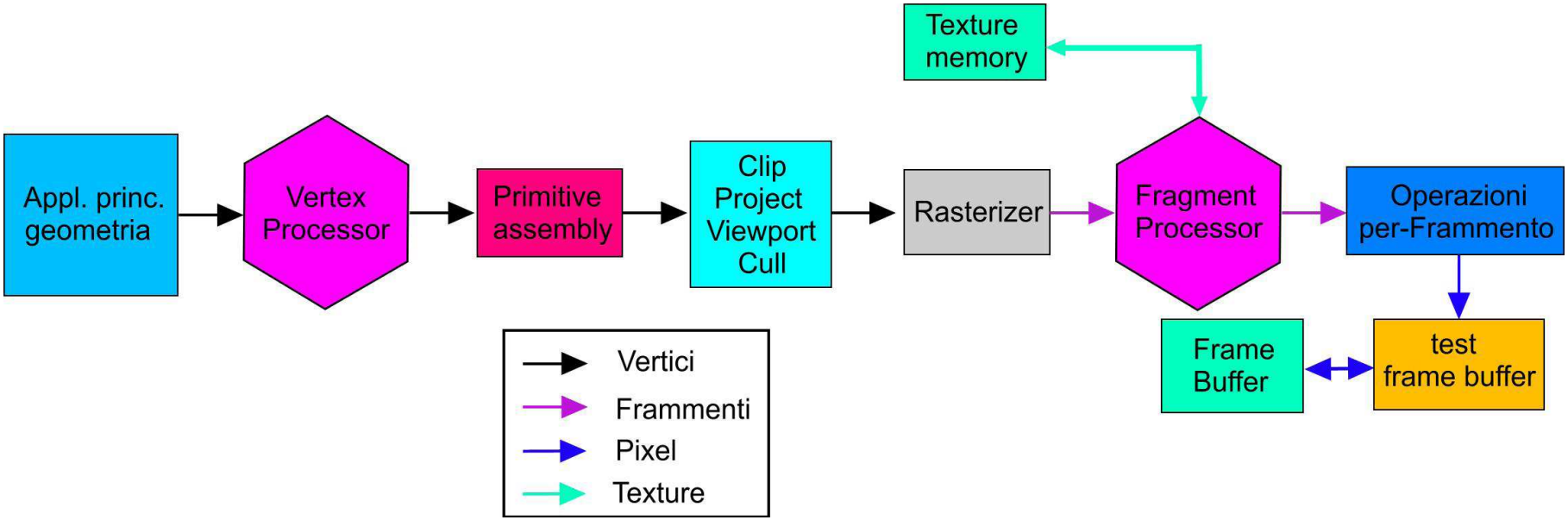
Grafica Computazionale

Introduzione agli shaders

