

Corso di
Grafica Computazionale

GLSL

Docente:
Massimiliano Corsini

Laurea Specialistica in Ing. Informatica

Facoltà di Ingegneria

Università degli Studi di Siena



Cosa sono gli shaders?

- Gli shaders sono programmi che vengono eseguiti dalla GPU (Graphics Processing Unit)
- Ci permettono di programmare gran parte della pipeline di rendering
- I Vertex Shaders lavorano a livello di vertice nel sottosistema geometrico
- I Pixel Shaders lavorano a livello di frammento nel sottosistema raster



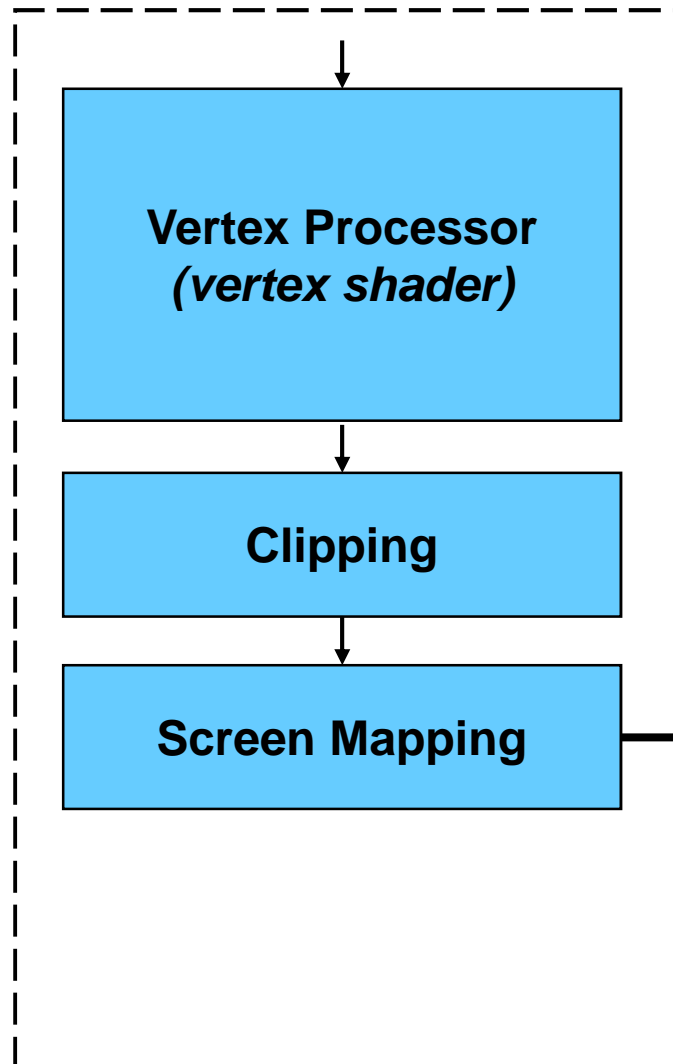
Cosa sono gli shaders?

- Fino a pochi anni fa l'hardware grafico non era programmabile. Le operazioni eseguite dalla pipeline erano quindi fisse e poco controllabili
- Adesso, grazie alla programmabilità della pipeline è possibile ottenere un gran numero di effetti visivi

