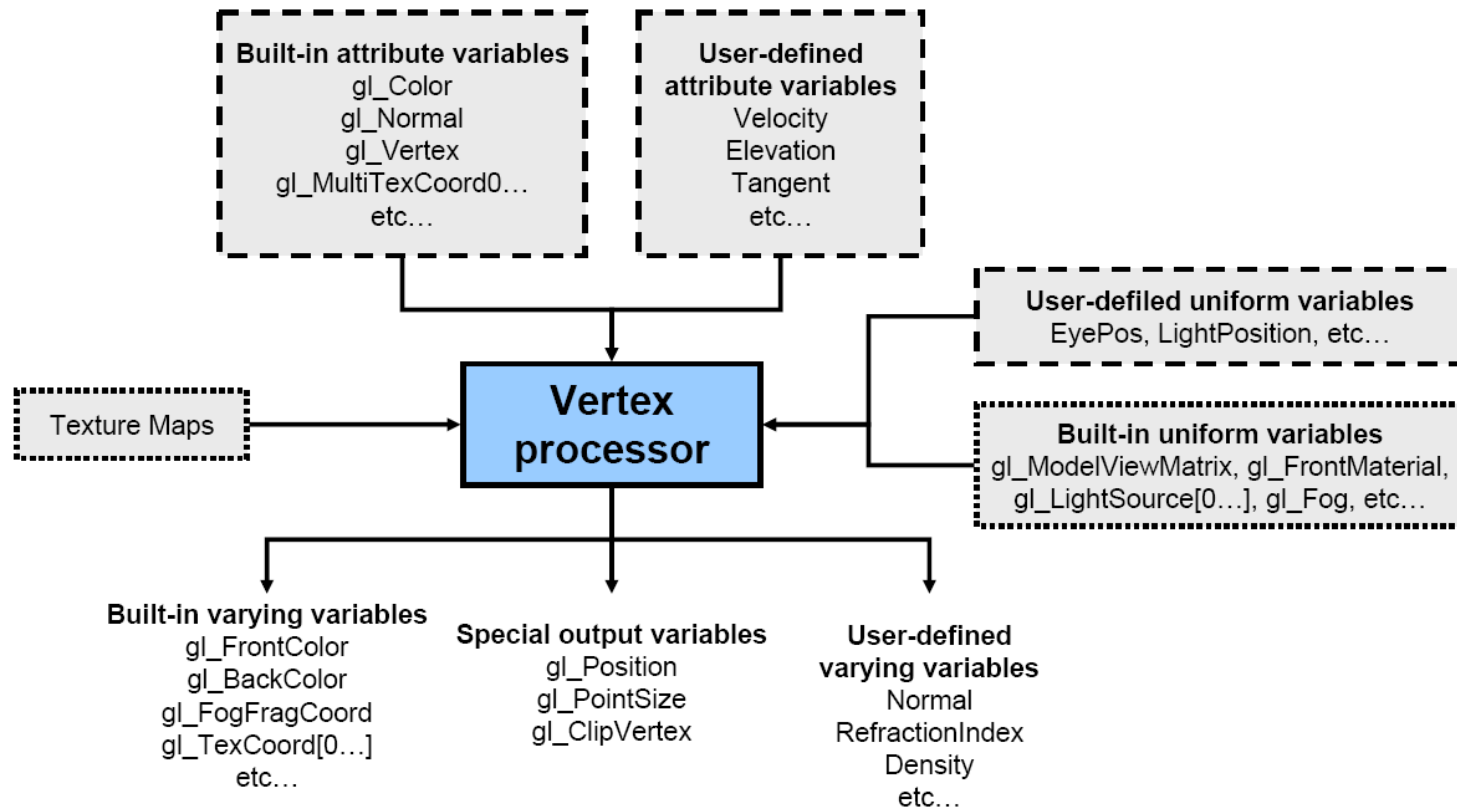
GLint j = glGetUniformLocation(P, "myUniform");

- Dopo aver recuperato l'indice della variabile di interesse, l'applicazione OpenGL può scriverne il contenuto per passarlo allo shader utilizzando varie funzioni a disposizione per i vari tipi di dato, per esempio:

glVertexAttrib1f(i, value);
glUniform1f(j, value);

Linguaggio GLSL

- Schema dettagliato di un vertex shader



Linguaggio GLSL

- **Schema dettagliato di un vertex shader**

Un vertex shader ha accesso in sola lettura agli attributi del vertice, come variabili globali:

```
vec4 gl_Vertex;           // posizione del vertice (xyzw) da glVertex
vec3 gl_Normal;           // normale del vertice (xyz) da glNormal
vec4 gl_Color;            // colore del vertice (rgba) da glColor
```

Il vertex shader ha accesso in scrittura ad altre variabili globali predefinite.
Tra cui:

```
vec4 gl_Position;         // nuova posizione del vertice (xyzw)
vec4 gl_FrontColor        // colore del vertice per la faccia frontale della primitiva
vec4 gl_BackColor         // colore del vertice per la faccia back della primitiva
```