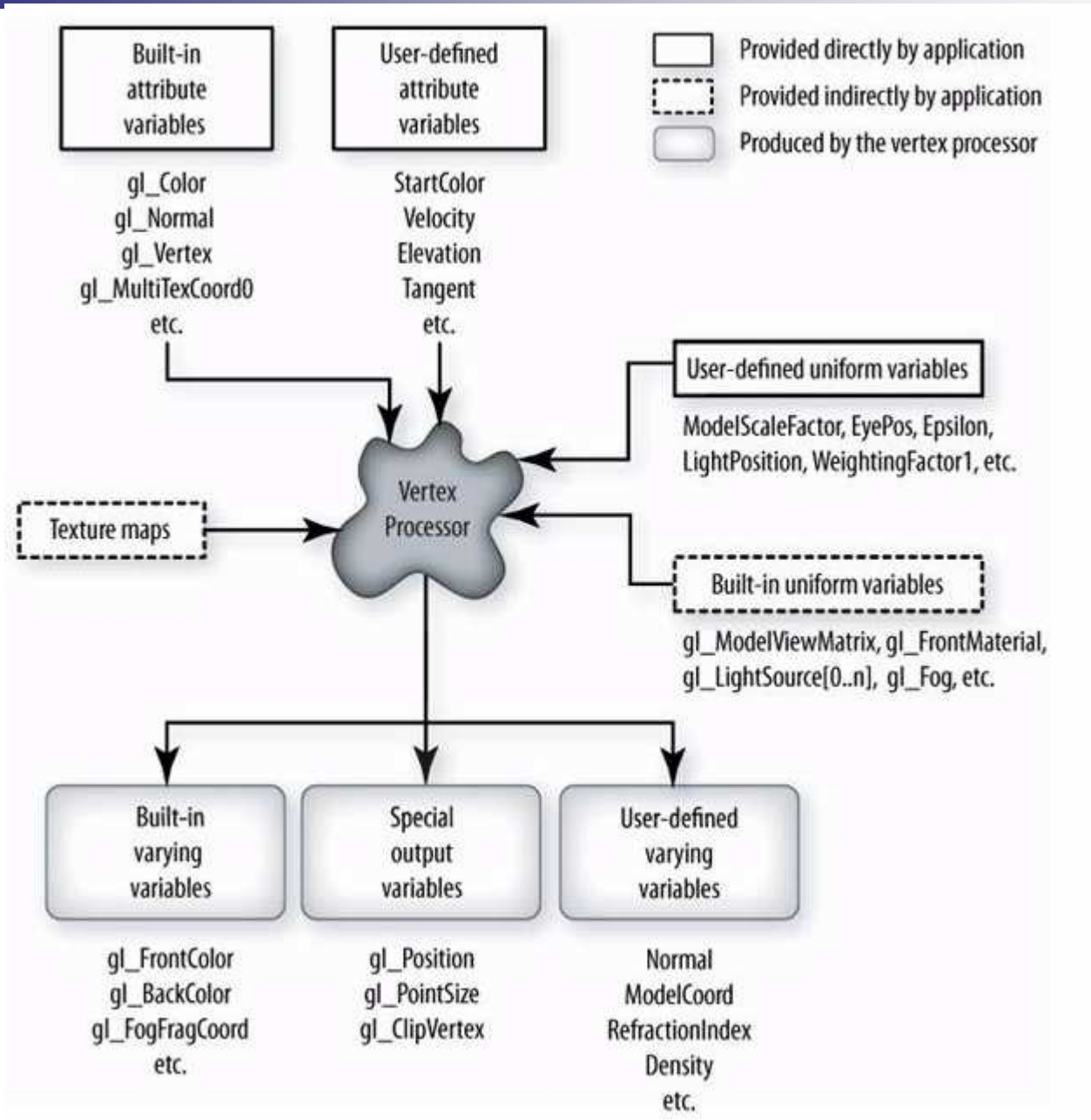
Fabio Ganovelli
fabio.ganovelli@gmail.com
a.a. 2006-2007