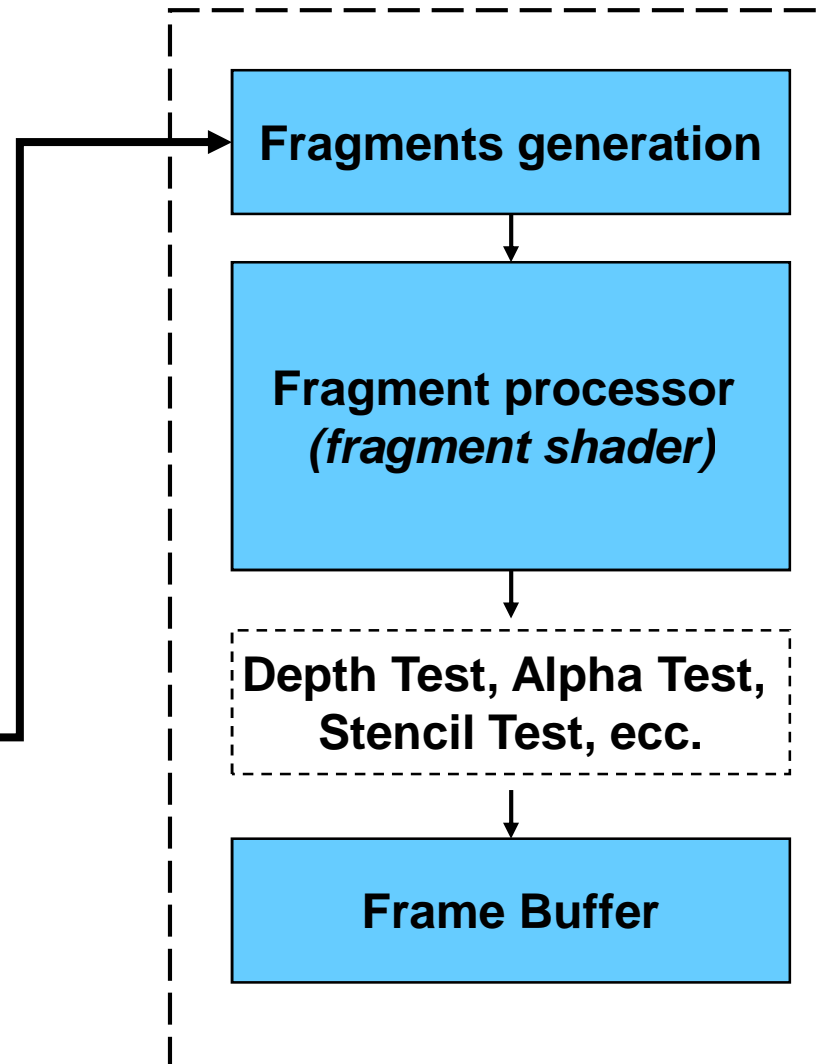


Shaders

Sottosistema geometrico



Sottosistema raster





Vertex Shaders

- Input: vertici (una per volta!!) ed i loro attributi
- Non si possono fare elaborazioni che coinvolgono più vertici
- Operazioni tipicamente implementate a livello di vertex shaders:
 - Trasformazioni geometriche sui vertici
 - Calcolo dell'illuminazione per vertice
 - Generazione coordinate texture
 - Trasformazioni sulla normale del vertice
- Output: posizione del vertice (in clip space)



Fragment Shaders

- Input: output del vertex shader interpolato
- Anche in questo caso si elabora un frammento per volta (i frammenti adiacenti non sono processabili)
- Operazioni tipicamente implementate a livello di pixel shaders:
 - Texturing
 - Calcolo dell'illuminazione (per-pixel)
 - Calcolo finale del colore del frammento
- Output: colore del frammento



Caratteristiche degli Shaders

- Gli shaders “conoscono” lo stato di rendering (GLSL conosce lo stato dell’OpenGL) e possono accedere alle textures
- Una passata di rendering è intesa come l’elaborazione di una scena da parte del vertex shader e del pixel shader
- Spesso la generazione di un’immagine richiede più di una passata (multi-pass rendering)
 - Ogni passata utilizzerà uno shader diverso
 - Una passata può inviare il risultato del rendering su una texture per essere utilizzato dalle successive passate (render-to-texture)



Linguaggi di Shaders

- Sintassi è C-like
- Tipi di dati:
 - Half (float a 16 bit)
 - Vettori
 - Matrici (3x3 e 4x4)
 - Textures (1D, 2D, 3D)
- Condizioni e cicli sono possibili
- Sono supportate operazioni vettori-matrici
- Funzioni matematiche particolari (esempio `reflect(...)`, funzioni per fare lookup dalle texture, ecc.)
- Linguaggi di shaders esistenti:
 - HLSL (DirectX, Microsoft, 2002)
 - GLSL (OpenGL, ARB, 2003)
 - Cg (NVidia, 2002)



Compilazione a Runtime

Facoltà di
Ingegneria

- Gli shaders vengono compilati durante l'esecuzione dell'applicazione 3D
 - Direct3D compila gli shaders HLSL
 - OpenGL compila gli shaders GLSL
 - Cg (NVIDIA) permette di compilare gli shaders Cg sia in applicazioni OpenGL che Direct3D



