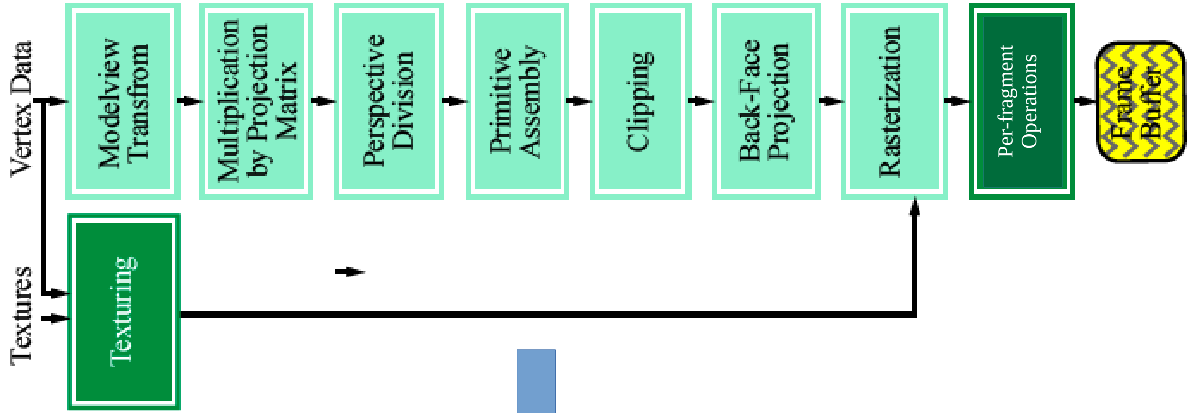
Computer Graphics
Lezione 19

Antonino Casile

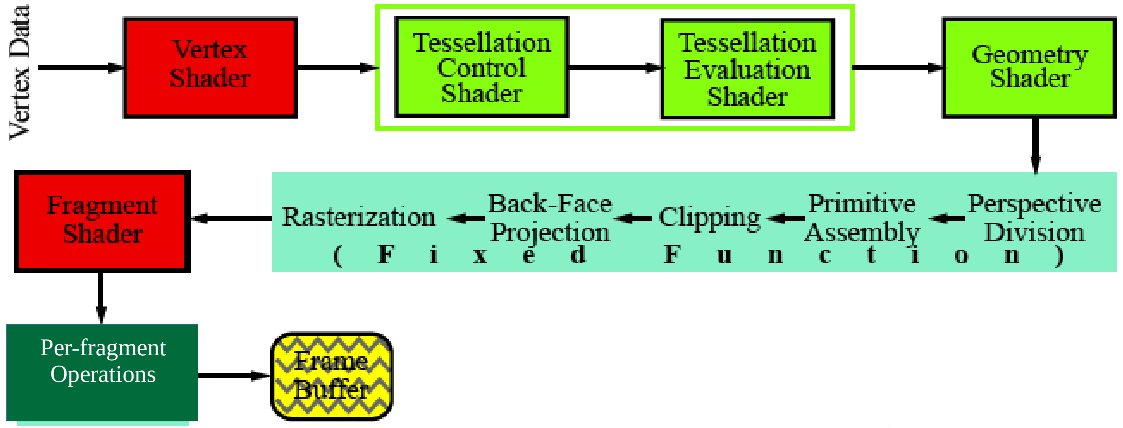
OpenGL4.3 - GLSL

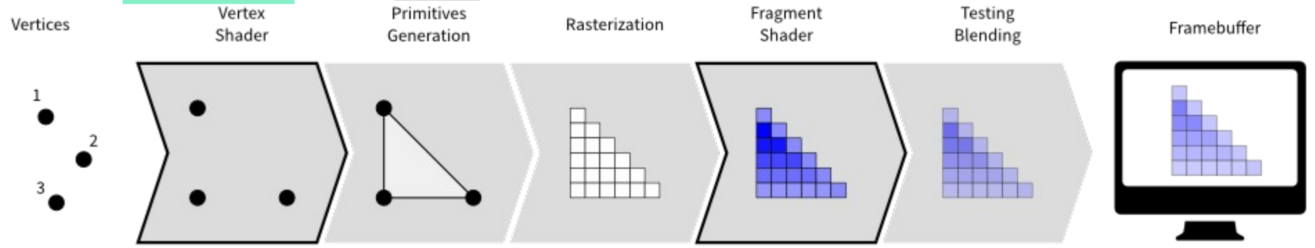
Shaders

Fixed functionality pipeline

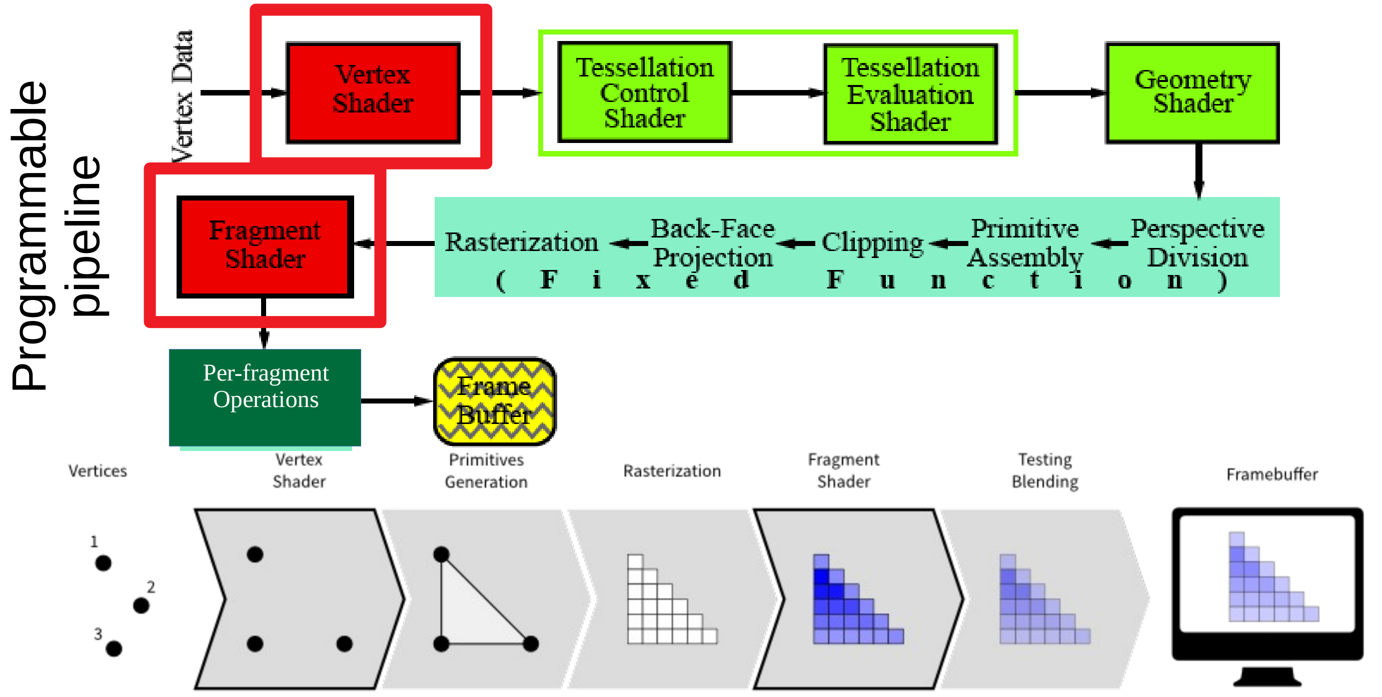


Programmable pipeline





Shaders



- **Vertex shader**: Viene applicato ad ogni vertice della scena processandone le caratteristiche (posizione, colore, normali, texture coordinates, etc.). Al minimo: deve restituirne le coordinate xyzw (potenzialmente dopo la moltiplicazione per le matrici **modelview** e **di proiezione**).
- **Fragment shader**: Viene applicato da ogni fragment della scena scartandolo (e.g. se e' occluso) o settandone il colore.

OpenGL 4.3

Cambiamenti piu' significativi

- Non c'e' piu' un "immediate mode" → Nessun comando `glBegin()-glEnd()`
- Sono ammessi comandi di disegno solo di tipo "retained mode" → `glDraw*()-glMultiDraw*()`
- Tutti i dati devono essere <u>obbligatoriamente</u> allocati mediante buffer objects → Devono risiedere nella memoria della GPU
- Nessun comando di trasformazione modelview → Niente piu' `glTranslatef()`, `glRotatef()`, `glScalef()`, `gluLookAt()` → L'implementazione delle trasformazioni spaziali e' a carico del programmatore
- Nessun comando per l'implementazione di lightining → Niente piu' `glLight*()`, `glMaterial*()` e `glShadeModel()` → L'implementazione del lightning e dello shading sono anch'essi a carico del programmatore
- Un nuovo linguaggio C-like, GLSL, per implementare tutte queste funzionalita' negli shaders.

GLSL

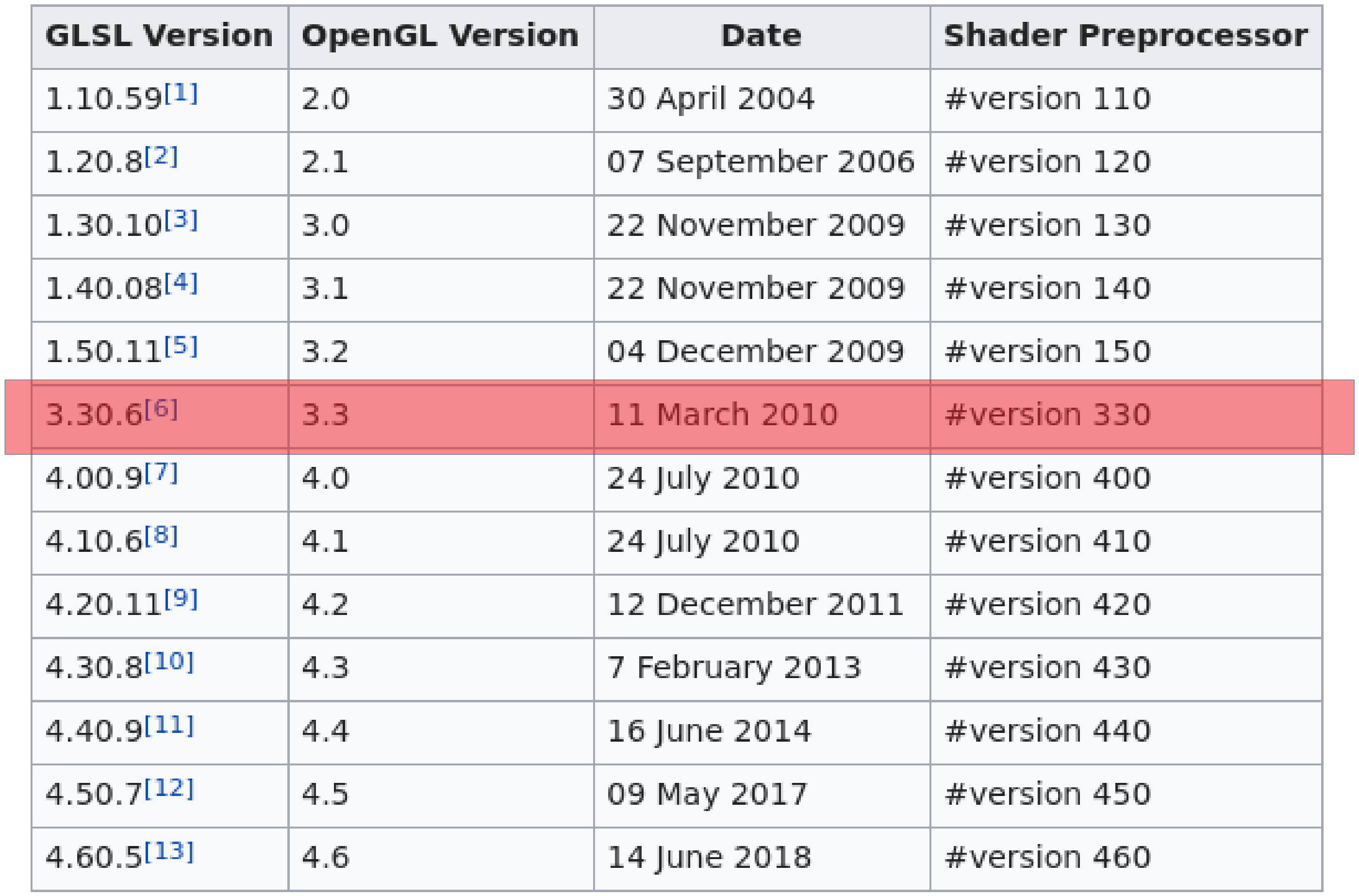
Steps e motivazioni per lo sviluppo di GLSL

- Con il passare del tempo i processori delle schede grafiche acquisivano la capacita’ di fare elaborazioni sempre piu’ complesse
- Originariamente gli shaders venivano sviluppati in linguaggio assembly → Poca compatibilita’ fra schede grafiche di modelli e marche diverse
- Necessita’ di avere una “piattaforma di programmazione” (API) uniforme.
  - Astrazione dall’hardware per mezzo di drivers sempre piu’ complessi → Attualmente nessuno sviluppatore di schede grafiche espone al programmatore i dettagli hardware delle proprie schede ma solo le API dei drivers
  - Creazione di un linguaggio standard di programmazione degli shaders → GLSL
- GLSL e’ cross platform → e.g. Linux, Windows e MacOS
- GLSL da’ agli sviluppatori hardware la flessibilita’ di cambiare l’hardware sottostante e a quelli software la tranquillita’ che il codice che sviluppano gira su ogni scheda OpenGL compatible.

https://en.wikipedia.org/wiki/OpenGL_Shading_Language

GLSL

Tappe dello sviluppo di GLSL



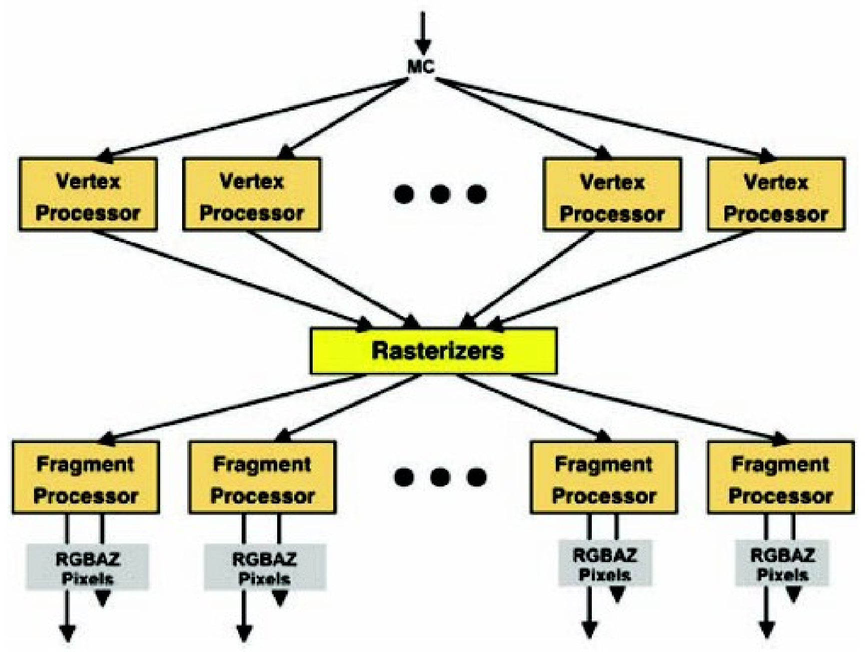
GLSL Version	OpenGL Version	Date	Shader Preprocessor
1.10.59[1]	2.0	30 April 2004	#version 110
1.20.8[2]	2.1	07 September 2006	#version 120
1.30.10[3]	3.0	22 November 2009	#version 130
1.40.08[4]	3.1	22 November 2009	#version 140
1.50.11[5]	3.2	04 December 2009	#version 150
3.30.6[6]	3.3	11 March 2010	#version 330
4.00.9[7]	4.0	24 July 2010	#version 400
4.10.6[8]	4.1	24 July 2010	#version 410
4.20.11[9]	4.2	12 December 2011	#version 420
4.30.8[10]	4.3	7 February 2013	#version 430
4.40.9[11]	4.4	16 June 2014	#version 440
4.50.7[12]	4.5	09 May 2017	#version 450
4.60.5[13]	4.6	14 June 2018	#version 460

https://en.wikipedia.org/wiki/OpenGL_Shading_Language

GLSL

Motivazione: GLSL e' stato sviluppato per trarre vantaggio dei diversi livelli di parallelismo presenti nelle moderne GPU.

- **Device-Level Parallelism** – multiple processors or multiple graphics cards can exist in the same system
- **Core-Level Parallelism** – each processor typically has multiple cores that are capable of independent execution
- **Thread-Level Parallelism** – each core can run multiple threads, that is, can have multiple instruction streams
- **Data-Level Parallelism** – many instructions can act on multiple data elements at once



GLSL

Differenze con il C

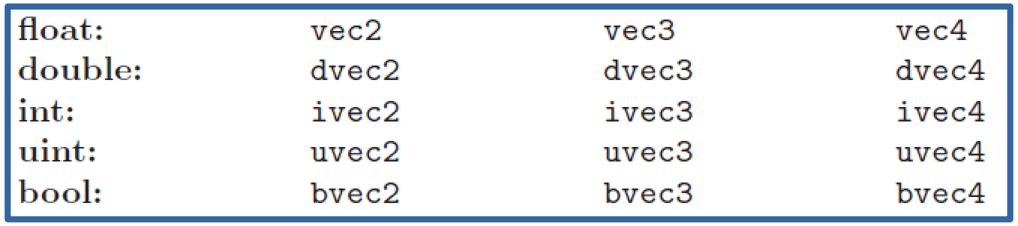
- Operatori e funzioni predefinite addizionali
- Gli shaders sono organizzati in un pipeline di programmazione
- Il passaggio di parametri fra shaders e’ esplicitamente definito dichiarando una variabile come *in* o *out*
- I parametri di una funzione sono passati per value-return anziche’ come valore
- Alcune caratteristiche dei linguaggi general-purpose sono mancanti:
  - Niente char, char *, tipo di dato stringa, funzioni per manipolare stringhe e <u>niente puntatori</u>
  - Niente operatore sizeof(). Ci sono gia’ i costruttori di matrici e array di tutte le dimensioni necessarie
  - Sono supportate solo conversioni di tipo esplicite per mezzo di costruttori (e.g. no (float)var ma float(var))

GLSL

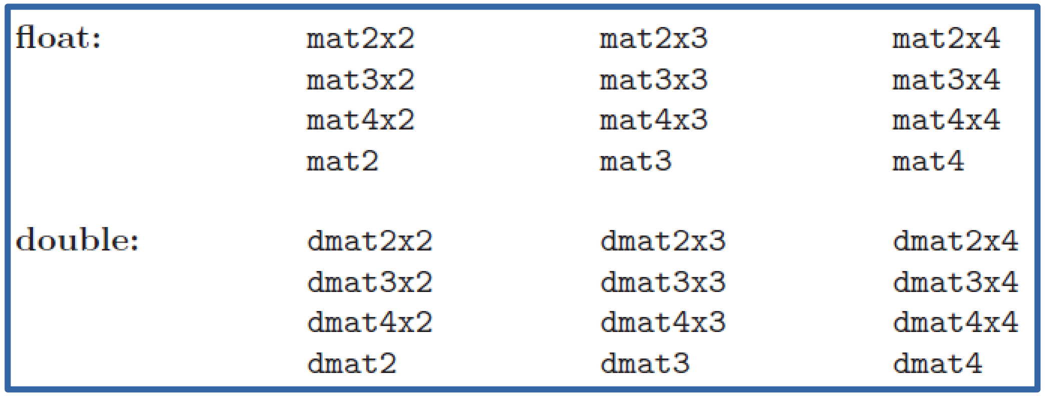
Tipi di dato base

`float`	32-bit floating point number
`double`	64-bit floating point number
`int`	signed 32-bit integer
`uint`	unsigned 32-bit integer
`bool`	Boolean (true/false)

Tipi di dato composti



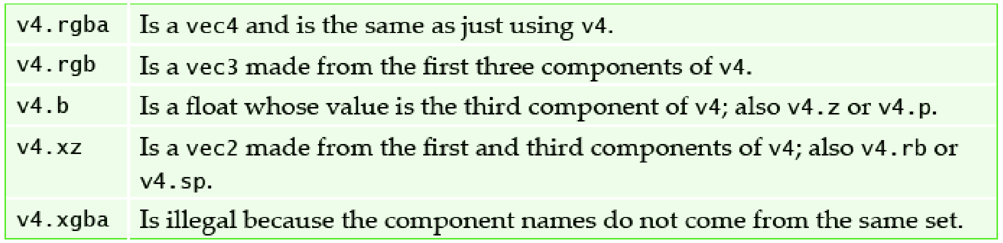
float:	vec2	vec3	vec4
double:	dvec2	dvec3	dvec4
int:	ivec2	ivec3	ivec4
uint:	uvec2	uvec3	uvec4
bool:	bvec2	bvec3	bvec4



float:	mat2x2	mat2x3	mat2x4
	mat3x2	mat3x3	mat3x4
	mat4x2	mat4x3	mat4x4
	mat2	mat3	mat4
double:	dmat2x2	dmat2x3	dmat2x4
	dmat3x2	dmat3x3	dmat3x4
	dmat4x2	dmat4x3	dmat4x4
	dmat2	dmat3	dmat4

Le componenti dei vettori si accedono mediante i campi:
`(x, y, z, w)` → componenti di coordinate
`(r, g, b, a)` → componenti colori
`(s, t, p, q)` → texture coordinates

Si puo’ utilizzare il tipo di “indirizzamento” che si vuole basta che i simboli appartengano allo stessi insieme.

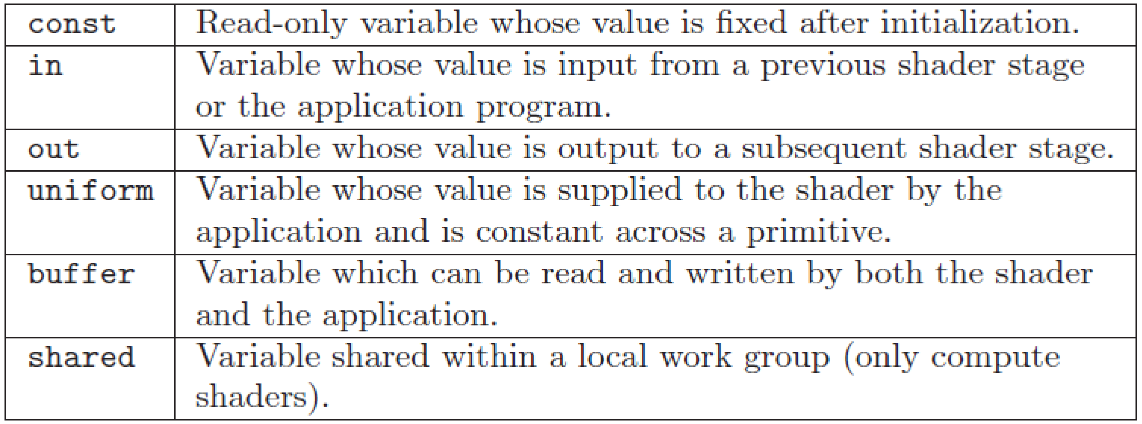


`v4.rgba`	Is a vec4 and is the same as just using v4.
`v4.rgb`	Is a vec3 made from the first three components of v4.
`v4.b`	Is a float whose value is the third component of v4; also `v4.z` or `v4.p`.
`v4.xz`	Is a vec2 made from the first and third components of v4; also `v4.rb` or `v4.sp`.
`v4.xgba`	Is illegal because the component names do not come from the same set.