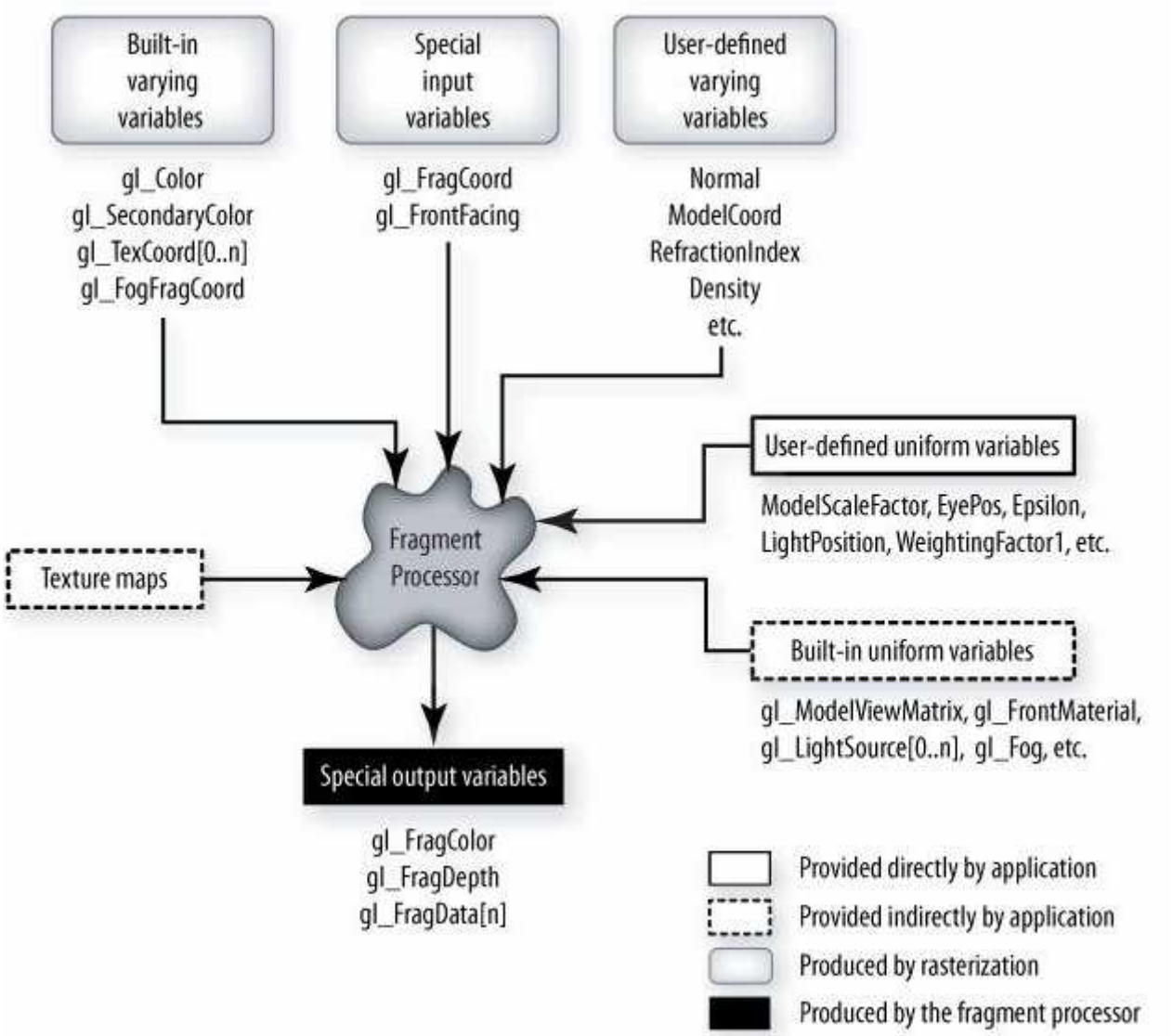
Vertex Processor & Fragment processor



Vertex Program (o vertex shader) : programma eseguito dal **vertex processor** che prende in input un **vertice** e calcola attributi che verranno interpolati nel triangolo;

Fragment Program (o pixel shader): programma che prende in input un **frammento** e calcola valori relativi al frammento (es: colore, depth)


Built-in attribute variables
User-defined attribute variables
Provided directly by application
Provided indirectly by application
Produced by the vertex processor
gl_Color
gl_Normal
gl_Vertex
gl_MultiTexCoord0
etc.
StartColor
Velocity
Elevation
Tangent
etc.
User-defined uniform variables
ModelScaleFactor, EyePos, Epsilon,
LightPosition, WeightingFactor1, etc.
Vertex Processor
Texture maps
Built-in uniform variables
gl_ModelViewMatrix, gl_FrontMaterial,
gl_LightSource[0..n], gl_Fog, etc.
Built-in varying variables
Special output variables
User-defined varying variables
gl_FrontColor
gl_BackColor
gl_FogFragCoord
etc.
gl_Position
gl_PointSize
gl_ClipVertex
Normal
ModelCoord
RefractionIndex
Density
etc.