Shaders & Tools

- RenderMonkey (ATI)
 - Supporta GLSL
 - IDE semplice e potente
 - Eccellente per sviluppare i propri shaders
- FX Composer (NVidia / Microsoft)
 - FX format (formato proprietario)
 - Debugging Tools
- Maya e 3DS Max hanno il loro ambiente di sviluppo di shaders
- Renderman (Pixar) ha il proprio linguaggio di shading



Imparare a scrivere shaders

- Partire dalle specifiche del linguaggio non aiuta moltissimo...
- Meglio: *RenderMonkey* → guardare gli *shaders* già scritti, capire quello che fanno e modificarli
- Provare, provare e ancora provare (!!)
- Debug è una nota dolente anche se stanno venendo fuori tool di supporto al debug (esempio GLSLDevil)
- Ottimizzare le performances → prima ottenere l'effetto voluto, poi affidarsi a qualche profiler



Usare GLSL nella propria applicazione OpenGL

Facoltà di
Ingegneria

- Abbiamo bisogno di poter gestire le estensioni del linguaggio (OpenGL 1.5):
 - `GL_ARB_vertex_shader`
 - `GL_ARB_fragment_shader`
- **GLEW** (OpenGL Extension Wrangler Library) semplifica enormemente l'utilizzo delle estensioni
- Se l'hardware grafico di cui si dispone è compatibile con le specifiche OpenGL 2.0 non si ha bisogno delle estensioni di cui sopra.



Creazione e Compilazione

- È possibile creare dei cosiddetti **Shader Object**, che fungono da contenitori per i programmi di shading.
- La funzione per creare uno shader ritorna un apposito handle:
 - `GLuint glCreateShaderObjectARB(GLenum shaderType);`
- La funzione per settare il codice sorgente dello shader è:

```
void glShaderSourceARB(GLhandleARB shader, int
  numOfStrings, const char **strings, int
  *lenOfStrings);
```
- Una volta settato il codice sorgente in formato testo si deve compilare.
 - `void glCompileShaderARB(GLhandleARB program);`



Attach e Linking

- Una volta creati degli shaders e compilati è possibile creare un programma di shading (***Program Object***).
- La funzione per creare un programma di shading è:
 - `GLuint glCreateProgramObjectARB(void);`
- Ad un programma di shading si possono attaccare (attach) gli shaders definiti negli shader objects (`glAttachObjectARB(...)`).
- Si possono avere più shader attaccati allo stesso programma così come si possono avere più sorgenti in un programma C.
- Lo shader base è caratterizzato dal *main* (come un programma C).



Attach e Linking

- Una volta attaccati gli Shader Object (e quindi definito il codice sorgente compilato dei vertex e pixel shaders del programma) al Program Object si può procedere al linking.
- Il programma di shading è pronto per essere utilizzato tramite la funzione:
 - `glUseProgramObjectARB(GLHandleARB program)`
- È possibile creare quanti programmi di shading si vogliono e decidere in ogni momento quale mandare in esecuzione.