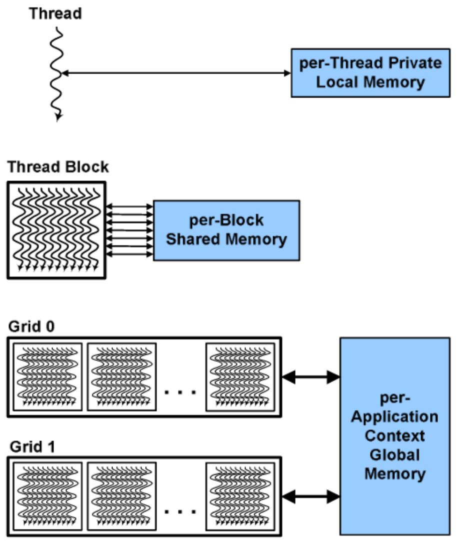
GLSL

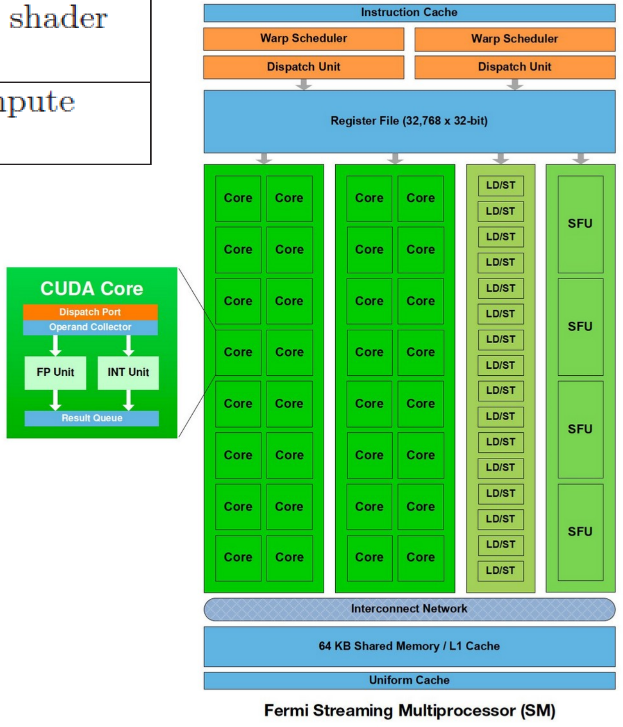
Storage qualifiers

La dichiazione di una variabile puo’ essere preceduta da al piu’ uno dei seguenti qualifiers:



const	Read-only variable whose value is fixed after initialization.
in	Variable whose value is input from a previous shader stage or the application program.
out	Variable whose value is output to a subsequent shader stage.
uniform	Variable whose value is supplied to the shader by the application and is constant across a primitive.
buffer	Variable which can be read and written by both the shader and the application.
shared	Variable shared within a local work group (only compute shaders).


Thread
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
Grid 1
per-Application Context Global Memory



Reading Material: NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

GLSL

Tipi di dato composti – Costruttori e indirizzamento

`vec3(float, float, float)`	Initializes each component of a vector with the explicit floats provided.
`vec4(ivec4)`	Makes a `vec4` with component-wise conversion.
`vec2(float)`	Initializes a `vec2` with the float value in each position.
`ivec3(int, int, int)`	Initializes an `ivec3` with three ints.
`bvec4(int,int,float,float)`	Performs four Boolean conversions.
`vec2(vec3)`	Drops the third component of a `vec3`.
`vec3(vec4)`	Drops the fourth component of a `vec4`.

`vec3(vec2, float)`	`vec3.xy = vec2`
	`vec3.z = float`

`vec3(float, vec2)`	`vec3.x = float`
	`vec3.yz = vec2`

`vec4(vec3, float)`	`vec4.xyz = vec3`
	`vec4.w = float`

`vec4(float, vec3)`	`vec4.x = float`
	`vec4.yzw = vec3`

`vec4(vec2a, vec2b)`	`vec4.xy = vec2a`
	`vec4.zw = vec2b`

`mat2(vec2, vec2)`
`mat3(vec3, vec3, vec3)`
`mat4(vec4, vec4, vec4, vec4)`
`mat3x2(vec2, vec2, vec2)`
	Each matrix is filled using one column per argument.

`mat2(float, float, float, float)`	Rows are first column and second column, respectively.

`mat3(float, float, float,`
`float, float, float,`
`float, float, float)`
	Rows are first column, second column, and third column, respectively.

`mat4(float,float,float,float,`
`float,float,float,float,`
`float,float,float,float,`
`float,float,float,float)`
	Rows are first column, second column, third column, and fourth column, respectively.

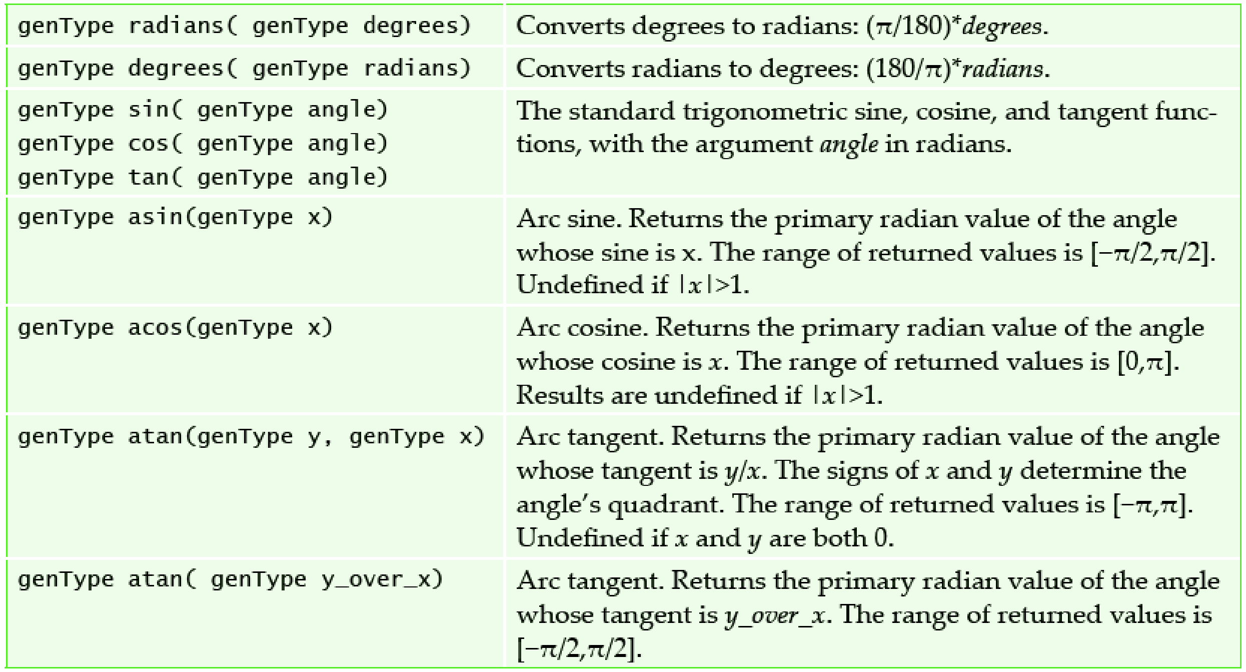
`mat2x3(vec2, float,`
`vec2, float)`
	Rows are first column and second column, respectively.

`mat3x3(mat4x4)`	Uses the upper-left 3×3 submatrix of the `mat4x4` matrix.
`mat2x3(mat4x2)`	Takes the upper-left 2×2 submatrix of the `mat4x2`, and sets the last column to `vec2(0.)`.
`mat4x4(mat3x3)`	Puts the `mat3x3` matrix in the upper-left submatrix and sets the lower right component to 1 and the rest to 0.

```
mat4 m;
m[i] refers to the i-th columns
m[1] = vec4(2.0); // sets the second column to all 2.0
m[0][0] = 1.0; // sets the upper left element to 1.0
m[2][3] = 2.0; // sets the 4th element of the third column to 2.0
```

GLSL

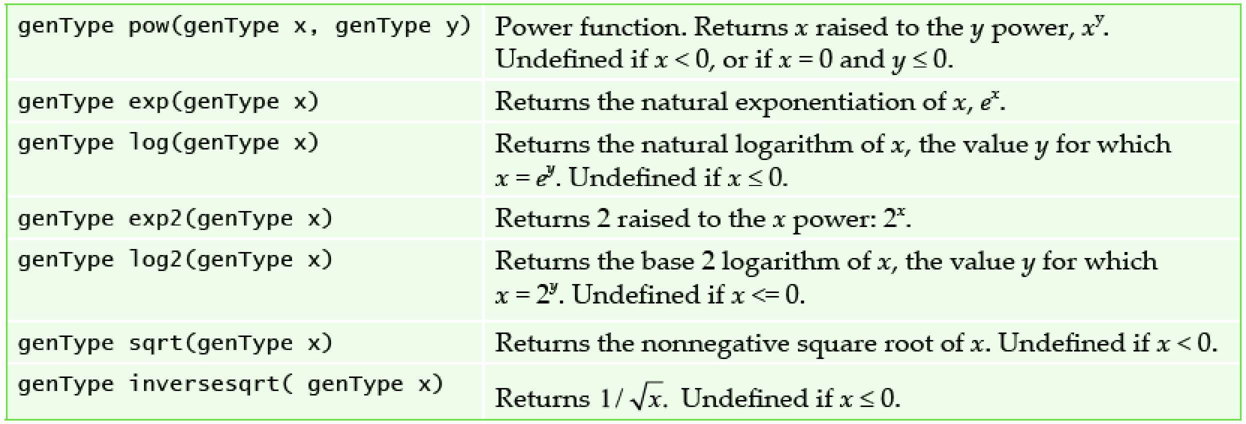
Tipi di dato composti – Funzioni trigonometriche



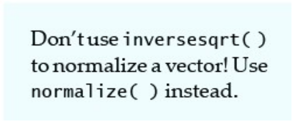
`genType radians( genType degrees)`	Converts degrees to radians: $(\pi/180)$**degrees*.
`genType degrees( genType radians)`	Converts radians to degrees: $(180/\pi)$**radians*.

`genType sin( genType angle)`
`genType cos( genType angle)`
`genType tan( genType angle)`
	The standard trigonometric sine, cosine, and tangent functions, with the argument *angle* in radians.

`genType asin(genType x)`	Arc sine. Returns the primary radian value of the angle whose sine is x. The range of returned values is $[-\pi/2,\pi/2]$. Undefined if $\lvert x\rvert>1$.
`genType acos(genType x)`	Arc cosine. Returns the primary radian value of the angle whose cosine is x. The range of returned values is $[0,\pi]$. Results are undefined if $\lvert x\rvert>1$.
`genType atan(genType y, genType x)`	Arc tangent. Returns the primary radian value of the angle whose tangent is y/x. The signs of x and y determine the angle's quadrant. The range of returned values is $[-\pi,\pi]$. Undefined if x and y are both 0.
`genType atan( genType y_over_x)`	Arc tangent. Returns the primary radian value of the angle whose tangent is *y_over_x*. The range of returned values is $[-\pi/2,\pi/2]$.



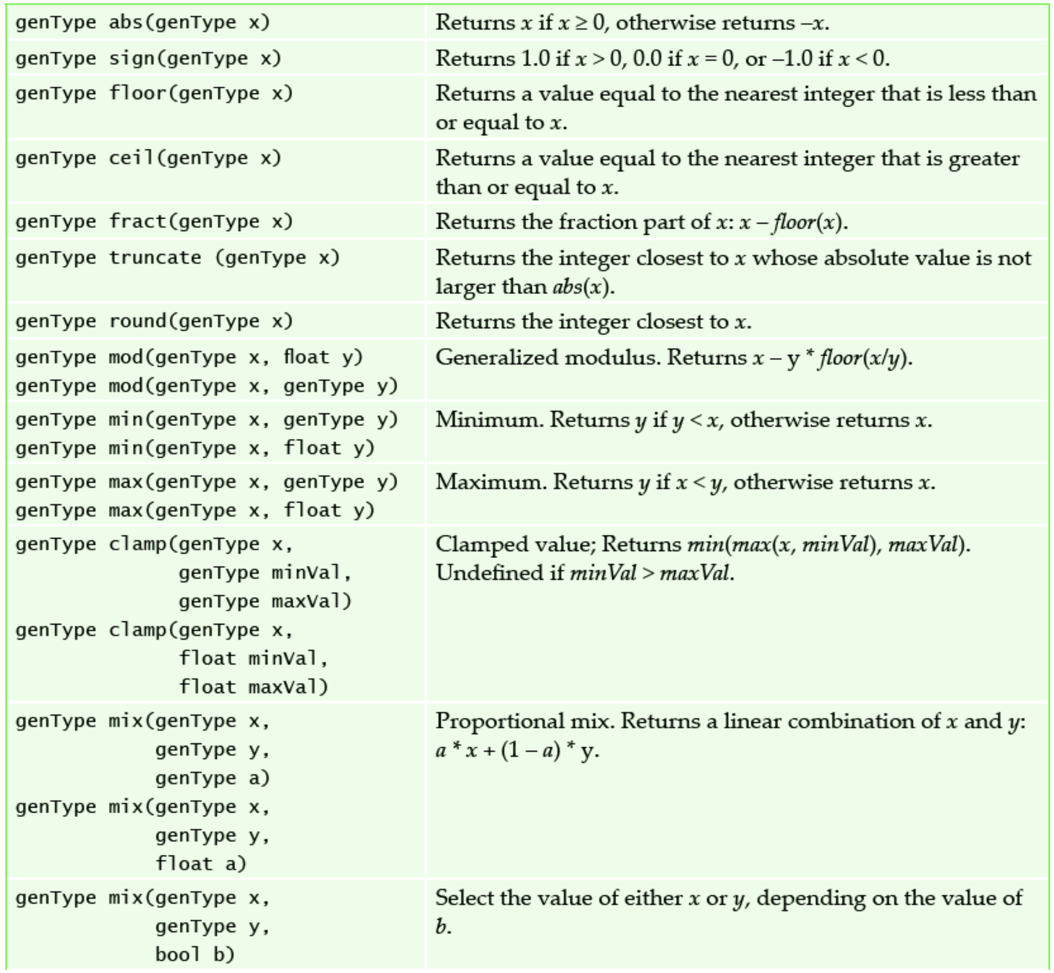
`genType pow(genType x, genType y)`	Power function. Returns x raised to the y power, x^y. Undefined if $x < 0$, or if $x = 0$ and $y \leq 0$.
`genType exp(genType x)`	Returns the natural exponentiation of x, e^x.
`genType log(genType x)`	Returns the natural logarithm of x, the value y for which $x = e^y$. Undefined if $x \leq 0$.
`genType exp2(genType x)`	Returns 2 raised to the x power: 2^x.
`genType log2(genType x)`	Returns the base 2 logarithm of x, the value y for which $x = 2^y$. Undefined if $x <= 0$.
`genType sqrt(genType x)`	Returns the nonnegative square root of x. Undefined if $x < 0$.
`genType inversesqrt( genType x)`	Returns $1/\sqrt{x}$. Undefined if $x \leq 0$.



Don't use `inversesqrt()` to normalize a vector! Use `normalize()` instead.

GLSL

Tipi di dato composti – Operazioni aritmetiche



`genType abs(genType x)`	Returns x if $x \geq 0$, otherwise returns $-x$.
`genType sign(genType x)`	Returns 1.0 if $x > 0$, 0.0 if $x = 0$, or –1.0 if $x < 0$.
`genType floor(genType x)`	Returns a value equal to the nearest integer that is less than or equal to x.
`genType ceil(genType x)`	Returns a value equal to the nearest integer that is greater than or equal to x.
`genType fract(genType x)`	Returns the fraction part of x: $x - floor(x)$.
`genType truncate (genType x)`	Returns the integer closest to x whose absolute value is not larger than $abs(x)$.
`genType round(genType x)`	Returns the integer closest to x.

`genType mod(genType x, float y)`
`genType mod(genType x, genType y)`
	Generalized modulus. Returns $x - y * floor(x/y)$.

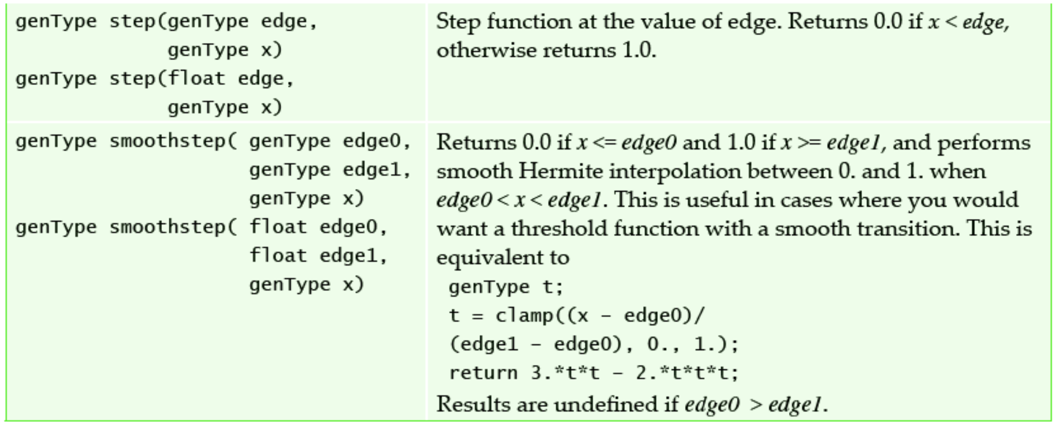
`genType min(genType x, genType y)`
`genType min(genType x, float y)`
	Minimum. Returns y if $y < x$, otherwise returns x.

`genType max(genType x, genType y)`
`genType max(genType x, float y)`
	Maximum. Returns y if $x < y$, otherwise returns x.

`genType clamp(genType x,`
`genType minVal,`
`genType maxVal)`
`genType clamp(genType x,`
`float minVal,`
`float maxVal)`
	Clamped value; Returns $min(max(x, minVal), maxVal)$. Undefined if $minVal > maxVal$.

`genType mix(genType x,`
`genType y,`
`genType a)`
`genType mix(genType x,`
`genType y,`
`float a)`
	Proportional mix. Returns a linear combination of x and y: $a * x + (1 - a) * y$.

`genType mix(genType x,`
`genType y,`
`bool b)`
	Select the value of either x or y, depending on the value of b.



`genType step(genType edge,`
`genType x)`
`genType step(float edge,`
`genType x)`
	Step function at the value of edge. Returns 0.0 if $x < edge$, otherwise returns 1.0.

`genType smoothstep( genType edge0,`
`genType edge1,`
`genType x)`
`genType smoothstep( float edge0,`
`float edge1,`
`genType x)`
	Returns 0.0 if $x <= edge0$ and 1.0 if $x >= edge1$, and performs smooth Hermite interpolation between 0. and 1. when $edge0 < x < edge1$. This is useful in cases where you would want a threshold function with a smooth transition. This is equivalent to
	`genType t;`
	`t = clamp((x - edge0)/`
	`(edge1 - edge0), 0., 1.);`
	`return 3.*t*t - 2.*t*t*t;`
	Results are undefined if $edge0 > edge1$.

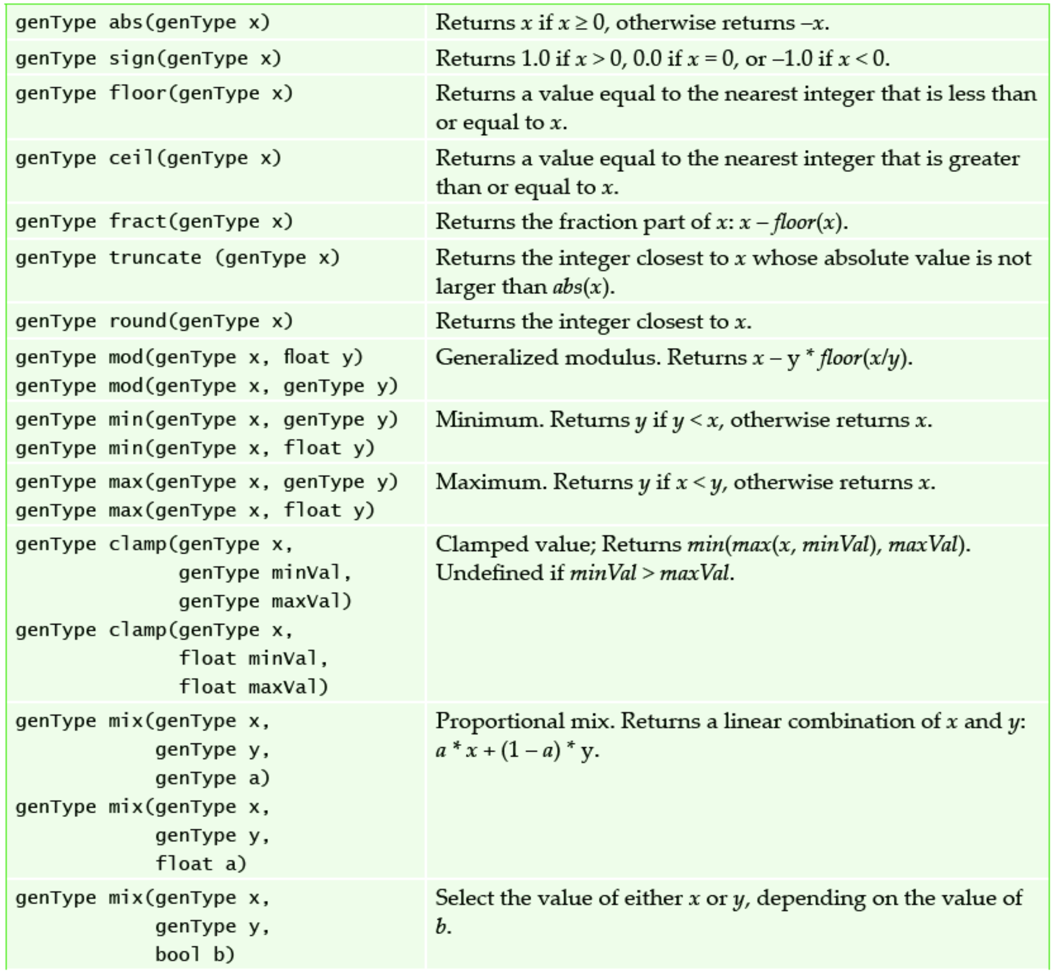
GLSL

Tipi di dato composti – Operazioni algebriche

- Su vettori e matrici sono definite le normali operazioni algebriche con le seguenti eccezioni:
  - +, -, / agiscono component-wise quando:
    - Un operando e’ scalare e l’altro e’ o un vettore o una matrice
    - Entrambi gli operandi sono o un vettore o una matrice
  - * agisce component-wise quando entrambi sono vettori → u*v **non** e’ prodotto scalare, per questo si usa la funzione `dot()`

GLSL

Tipi di dato composti – Operazioni “varie”



`genType abs(genType x)`	Returns x if $x \geq 0$, otherwise returns $-x$.
`genType sign(genType x)`	Returns 1.0 if $x > 0$, 0.0 if $x = 0$, or –1.0 if $x < 0$.
`genType floor(genType x)`	Returns a value equal to the nearest integer that is less than or equal to x.
`genType ceil(genType x)`	Returns a value equal to the nearest integer that is greater than or equal to x.
`genType fract(genType x)`	Returns the fraction part of x: $x - floor(x)$.
`genType truncate (genType x)`	Returns the integer closest to x whose absolute value is not larger than $abs(x)$.
`genType round(genType x)`	Returns the integer closest to x.

`genType mod(genType x, float y)`
`genType mod(genType x, genType y)`
	Generalized modulus. Returns $x - y * floor(x/y)$.

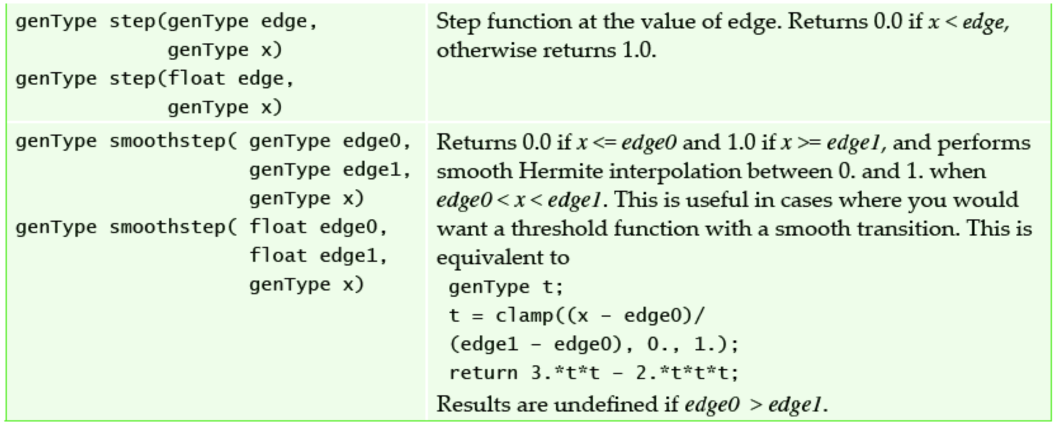
`genType min(genType x, genType y)`
`genType min(genType x, float y)`
	Minimum. Returns y if $y < x$, otherwise returns x.

`genType max(genType x, genType y)`
`genType max(genType x, float y)`
	Maximum. Returns y if $x < y$, otherwise returns x.

`genType clamp(genType x, genType minVal, genType maxVal)`
`genType clamp(genType x, float minVal, float maxVal)`
	Clamped value; Returns $min(max(x, minVal), maxVal)$. Undefined if $minVal > maxVal$.

`genType mix(genType x, genType y, genType a)`
`genType mix(genType x, genType y, float a)`
	Proportional mix. Returns a linear combination of x and y: $a * x + (1 - a) * y$.

`genType mix(genType x, genType y, bool b)`	Select the value of either x or y, depending on the value of b.

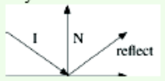


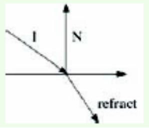
`genType step(genType edge, genType x)`
`genType step(float edge, genType x)`
	Step function at the value of edge. Returns 0.0 if $x < edge$, otherwise returns 1.0.

