# **OPENGL**

## Schema di funzionamento overall

**Input Obbligatori:**

VAO???

**Input Opzionali:**

Vertex Buffer object

I

**Shaders**

(i vertici vengono definiti qui o passati in input (ad esempio con un buffer object)

[**Vertex Attributes**](#_Vertex_Attribute)

Rendering







**void glCreateVertexArrays(GLsizei n,**

**GLuint \* arrays);**

**void glBindVertexArray(GLuint array);**

**glUseProgram() tell OpenGL to use our program object for rendering and then**

**call our first drawing command, glDrawArrays().**

**glCreateShader()** creates an empty shader object, ready to accept source

code and be compiled.

• **glShaderSource()** hands shader source code to the shader object so that it can

keep a copy of it.

• **glCompileShader()** compiles whatever source code is contained in the

shader object.

• **glCreateProgram()** creates a program object to which you can attach shader

objects.

• **glAttachShader()** attaches a shader object to a program object.

• **glLinkProgram()** links all of the shader objects attached to a program object

together.

• **glDeleteShader()** deletes a shader object. Once a shader has been linked into

a program object, the program contains the binary code and the shader is no longer

needed.

## **CAPITOLO 2 : First program in OGL**

### **First program**

#### Using Shaders

The shader stages of OpenGL are vertex shaders, tessellation control and evaluation shaders, geometry shaders,

fragment shaders, and compute shaders. The minimal useful pipeline configuration

consists of only a vertex shader4 (or just a compute shader), but if you wish to see any

pixels on the screen, you will also need a fragment shader.

**#version** 450 core

**void** main(**void**)

{

gl\_Position = **vec4**(0.0, 0.0, 0.5, 1.0);

}

Listing 2.3: Our first vertex shader

---------------------------------------------------------------

**#version** 450 core

**out vec4** color;

**void** main(**void**)

{

color = **vec4**(0.0, 0.8, 1.0, 1.0);

}

Listing 2.4: Our first fragment shader

#### Shader Program

Now that we have both a vertex and a fragment shader, it’s time to compile them and

link them together into a program that can be run by OpenGL.

GLuint compile\_shaders(**void**)

{

GLuint vertex\_shader;

GLuint fragment\_shader;

GLuint program;

// Source code for vertex shader

**static const** GLchar \* vertex\_shader\_source[] =

{

"#version 450 core \n"

" \n"

"void main(void) \n"

"{ \n"

" gl\_Position = vec4(0.0, 0.0, 0.5, 1.0); \n"

"} \n"

};

// Source code for fragment shader

**static const** GLchar \* fragment\_shader\_source[] =

{

"#version 450 core \n"

" \n"

"out vec4 color; \n"

" \n"

"void main(void) \n"

"{ \n"

" color = vec4(0.0, 0.8, 1.0, 1.0); \n"

"} \n"

};

// Create and compile vertex shader

vertex\_shader = glCreateShader(GL\_VERTEX\_SHADER);

glShaderSource(vertex\_shader, 1, vertex\_shader\_source, NULL);

glCompileShader(vertex\_shader);

// Create and compile fragment shader

fragment\_shader = glCreateShader(GL\_FRAGMENT\_SHADER);

glShaderSource(fragment\_shader, 1, fragment\_shader\_source, NULL);

glCompileShader(fragment\_shader);

// Create program, attach shaders to it, and link it

program = glCreateProgram();

glAttachShader(program, vertex\_shader);

glAttachShader(program, fragment\_shader);

glLinkProgram(program);

// Delete the shaders as the program has them now

glDeleteShader(vertex\_shader);

glDeleteShader(fragment\_shader);

**return** program;

}

Listing 2.5: Compiling a simple shader

When we call this function, we need to keep the returned program object somewhere so

that we can use it to draw things. Also, we really don’t want to recompile the whole

program every time we want to use it. So, we need a function that is called once when the program starts up (see paragraph : [Structure of a Graphics Application](#_Structure_of_a))

#### VAO

*( non supportato in versioni < 4.5)*

*Vertex array object* (VAO), which is an object that represents the vertex fetch stage of the

OpenGL pipeline and is used to supply input to the vertex shader.