Esempio di Base

```
GLHandleARB p,f,v;
char *vs,*fs;

v = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
f = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);

vs = textFileRead("toon.vert");
fs = textFileRead("toon.frag");

const char * vv = vs; // vv diventa ** al testo del codice sorgente
const char * ff = fs; // ff diventa ** al testo del codice sorgente

glShaderSourceARB(v, 1, &vv,NULL);
glShaderSourceARB(f, 1, &ff,NULL);
free(vs);free(fs);

glCompileShaderARB(v);
glCompileShaderARB(f);

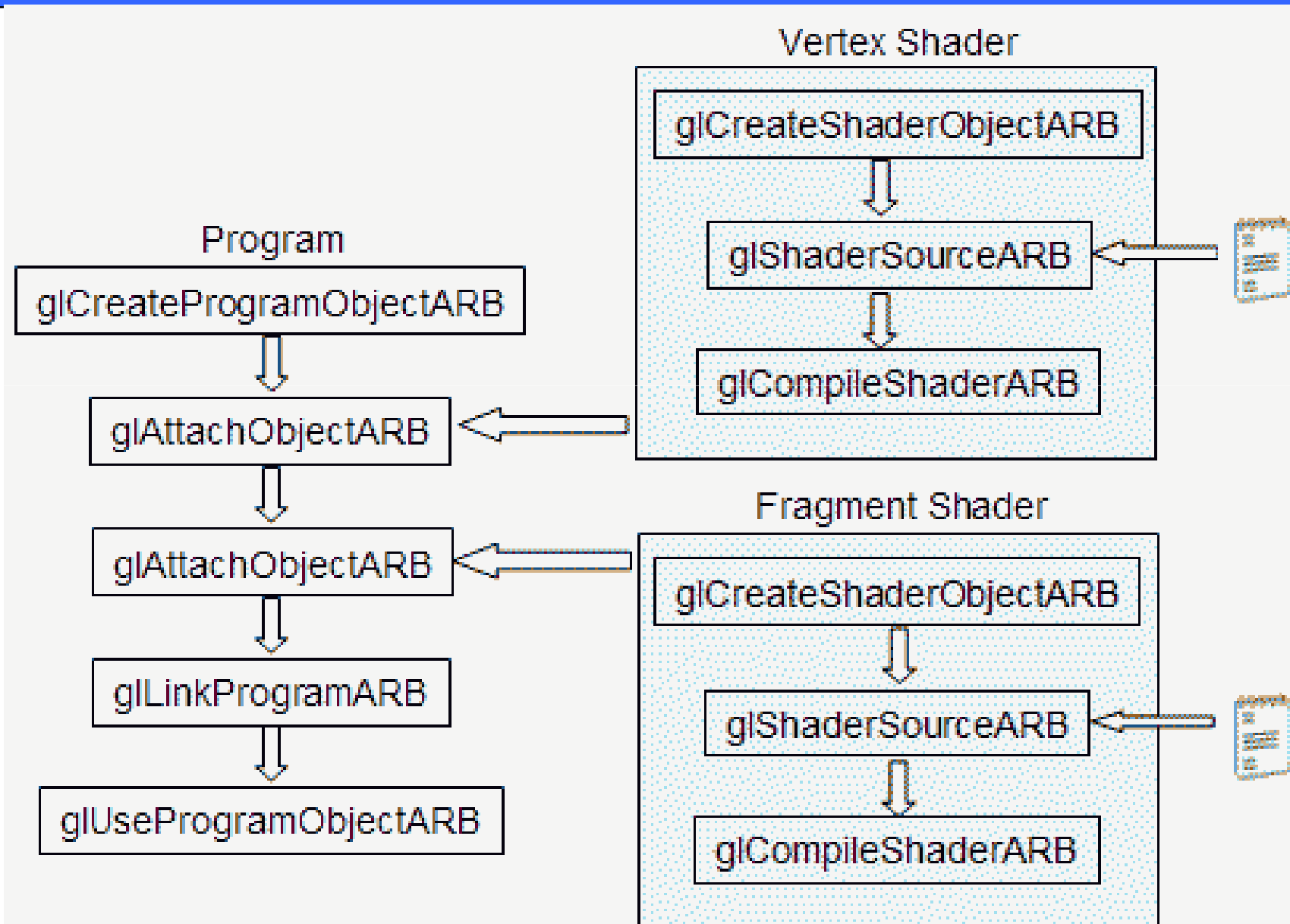
p = glCreateProgramObjectARB();

glAttachObjectARB(p,v);
glAttachObjectARB(p,f);

glLinkProgramARB(p);
glUseProgramObjectARB(p);
```




Riassumendo...





Cleaning Up

- Per effettuare il detach di uno shader:
 - `void glDetachObjectARB(GLhandleARB program, GLhandleARB shader);`
- Gli shader attached non possono essere cancellati dalla memoria. Ecco perchè l'operazione di detach è importante.
- Per cancellare uno shader dalla memoria (Program Object o Shader Object):
 - `void glDeleteObjectARB(GLhandleARB id);`
- Se si fa un'operazione di delete è lo Shader Object fa sempre parte di un Program Object lo shader non viene cancellato ma marchiato come *da cancellare*.