`genType smoothstep( genType edge0, genType edge1, genType x)`
`genType smoothstep( float edge0, float edge1, genType x)`
	Returns 0.0 if $x <= edge0$ and 1.0 if $x >= edge1$, and performs smooth Hermite interpolation between 0. and 1. when $edge0 < x < edge1$. This is useful in cases where you would want a threshold function with a smooth transition. This is equivalent to
	`genType t;`
	`t = clamp((x - edge0)/`
	`(edge1 - edge0), 0., 1.);`
	`return 3.*t*t - 2.*t*t*t;`
	Results are undefined if $edge0 > edge1$.

GLSL

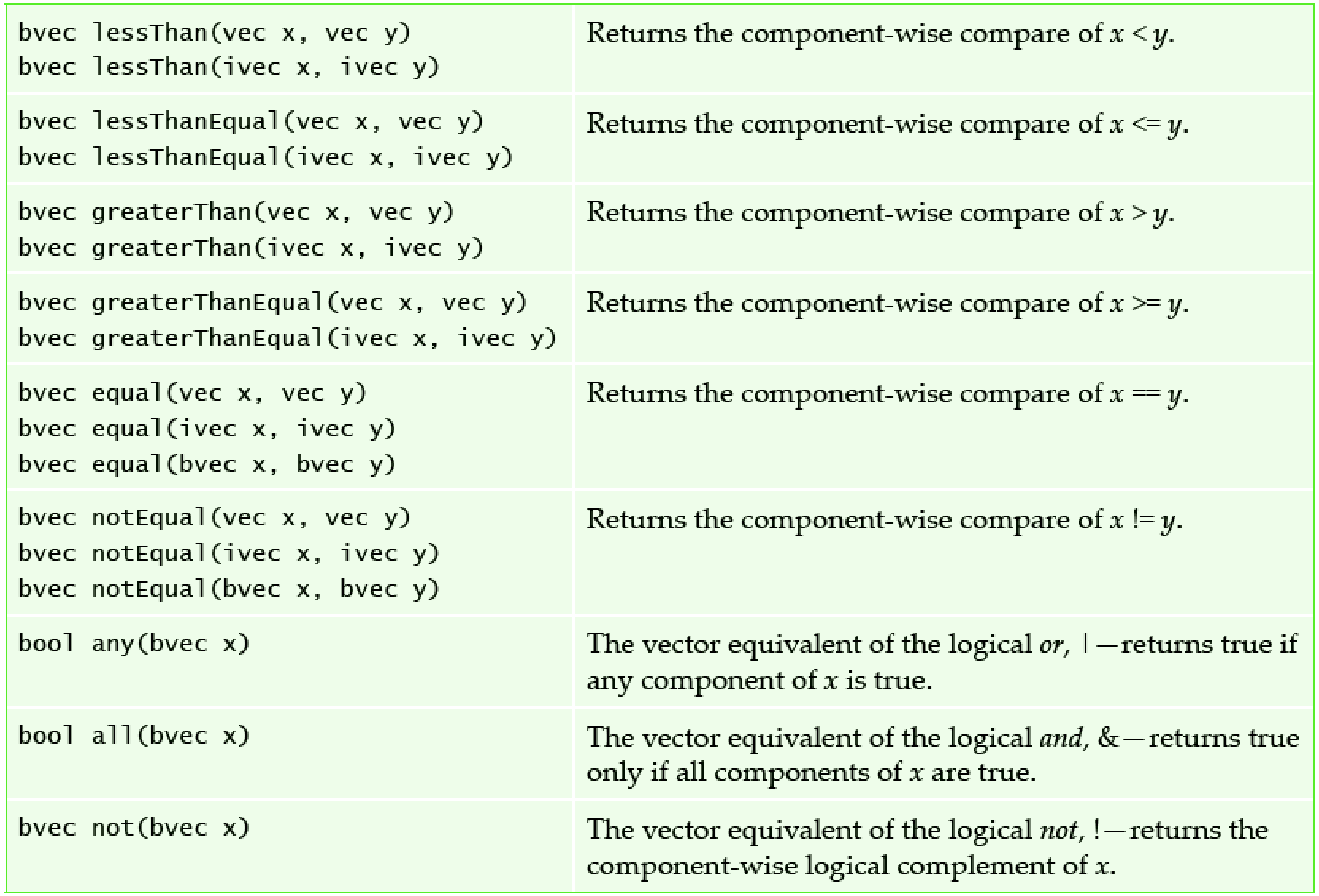
Tipi di dato composti – Operazioni geometriche

`float length(genType x)`	Returns the length of the vector *x*, $\sqrt{(x[0]^2 + x[1]^2 + \ldots)}$.
`float distance(genType p0, genType p1)`	Returns the distance between *p0* and *p1*: `length(p0 - p1)`.
`float dot(genType x, genType y)`	Returns the dot product of *x* and *y*: `x[0]*y[0]+x[1]*y[1]+...` .
`vec3 cross(vec3 x, vec3 y)`	Returns the cross product of *x* and *y*, $\begin{Bmatrix} x[1]y[2] - y[1]x[2] \\ x[2]y[0] - y[2]x[0] \\ x[0]y[1] - y[0]x[1] \end{Bmatrix}$.
`genType normalize(genType x)`	Returns a vector in the same direction as *x*, but with a length of 1, or $\frac{x}{\lVert x \rVert}$.
`genType faceforward(genType N, genType I, genType Nref)`	Make *N* face in the direction of *Nref*. If $dot(Nref, I) < 0$ return *N*, otherwise return –*N*.
`genType reflect(genType I, genType N)`	For the incident vector *I* and surface orientation *N*, returns the reflection direction: $I - 2 * dot(N,I) * N$. The normal vector *N* must already be normalized in order to achieve the correct result. 

`genType refract(genType I, genType N, float eta)`	For the incident vector *I* and surface normal *N*, and the ratio of indices of refraction *eta*, return the refraction vector. The result is computed by
	`k=1.0-eta*eta*(1.0-dot(N,I)*dot(N,I))`
	`if (k < 0.0)`
	`   return genType(0.0)`
	`else`
	`   return eta*I-(eta*dot(N,I)+sqrt(k))*N.`
	The incident vector *I* and the normal vector *N* must already be normalized in order to achieve the correct result. 

GLSL

Tipi di dato composti – Operazioni relazionali



`bvec lessThan(vec x, vec y)`
`bvec lessThan(ivec x, ivec y)`
	Returns the component-wise compare of $x < y$.

`bvec lessThanEqual(vec x, vec y)`
`bvec lessThanEqual(ivec x, ivec y)`
	Returns the component-wise compare of $x <= y$.

`bvec greaterThan(vec x, vec y)`
`bvec greaterThan(ivec x, ivec y)`
	Returns the component-wise compare of $x > y$.

`bvec greaterThanEqual(vec x, vec y)`
`bvec greaterThanEqual(ivec x, ivec y)`
	Returns the component-wise compare of $x >= y$.

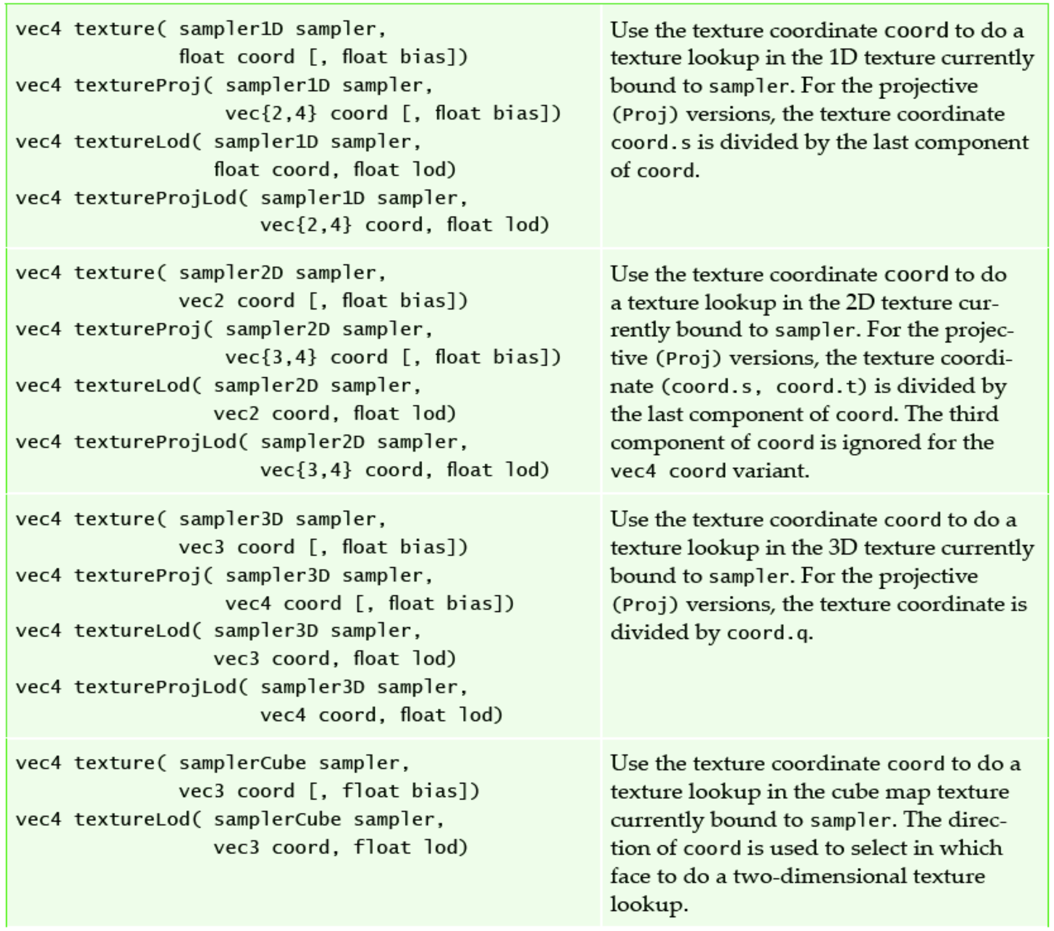
`bvec equal(vec x, vec y)`
`bvec equal(ivec x, ivec y)`
`bvec equal(bvec x, bvec y)`
	Returns the component-wise compare of $x == y$.

`bvec notEqual(vec x, vec y)`
`bvec notEqual(ivec x, ivec y)`
`bvec notEqual(bvec x, bvec y)`
	Returns the component-wise compare of $x != y$.

`bool any(bvec x)`	The vector equivalent of the logical *or*, |—returns true if any component of *x* is true.
`bool all(bvec x)`	The vector equivalent of the logical *and*, &—returns true only if all components of *x* are true.
`bvec not(bvec x)`	The vector equivalent of the logical *not*, !—returns the component-wise logical complement of *x*.

GLSL

Tipi di dato composti – Funzioni di texture look-up

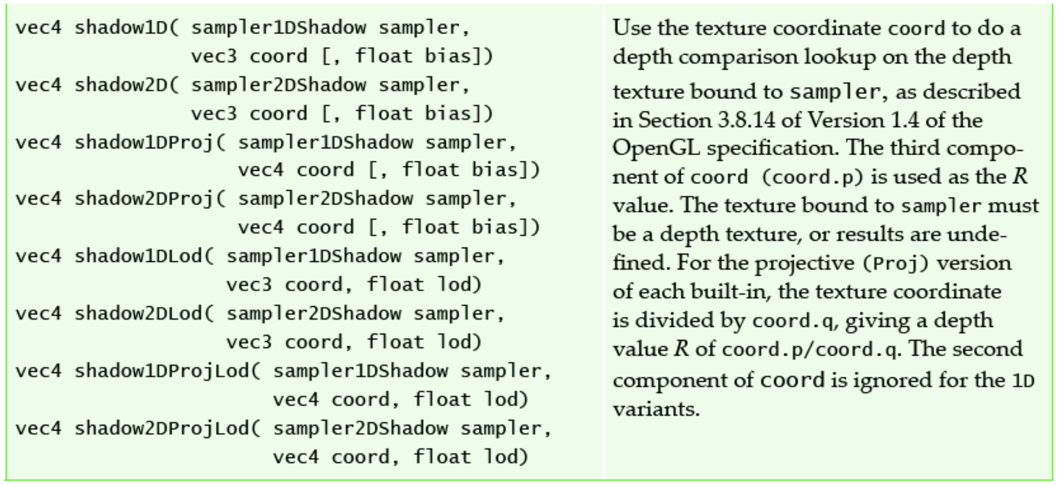


`vec4 texture( sampler1D sampler,`
`               float coord [, float bias])`
`vec4 textureProj( sampler1D sampler,`
`                   vec{2,4} coord [, float bias])`
`vec4 textureLod( sampler1D sampler,`
`                  float coord, float lod)`
`vec4 textureProjLod( sampler1D sampler,`
`                      vec{2,4} coord, float lod)`
	Use the texture coordinate `coord` to do a texture lookup in the 1D texture currently bound to `sampler`. For the projective (`Proj`) versions, the texture coordinate `coord.s` is divided by the last component of `coord`.

`vec4 texture( sampler2D sampler,`
`               vec2 coord [, float bias])`
`vec4 textureProj( sampler2D sampler,`
`                   vec{3,4} coord [, float bias])`
`vec4 textureLod( sampler2D sampler,`
`                  vec2 coord, float lod)`
`vec4 textureProjLod( sampler2D sampler,`
`                      vec{3,4} coord, float lod)`
	Use the texture coordinate `coord` to do a texture lookup in the 2D texture currently bound to `sampler`. For the projective (`Proj`) versions, the texture coordinate (`coord.s`, `coord.t`) is divided by the last component of `coord`. The third component of `coord` is ignored for the `vec4 coord` variant.

`vec4 texture( sampler3D sampler,`
`               vec3 coord [, float bias])`
`vec4 textureProj( sampler3D sampler,`
`                   vec4 coord [, float bias])`
`vec4 textureLod( sampler3D sampler,`
`                  vec3 coord, float lod)`
`vec4 textureProjLod( sampler3D sampler,`
`                      vec4 coord, float lod)`
	Use the texture coordinate `coord` to do a texture lookup in the 3D texture currently bound to `sampler`. For the projective (`Proj`) versions, the texture coordinate is divided by `coord.q`.

`vec4 texture( samplerCube sampler,`
`               vec3 coord [, float bias])`
`vec4 textureLod( samplerCube sampler,`
`                  vec3 coord, float lod)`
	Use the texture coordinate `coord` to do a texture lookup in the cube map texture currently bound to `sampler`. The direction of `coord` is used to select in which face to do a two-dimensional texture lookup.

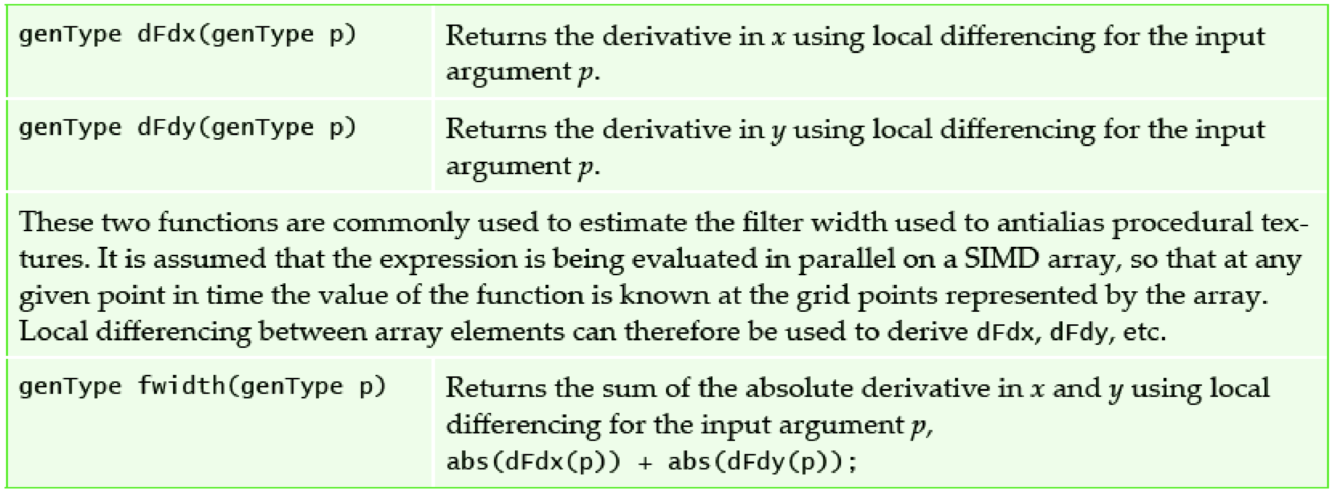


`vec4 shadow1D( sampler1DShadow sampler,`
`               vec3 coord [, float bias])`
`vec4 shadow2D( sampler2DShadow sampler,`
`               vec3 coord [, float bias])`
`vec4 shadow1DProj( sampler1DShadow sampler,`
`                   vec4 coord [, float bias])`
`vec4 shadow2DProj( sampler2DShadow sampler,`
`                   vec4 coord [, float bias])`
`vec4 shadow1DLod( sampler1DShadow sampler,`
`                  vec3 coord, float lod)`
`vec4 shadow2DLod( sampler2DShadow sampler,`
`                  vec3 coord, float lod)`
`vec4 shadow1DProjLod( sampler1DShadow sampler,`
`                      vec4 coord, float lod)`
`vec4 shadow2DProjLod( sampler2DShadow sampler,`
`                      vec4 coord, float lod)`
	Use the texture coordinate coord to do a depth comparison lookup on the depth texture bound to `sampler`, as described in Section 3.8.14 of Version 1.4 of the OpenGL specification. The third component of coord (`coord.p`) is used as the *R* value. The texture bound to `sampler` must be a depth texture, or results are undefined. For the projective (`Proj`) version of each built-in, the texture coordinate is divided by `coord.q`, giving a depth value *R* of `coord.p/coord.q`. The second component of `coord` is ignored for the `1D` variants.

GLSL

Tipi di dato composti – Funzioni che operano su fragments

Utili per l’antialias di texture procedurali (i.e. generate online)



`genType dFdx(genType p)`	Returns the derivative in *x* using local differencing for the input argument *p*.
`genType dFdy(genType p)`	Returns the derivative in *y* using local differencing for the input argument *p*.
These two functions are commonly used to estimate the filter width used to antialias procedural textures. It is assumed that the expression is being evaluated in parallel on a SIMD array, so that at any given point in time the value of the function is known at the grid points represented by the array. Local differencing between array elements can therefore be used to derive `dFdx`, `dFdy`, etc.	
`genType fwidth(genType p)`	Returns the sum of the absolute derivative in *x* and *y* using local differencing for the input argument *p*, `abs(dFdx(p)) + abs(dFdy(p));`

GLSL

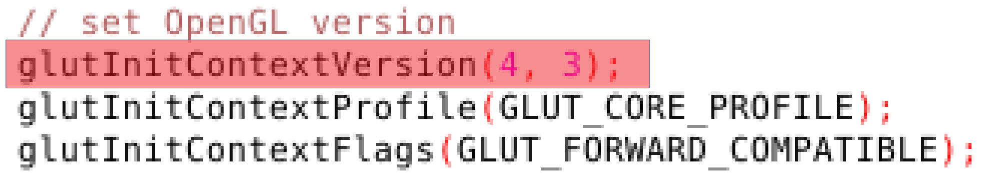
Passaggio di parametri e modificatori

In GLSL i parametri di una funzione sono passati per value-return →
Modificare il valore del parametro formale (i.e. x in f(x)) modifica il valore della variabile passata al momento del ritorno dalla funzione!
Sono definiti i seguenti modificatori:

- **`const`**: The value of the input parameter is copied to the formal parameter, but no change to the formal parameter is allowed in the function.
- **`in`**: The value of the actual parameter is copied to the formal parameter, but no changed value will be returned. The actual parameter may be an expression that sets the value to be copied into the function. The formal parameter may be changed during the execution of the function. The keyword in may be preceded by const, in which case the formal parameter will be treated as a const in the function.
- **`out`**: The formal parameter must be an lvalue and will have no value until it is set inside the function. Any function operations may use this parameter, but a value must be set in the function. The value of the formal parameter in the function is copied to the actual parameter when the function terminates.
- **`input`**: The formal parameter must be an lvalue and is assumed to have a value when it is copied to the function, and this value may be used or changed during the function execution. When the function terminates, the value of the formal parameter is copied to the actual parameter.

I miei primi shader

glutInitContextVersion(major, minor)
Definisce la versione di OpenGL in cui si sta sviluppando (e.g. 4.3)



```
// set OpenGL version
glutInitContextVersion(4, 3);
glutInitContextProfile(GLUT_CORE_PROFILE);
glutInitContextFlags(GLUT_FORWARD_COMPATIBLE);
```

main()

OpenGL – Definizione del contesto

Motivazione: Con il cambiamento nel tempo della OpenGL e' sorta la necessita' di mantenere da un lato "usabile" codice sviluppato per versioni precedenti di OpenGL e dall'altro di essere sicuri di non stare usando funzionalita' "deprecate" o obsolete nello sviluppo di nuovo software

Soluzione: Glut offre meccanismi sofisticati per specificare nel contesto di quale versione deve girare il nostro programma.

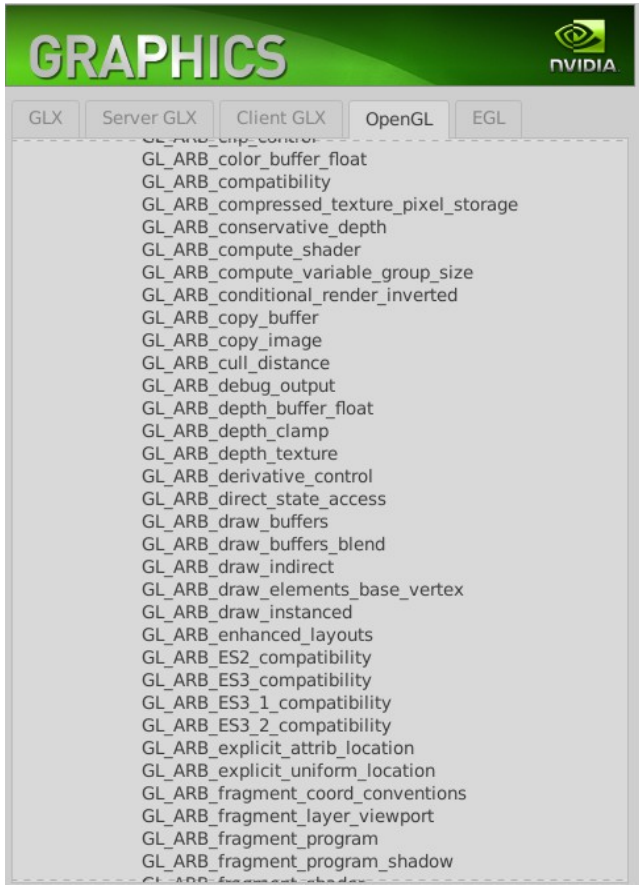
- 1992 **Versione 1.0**
- 1995 OpenGL 1.1 (vertex array, texture object)
- 1997 OpenGL + Direct3D = Fahrenheit
- 1998 OpenGL 1.2 (3D textures, separate specular imaging)
- 2001 OpenGL 1.3 (compressed texture, cube maps, dot3)
- 2002 OpenGL 1.4 (mipmap, shadow)
- 2003 OpenGL 1.5 (vertex buffer object, occlusion query)
- 2004 **Versione 2.0**
- 2005 OpenGL Embedded Systems (OpenGL ES) per dispositivi mobile
- 2006 OpenGL 2.1 (GLSL - OpenGL shading language)
- 2008 **Versione 3.0**
- 2009 OpenGL 3.1 (texture objects)
- 2009 OpenGL 3.2 (Geometry shader)
- 2010 **Versione 4.0**
- 2010 Accesso alle GPU di ultima generazione
- 2010 OpenGL 4.1 (tessellation-control & tessellation-evaluation shaders)
- 2011 OpenGL 4.2
- 2012 OpenGL 4.3 (versione 4.3 di GLSL, nuovo metodo di compressione texture)
- 2013 OpenGL 4.4
- 2014 OpenGL 4.5
- 2017 OpenGL 4.6

https://www.khronos.org/opengl/wiki/History_of_OpenGL

OpenGL – Definizione del contesto

Stato delle funzionalita’ OpenGL

- **Current** – La funzionalita’ fa parte delle core specification di quella versione
- **Deprecated** – La funzionalita’ e’ “deprecated” ma ancora parte delle core specification
- **Removed** – La funzionalita’ e’ stata rimossa dalle core specification
- **Not yet introduced** – La funzionalita’ non e’ ancora stata introdotta in OpenGL