##### When you need a VAO?

VAO ***it’s not*** mandatory in order to draw something, that is, you can render whatever you like with no use of any VAO.

The point of a VAO is to run all the methods necessary to draw an object once during initialization and cut out all the extra method call overhead during the main loop.

The best practise is to have multiple VAOs and switch between them when drawing.

here's how you should organize your code:

void Setup() {

glGenVertexArrays(..);

glBindVertexArray(..);

// now setup all your VertexAttribPointers that will be bound to this VAO

glBindBuffer(..);

glVertexAttribPointer(..);

glEnableVertexAttribArray(..);

}

void Render() {

glBindVertexArray(vao);

// that's it, now call one of glDraw... functions

// no need to set up vertex attrib pointers and buffers!

glDrawXYZ(..)

glBindVertexArray(0); //call this to unbind the VAO;

}

This avoids the mess of binding/unbinding buffers and passing all the settings for each vertex attribute and replaces it with just a single method call, binding a VAO.

##### How to create VAO

**void** glCreateVertexArrays(GLsizei n, GLuint \* arrays);

**void** glBindVertexArray(GLuint array);

The vertex array object maintains all of the state related to the input to the OpenGL

pipeline.

#### Structure of a Graphics Application

**class** my\_application : **public** sb7::application

{

**public**:

// <snip>

**void** startup()

{

rendering\_program = compile\_shaders();

glCreateVertexArrays(1, &vertex\_array\_object);

glBindVertexArray(vertex\_array\_object);

}

**void** shutdown()

{

glDeleteVertexArrays(1, &vertex\_array\_object);

glDeleteProgram(rendering\_program);

glDeleteVertexArrays(1, &vertex\_array\_object);

}

**private**:

GLuint rendering\_program;

GLuint vertex\_array\_object;

};

#### Rendering Function

glDrawArrays — render primitives from array data

**C Specification**

|  |  |
| --- | --- |
| void **glDrawArrays**( | GLenum *mode*, |
|  | GLint *first*, |
|  | GLsizei *count*); |

**Parameters**

*mode*

Specifies what kind of primitives to render. Symbolic constants GL\_POINTS, GL\_LINE\_STRIP, GL\_LINE\_LOOP, GL\_LINES, GL\_LINE\_STRIP\_ADJACENCY, GL\_LINES\_ADJACENCY, GL\_TRIANGLE\_STRIP, GL\_TRIANGLE\_FAN, GL\_TRIANGLES, GL\_TRIANGLE\_STRIP\_ADJACENCY, GL\_TRIANGLES\_ADJACENCY and GL\_PATCHES are accepted.

*first*

Specifies the starting index in the enabled arrays.

count

Specifies the number of indices to be rendered.

**Description**

glDrawArrays specifies multiple geometric primitives with very few subroutine calls. Instead of calling a GL procedure to pass each individual vertex, normal, texture coordinate, edge flag, or color, you can prespecify separate arrays of vertices, normals, and colors and use them to construct a sequence of primitives with a single call to glDrawArrays.

When glDrawArrays is called, it uses count sequential elements from each enabled array to construct a sequence of geometric primitives, beginning with element first. mode specifies what kind of primitives are constructed and how the array elements construct those primitives.

Vertex attributes that are modified by glDrawArrays have an unspecified value after glDrawArrays returns. Attributes that aren't modified remain well defined.

**Notes**

GL\_LINE\_STRIP\_ADJACENCY, GL\_LINES\_ADJACENCY, GL\_TRIANGLE\_STRIP\_ADJACENCY and GL\_TRIANGLES\_ADJACENCY are available only if the GL version is 3.2 or greater.