Tipi di Dati

- Floating point, interi e boolean sono disponibili:
 - `float`
 - `bool`
 - `int`
- Vettori a 2,3 o 4 componenti:
 - `vec2` , `vec3` , `vec4` (vettori di float)
 - `bvec2` , `bvec3` , `bvec4` (vettori di boolean)
 - `ivec2` , `ivec3` , `ivec4` (vettori di interi)
- Matrici quadrate 2x2, 3x3 e 4x4:
 - `mat2`
 - `mat3`
 - `mat4`



Tipi di Dati

- L'accesso alle texture avviene tramite speciali tipi di dati: i campionatori (sampler):
 - `sampler1D` – per le texture 1D
 - `sampler2D` – per le texture 2D
 - `sampler3D` – per le texture 3D
 - `samplerCube` – per le CUBE MAP



Dichiarazioni variabili ed inizializzazione

- La dichiarazione delle variabili è simile a quella del C/C++.
 - `float a,b;`
 - `int c = 2; // commento (come in C++)`
- GLSL gestisce il type-casting tramite costruttori e l'inizializzazione tramite costruttori di copia:
 - `float b = 2; // non va bene, l'intero non viene automaticamente castato a float`
 - `float e = (float)2; // non funziona ugualmente anche se ci si potrebbe aspettare di si`
 - `float c = float(2.0); // corretto`
 - `vec3 g = vec3(1.0,2.0,3.0); // corretto`
- Un pò di flessibilità nel costruttore c'è(!)
 - `vec2 a = vec2(1.0,2.0);`
 - `vec2 b = vec2(3.0,4.0);`
 - `vec4 c = vec4(a,b); // c = vec4(1.0,2.0,3.0,4.0);`
 - `vec4 g = vec4(1.0); // tutte le componenti sono inizializzate ad 1.0`



Variabili 'Uniform'

- Una variabile di tipo **uniform** è costante rispetto ad una primitiva, ossia non può modificare il proprio valore tra una chiamata a *glBegin* ed una a *glEnd*. Le variabili di tipo uniform possono essere lette ma non scritte dal vertex o fragment shaders.
- La funzione per recuperare la **location** all'interno di un programma di shading di una variabile uniform (dato il suo nome) è:
 - `GLint glGetUniformLocationARB(GLhandleARB program, const char *name);`
- La location della variabile (ritornata dalla funzione) permette di settare la variabile uniform in questione utilizzando:
 - `void glUniform1fARB(GLint location, GLfloat v0);`
`void glUniform2fARB(GLint location, GLfloat v0, GLfloat v1);`
`void glUniform3fARB(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);`
`void glUniform4fARB(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3);`
 - `GLint glUniform{1,2,3,4}fvARB(GLint location, GLsizei count, GLfloat *v);`



Variabili 'Uniform'

- Esiste un set di funzioni analogo per i valori di tipo intero e matriciale ma non per i valori booleani.
 - `GLint glUniformMatrix{2,3,4}fvARB(GLint location, GLsizei count, GLboolean transpose, GLfloat *v);`
 - location – location della variabile
 - count – numero di matrici da settare
 - transpose - 1 indica row-major order, 0 column-major order
 - v – array di float



Uniform (esempio)

```
uniform float specIntensity;
uniform vec4 specColor;
uniform float t[2];
uniform vec4 colors[3];

GLint loc1,loc2,loc3,loc4;
float specIntensity = 0.98;
float sc[4] = {0.8,0.8,0.8,1.0};
float threshold[2] = {0.5,0.25};
float colors[12] = {0.4,0.4,0.8,1.0,
                  0.2,0.2,0.4,1.0,
                  0.1,0.1,0.1,1.0};

loc1 = glGetUniformLocationARB(p,"specIntensity");
glUniform1fARB(loc1,specIntensity);
loc2 = glGetUniformLocationARB(p,"specColor");
glUniform4fvARB(loc2,1,sc);
loc3 = glGetUniformLocationARB(p,"t");
glUniform1fvARB(loc3,2,threshold);
loc4 = glGetUniformLocationARB(p,"colors");
glUniform4fvARB(loc4,3,colors);
```




Attributi

- Variabili di tipo attributo possono essere assegnate per ogni vertice.
- Ci permettono di settare i dati degli attributi dei vertici durante il passaggio delle primitive alla scheda grafica.
- Analogamente alle variabili di tipo uniform, possono essere soltanto letti (nel vertex shader). Per il settaggio si devono utilizzare delle funzioni ad hoc.