Come tutti gli shader, un vertex shader ha accesso a molte delle variabili di stato di OpenGL, tra cui le matrici:

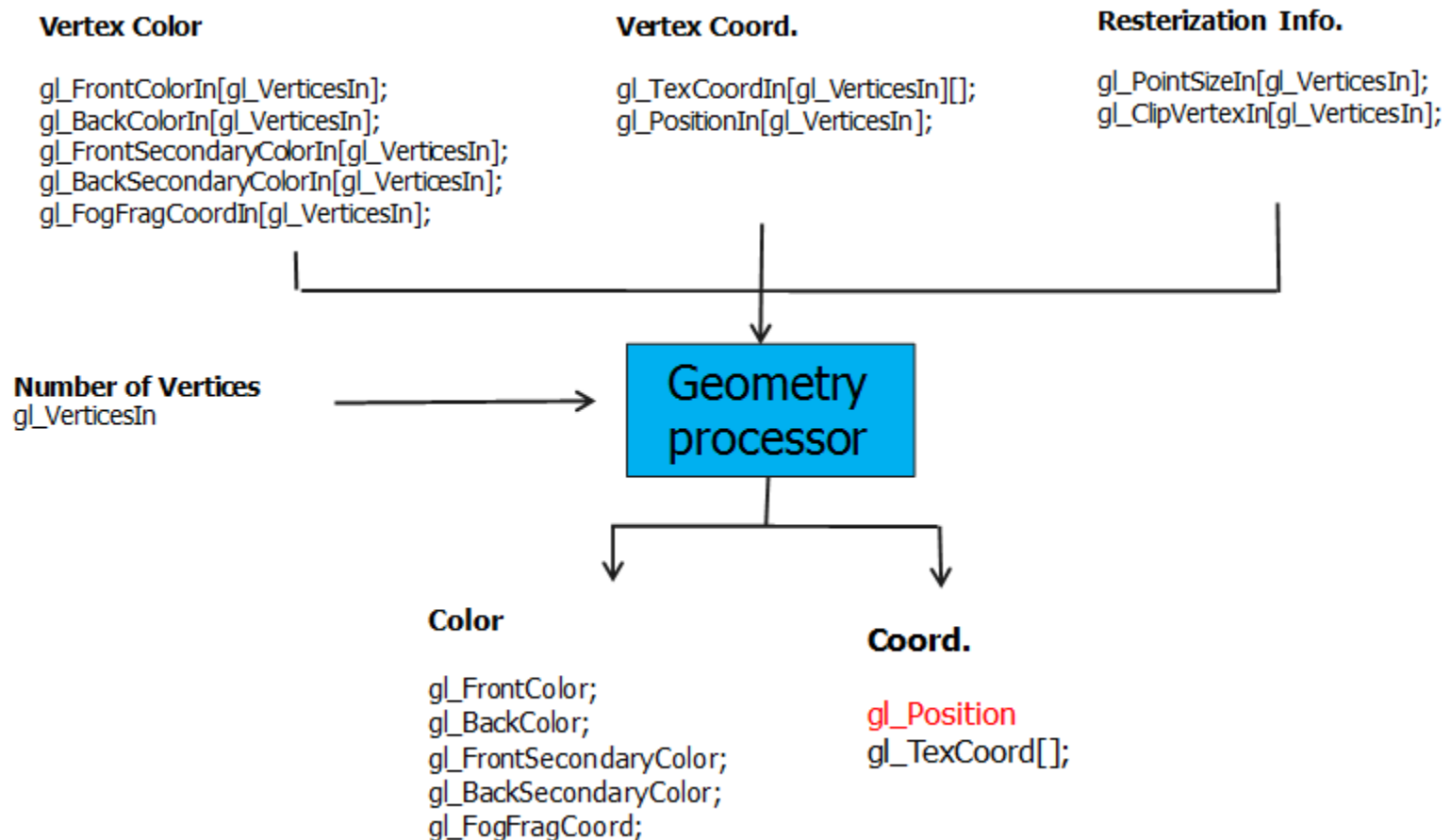
```
mat4 gl_ProjectionMatrix; // matrice di proiezione
mat4 gl_ModelViewMatrix;  // matrice di modelview
mat4 gl_ModelViewProjectionMatrix; // prodotto già precalcolato di
                                gl_ProjectionMatrix * gl_ModelViewMatrix
mat3 gl_NormalMatrix;     // matrice di trasformazione per le normali
```

Linguaggio GLSL

- **Esempio 1, un semplice vertex shader**
- **Esempio 2, ri-scrittura del Gouraud shading utilizzando gli shader**