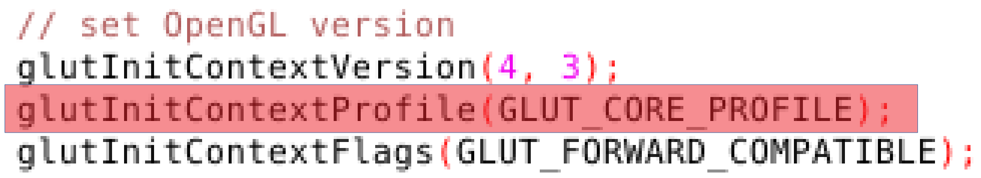
```
GL_ARB_color_buffer_float
GL_ARB_compatibility
GL_ARB_compressed_texture_pixel_storage
GL_ARB_conservative_depth
GL_ARB_compute_shader
GL_ARB_compute_variable_group_size
GL_ARB_conditional_render_inverted
GL_ARB_copy_buffer
GL_ARB_copy_image
GL_ARB_cull_distance
GL_ARB_debug_output
GL_ARB_depth_buffer_float
GL_ARB_depth_clamp
GL_ARB_depth_texture
GL_ARB_derivative_control
GL_ARB_direct_state_access
GL_ARB_draw_buffers
GL_ARB_draw_buffers_blend
GL_ARB_draw_indirect
GL_ARB_draw_elements_base_vertex
GL_ARB_draw_instanced
GL_ARB_enhanced_layouts
GL_ARB_ES2_compatibility
GL_ARB_ES3_compatibility
GL_ARB_ES3_1_compatibility
GL_ARB_ES3_2_compatibility
GL_ARB_explicit_attrib_location
GL_ARB_explicit_uniform_location
GL_ARB_fragment_coord_conventions
GL_ARB_fragment_layer_viewport
GL_ARB_fragment_program
GL_ARB_fragment_program_shadow
```

Implicazioni per il programmatore

Compatibilita’ hardware – Un programma sviluppato per una versione x.y di OpenGL o contenente funzionalita’ rimosse o deprecated potrebbe non girare su schede grafiche che supportano versioni di OpenGL precedenti → In generale i produttori di schede grafiche implementano tali funzionalita’ nei loro drivers ma lasciano al programmatore la scelta se usarle o meno.

https://sites.google.com/site/gsucomputergraphics/educational/initialization-tasks-in-an-opengl-program/opengl-context-and-profile

OpenGL – Definizione del contesto



```
// set OpenGL version
glutInitContextVersion(4, 3);
glutInitContextProfile(GLUT_CORE_PROFILE);
glutInitContextFlags(GLUT_FORWARD_COMPATIBLE);
```

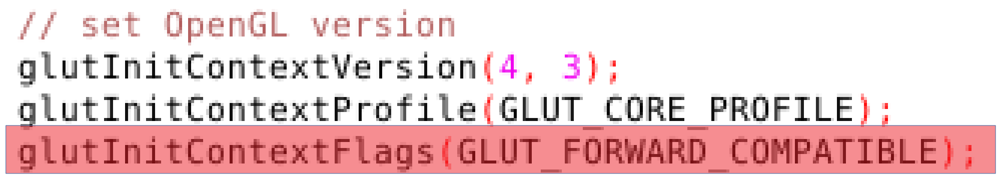
main()

`glutInitContextVersion(major, minor)`
Definisce la versione di OpenGL in cui si sta sviluppando (e.g. 4.3)

`glutInitContextProfile(GLUT_CORE_PROFILE | GLUT_COMPATIBILITY_PROFILE)`
Definisce quali funzionalita' (core o presenti anche in precedenti versioni di OpenGL si vogliono usare). Le funzionalita' core sono quelle con stato current o deprecated nella versione major.minor di OpenGL.

https://www.khronos.org/opengl/wiki/History_of_OpenGL

OpenGL – Definizione del contesto



```
// set OpenGL version
glutInitContextVersion(4, 3);
glutInitContextProfile(GLUT_CORE_PROFILE);
glutInitContextFlags(GLUT_FORWARD_COMPATIBLE);
```

`main()`

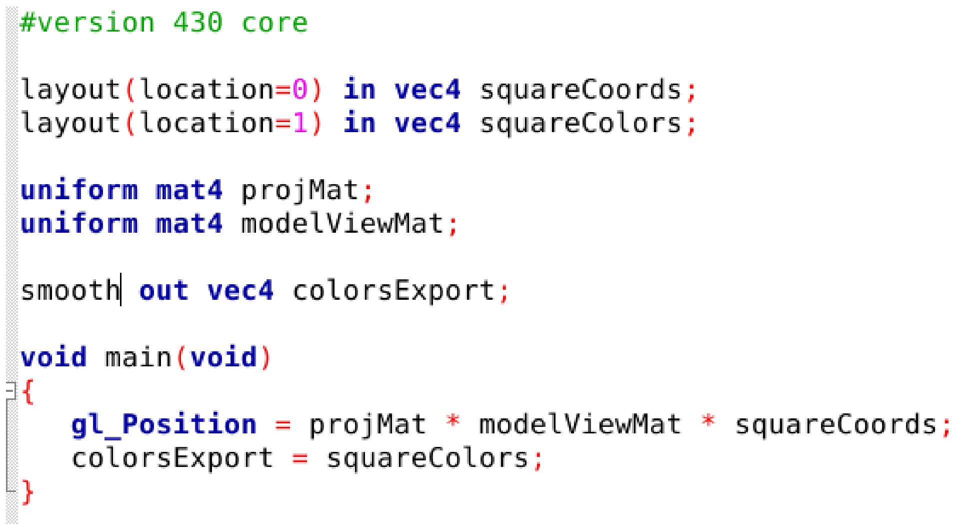
`glutInitContextVersion(major, minor)`
Definisce la versione di OpenGL in cui si sta sviluppando (e.g. 4.3)

`glutInitContextProfile(GLUT_CORE_PROFILE | GLUT_COMPATIBILITY_PROFILE)`
Definisce quali funzionalita' (core o presenti anche in precedenti versioni di OpenGL si vogliono usare). Le funzionalita' core sono quelle con stato current o deprecated nella versione major.minor di OpenGL.

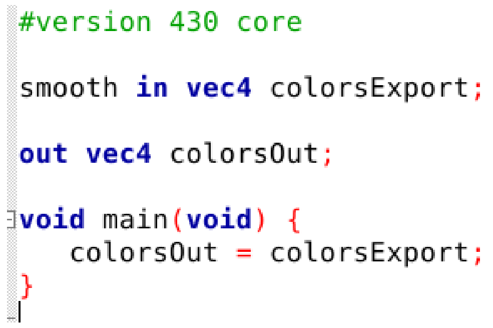
`glutInitContextFlags(flags)`
Passa vari flags alla OpenGL (e.g. GLUT_DEBUG).
GLUT_FORWARD_COMPATIBLE esclude le funzionalita' deprecated.

https://www.khronos.org/opengl/wiki/History_of_OpenGL

Vertex and Fragment shaders



```
#version 430 core

layout(location=0) in vec4 squareCoords;
layout(location=1) in vec4 squareColors;

uniform mat4 projMat;
uniform mat4 modelViewMat;

smooth out vec4 colorsExport;

void main(void)
{
   gl_Position = projMat * modelViewMat * squareCoords;
   colorsExport = squareColors;
}
```

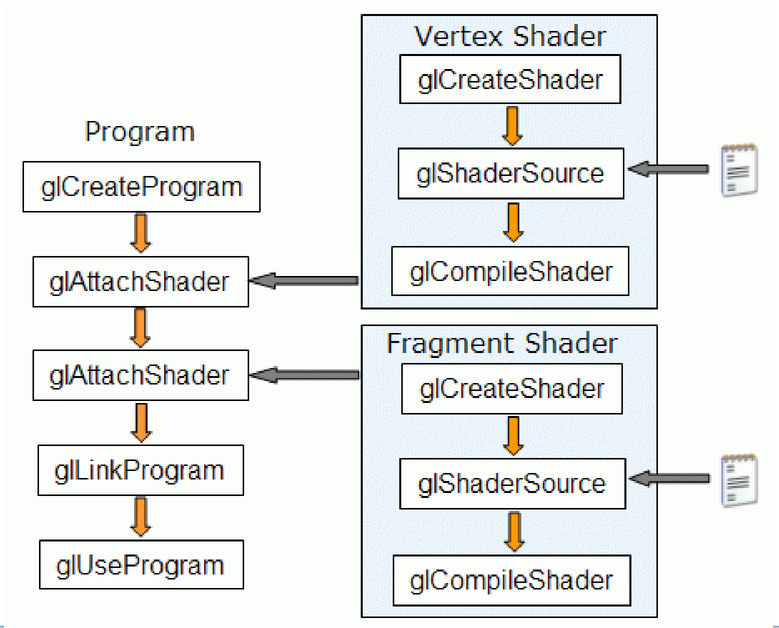


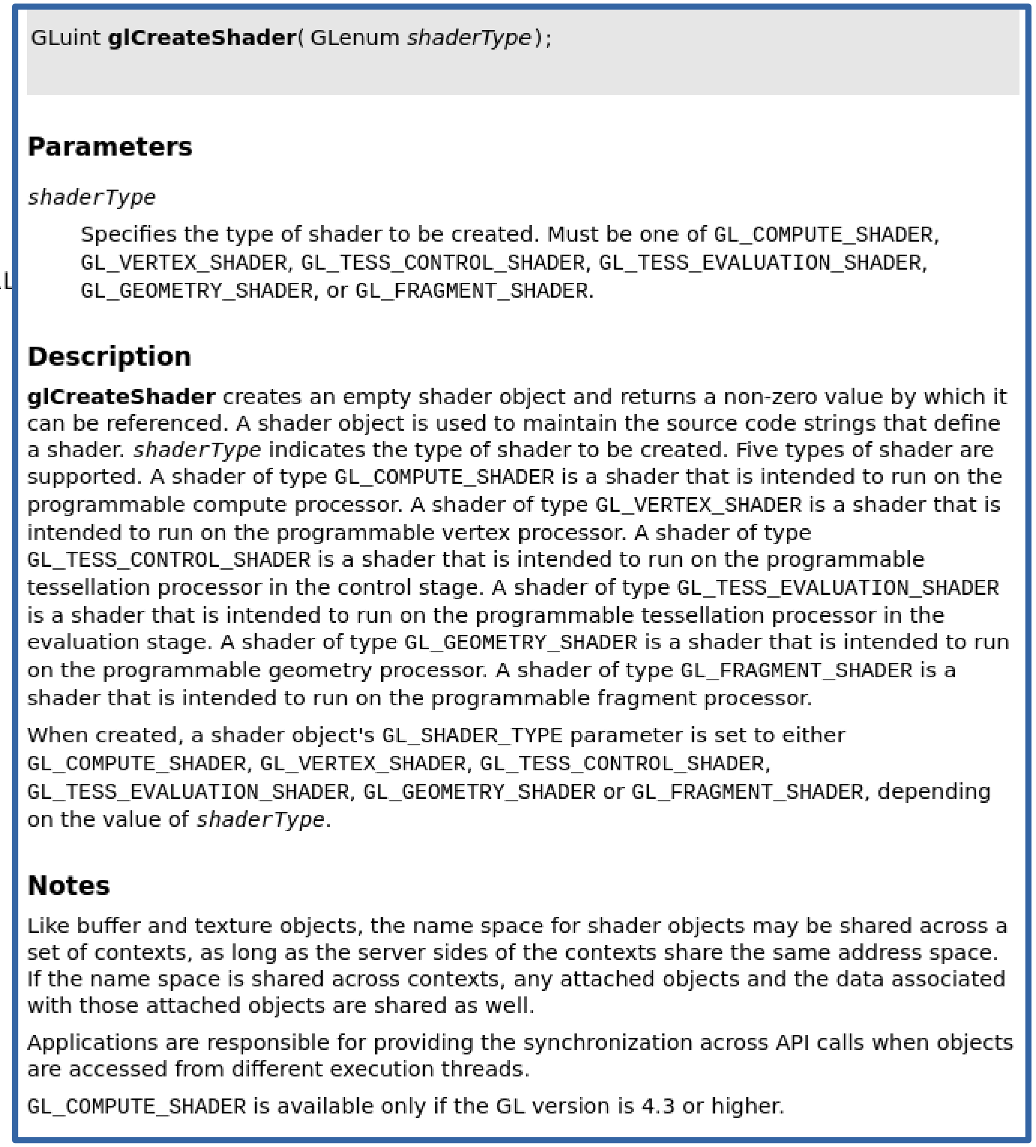
```
#version 430 core

smooth in vec4 colorsExport;

out vec4 colorsOut;

void main(void) {
   colorsOut = colorsExport;
}
```

Compilazione degli shaders

```
// Create shader program executable.
char* vertexShader = readTextFile("vertexShader.glsl");
vertexShaderId = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShaderId, 1, (const char**) &vertexShader, NULL);
glCompileShader(vertexShaderId);

char* fragmentShader = readTextFile("fragmentShader.glsl");
fragmentShaderId = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShaderId, 1, (const char**) &fragmentShader, NULL
glCompileShader(fragmentShaderId);

programId = glCreateProgram();
glAttachShader(programId, vertexShaderId);
glAttachShader(programId, fragmentShaderId);
glLinkProgram(programId);
glUseProgram(programId);
// ------------------------------------------
```

init()




GLuint **glCreateShader**(GLenum *shaderType*);

Parameters

shaderType

Specifies the type of shader to be created. Must be one of GL_COMPUTE_SHADER, GL_VERTEX_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER, GL_GEOMETRY_SHADER, or GL_FRAGMENT_SHADER.

Description

glCreateShader creates an empty shader object and returns a non-zero value by which it can be referenced. A shader object is used to maintain the source code strings that define a shader. *shaderType* indicates the type of shader to be created. Five types of shader are supported. A shader of type GL_COMPUTE_SHADER is a shader that is intended to run on the programmable compute processor. A shader of type GL_VERTEX_SHADER is a shader that is intended to run on the programmable vertex processor. A shader of type GL_TESS_CONTROL_SHADER is a shader that is intended to run on the programmable tessellation processor in the control stage. A shader of type GL_TESS_EVALUATION_SHADER is a shader that is intended to run on the programmable tessellation processor in the evaluation stage. A shader of type GL_GEOMETRY_SHADER is a shader that is intended to run on the programmable geometry processor. A shader of type GL_FRAGMENT_SHADER is a shader that is intended to run on the programmable fragment processor.

When created, a shader object's GL_SHADER_TYPE parameter is set to either GL_COMPUTE_SHADER, GL_VERTEX_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER, GL_GEOMETRY_SHADER or GL_FRAGMENT_SHADER, depending on the value of *shaderType*.

Notes

Like buffer and texture objects, the name space for shader objects may be shared across a set of contexts, as long as the server sides of the contexts share the same address space. If the name space is shared across contexts, any attached objects and the data associated with those attached objects are shared as well.

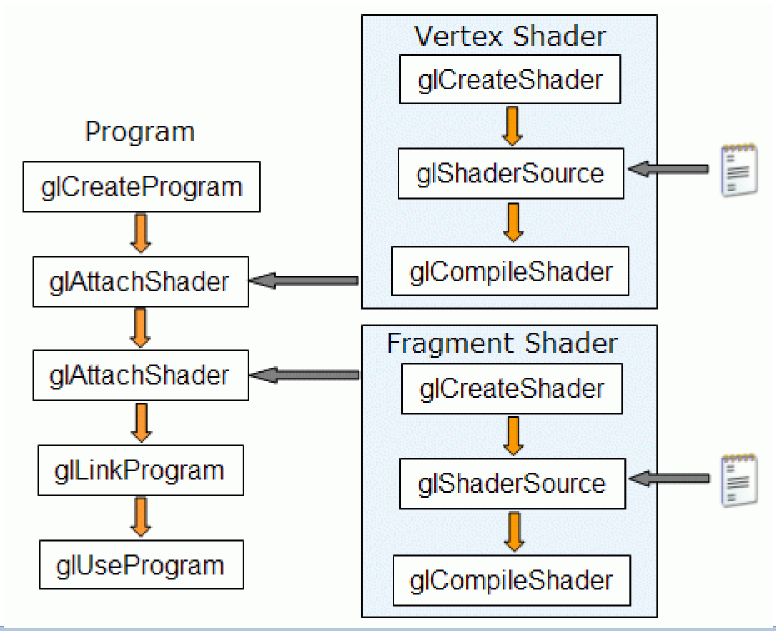
Applications are responsible for providing the synchronization across API calls when objects are accessed from different execution threads.

GL_COMPUTE_SHADER is available only if the GL version is 4.3 or higher.

Compilazione degli shaders

```
// Create shader program executable.
char* vertexShader = readTextFile("vertexShader.glsl");
vertexShaderId = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShaderId, 1, (const char**) &vertexShader, NULL);
glCompileShader(vertexShaderId);

char* fragmentShader = readTextFile("fragmentShader.glsl");
fragmentShaderId = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShaderId, 1, (const char**) &fragmentShader, NULL)
glCompileShader(fragmentShaderId);

programId = glCreateProgram();
glAttachShader(programId, vertexShaderId);
glAttachShader(programId, fragmentShaderId);
glLinkProgram(programId);
glUseProgram(programId);
// ------------------------------------------
```

init()



glShaderSource — Replaces the source code in a shader object

C Specification

void **glShaderSource**(GLuint *shader*,
GLsizei *count*,
const GLchar ***string*,
const GLint **length*);

Parameters

shader

Specifies the handle of the shader object whose source code is to be replaced.

count

Specifies the number of elements in the *string* and *length* arrays.

string

Specifies an array of pointers to strings containing the source code to be loaded into the shader.

length

Specifies an array of string lengths.

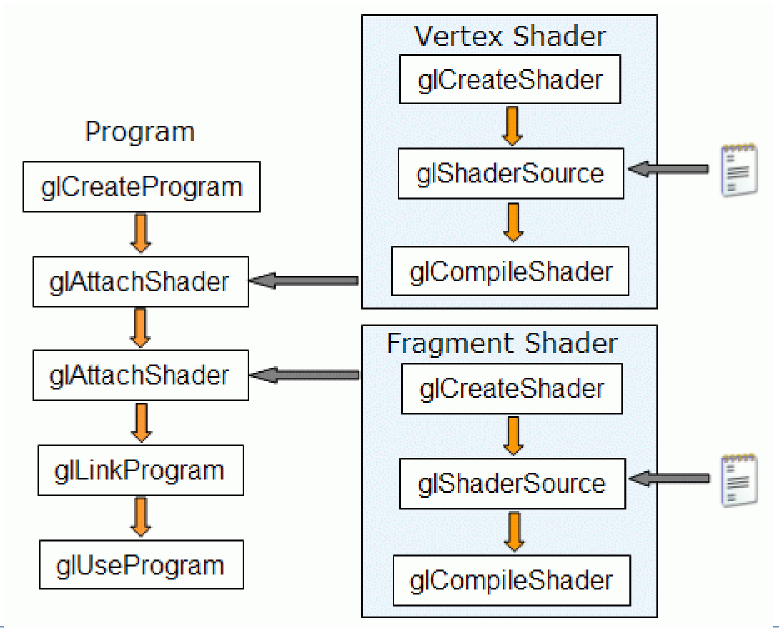
Description

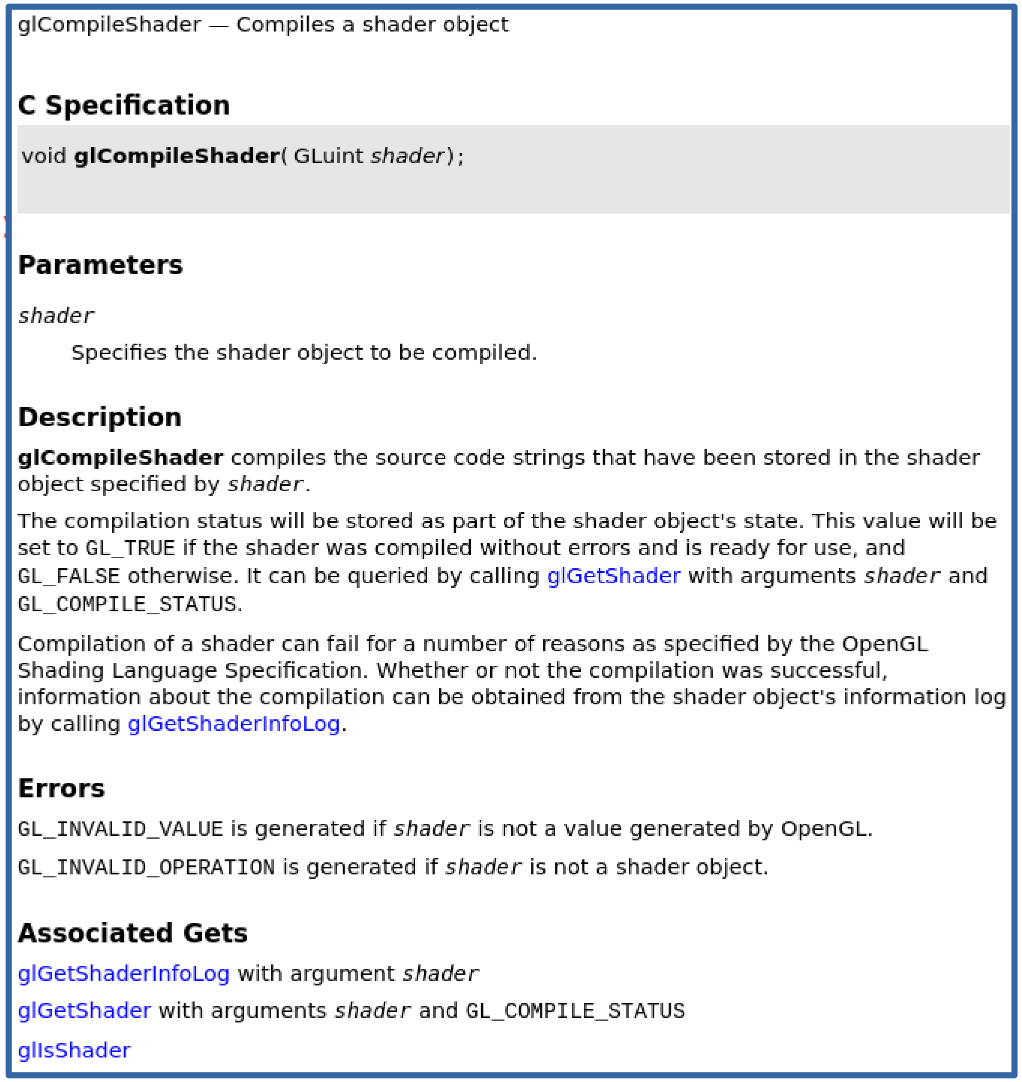
glShaderSource sets the source code in *shader* to the source code in the array of strings specified by *string*. Any source code previously stored in the shader object is completely replaced. The number of strings in the array is specified by *count*. If *length* is NULL, each string is assumed to be null terminated. If *length* is a value other than NULL, it points to an array containing a string length for each of the corresponding elements of *string*. Each element in the *length* array may contain the length of the corresponding string (the null character is not counted as part of the string length) or a value less than 0 to indicate that the string is null terminated. The source code strings are not scanned or parsed at this time; they are simply copied into the specified shader object.

Compilazione degli shaders

```
// Create shader program executable.
char* vertexShader = readTextFile("vertexShader.glsl");
vertexShaderId = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShaderId, 1, (const char**) &vertexShader, NULL);
glCompileShader(vertexShaderId);

char* fragmentShader = readTextFile("fragmentShader.glsl");
fragmentShaderId = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShaderId, 1, (const char**) &fragmentShader, NULL)
glCompileShader(fragmentShaderId);

programId = glCreateProgram();
glAttachShader(programId, vertexShaderId);
glAttachShader(programId, fragmentShaderId);
glLinkProgram(programId);
glUseProgram(programId);
// ------------------------------------------
```

init()




glCompileShader — Compiles a shader object

C Specification

void **glCompileShader**(GLuint *shader*);

Parameters

shader

Specifies the shader object to be compiled.

Description

glCompileShader compiles the source code strings that have been stored in the shader object specified by *shader*.

The compilation status will be stored as part of the shader object's state. This value will be set to GL_TRUE if the shader was compiled without errors and is ready for use, and GL_FALSE otherwise. It can be queried by calling glGetShader with arguments *shader* and GL_COMPILE_STATUS.

Compilation of a shader can fail for a number of reasons as specified by the OpenGL Shading Language Specification. Whether or not the compilation was successful, information about the compilation can be obtained from the shader object's information log by calling glGetShaderInfoLog.

Errors

GL_INVALID_VALUE is generated if *shader* is not a value generated by OpenGL.

GL_INVALID_OPERATION is generated if *shader* is not a shader object.