Built-in varying variables
Special input variables
User-defined varying variables
gl_Color
gl_SecondaryColor
gl_TexCoord[0..n]
gl_FogFragCoord
gl_FragCoord
gl_FrontFacing
Normal
ModelCoord
RefractionIndex
Density
etc.
User-defined uniform variables
ModelScaleFactor, EyePos, Epsilon,
LightPosition, WeightingFactor1, etc.
Fragment Processor
Texture maps
Built-in uniform variables
gl_ModelViewMatrix, gl_FrontMaterial,
gl_LightSource[0..n], gl_Fog, etc.
Special output variables
gl_FragColor
gl_FragDepth
gl_FragData[n]
Provided directly by application
Provided indirectly by application
Produced by rasterization
Produced by the fragment processor

Linguaggi di shading

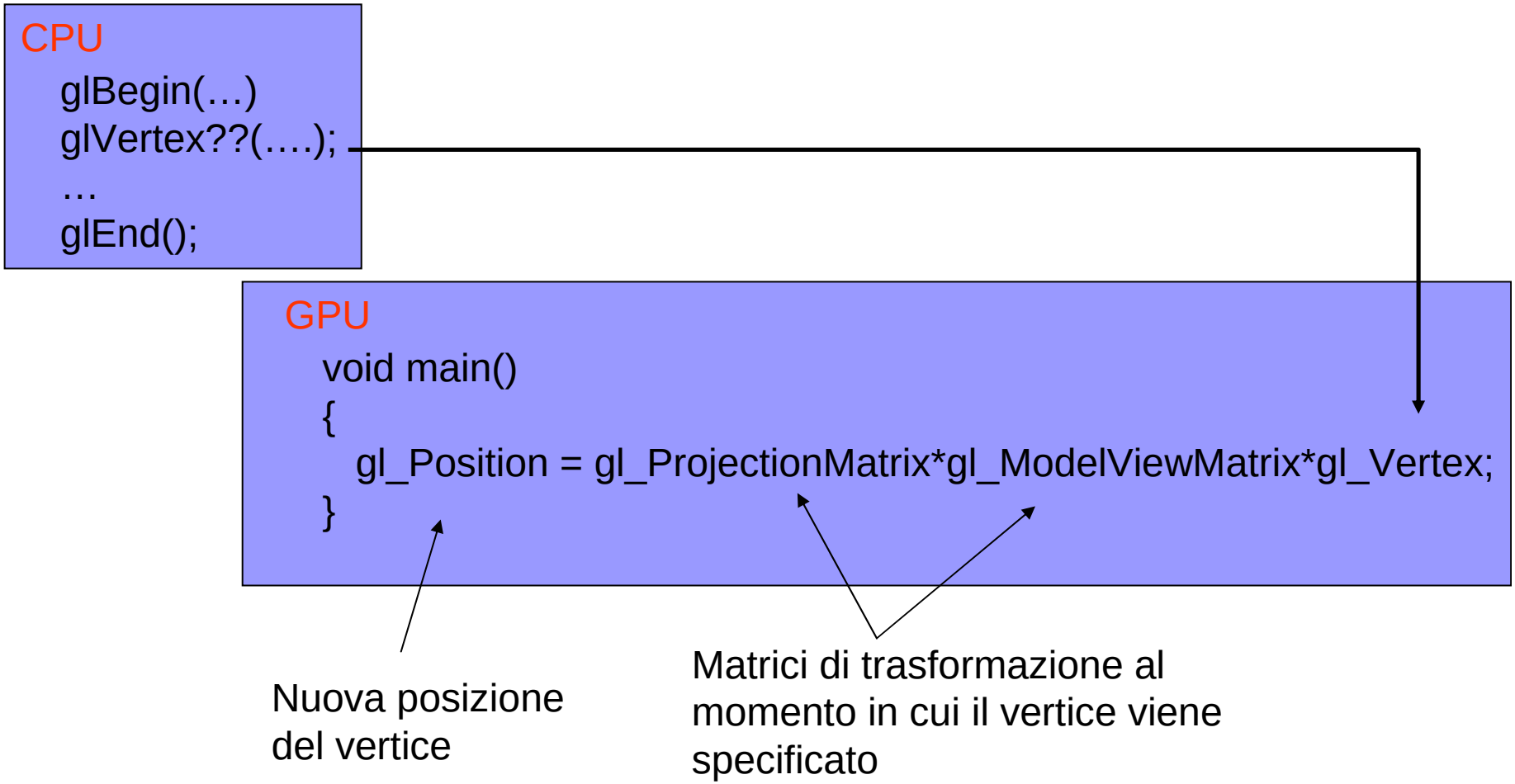
- HLSL (High Level Shader Language, Microsoft)
- OGLSL (OpenGL Shading Language)
- CG (C for Graphics, NVidia)