Variabili Attributo

- Funzione per recuperare la location:
 - `GLint glGetAttribLocationARB(GLhandleARB program, char *name);`
- Funzioni per il setting:
 - `void glVertexAttrib1fARB(GLint location, GLfloat v0);`
 - `void glVertexAttrib2fARB(GLint location, GLfloat v0, GLfloat v1);`
 - `void glVertexAttrib3fARB(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);`
 - `void glVertexAttrib4fARB(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3);`
 - `GLint glVertexAttrib{1,2,3,4}fvARB(GLint location, GLfloat *v);`



Attributi (esempio)

```
loc = glGetAttribLocationARB(p, "height");  
  
glBegin(GL_TRIANGLE_STRIP);  
    glVertexAttrib1fARB(loc, 2.0);  
    glVertex2f(-1, 1);  
    glVertexAttrib1fARB(loc, 2.0);  
    glVertex2f(1, 1);  
    glVertexAttrib1fARB(loc, -2.0);  
    glVertex2f(-1, -1);  
    glVertexAttrib1fARB(loc, -2.0);  
    glVertex2f(1, -1);  
glEnd();
```



Attributi (esempio)

```
// attribute variables
// www.lighthouse3d.com

attribute float height;

void main()
{
    vec4 p;
    p.xz = gl_Vertex.xy;
    p.y = height;
    p.w = 1.0;

    gl_Position = gl_ModelViewProjectionMatrix * p;
}
```



Variabili 'Varying'

- Una volta che i vertici sono stati elaborati, vengono trasformati in frammenti durante la fase di rasterizzazione e passati al fragment shader. Per ogni frammento esistono un set di variabili che vengono interpolate automaticamente in fase di rasterizzazione (esempio colore).
- GLSL ha alcune variabili predefinite di questo tipo; queste variabili vengono chiamate **varying variable**. GLSL permette di definire variabili varying.
- Le variabili varying devono essere dichiarate sia nel vertex shader che nel fragment shader.
- Questo ci permette di assegnare ai vertici degli attributi che saranno interpolati automaticamente.
- Per esempio si può assegnare la normale di un vertice come varying variable e trovarsela così interpolata per ogni pixel → **calcolo del lighting per-pixel**.



Hello GLSL (!)

- Vertex Shader:

- `void main()`
 - `{`
 - `gl_Position = gl_ProjectionMatrix *`
 - `gl_ModelViewMatrix * gl_Vertex;`
 - `}`

- Fragment Shader:

- `void main()`
 - `{`
 - `gl_FragColor = vec4(1.0,0.0,0.0,0.0);`
 - `}`



Hello GLSL (!)

- Vertex Shader:

- `void main()`
 - `{`
 - `gl_Position = ftransform();`
 - `}`

- Fragment Shader:

- `void main()`
 - `{`
 - `gl_FragColor = glColor;`
 - `}`



Flatten Shader

- Vertex Shader:

- ```
void main()
{
 vec4 v = vec4(gl_Vertex);
 v.z = 0.0; // il modello viene appiattito
 gl_Position = gl_ModelViewProjectionMatrix * v;
}
```

- Fragment Shader:

- ```
void main()
{
    gl_FragColor = vec4(1.0,0.0,0.0,0.0);
}
```




Phong Shader (per-vertex)

Vertex Shader:

```
varying vec4 color;
uniform vec4 mViewDirection;
uniform float shininess;
uniform vec4 Kd;
uniform vec4 Ks;

void main(void)
{
    // vertex normal
    vec3 N = normalize(gl_NormalMatrix * gl_Normal);

    // diffuse term
    vec4 vpos = gl_ModelViewMatrix * gl_Vertex;
    vec4 diffuse = vec4(0.0);
    vec3 lightDir = normalize(gl_LightSource[0].position.xyz -
                              vpos.xyz);
    float NdotL = dot(N, lightDir);
    if (NdotL > 0.0)
        diffuse = Kd * gl_LightSource[0].diffuse * NdotL;
    ...
}
```



Phong Shader (per-vertex)