// Our rendering function

**void** render(**double** currentTime)

{

**const** GLfloat color[] = { (**float**)sin(currentTime) \* 0.5f + 0.5f,

(**float**)cos(currentTime) \* 0.5f + 0.5f,

0.0f, 1.0f };

glClearBufferfv(GL\_COLOR, 0, color);

// Use the program object we created earlier for rendering

glUseProgram(rendering\_program);

// Draw one point

glDrawArrays(GL\_POINTS, 0, 1);

}

Listing 2.7: Rendering a single point

#### First Triangle

## **CAPITOLO 3 : PIPELINE**

### Passare dati al Vertex Shader

Il vertex shader è la prima fase della pipeline, ed’è anche l’unico shader programmabile obbligatorio , eg non posso non scrivere il vertex shader. Prima del Vertex Shader c’è comunque una fase gestita automaticamente dal sistema che manda gli input al vertex shader (Vertex Pulling).

Vertex Attributes

In GLSL gli input/output vengono gestiti attraverso variabili che si dichiarano attraverso i qualificatori in/out.

Questi qualificatori possono essere in tutti gli shader, e possono formare una sorta di condotto attraverso il quale l’input di uno shader viene propagato agli altri.

Una variabile dichiarata come in/out è un **vertex attribute**.

Esempio:

Listing 3.1: Declaration of a vertex attribute

**#version** 450 core

// 'offset' is an input vertex attribute

**layout** (location = 0) **in vec4 offset**;

**void** main(**void**)

{

**const vec4** vertices[3] = **vec4**[3](**vec4**(0.25, -0.25, 0.5, 1.0),

**vec4**(-0.25, -0.25, 0.5, 1.0),

**vec4**(0.25, 0.25, 0.5, 1.0));

// Add 'offset' to our hard-coded vertex position

gl\_Position = vertices[gl\_VertexID] + **offset**;

}

La variabile offset sarà riempita dalla fase di vertex pulling, i dati con cui tale fase riempie il vertex attribute è specificato dalla funzione

**void glVertexAttrib4fv**(GLuint index, **const** GLfloat \* v);

**index** è la location del vertex attribute, e **v** è il puntatore ai dati che alimentano il vertex attribute.

Altre funzioni del genere :

void glVertexAttrib1f( GLuint index,

GLfloat v0);

void glVertexAttrib1s( GLuint index,

GLshort v0);

void glVertexAttrib1d( GLuint index,

GLdouble v0);

void glVertexAttribI1i( GLuint index,

GLint v0);

…..

### Vertex Attribute

Vertex attributes are used to communicate from "outside" ***to the vertex shader***. Unlike uniform variables, values are provided per vertex (and not globally for all vertices).

In GLSL, the mechanism for getting data in and out of shaders is to declare global

variables with the **in** and **out** storage qualifiers

**in** storage qualifier: This marks the variable as an input to the vertex shader, which means that it is essentially an input to the OpenGL graphics pipeline. It is automatically filled in by the fixed-function vertex fetch stage. The variable becomes known as a ***vertex attribute***.

Esempio :

**#version** 450 core

// 'offset' is an input vertex attribute

**layout** (location = 0) **in vec4 offset**;

**void** main(**void**)

{

**const vec4** vertices[3] = **vec4**[3](**vec4**(0.25, -0.25, 0.5, 1.0),

**vec4**(-0.25, -0.25, 0.5, 1.0),

**vec4**(0.25, 0.25, 0.5, 1.0));

// Add 'offset' to our hard-coded vertex position

gl\_Position = vertices[gl\_VertexID] + **offset**;

}

Listing 3.1: Declaration of a vertex attribute

**offset :** As it is an input to the first shader in the pipeline, it will be filled automatically by the vertex

fetch stage. We can tell this stage what to fill the variable with by using one of the many

variants of the vertex attribute functions, **glVertexAttrib\*()**

index is used to reference the attribute and v is a pointer to the

new data to put into the attribute. You may have noticed the **layout** (location =

0) code in the declaration of the **offset** attribute. This is a *layout qualifier*, which we

have used to set the *location* of the vertex attribute to zero. This location is the value

we’ll pass in index to refer to the attribute

(example **void** glVertexAttrib4fv(GLuint index, **const** GLfloat \* v).

### Fragment Shaders

This stage is responsible for determining the colour of each fragment.

Just as with other shader stages we can define our won inputs to the fragment shader , which will be filled based on the outputs of whichever stage is before the fragment shader .