- HLSL e OGLSL equivalenti a meno della sintassi
- CG più ad alto livello e pensato per utilizzare sia HLSL che OGLSL

Cosa serve di sapere

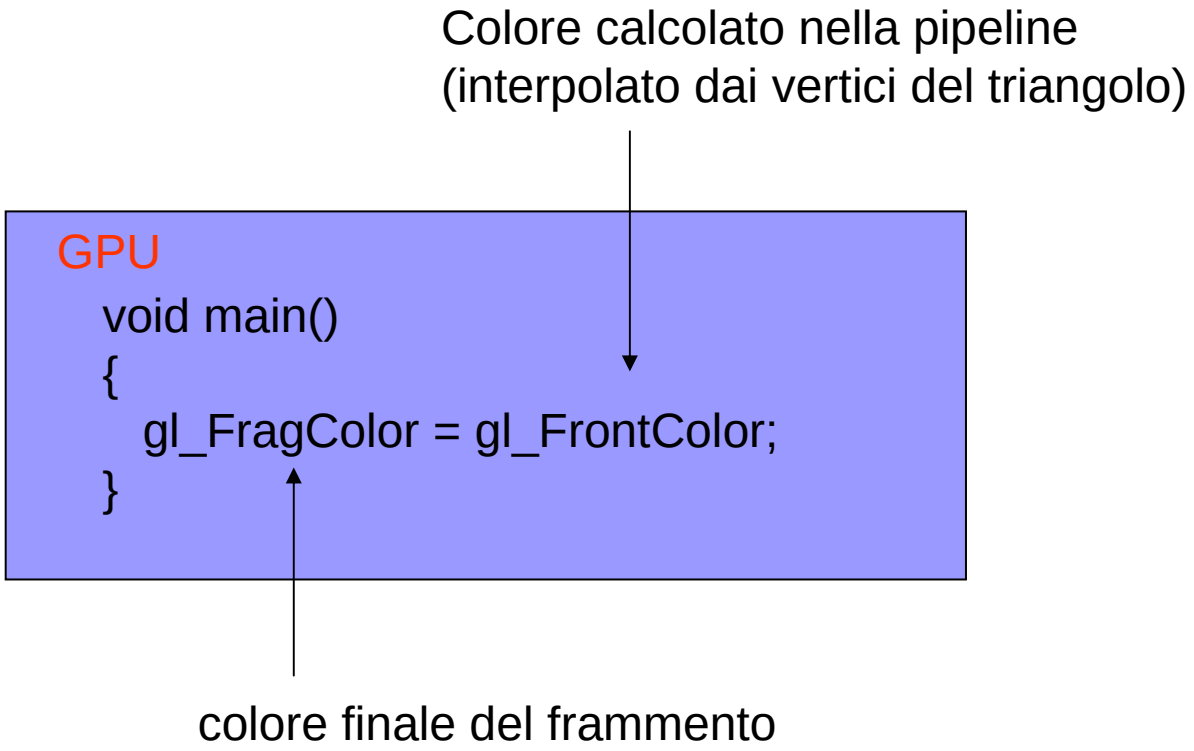
- Le istruzioni del linguaggio di shading (la sintassi è quella del c++)
- Le istruzioni per passare informazioni dal programma (in CPU) agli shader.