Vertex Shader (continua):

```
// specular term
vec4 specular = vec4(0.0);
vec3 rVector = normalize(2.0 * N * dot(N, lightDir) - lightDir);
vec3 viewVector = normalize(-vViewDirection.xyz);
float RdotV = dot(rVector, viewVector);

if (RdotV > 0.0)
    specular = Ks * gl_LightSource[0].specular *
               pow(RdotV, shininess);

// final color (ambient term is not considered)
color = diffuse + specular;

// output: vertex position
gl_Position = ftransform();
}
```



Phong Shader (per-vertex)

Fragment Shader:

```
varying vec4 color;  
  
void main(void)  
{  
    gl_FragColor = color;  
}
```



Phong Shader (per-pixel)

Vertex Shader:

```
varying vec3 normal;
varying vec3 vpos;

void main()
{
    // vertex normal
    normal = gl_NormalMatrix * gl_Normal;

    // vertex position
    vpos = vec3(gl_ModelViewMatrix * gl_Vertex);

    // vertex position
    gl_Position = ftransform();
}
```



Phong Shader (per-pixel)

Facoltà di
Ingegneria

Fragment Shader:

```
varying vec3 normal;  
varying vec3 vpos;
```

```
uniform vec4 mViewDirection;  
uniform float shininess;  
uniform vec4 Kd;  
uniform vec4 Ks;
```

```
void main()
```

```
{
```

```
    // per-pixel normal
```

```
    vec3 n = normalize(normal);
```

```
    // diffuse term
```

```
    vec4 diffuse = vec4(0.0);
```

```
    vec3 lightDir = normalize(gl_LightSource[0].position.xyz - vpos);
```

```
    float NdotL = dot(n, lightDir);
```

```
    if (NdotL > 0.0)
```

```
        diffuse = Kd * gl_LightSource[0].diffuse * NdotL;
```

```
    ...
```



Phong Shader (per-pixel)

Fragment Shader (continua):

```
...

// specular term
vec4 specular = vec4(0.0);
vec3 rVector = normalize(2.0 * n * dot(n, lightDir) -
    lightDir);
vec3 viewVector = normalize(-vViewDirection.xyz);
float RdotV = dot(rVector, viewVector);

if (RdotV > 0.0)
    specular = Ks * gl_LightSource[0].specular *
        pow(RdotV, shininess);

// final color
gl_FragColor = diffuse + specular;
}
```



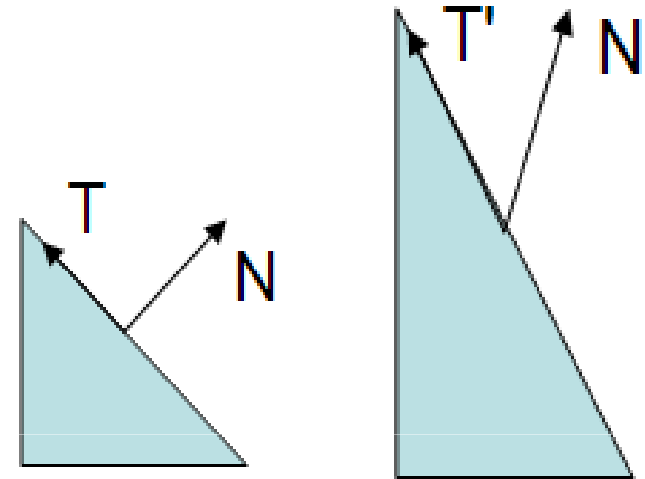
- Quando si lavora con le normali (cioè spesso) si deve fare attenzione allo spazio di coordinate in cui si lavora (***object space*** o ***eye space***).
- Ossia anche le normali devono essere trasformate opportunamente, così come i vertici.
- Talvolta questo equivale semplicemente ad applicare la matrice di rototraslazione:
 - `normalEyeSpace = vec3(gl_ModelViewMatrix *
vec4(gl_Normal,0.0)); // gl_Normal ha 3
componenti`



Trasformazioni e Normali

Facoltà di
Ingegneria

- Talvolta questo non è valido:



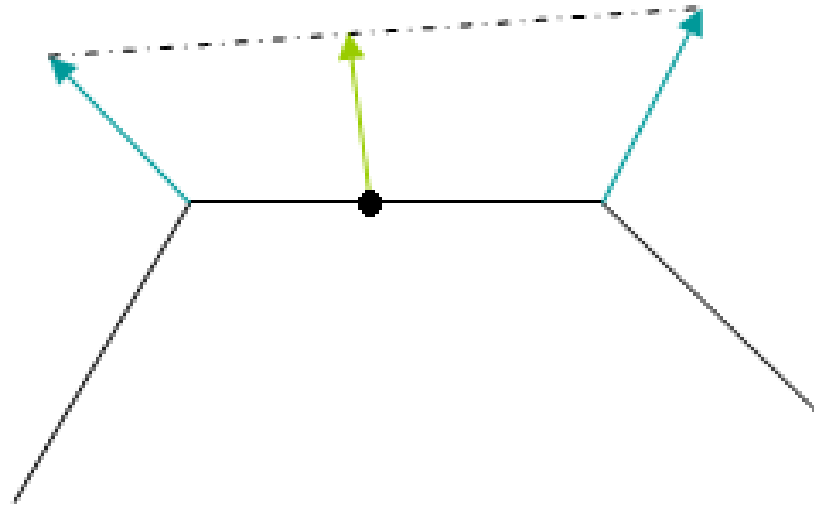
- In particolare se ho trasformazioni di scala la normale risulta deformata. Si dimostra matematicamente che in questi casi si deve utilizzare l'inversa della trasposta della matrice MODELVIEW.

$$G = (M^{-1})^T$$



Normal Matrix

- In GLSL potete trasformare la normale in eye-space semplicemente moltiplicandola per la matrice predefinita ***gl_NormalMatrix***:
 - `normal = gl_NormalMatrix * gl_Normal;`
- Ricordatevi sempre la normalizzazione (`normalize(normal)`), specie se lavorate nel fragment shader, dove l'interpolazione spesso rende il vettore di lunghezza non unitaria.





Simple Texture Shader

- Per quanto riguarda l'accesso alle coordinate texture, il linguaggio GLSL prevede le seguenti variabili (attribute) predefinite:
 - `attribute vec4 gl_MultiTexCoord0`
 - `attribute vec4 gl_MultiTexCoord1`
 - `attribute vec4 gl_MultiTexCoord2`
 - ...
 - `attribute vec4 gl_MultiTexCoord7`
- Come si nota, si ha una variabile per ogni texture unit.



Simple Texture Shader

Vertex Shader:

```
varying vec4 texCoord;  
  
void main()  
{  
    // texture coordinates  
    texCoord = gl_MultiTexCoord0;  
  
    // output (vertex position)  
    gl_Position = ftransform();  
}
```



Simple Texture Shader

Facoltà di
Ingegneria

Fragment Shader:

```
varying vec4 texCoord;  
uniform sampler2D tex;  
  
void main()  
{  
    // texture lookup  
    vec4 color = texture2D(tex, texCoord.st);  
  
    // output  
    gl_FragColor = color;  
}
```

NOTA: Questo modo di procedere equivale ad effettuare un'operazione di texture mapping in modalità GL_REPLACE



Simple Texture Shader (2)

Facoltà di
Ingegneria

Vediamo il caso di texture mapping in modalità GL_MODULATE:

Vertex Shader:

```
varying vec3 lightDir;
varying vec3 normal;

void main()
{
    // mi servono per calcolare l'illuminazione per-pixel
    normal = gl_NormalMatrix * gl_Normal;
    lightDir = vec3(1.0); // directional light

    // pre-defined varying variable
    gl_TexCoord[0] = gl_MultiTexCoord0;

    // output (vertex position)
    gl_Position = ftransform();
}
```



Simple Texture Shader (2)

Fragment Shader:

```
varying vec3 lightDir, normal;
uniform sampler2D baseMap;

void main()
{
    float intensity;
    intensity = max(dot(normalize(lightDir),normalize(normal)),0.0);
    vec3 cf = vec3(intensity * gl_FrontMaterial.diffuse.rgb);
    float af = float(gl_FrontMaterial.diffuse.a);

    // texture lookup
    vec4 texel = vec4(texture2D(baseMap, gl_TexCoord[0].st));
    vec3 ct = vec3(texel.rgb);
    float at = float(texel.a);

    // output (modulazione della texture secondo il contributo
    //          dell'illuminazione)
    gl_FragColor = vec4(ct * cf, at * af);
}
```



Domande?