Associated Gets

glGetShaderInfoLog with argument *shader*

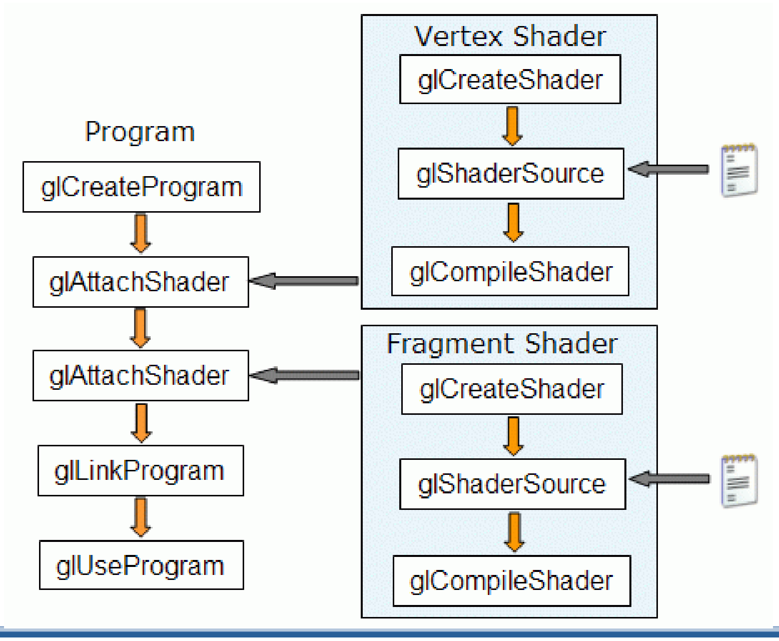
glGetShader with arguments *shader* and GL_COMPILE_STATUS

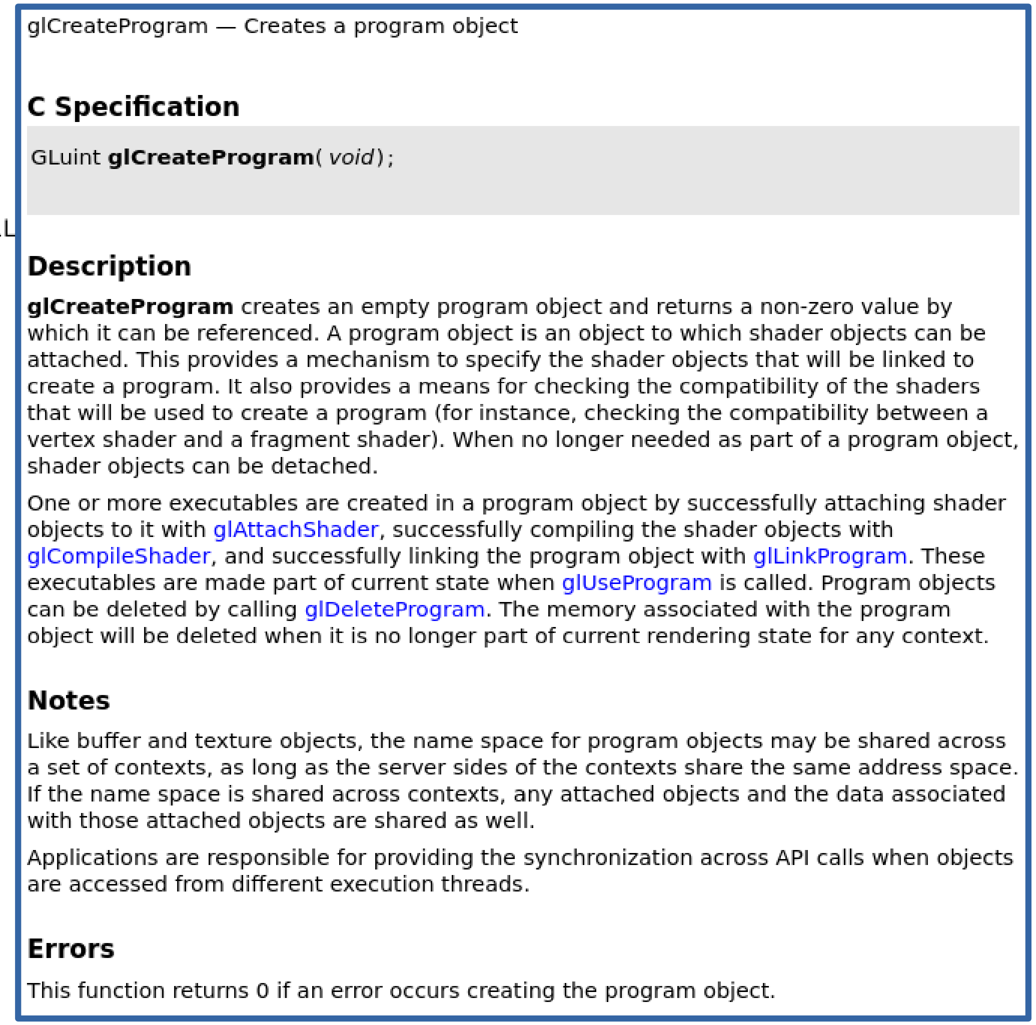
glIsShader

Compilazione degli shaders

```
// Create shader program executable.
char* vertexShader = readTextFile("vertexShader.glsl");
vertexShaderId = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShaderId, 1, (const char**) &vertexShader, NULL);
glCompileShader(vertexShaderId);

char* fragmentShader = readTextFile("fragmentShader.glsl");
fragmentShaderId = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShaderId, 1, (const char**) &fragmentShader, NULL
glCompileShader(fragmentShaderId);

programId = glCreateProgram();
glAttachShader(programId, vertexShaderId);
glAttachShader(programId, fragmentShaderId);
glLinkProgram(programId);
glUseProgram(programId);
// ------------------------------------------
```

init()




glCreateProgram — Creates a program object

C Specification

GLuint **glCreateProgram**(*void*);

Description

glCreateProgram creates an empty program object and returns a non-zero value by which it can be referenced. A program object is an object to which shader objects can be attached. This provides a mechanism to specify the shader objects that will be linked to create a program. It also provides a means for checking the compatibility of the shaders that will be used to create a program (for instance, checking the compatibility between a vertex shader and a fragment shader). When no longer needed as part of a program object, shader objects can be detached.

One or more executables are created in a program object by successfully attaching shader objects to it with glAttachShader, successfully compiling the shader objects with glCompileShader, and successfully linking the program object with glLinkProgram. These executables are made part of current state when glUseProgram is called. Program objects can be deleted by calling glDeleteProgram. The memory associated with the program object will be deleted when it is no longer part of current rendering state for any context.

Notes

Like buffer and texture objects, the name space for program objects may be shared across a set of contexts, as long as the server sides of the contexts share the same address space. If the name space is shared across contexts, any attached objects and the data associated with those attached objects are shared as well.

Applications are responsible for providing the synchronization across API calls when objects are accessed from different execution threads.

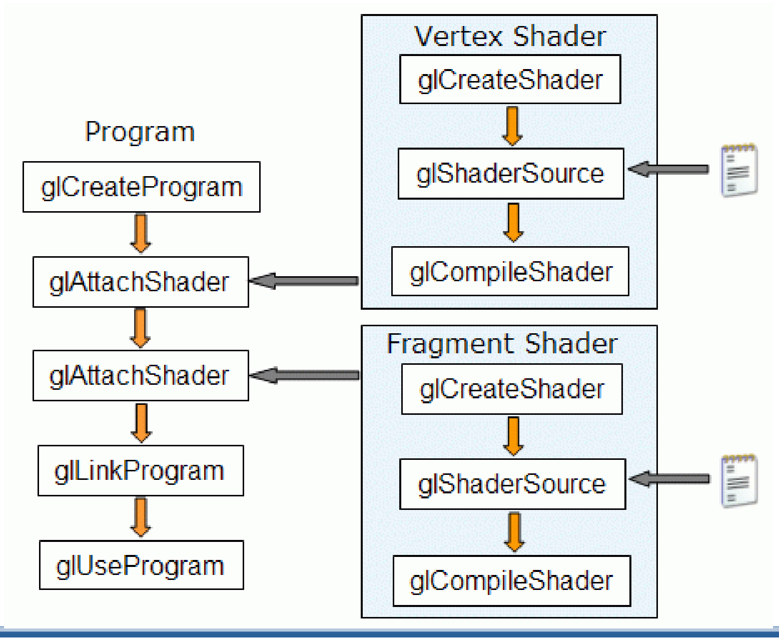
Errors

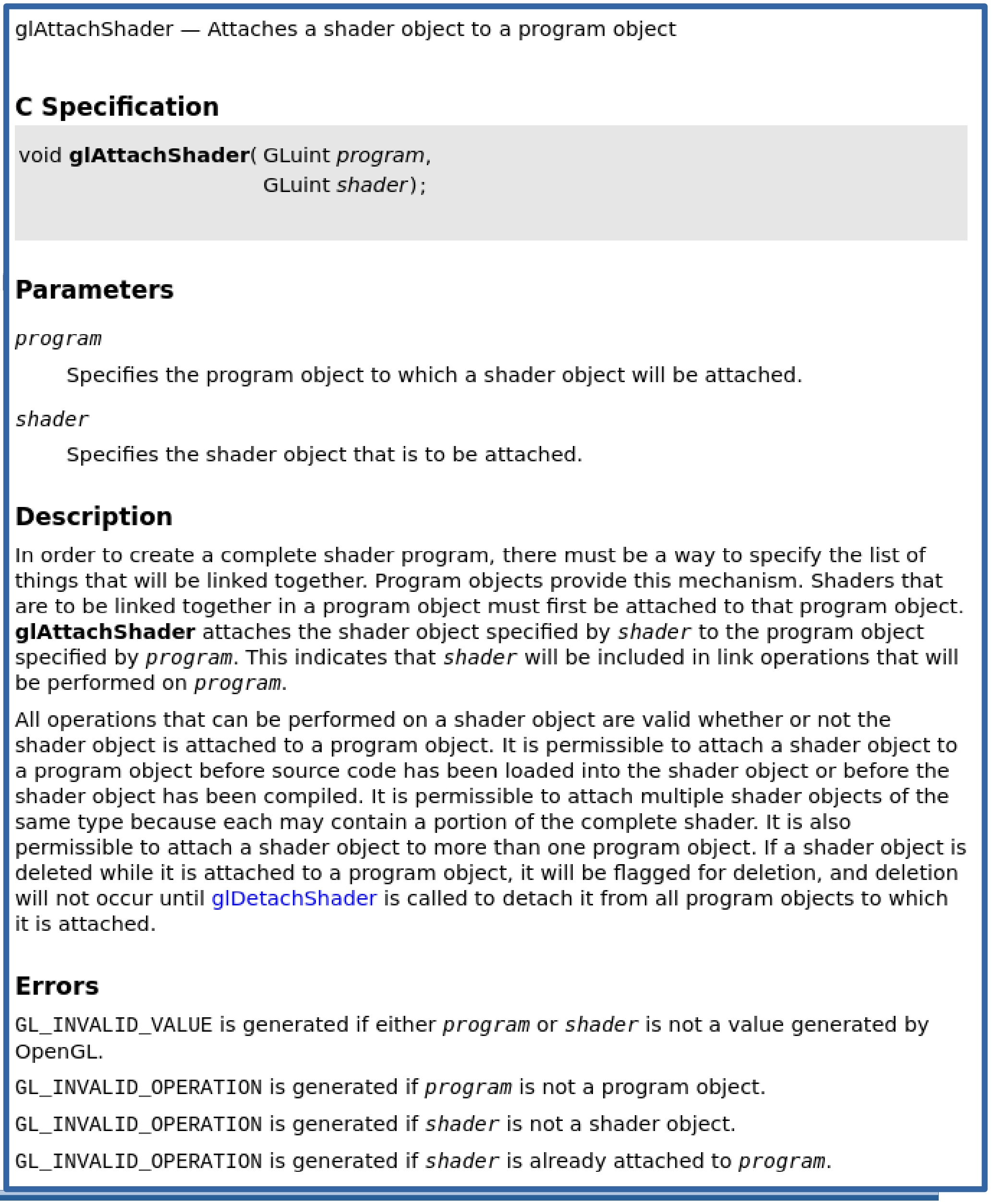
This function returns 0 if an error occurs creating the program object.

Compilazione degli shaders

```
// Create shader program executable.
char* vertexShader = readTextFile("vertexShader.glsl");
vertexShaderId = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShaderId, 1, (const char**) &vertexShader, NULL);
glCompileShader(vertexShaderId);

char* fragmentShader = readTextFile("fragmentShader.glsl");
fragmentShaderId = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShaderId, 1, (const char**) &fragmentShader, NUL
glCompileShader(fragmentShaderId);

programId = glCreateProgram();
glAttachShader(programId, vertexShaderId);
glAttachShader(programId, fragmentShaderId);
glLinkProgram(programId);
glUseProgram(programId);
// ------------------------------------------
```

init()




glAttachShader — Attaches a shader object to a program object

C Specification

void **glAttachShader**(GLuint *program*,
GLuint *shader*);

Parameters

program

Specifies the program object to which a shader object will be attached.

shader

Specifies the shader object that is to be attached.

Description

In order to create a complete shader program, there must be a way to specify the list of things that will be linked together. Program objects provide this mechanism. Shaders that are to be linked together in a program object must first be attached to that program object. **glAttachShader** attaches the shader object specified by *shader* to the program object specified by *program*. This indicates that *shader* will be included in link operations that will be performed on *program*.

All operations that can be performed on a shader object are valid whether or not the shader object is attached to a program object. It is permissible to attach a shader object to a program object before source code has been loaded into the shader object or before the shader object has been compiled. It is permissible to attach multiple shader objects of the same type because each may contain a portion of the complete shader. It is also permissible to attach a shader object to more than one program object. If a shader object is deleted while it is attached to a program object, it will be flagged for deletion, and deletion will not occur until glDetachShader is called to detach it from all program objects to which it is attached.

Errors

GL_INVALID_VALUE is generated if either *program* or *shader* is not a value generated by OpenGL.

GL_INVALID_OPERATION is generated if *program* is not a program object.

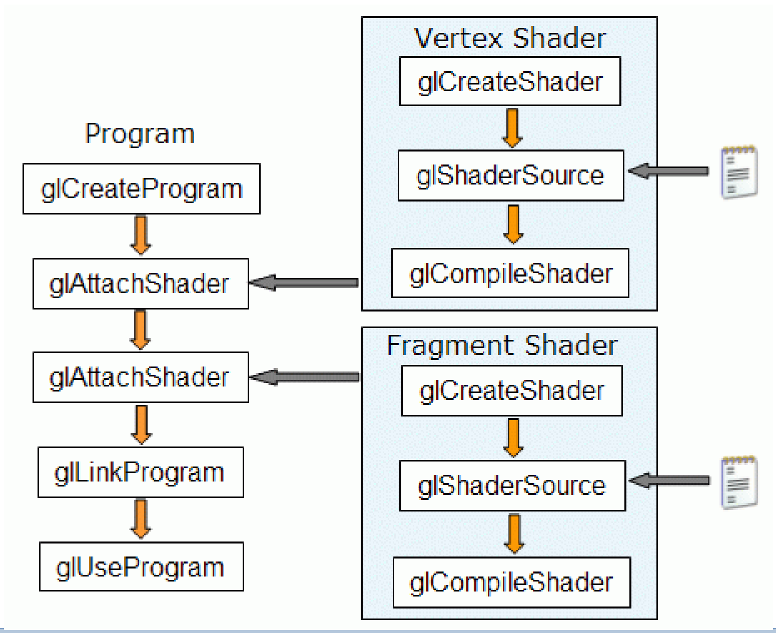
GL_INVALID_OPERATION is generated if *shader* is not a shader object.

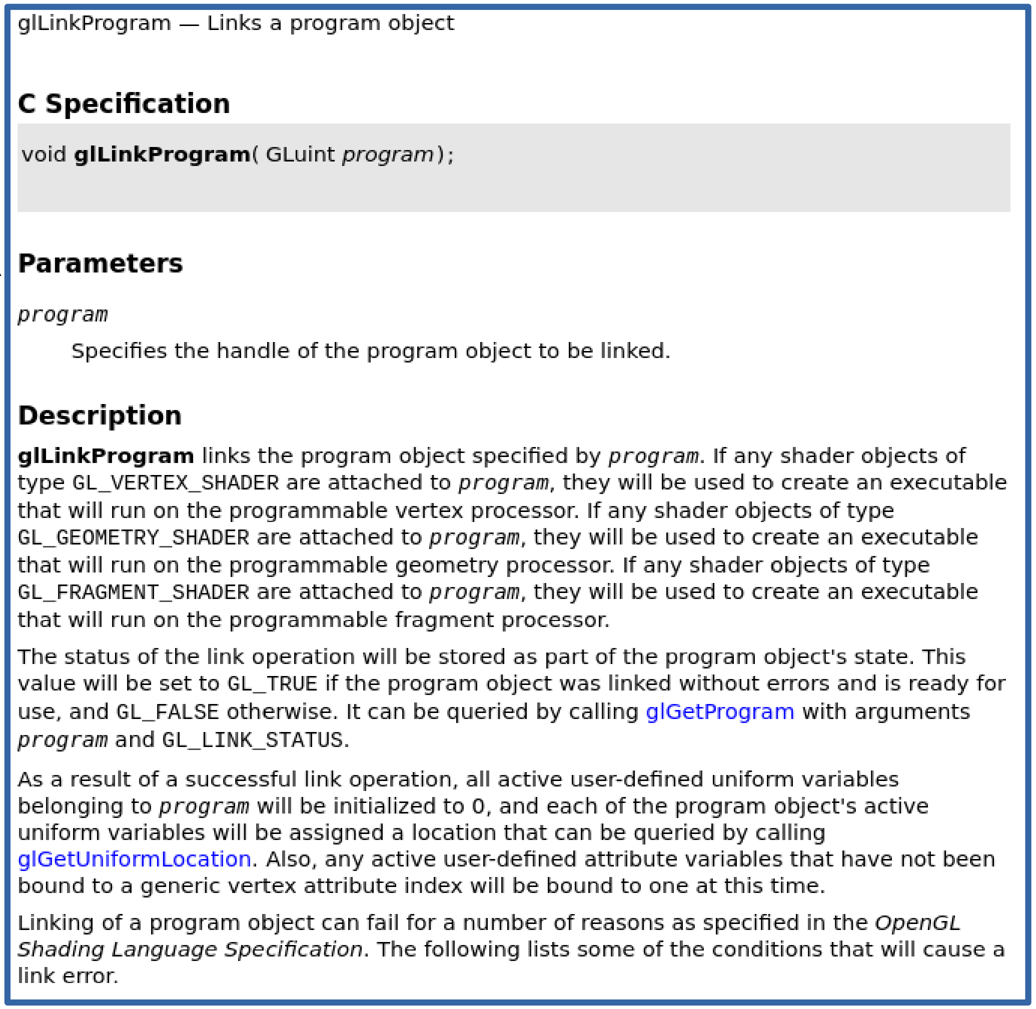
GL_INVALID_OPERATION is generated if *shader* is already attached to *program*.

Compilazione degli shaders

```
// Create shader program executable.
char* vertexShader = readTextFile("vertexShader.glsl");
vertexShaderId = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShaderId, 1, (const char**) &vertexShader, NULL);
glCompileShader(vertexShaderId);

char* fragmentShader = readTextFile("fragmentShader.glsl");
fragmentShaderId = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShaderId, 1, (const char**) &fragmentShader, NUL
glCompileShader(fragmentShaderId);

programId = glCreateProgram();
glAttachShader(programId, vertexShaderId);
glAttachShader(programId, fragmentShaderId);
glLinkProgram(programId);
glUseProgram(programId);
// ----------------------------------------
```

`init()`





glLinkProgram — Links a program object

C Specification

void **glLinkProgram**(GLuint *program*);

Parameters

program

Specifies the handle of the program object to be linked.

Description

glLinkProgram links the program object specified by *program*. If any shader objects of type GL_VERTEX_SHADER are attached to *program*, they will be used to create an executable that will run on the programmable vertex processor. If any shader objects of type GL_GEOMETRY_SHADER are attached to *program*, they will be used to create an executable that will run on the programmable geometry processor. If any shader objects of type GL_FRAGMENT_SHADER are attached to *program*, they will be used to create an executable that will run on the programmable fragment processor.

The status of the link operation will be stored as part of the program object's state. This value will be set to GL_TRUE if the program object was linked without errors and is ready for use, and GL_FALSE otherwise. It can be queried by calling glGetProgram with arguments *program* and GL_LINK_STATUS.

As a result of a successful link operation, all active user-defined uniform variables belonging to *program* will be initialized to 0, and each of the program object's active uniform variables will be assigned a location that can be queried by calling glGetUniformLocation. Also, any active user-defined attribute variables that have not been bound to a generic vertex attribute index will be bound to one at this time.

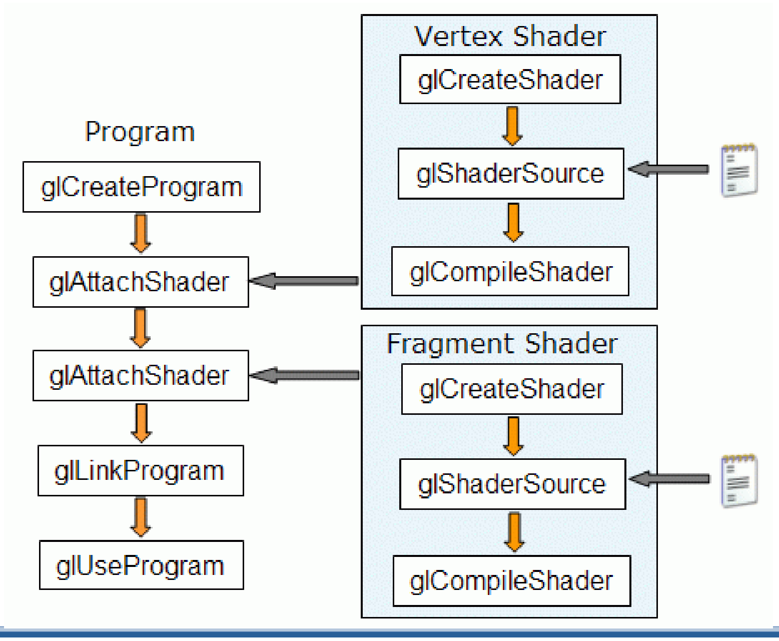
Linking of a program object can fail for a number of reasons as specified in the *OpenGL Shading Language Specification*. The following lists some of the conditions that will cause a link error.

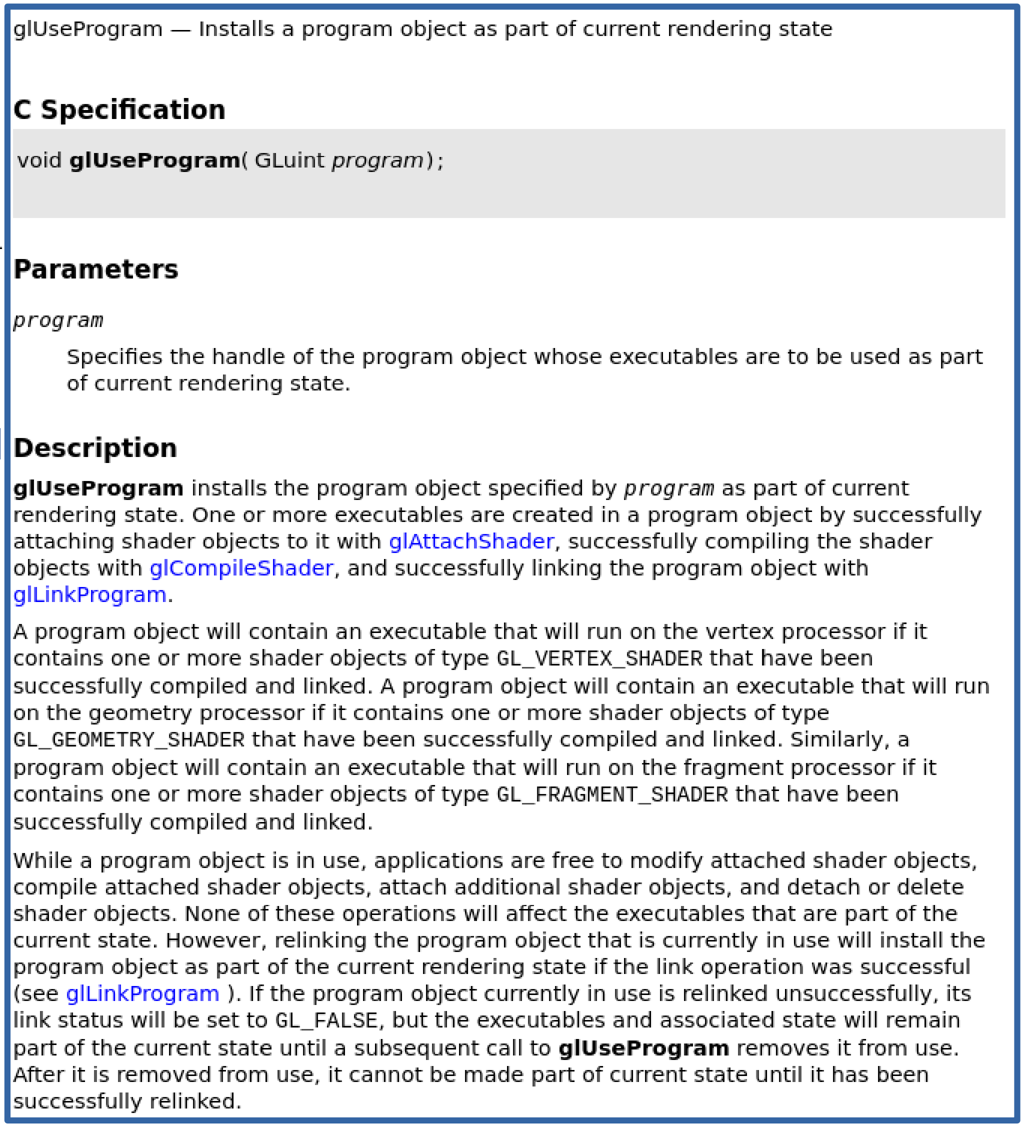
Una sola funzione `main` per tutti gli shaders dello stesso tipo

Compilazione degli shaders

```
// Create shader program executable.
char* vertexShader = readTextFile("vertexShader.glsl");
vertexShaderId = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShaderId, 1, (const char**) &vertexShader, NULL);
glCompileShader(vertexShaderId);

char* fragmentShader = readTextFile("fragmentShader.glsl");
fragmentShaderId = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShaderId, 1, (const char**) &fragmentShader, NUL
glCompileShader(fragmentShaderId);

programId = glCreateProgram();
glAttachShader(programId, vertexShaderId);
glAttachShader(programId, fragmentShaderId);
glLinkProgram(programId);
glUseProgram(programId);
// ----------------------------------------
```

init()




glUseProgram — Installs a program object as part of current rendering state

C Specification

void **glUseProgram**(GLuint *program*);

Parameters

program

Specifies the handle of the program object whose executables are to be used as part of current rendering state.