Il mio primo vertex program



Le parole riservate di GLSL sono composte dal prefisso “gl_” seguite dal nome con la prima lettera maiuscola

Il mio primo fragment shader



…ovvero un fragment program che non fa niente!

Cosa si perde?

- Il fragment shader è messo “dopo” la rasterizzazione. Il programma trova gli attributi interpolati
- Il vertex shader è “al posto” dello stadio geometrico, quindi usando un vertex shader “perdiamo” (cioè dobbiamo implementare nel vertex shader):
  - Le trasformazioni (posizioni, normali,coordinate textures)
  - Il lighting
- Il programma precedente implementa esclusivamente la trasformazione della posizione dei vertici

Il linguaggio GLSL

- Tipi primitivi:
  - Scalari:
    - float,int, bool
  - Vettori:
    - vec2,vec3,vec4 (di float)
    - Ivec2,ivec3,ivec4 (di interi)
    - bvec2,bvec3,bvec4 (di booleani)
  - Matrici:
    - mat2,mat3,mat3
  - Samplers: …dopo…
  - Array:
    - vec4 points[10];

Il linguaggio GLSL

- Qualificatori: il mezzo con cui si passano dati da programma utente e tra vertex e fragment prog.
  - **Uniform:** una variabile uniform può essere impostata dall'applicazione per "passare" un valore. Il passaggio per uniform è una operazione "costosa", non da fare per ogni vertice..
  - **Attribute:** come uniform ma specifica un valore associato al vertice alla pari della posizione, normale, coordinate texture, material etc..
  - **Varying:** una valore che viene calcolato nel vertex program e il cui valore (interpolato) viene letto dal fragment program

Application side

- Gli shader program sono compilati a run time, cioè la API (es. OpenGL) contiene un compilatore che compila il programma in binario per la GPU
- La sequenza di operazioni:
  - Creare l'oggetto shader
  - Compilare il vertex[fragment] program
  - "attaccare" vertex e fragment shader a uno shader program
  - Linkare lo shader program
  - Dire alla API che vogliamo usare lo shader program

```
void set_shaders( char *nameV,  char *nameF, GLuint & vs, GLuint &fs,  GLuint &pr){

fs= glCreateShader(GL_FRAGMENT_SHADER);
vs= glCreateShader(GL_VERTEX_SHADER);

const char * code;
if(nameV!=NULL){
        code = textFileRead(nameV);
        glShaderSource(vs, 1, &code,NULL);
        glCompileShader(vs);
        int errV;
        glGetShaderiv(vs,GL_COMPILE_STATUS,&errV);
        assert(errV==GL_TRUE);
}
if(nameF!=NULL){
        code = textFileRead(nameF);
        glShaderSource(fs, 1, &code,NULL);
        glCompileShader(fs);
        int errF;
        glGetShaderiv(fs,GL_COMPILE_STATUS,&errF);
        assert(errF==GL_TRUE);
}
pr = glCreateProgram();
if(nameV!=NULL)glAttachShader(pr,vs);
if(nameF!=NULL) glAttachShader(pr,fs);
glLinkProgram(pr);
}
```

textFileRead(..):
Utiliti per leggere un file di testo da disco.

Files:
textFile.cpp
textFile.h

Impostare variabili uniform

- Passare un valore **uniform:**
  - **glUniform{1234}{fi}[v](location, valore/i)**
- **Location:** la locazione di memoria in GPU relativo alla variabile uniform che vogliamo impostare. Come si trova?
- GLint glGetUniformLocation(GLuint program, const GLchar *name)
  dove name è il nome della variabile

Impostare variabili “attribute”