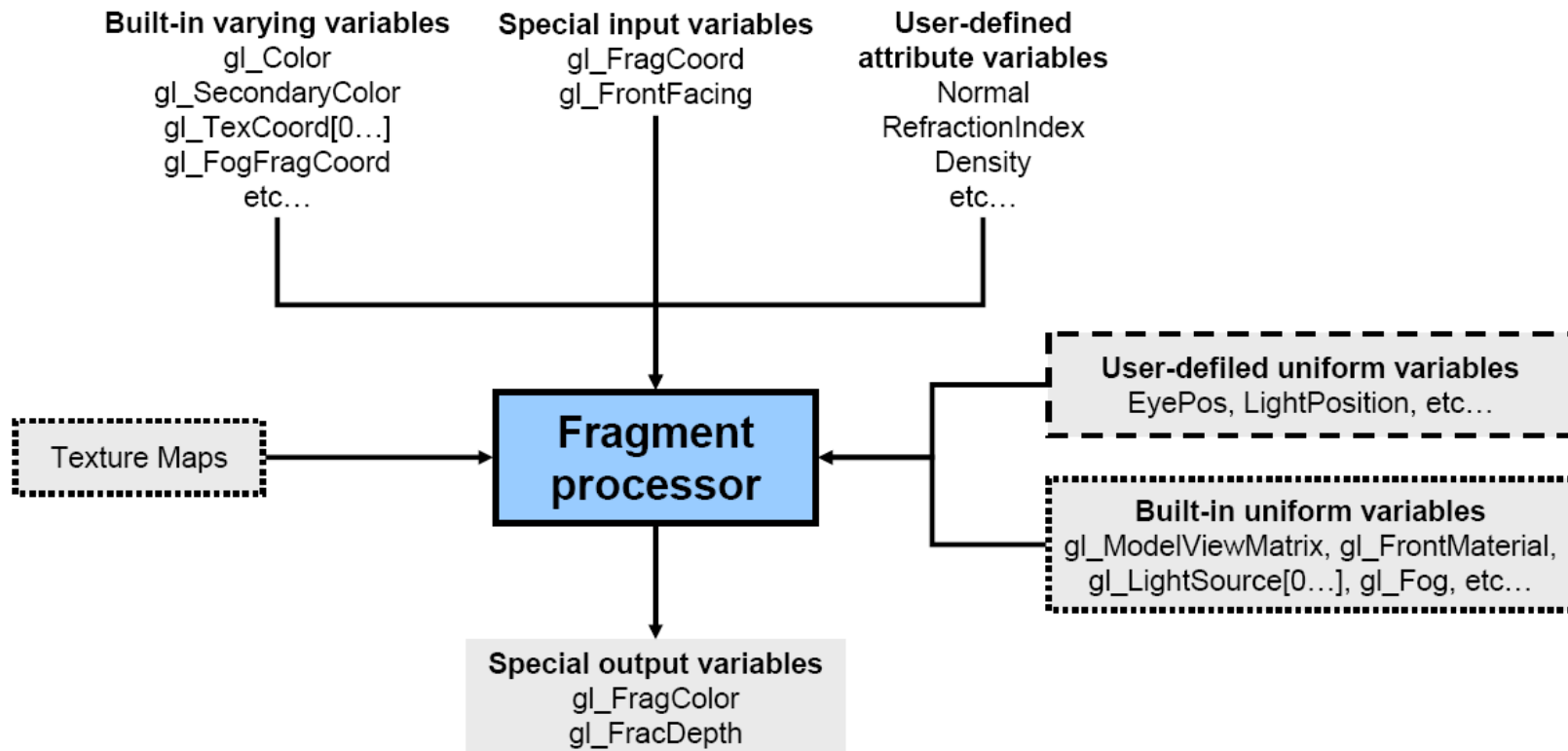
Linguaggio GLSL

- Schema dettagliato di un geometry shader
(usato meno frequentemente, molti esempi utilizzano solo vertex shaders e fragment shaders)



Linguaggio GLSL

- Schema dettagliato di un fragment shader



Linguaggio GLSL

- **Schema dettagliato di un fragment shader**

Il fragment shader ha accesso ad alcune variabili globali predefinite, tra cui:

```
vec4 gl_Color;  
vec4 gl_FragColor;
```

gl_Color contiene il colore del frammento calcolato automaticamente per interpolazione dei colori (prodotti dal vertex shader) dei vertici della primitiva in **gl_FrontColor** o **gl_BackColor** per facce front o back rispettivamente.

In **gl_FragColor** deve essere scritto il colore finale del frammento, cioè quello che verrà visualizzato sullo schermo se il depth test ha successo.

Il fragment shader è l'unico shader che può usare la parola chiave **discard**. Quando questa istruzione è eseguita, l'esecuzione dell'istanza dello shader si interrompe e il frammento non è mai visualizzato sullo schermo.

Esempio 3, un semplice fragment shader che si limita ad applicare il colore che riceve in ingresso.

Linguaggio GLSL

Esempio 4, illuminazione per pixel (phong-shading).

Altri esempi: Bump mapping, erosione, esempi di geometry shader