Description

glUseProgram installs the program object specified by *program* as part of current rendering state. One or more executables are created in a program object by successfully attaching shader objects to it with glAttachShader, successfully compiling the shader objects with glCompileShader, and successfully linking the program object with glLinkProgram.

A program object will contain an executable that will run on the vertex processor if it contains one or more shader objects of type GL_VERTEX_SHADER that have been successfully compiled and linked. A program object will contain an executable that will run on the geometry processor if it contains one or more shader objects of type GL_GEOMETRY_SHADER that have been successfully compiled and linked. Similarly, a program object will contain an executable that will run on the fragment processor if it contains one or more shader objects of type GL_FRAGMENT_SHADER that have been successfully compiled and linked.

While a program object is in use, applications are free to modify attached shader objects, compile attached shader objects, attach additional shader objects, and detach or delete shader objects. None of these operations will affect the executables that are part of the current state. However, relinking the program object that is currently in use will install the program object as part of the current rendering state if the link operation was successful (see glLinkProgram). If the program object currently in use is relinked unsuccessfully, its link status will be set to GL_FALSE, but the executables and associated state will remain part of the current state until a subsequent call to **glUseProgram** removes it from use. After it is removed from use, it cannot be made part of current state until it has been successfully relinked.

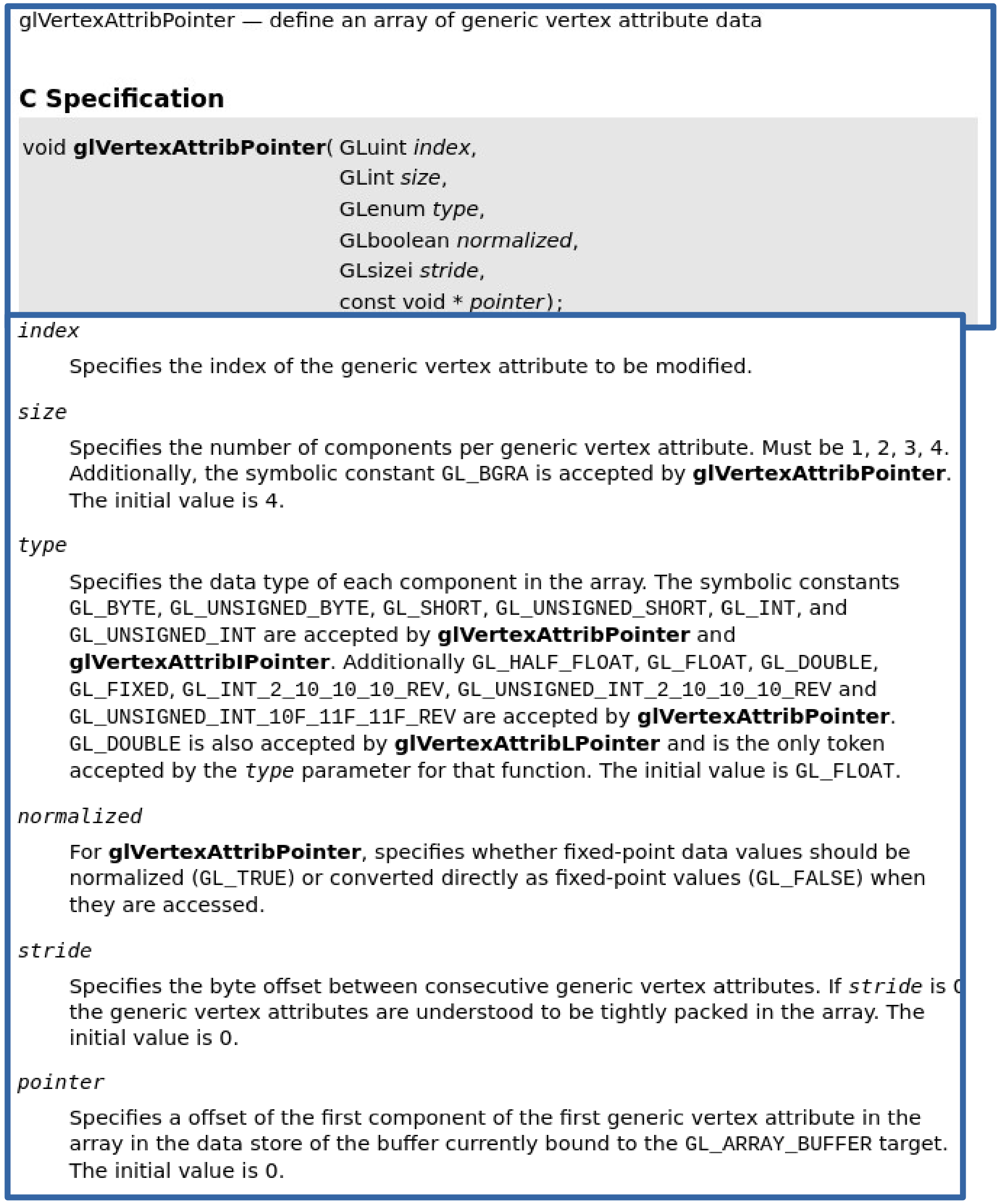
Creazione dei VAOs

```c
// Here wde define our data structures
typedef struct Vertex
{
    float coords[4];
    float colors[4];
} Vertex;

// vertex coordinates and colors
static Vertex squareVertices[4] =
{
    { { 20.0, 20.0, 0.0, 1.0 }, { 1.0, 0.0, 0.0, 1.0 } },
    { { 80.0, 20.0, 0.0, 1.0 }, { 1.0, 1.0, 0.0, 1.0 } },
    { { 20.0, 80.0, 0.0, 1.0 }, { 0.0, 0.0, 1.0, 1.0 } },
    { { 80.0, 80.0, 0.0, 1.0 }, { 0.0, 1.0, 0.0, 1.0 } }
};
```

```c
// Create VAO and VBO and associate data with vertex shader.
glGenVertexArrays(1, vao);
glGenBuffers(1, buffer);

glBindVertexArray(vao[0]);
glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(squareVertices),
    squareVertices, GL_STATIC_DRAW);

glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE,
    sizeof(squareVertices[0]), 0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE,
                      sizeof(squareVertices[0]),
                      (GLvoid*)sizeof(squareVertices[0].coords));
glEnableVertexAttribArray(1);
```

init()

```glsl
#version 430 core

layout(location=0) in vec4 squareCoords;
layout(location=1) in vec4 squareColors;
```



glVertexAttribPointer — define an array of generic vertex attribute data

C Specification

```
void glVertexAttribPointer( GLuint index,
                            GLint size,
                            GLenum type,
                            GLboolean normalized,
                            GLsizei stride,
                            const void * pointer);
```

index

Specifies the index of the generic vertex attribute to be modified.

size

Specifies the number of components per generic vertex attribute. Must be 1, 2, 3, 4. Additionally, the symbolic constant GL_BGRA is accepted by **glVertexAttribPointer**. The initial value is 4.

type

Specifies the data type of each component in the array. The symbolic constants GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, and GL_UNSIGNED_INT are accepted by **glVertexAttribPointer** and **glVertexAttribIPointer**. Additionally GL_HALF_FLOAT, GL_FLOAT, GL_DOUBLE, GL_FIXED, GL_INT_2_10_10_10_REV, GL_UNSIGNED_INT_2_10_10_10_REV and GL_UNSIGNED_INT_10F_11F_11F_REV are accepted by **glVertexAttribPointer**. GL_DOUBLE is also accepted by **glVertexAttribLPointer** and is the only token accepted by the *type* parameter for that function. The initial value is GL_FLOAT.

normalized

For **glVertexAttribPointer**, specifies whether fixed-point data values should be normalized (GL_TRUE) or converted directly as fixed-point values (GL_FALSE) when they are accessed.

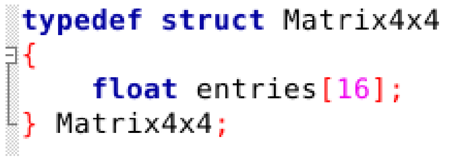
stride

Specifies the byte offset between consecutive generic vertex attributes. If *stride* is the generic vertex attributes are understood to be tightly packed in the array. The initial value is 0.

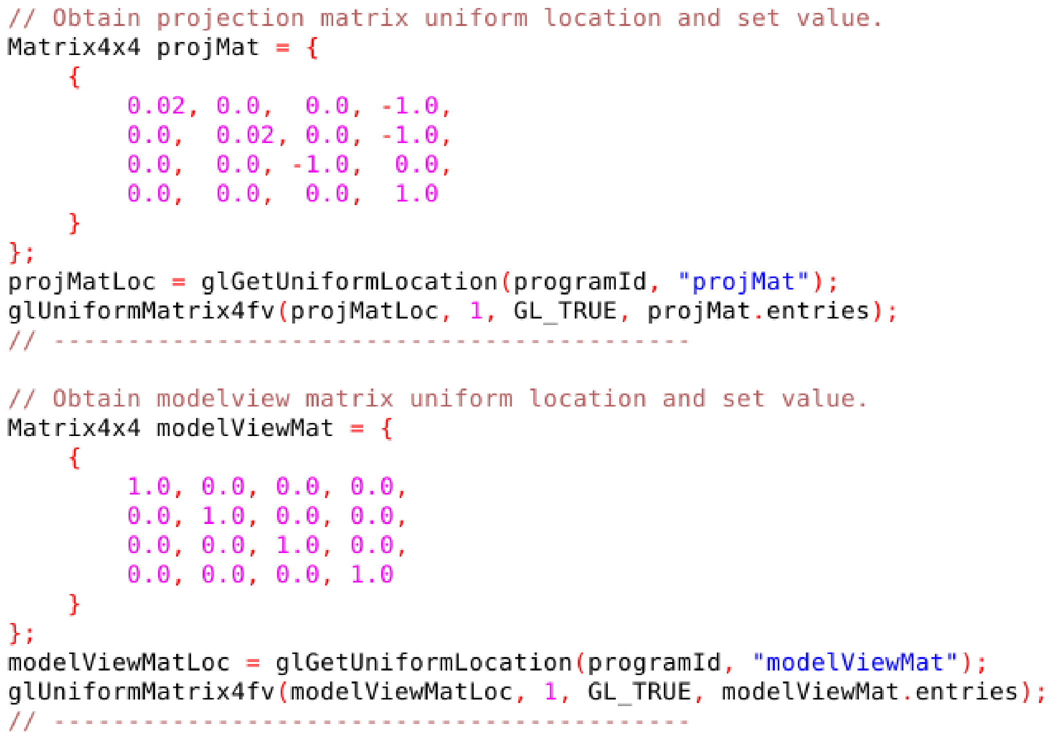
pointer

Specifies a offset of the first component of the first generic vertex attribute in the array in the data store of the buffer currently bound to the GL_ARRAY_BUFFER target. The initial value is 0.

Modelview e projection matrices



```
typedef struct Matrix4x4
{
    float entries[16];
} Matrix4x4;
```



```
// Obtain projection matrix uniform location and set value.
Matrix4x4 projMat = {
    {
        0.02, 0.0,  0.0, -1.0,
        0.0,  0.02, 0.0, -1.0,
        0.0,  0.0, -1.0,  0.0,
        0.0,  0.0,  0.0,  1.0
    }
};
projMatLoc = glGetUniformLocation(programId, "projMat");
glUniformMatrix4fv(projMatLoc, 1, GL_TRUE, projMat.entries);
// ------------------------------------------

// Obtain modelview matrix uniform location and set value.
Matrix4x4 modelViewMat = {
    {
        1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0
    }
};
modelViewMatLoc = glGetUniformLocation(programId, "modelViewMat");
glUniformMatrix4fv(modelViewMatLoc, 1, GL_TRUE, modelViewMat.entries);
// ------------------------------------------
```

init()

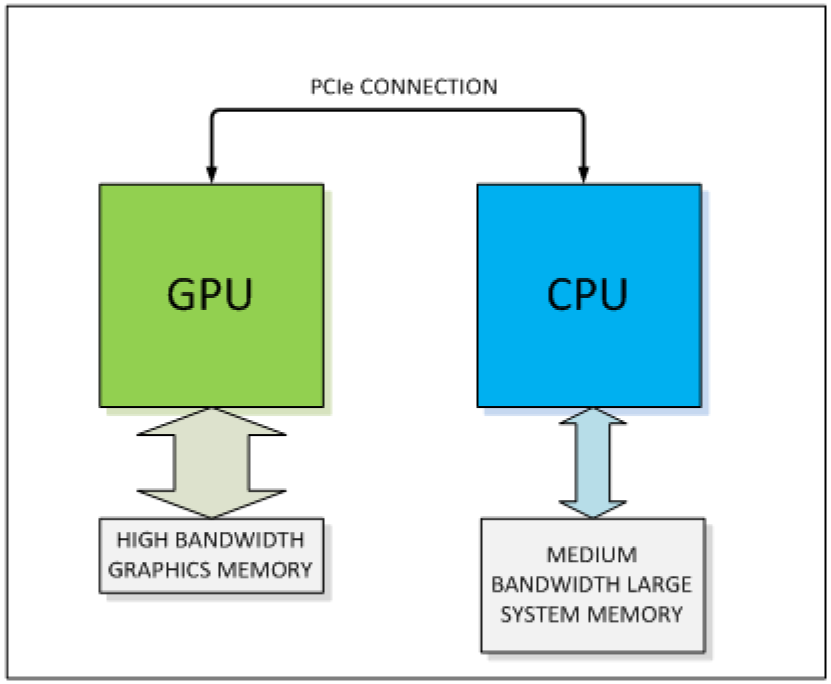
Domanda: Se GLSL offre gia' il tipo mat4x4, perche' dobbiamo definirlo?

Modelview e projection matrices

```
typedef struct Matrix4x4
{
    float entries[16];
} Matrix4x4;
```

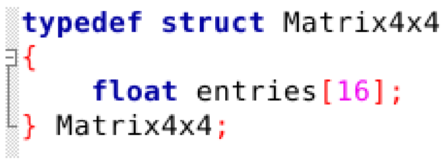
```
// Obtain projection matrix uniform location and set value.
Matrix4x4 projMat = {
    {
        0.02, 0.0,  0.0, -1.0,
        0.0,  0.02, 0.0, -1.0,
        0.0,  0.0, -1.0,  0.0,
        0.0,  0.0,  0.0,  1.0
    }
};
projMatLoc = glGetUniformLocation(programId, "projMat");
glUniformMatrix4fv(projMatLoc, 1, GL_TRUE, projMat.entries);
// ------------------------------------------

// Obtain modelview matrix uniform location and set value.
Matrix4x4 modelViewMat = {
    {
        1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0
    }
};
modelViewMatLoc = glGetUniformLocation(programId, "modelViewMat");
glUniformMatrix4fv(modelViewMatLoc, 1, GL_TRUE, modelViewMat.entries);
// ------------------------------------------
```

init()

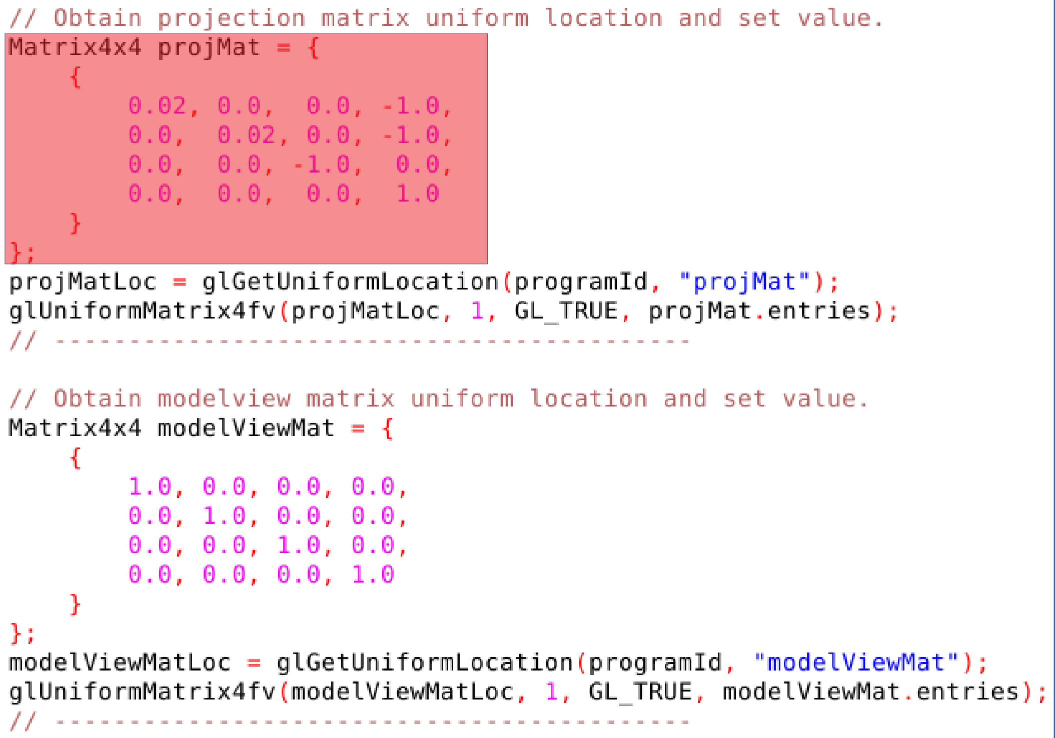

PCIe CONNECTION
GPU
CPU
HIGH BANDWIDTH GRAPHICS MEMORY
MEDIUM BANDWIDTH LARGE SYSTEM MEMORY

Risposta: GLSL e C sono <u>due linguaggi diversi</u> (anche se molto simili) con i propri tipi di dato. GLSL gira sulla GPU mentre il programma C gira sulla CPU!

Modelview e projection matrices

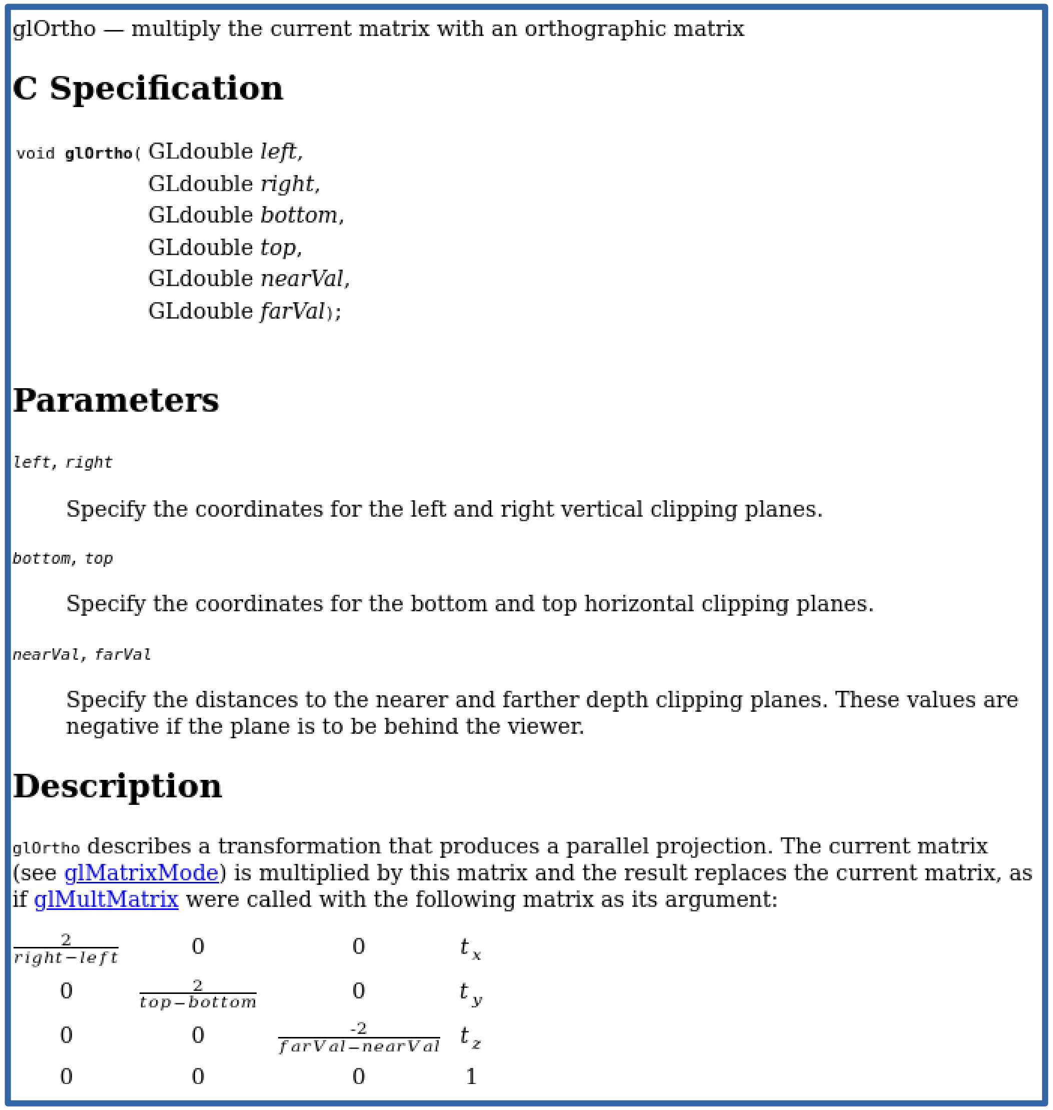


```c
typedef struct Matrix4x4
{
    float entries[16];
} Matrix4x4;
```



```c
// Obtain projection matrix uniform location and set value.
Matrix4x4 projMat = {
    {
        0.02, 0.0,  0.0, -1.0,
        0.0,  0.02, 0.0, -1.0,
        0.0,  0.0, -1.0,  0.0,
        0.0,  0.0,  0.0,  1.0
    }
};
projMatLoc = glGetUniformLocation(programId, "projMat");
glUniformMatrix4fv(projMatLoc, 1, GL_TRUE, projMat.entries);
// -------------------------------------------

// Obtain modelview matrix uniform location and set value.
Matrix4x4 modelViewMat = {
    {
        1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0
    }
};
modelViewMatLoc = glGetUniformLocation(programId, "modelViewMat");
glUniformMatrix4fv(modelViewMatLoc, 1, GL_TRUE, modelViewMat.entries);
// -------------------------------------------
```

init()



glOrtho — multiply the current matrix with an orthographic matrix

C Specification

```
void glOrtho( GLdouble left,
              GLdouble right,
              GLdouble bottom,
              GLdouble top,
              GLdouble nearVal,
              GLdouble farVal);
```

Parameters

left, right

Specify the coordinates for the left and right vertical clipping planes.

bottom, top

Specify the coordinates for the bottom and top horizontal clipping planes.

nearVal, farVal

Specify the distances to the nearer and farther depth clipping planes. These values are negative if the plane is to be behind the viewer.