- Simile, ma le variabili attribute sono tenute su appositi registri numero da 1 a GL_MAX_VERTEX_ATTRIBS
- Assegnare un registro a un attributo:
  - void glBindAttribLocation(GLuint program, GLuint **index**, const GLchar *name)
- Impostare il valore di un attributo:
  - void glVertexAttrib… (GLuint **index**, GLfloat v0)

Accesso alle texture

- Dallo shader model 2.0 le textures sono accedibili sia dal vertex prog. che dal fragment prog.
- È quello che rende possibile implementare la fisica di un sistema interamente in GPU senza mai read back in memoria
- Le nostre (ATI Radeon 9250) supportano lo shader model 1.3, e le textures sono accedibili solo dal fragment

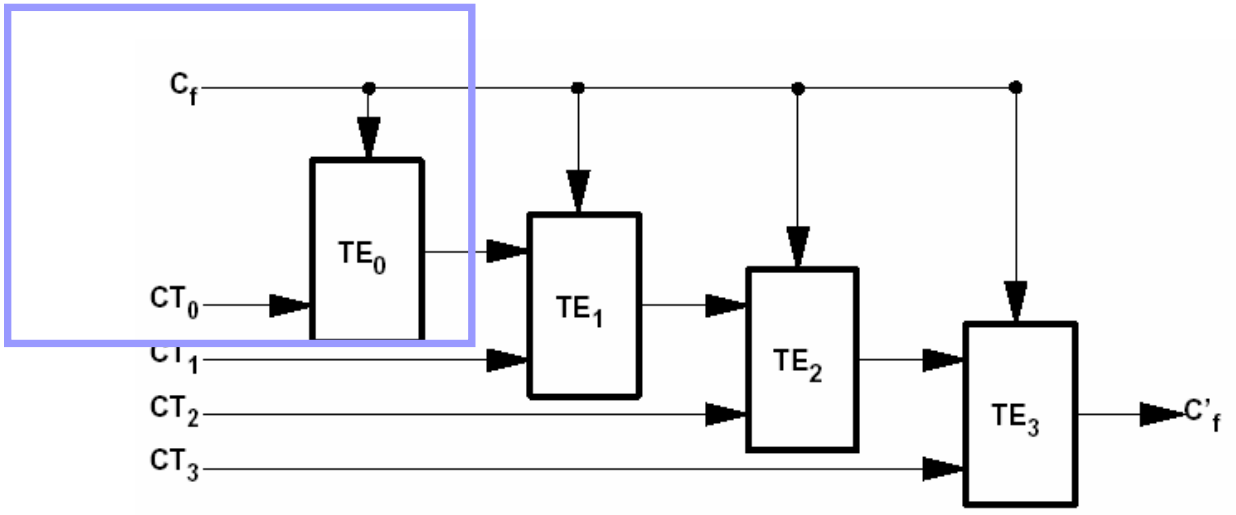
Accesso alle texture

- Funzioni di GLSL:
  - **vec4 texture[123]D** (sampler2D sampler, vec2 coord [, float bias])
- sampler[123]D specifica su quale **texture unit** viene letto il valore
- Cos’è una texture unit ?

Multitexturing

- Finora abbiamo assunto che per applicare due texture allo stesso frammento servisse cumulare due rendering della scena o comunque disegnare due volte i poligoni interessati e usare il blending
- Una volta era così
- Oggi le schede hanno una serie di Texture Unit, da 0 a GL_MAX_TEXTURE_UNITS in cascata.

Multitexturing



C_f = fragment primary color input to texturing

C'_f = fragment color output from texturing

CT_i = texture color from texture lookup *i*

TE_i = texture environment *i*

Multitexturing

- Ogni texture unit ha il proprio “stato” (parametri di sampling, environment, texture matrix)
- Con la funzione:
  - glActiveTexture(GL_TEXTUREi) i=0,.. Si dichiara che tutti comandi opengl seguenti useranno la texture “i” come unit
  - Il default è le texture 0.
- Un vertice può avere coordinate texture diverse per texture unit diverse:
  - gl**Multi**TexCoord**i**..(..)

Demo lab.08.shaders

- …