Description

`glOrtho` describes a transformation that produces a parallel projection. The current matrix (see glMatrixMode) is multiplied by this matrix and the result replaces the current matrix, as if glMultMatrix were called with the following matrix as its argument:

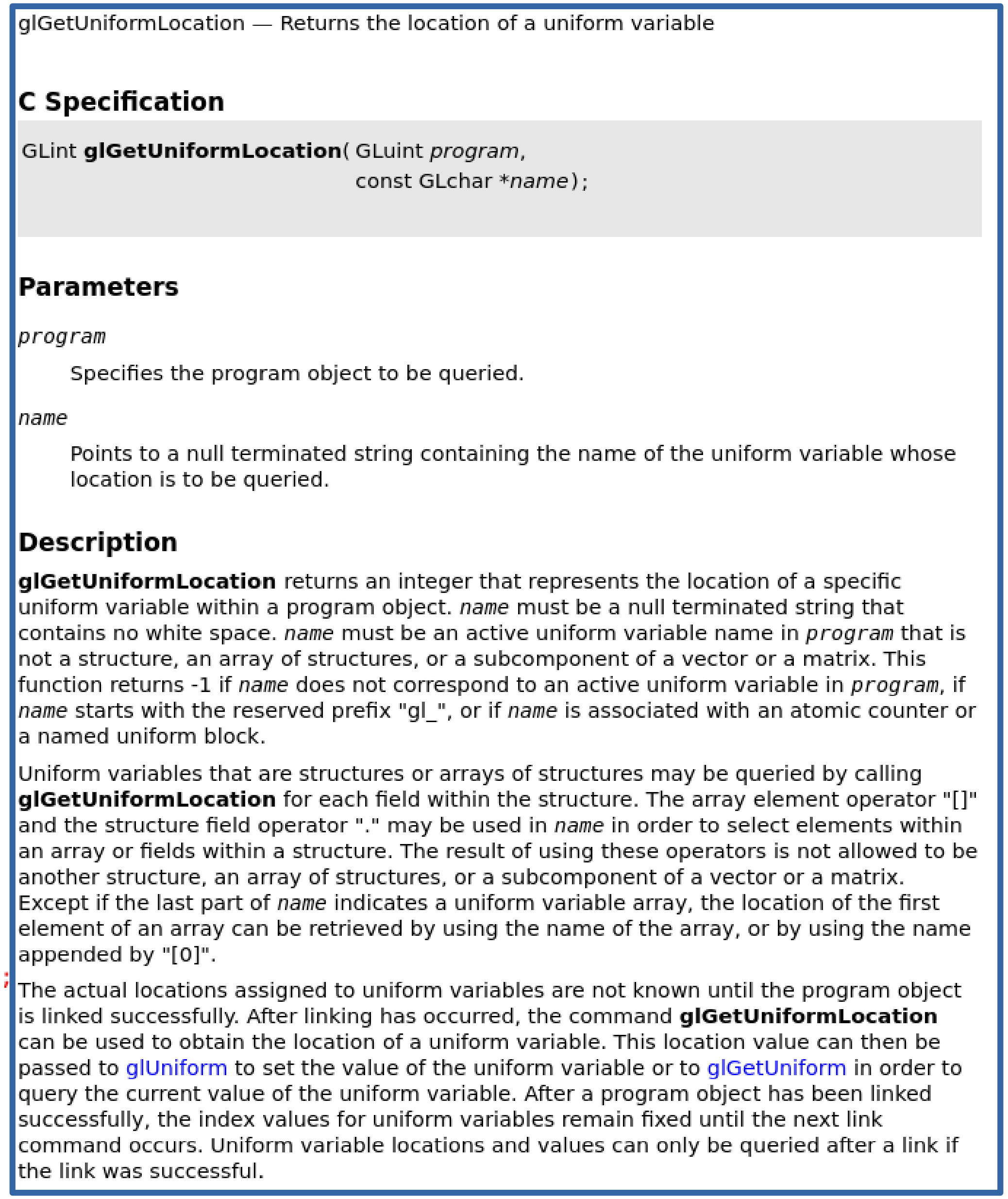
$$\begin{matrix} \frac{2}{right-left} & 0 & 0 & t_x \\ 0 & \frac{2}{top-bottom} & 0 & t_y \\ 0 & 0 & \frac{-2}{farVal-nearVal} & t_z \\ 0 & 0 & 0 & 1 \end{matrix}$$

Modelview e projection matrices

```
typedef struct Matrix4x4
{
    float entries[16];
} Matrix4x4;
```

```
// Obtain projection matrix uniform location and set value.
Matrix4x4 projMat = {
    {
        0.02, 0.0,  0.0, -1.0,
        0.0,  0.02, 0.0, -1.0,
        0.0,  0.0, -1.0,  0.0,
        0.0,  0.0,  0.0,  1.0
    }
};
projMatLoc = glGetUniformLocation(programId, "projMat");
glUniformMatrix4fv(projMatLoc, 1, GL_TRUE, projMat.entries);
// -------------------------------------------

// Obtain modelview matrix uniform location and set value.
Matrix4x4 modelViewMat = {
    {
        1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0
    }
};
modelViewMatLoc = glGetUniformLocation(programId, "modelViewMat");
glUniformMatrix4fv(modelViewMatLoc, 1, GL_TRUE, modelViewMat.entries);
// -------------------------------------------
```

init()



glGetUniformLocation — Returns the location of a uniform variable

C Specification

```
GLint glGetUniformLocation( GLuint program,
                            const GLchar *name);
```

Parameters

program

Specifies the program object to be queried.

name

Points to a null terminated string containing the name of the uniform variable whose location is to be queried.

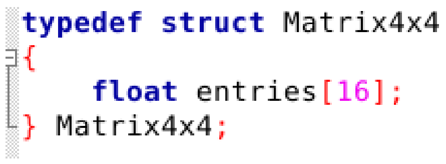
Description

glGetUniformLocation returns an integer that represents the location of a specific uniform variable within a program object. *name* must be a null terminated string that contains no white space. *name* must be an active uniform variable name in *program* that is not a structure, an array of structures, or a subcomponent of a vector or a matrix. This function returns -1 if *name* does not correspond to an active uniform variable in *program*, if *name* starts with the reserved prefix "gl_", or if *name* is associated with an atomic counter or a named uniform block.

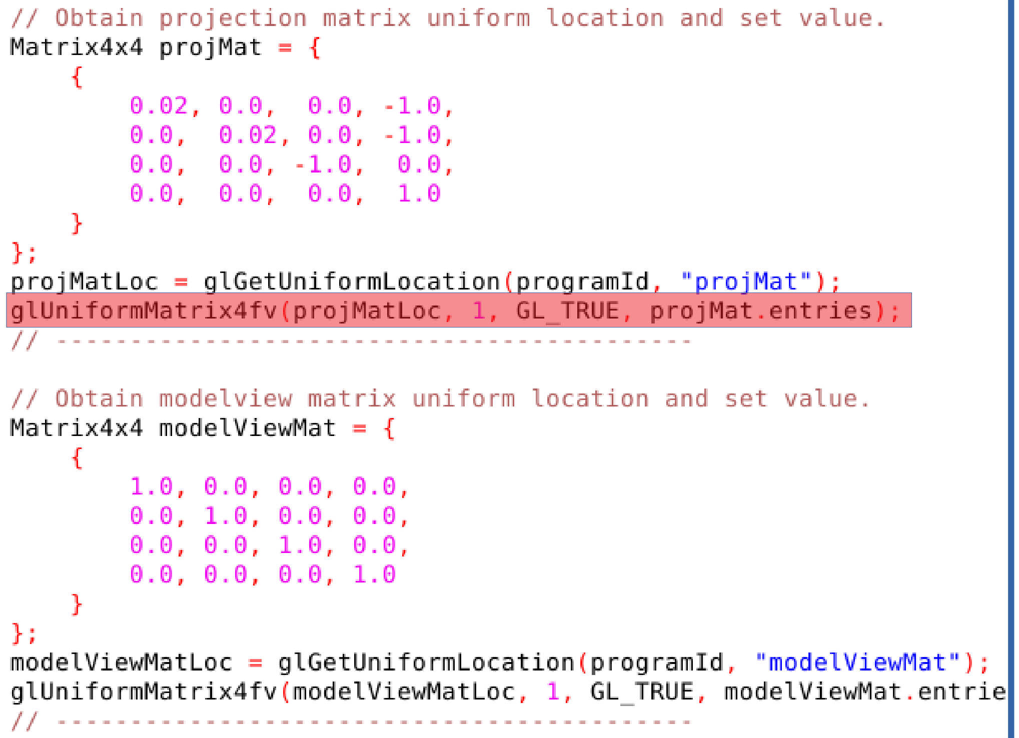
Uniform variables that are structures or arrays of structures may be queried by calling **glGetUniformLocation** for each field within the structure. The array element operator "[]" and the structure field operator "." may be used in *name* in order to select elements within an array or fields within a structure. The result of using these operators is not allowed to be another structure, an array of structures, or a subcomponent of a vector or a matrix. Except if the last part of *name* indicates a uniform variable array, the location of the first element of an array can be retrieved by using the name of the array, or by using the name appended by "[0]".

The actual locations assigned to uniform variables are not known until the program object is linked successfully. After linking has occurred, the command **glGetUniformLocation** can be used to obtain the location of a uniform variable. This location value can then be passed to glUniform to set the value of the uniform variable or to glGetUniform in order to query the current value of the uniform variable. After a program object has been linked successfully, the index values for uniform variables remain fixed until the next link command occurs. Uniform variable locations and values can only be queried after a link if the link was successful.

Modelview e projection matrices

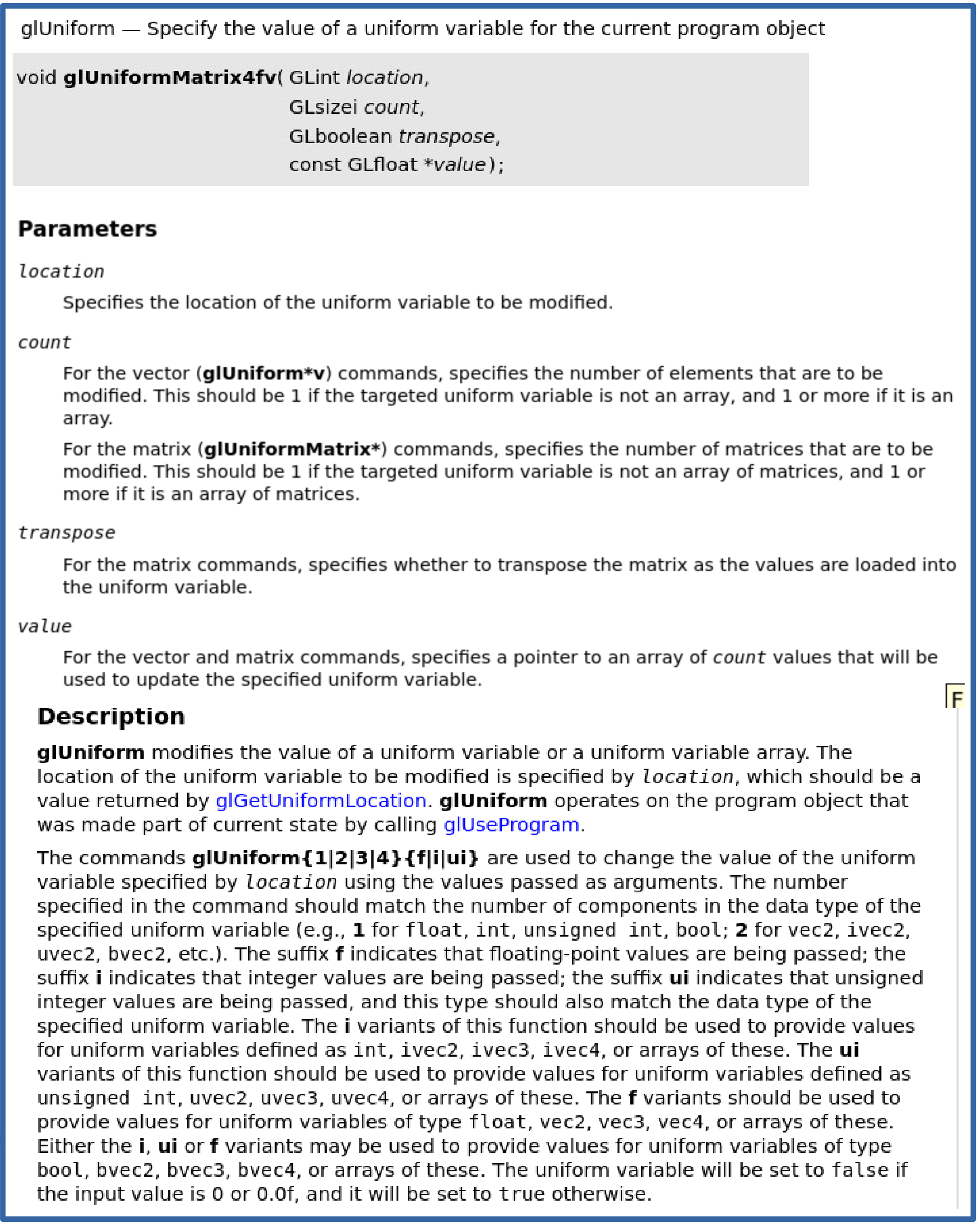


```
typedef struct Matrix4x4
{
    float entries[16];
} Matrix4x4;
```



```
// Obtain projection matrix uniform location and set value.
Matrix4x4 projMat = {
    {
        0.02, 0.0,  0.0, -1.0,
        0.0,  0.02, 0.0, -1.0,
        0.0,  0.0, -1.0,  0.0,
        0.0,  0.0,  0.0,  1.0
    }
};
projMatLoc = glGetUniformLocation(programId, "projMat");
glUniformMatrix4fv(projMatLoc, 1, GL_TRUE, projMat.entries);
// ------------------------------------------

// Obtain modelview matrix uniform location and set value.
Matrix4x4 modelViewMat = {
    {
        1.0, 0.0, 0.0, 0.0,
        0.0, 1.0, 0.0, 0.0,
        0.0, 0.0, 1.0, 0.0,
        0.0, 0.0, 0.0, 1.0
    }
};
modelViewMatLoc = glGetUniformLocation(programId, "modelViewMat");
glUniformMatrix4fv(modelViewMatLoc, 1, GL_TRUE, modelViewMat.entrie
// ------------------------------------------
```

init()



glUniform — Specify the value of a uniform variable for the current program object

void **glUniformMatrix4fv**(GLint *location*,
GLsizei *count*,
GLboolean *transpose*,
const GLfloat **value*);

Parameters

location

Specifies the location of the uniform variable to be modified.

count

For the vector (**glUniform*v**) commands, specifies the number of elements that are to be modified. This should be 1 if the targeted uniform variable is not an array, and 1 or more if it is an array.

For the matrix (**glUniformMatrix***) commands, specifies the number of matrices that are to be modified. This should be 1 if the targeted uniform variable is not an array of matrices, and 1 or more if it is an array of matrices.

transpose

For the matrix commands, specifies whether to transpose the matrix as the values are loaded into the uniform variable.

value

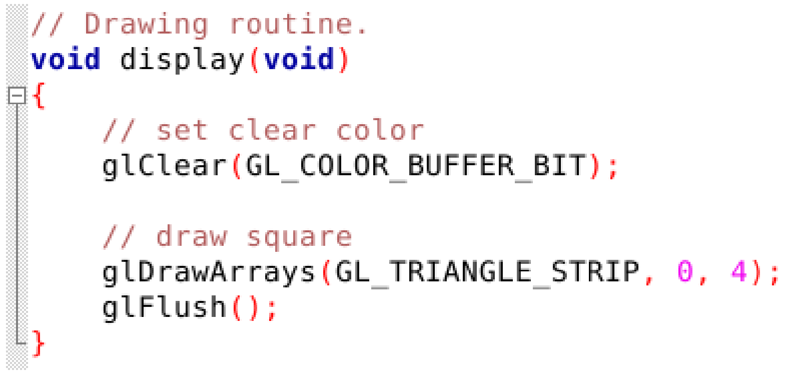
For the vector and matrix commands, specifies a pointer to an array of *count* values that will be used to update the specified uniform variable.

Description

glUniform modifies the value of a uniform variable or a uniform variable array. The location of the uniform variable to be modified is specified by *location*, which should be a value returned by glGetUniformLocation. **glUniform** operates on the program object that was made part of current state by calling glUseProgram.

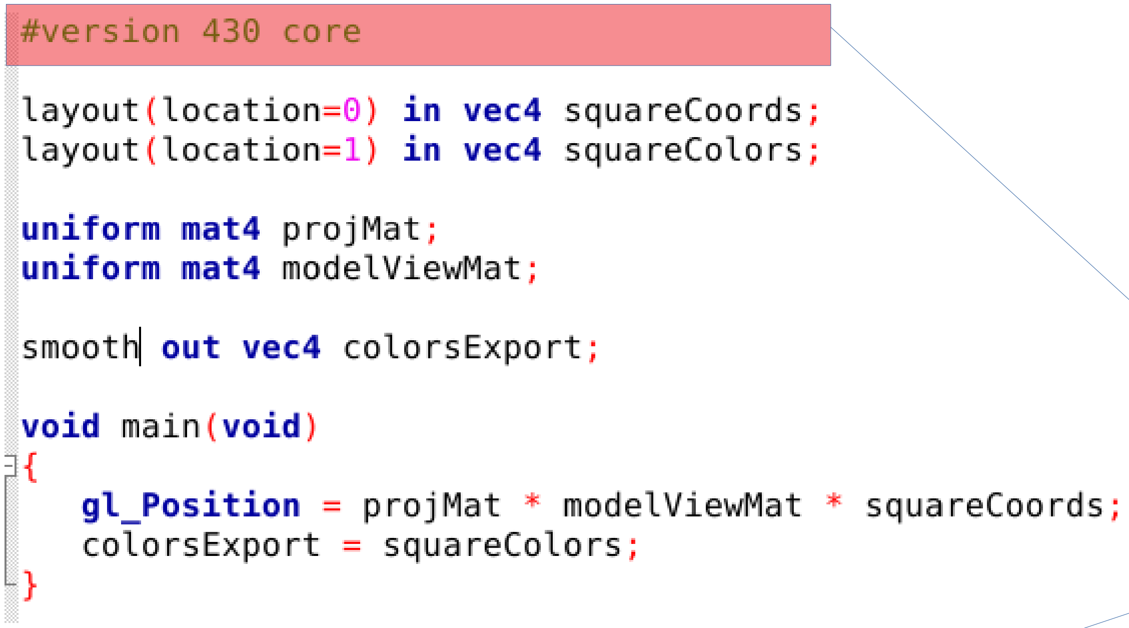
The commands **glUniform{1|2|3|4}{f|i|ui}** are used to change the value of the uniform variable specified by *location* using the values passed as arguments. The number specified in the command should match the number of components in the data type of the specified uniform variable (e.g., **1** for float, int, unsigned int, bool; **2** for vec2, ivec2, uvec2, bvec2, etc.). The suffix **f** indicates that floating-point values are being passed; the suffix **i** indicates that integer values are being passed; the suffix **ui** indicates that unsigned integer values are being passed, and this type should also match the data type of the specified uniform variable. The **i** variants of this function should be used to provide values for uniform variables defined as int, ivec2, ivec3, ivec4, or arrays of these. The **ui** variants of this function should be used to provide values for uniform variables defined as unsigned int, uvec2, uvec3, uvec4, or arrays of these. The **f** variants should be used to provide values for uniform variables of type float, vec2, vec3, vec4, or arrays of these. Either the **i**, **ui** or **f** variants may be used to provide values for uniform variables of type bool, bvec2, bvec3, bvec4, or arrays of these. The uniform variable will be set to false if the input value is 0 or 0.0f, and it will be set to true otherwise.

Display function



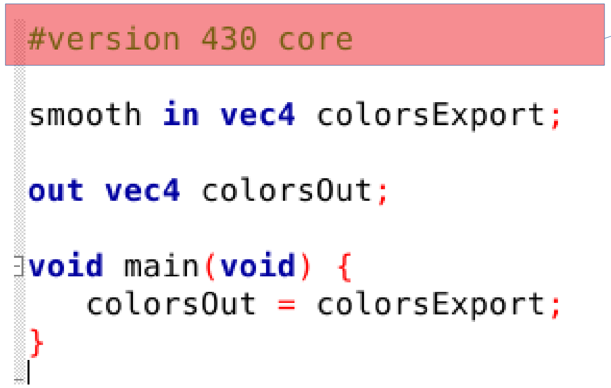
```
// Drawing routine.
void display(void)
{
    // set clear color
    glClear(GL_COLOR_BUFFER_BIT);

    // draw square
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    glFlush();
}
```

Vertex shader



```
#version 430 core

layout(location=0) in vec4 squareCoords;
layout(location=1) in vec4 squareColors;

uniform mat4 projMat;
uniform mat4 modelViewMat;

smooth out vec4 colorsExport;

void main(void)
{
   gl_Position = projMat * modelViewMat * squareCoords;
   colorsExport = squareColors;
}
```

vertex shader

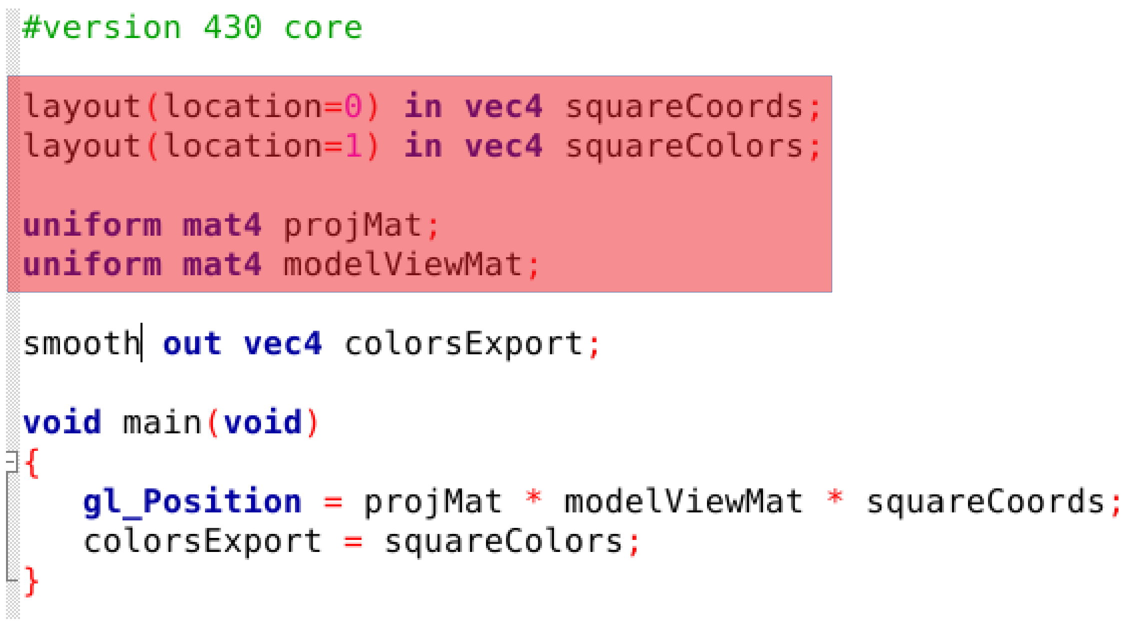


```
#version 430 core

smooth in vec4 colorsExport;

out vec4 colorsOut;

void main(void) {
   colorsOut = colorsExport;
}
```

fragment shader

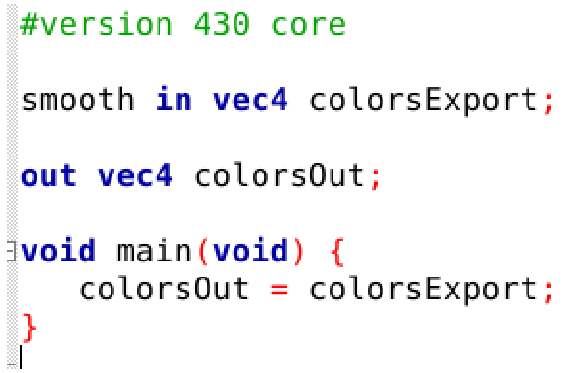
Definizione del contesto (i.e. versione) dello shader

Vertex shader



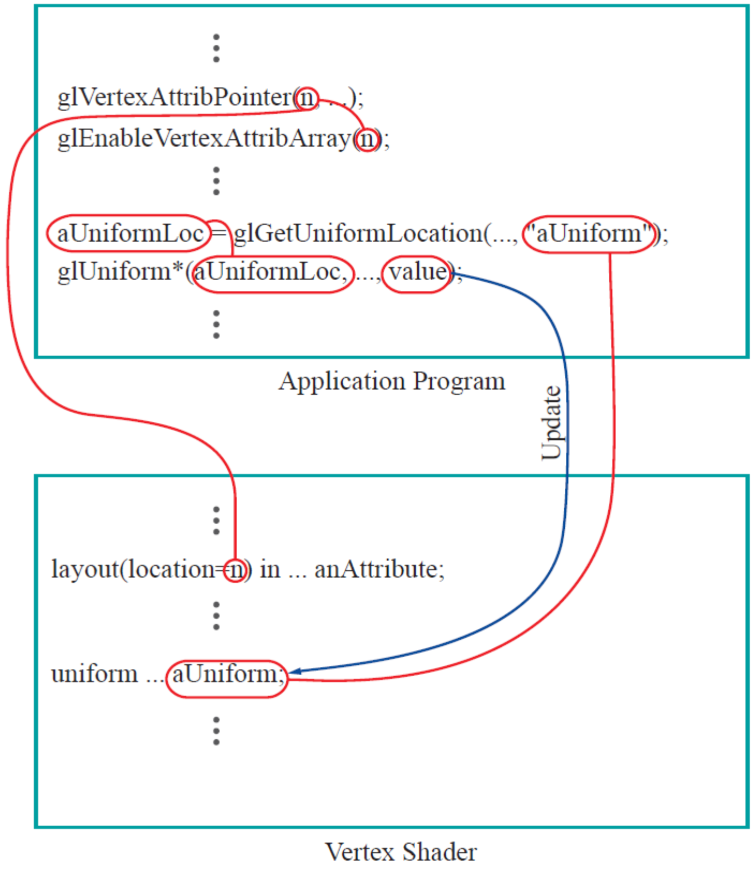
```
#version 430 core

layout(location=0) in vec4 squareCoords;
layout(location=1) in vec4 squareColors;

uniform mat4 projMat;
uniform mat4 modelViewMat;

smooth out vec4 colorsExport;

void main(void)
{
   gl_Position = projMat * modelViewMat * squareCoords;
   colorsExport = squareColors;
}
```

vertex shader



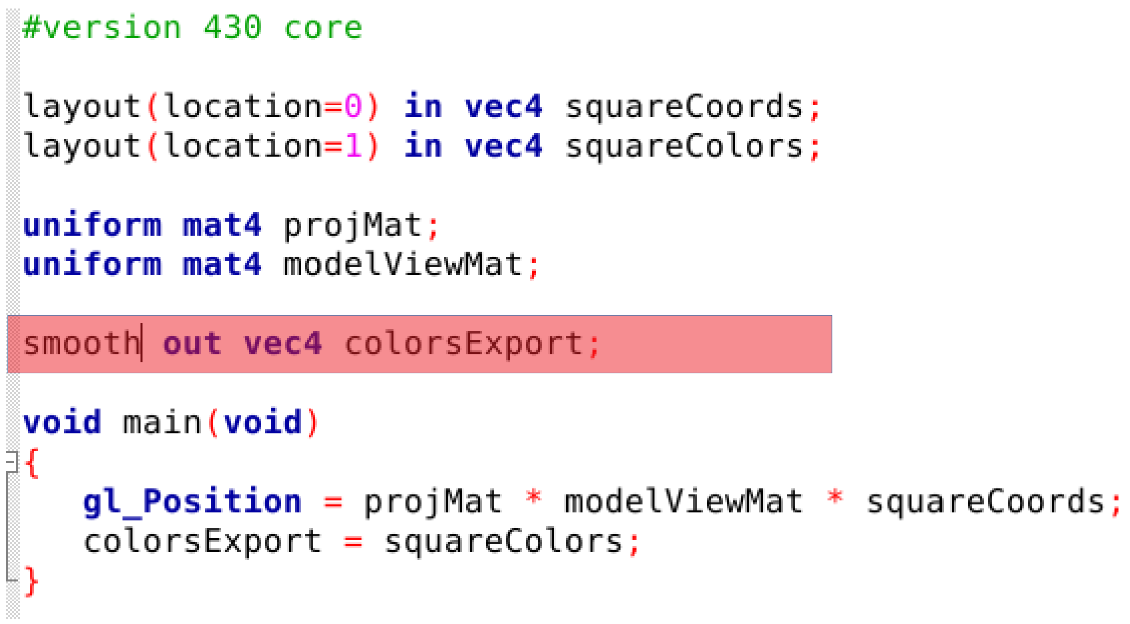
```
#version 430 core

smooth in vec4 colorsExport;

out vec4 colorsOut;

void main(void) {
   colorsOut = colorsExport;
}
```

fragment shader


glVertexAttribPointer(n, ...);
glEnableVertexAttribArray(n);
aUniformLoc = glGetUniformLocation(..., "aUniform");
glUniform*(aUniformLoc, ..., value);
Application Program
Update
layout(location=n) in ... anAttribute;
uniform ... aUniform;
Vertex Shader

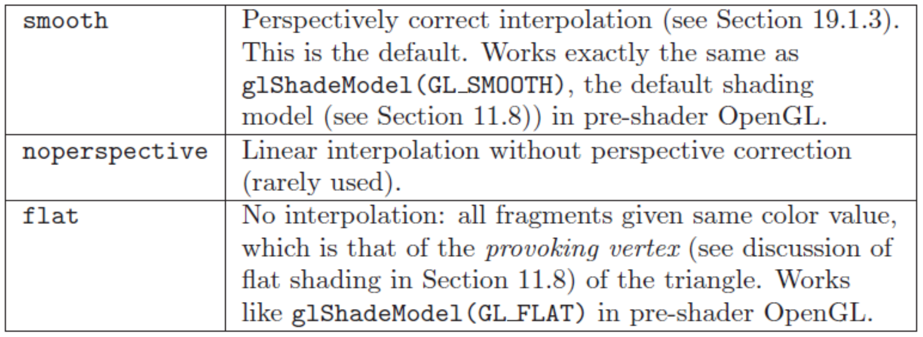
`const`	Read-only variable whose value is fixed after initialization.
`in`	Variable whose value is input from a previous shader stage or the application program.
`out`	Variable whose value is output to a subsequent shader stage.
`uniform`	Variable whose value is supplied to the shader by the application and is constant across a primitive.
`buffer`	Variable which can be read and written by both the shader and the application.
`shared`	Variable shared within a local work group (only compute shaders).

Vertex shader



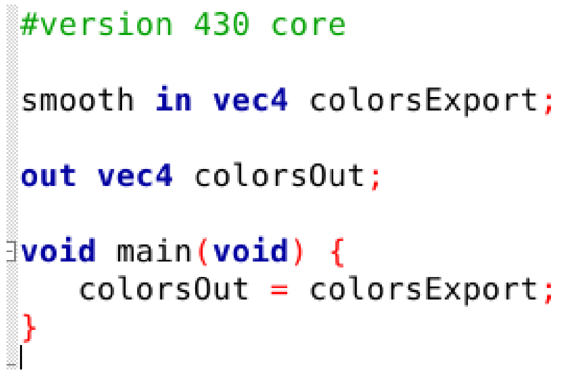
```
#version 430 core

layout(location=0) in vec4 squareCoords;
layout(location=1) in vec4 squareColors;

uniform mat4 projMat;
uniform mat4 modelViewMat;

smooth out vec4 colorsExport;

void main(void)
{
   gl_Position = projMat * modelViewMat * squareCoords;
   colorsExport = squareColors;
}
```

vertex shader



`smooth`	Perspectively correct interpolation (see Section 19.1.3). This is the default. Works exactly the same as `glShadeModel(GL_SMOOTH)`, the default shading model (see Section 11.8)) in pre-shader OpenGL.
`noperspective`	Linear interpolation without perspective correction (rarely used).
`flat`	No interpolation: all fragments given same color value, which is that of the *provoking vertex* (see discussion of flat shading in Section 11.8) of the triangle. Works like `glShadeModel(GL_FLAT)` in pre-shader OpenGL.

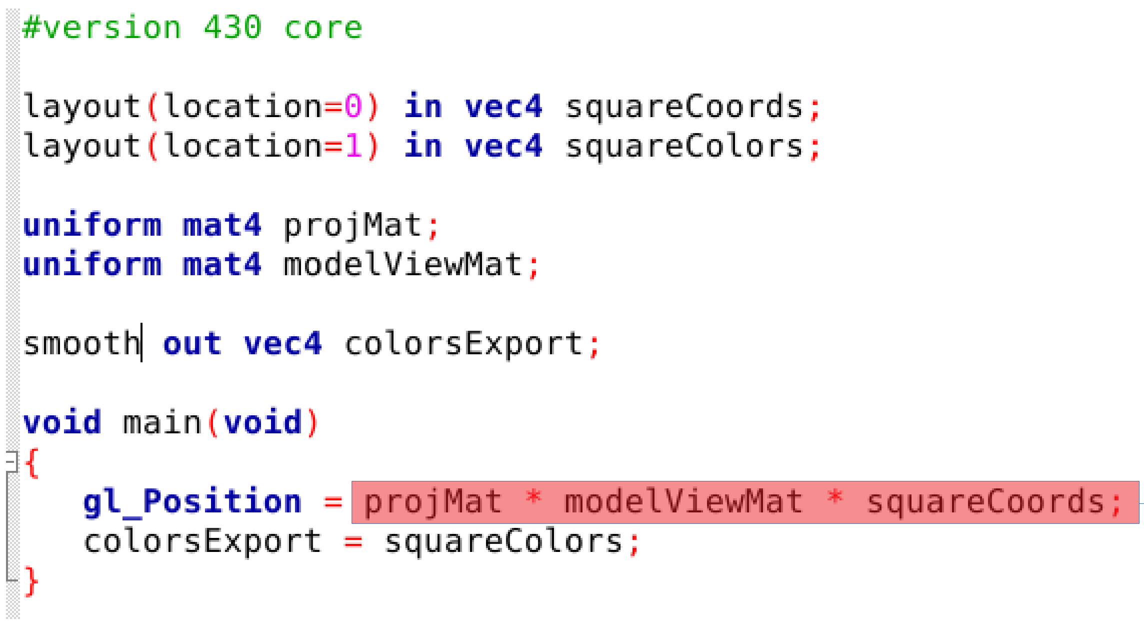
Metodo di interpolazione del colore



```
#version 430 core

smooth in vec4 colorsExport;

out vec4 colorsOut;

void main(void) {
   colorsOut = colorsExport;
}
```

fragment shader

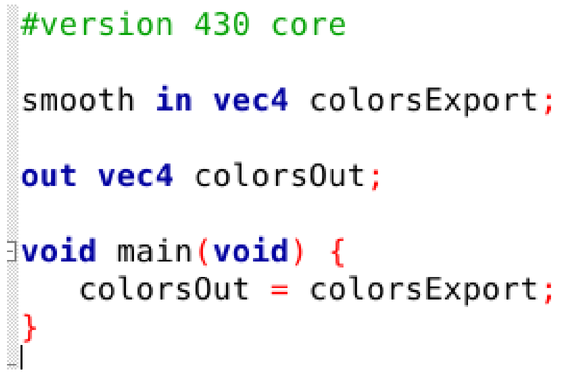
Vertex shader



```
#version 430 core

layout(location=0) in vec4 squareCoords;
layout(location=1) in vec4 squareColors;

uniform mat4 projMat;
uniform mat4 modelViewMat;

smooth out vec4 colorsExport;

void main(void)
{
   gl_Position = projMat * modelViewMat * squareCoords;
   colorsExport = squareColors;
}
```

Applicazione delle trasformazioni di modelview e proiezione

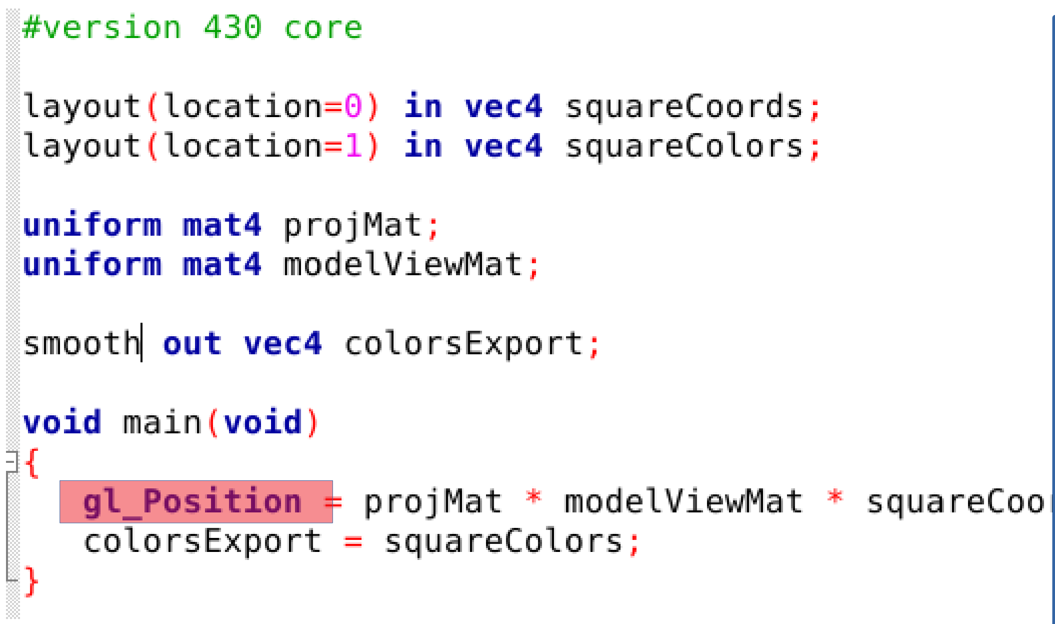
vertex shader



```
#version 430 core

smooth in vec4 colorsExport;

out vec4 colorsOut;

void main(void) {
   colorsOut = colorsExport;
}
```

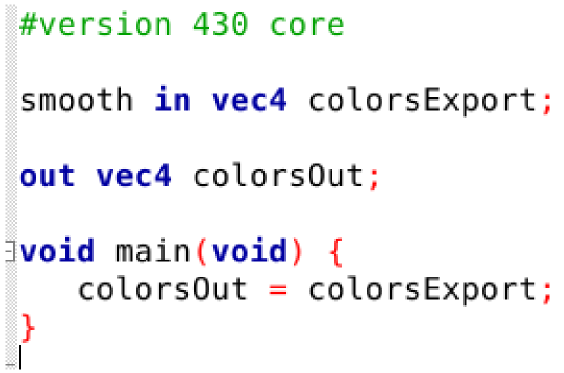
fragment shader

Vertex shader



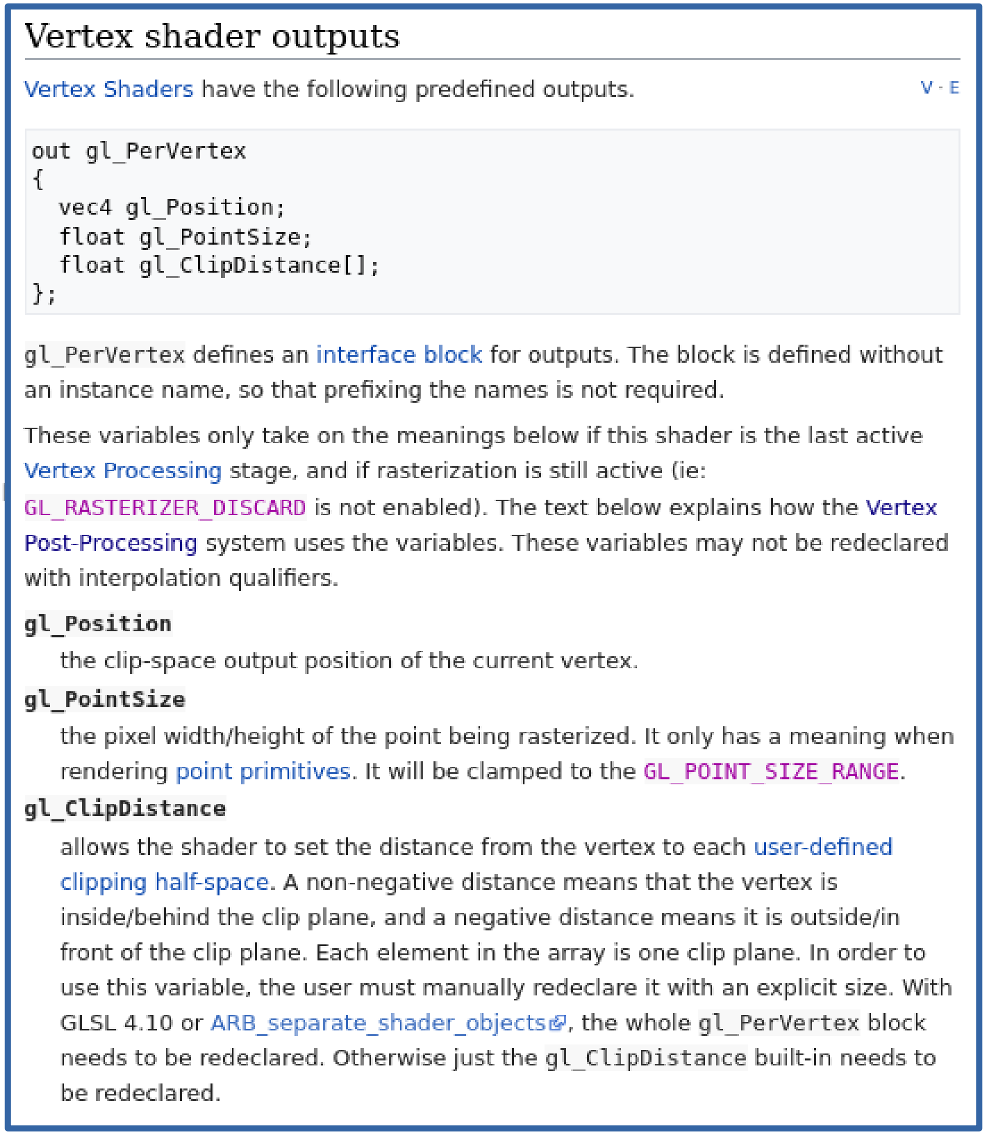
```
#version 430 core

layout(location=0) in vec4 squareCoords;
layout(location=1) in vec4 squareColors;

uniform mat4 projMat;
uniform mat4 modelViewMat;

smooth out vec4 colorsExport;

void main(void)
{
   gl_Position = projMat * modelViewMat * squareCoo
   colorsExport = squareColors;
}
```

vertex shader



```
#version 430 core

smooth in vec4 colorsExport;

out vec4 colorsOut;

void main(void) {
   colorsOut = colorsExport;
}
```

fragment shader



Vertex shader outputs

Vertex Shaders have the following predefined outputs. V · E

```
out gl_PerVertex
{
  vec4 gl_Position;
  float gl_PointSize;
  float gl_ClipDistance[];
};
```

`gl_PerVertex` defines an interface block for outputs. The block is defined without an instance name, so that prefixing the names is not required.

These variables only take on the meanings below if this shader is the last active Vertex Processing stage, and if rasterization is still active (ie: `GL_RASTERIZER_DISCARD` is not enabled). The text below explains how the Vertex Post-Processing system uses the variables. These variables may not be redeclared with interpolation qualifiers.

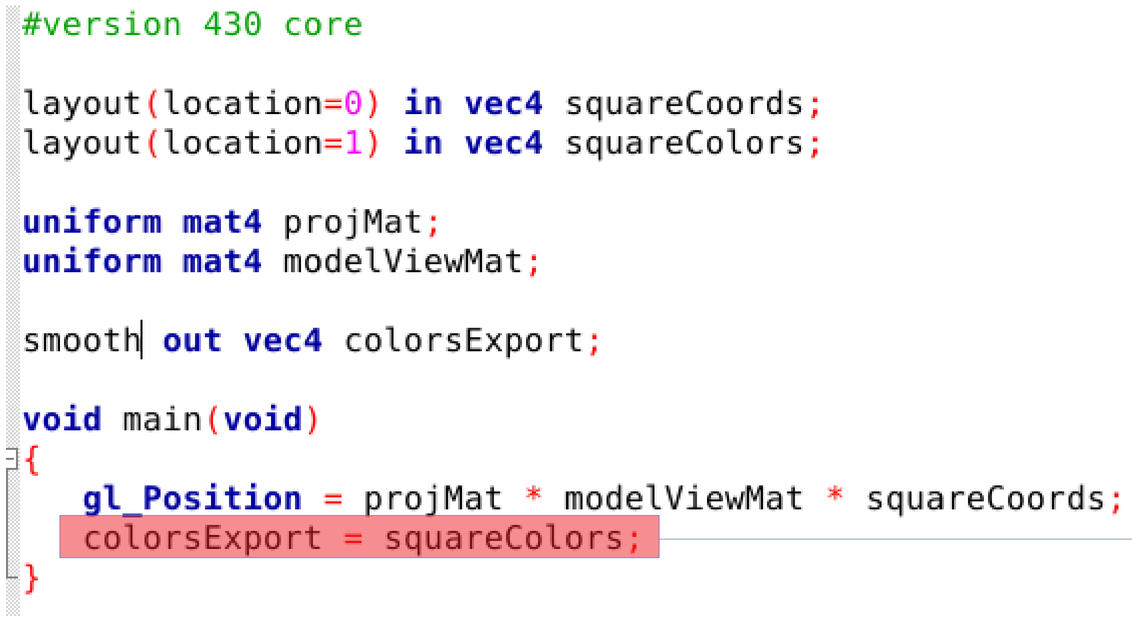
`gl_Position`
the clip-space output position of the current vertex.

`gl_PointSize`
the pixel width/height of the point being rasterized. It only has a meaning when rendering point primitives. It will be clamped to the `GL_POINT_SIZE_RANGE`.

`gl_ClipDistance`
allows the shader to set the distance from the vertex to each user-defined clipping half-space. A non-negative distance means that the vertex is inside/behind the clip plane, and a negative distance means it is outside/in front of the clip plane. Each element in the array is one clip plane. In order to use this variable, the user must manually redeclare it with an explicit size. With GLSL 4.10 or ARB_separate_shader_objects, the whole `gl_PerVertex` block needs to be redeclared. Otherwise just the `gl_ClipDistance` built-in needs to be redeclared.

https://www.khronos.org/opengl/wiki/Built-in_Variable_(GLSL)

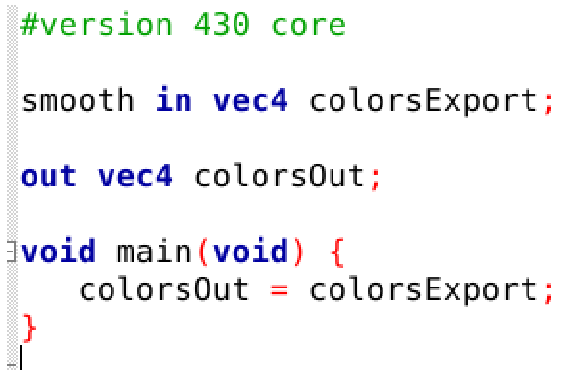
Vertex shader



```
#version 430 core

layout(location=0) in vec4 squareCoords;
layout(location=1) in vec4 squareColors;

uniform mat4 projMat;
uniform mat4 modelViewMat;

smooth out vec4 colorsExport;

void main(void)
{
    gl_Position = projMat * modelViewMat * squareCoords;
    colorsExport = squareColors;
}
```

Si setta il colore del vertice

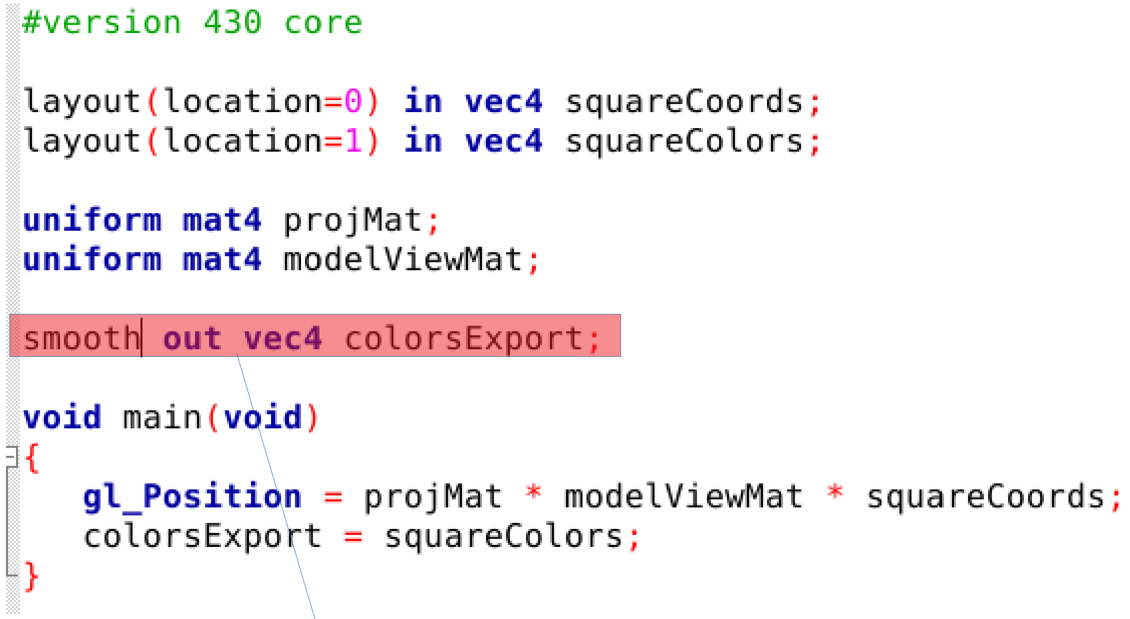
vertex shader



```
#version 430 core

smooth in vec4 colorsExport;

out vec4 colorsOut;

void main(void) {
    colorsOut = colorsExport;
}
```

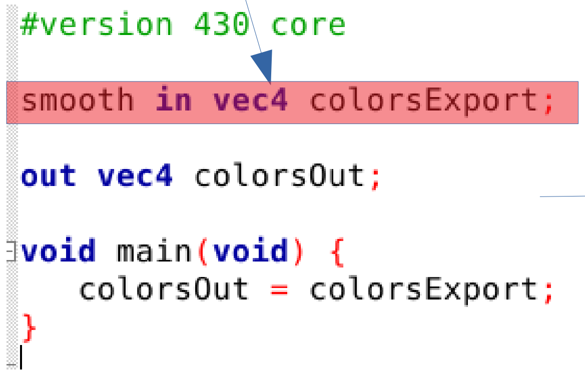
fragment shader

Fragment shader



```
#version 430 core

layout(location=0) in vec4 squareCoords;
layout(location=1) in vec4 squareColors;

uniform mat4 projMat;
uniform mat4 modelViewMat;

smooth out vec4 colorsExport;

void main(void)
{
   gl_Position = projMat * modelViewMat * squareCoords;
   colorsExport = squareColors;
}
```

vertex shader



```
#version 430 core

smooth in vec4 colorsExport;

out vec4 colorsOut;

void main(void) {
   colorsOut = colorsExport;
}
```

fragment shader

Semplice pass-through fragment shader