The inputs to the fragment shader are somewhat unlike inputs to other shader stages, in that openGL interpolates their values across the primitive that’s being rendered.

#version 450 core

// 'vs\_color' is an output that will be sent to the next shader stage

out vec4 vs\_color;

void main(void)

{

const vec4 vertices[3] = vec4[3](vec4(0.25, -0.25, 0.5, 1.0),

vec4(-0.25, -0.25, 0.5, 1.0),

vec4(0.25, 0.25, 0.5, 1.0));

const vec4 colors[] = vec4[3](vec4(1.0, 0.0, 0.0, 1.0),

vec4(0.0, 1.0, 0.0, 1.0),

vec4(0.0, 0.0, 1.0, 1.0));

// Add 'offset' to our hard-coded vertex position

gl\_Position = vertices[gl\_VertexID] + offset;

// Output a fixed value for vs\_color

vs\_color = color[gl\_VertexID];

}

Listing 3.11: Vertex shader with an output

As you can see, in Listing 3.11 we added a second constant array that contains colors

and index into it using gl\_VertexID, writing its content to the vs\_color output. In

Listing 3.12 we modify our simple fragment shader to include the corresponding input

and write its value to the output.

Click here to view code image

#version 450 core

// 'vs\_color' is the color produced by the vertex shader

in vec4 vs\_color;

out vec4 color;

void main(void)

{

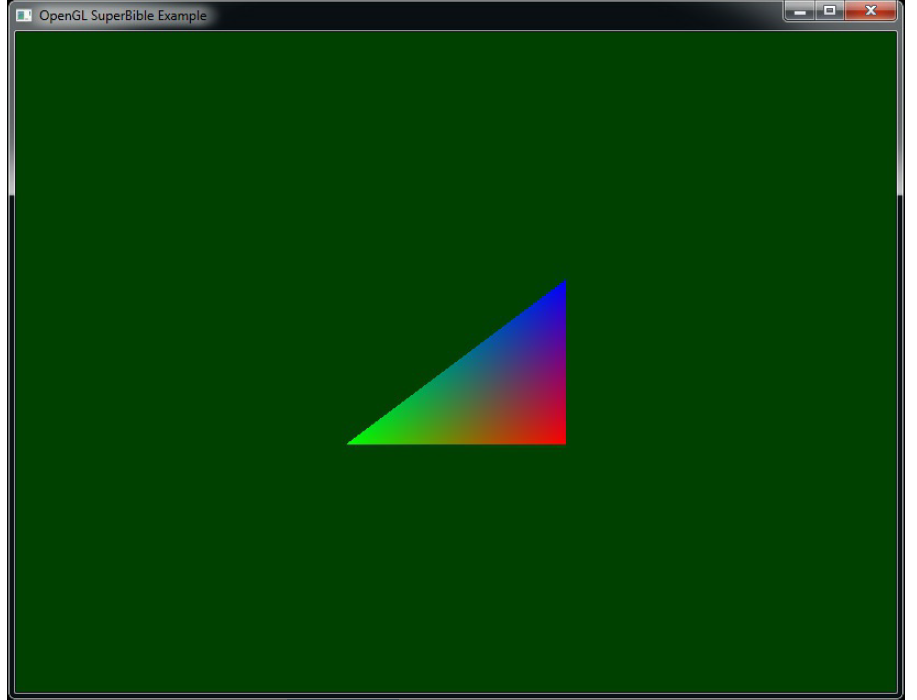
color = vs\_color;

}

Listing 3.12: Deriving a fragment’s color from its position

The result of using this new pair of shaders is shown in Figure 3.5. As you can see, the

color changes smoothly across the triangle.



## **CAPITOLO 4 : MATH FOR 3D GRAPHICS**

### Model-View Transformation

There are usually three elementary transformation steps.

We want to take a model from

**MODEL SPACE -> WORLD SPACE -> VIEW SPACE**

The **model transformation (**also called **model-world transformation)** essentially places objects into world space. Each object is likely to have its own model transform which will generally consist of a sequence of scale, rotation and translation operations.

The **view transformation** allows you to place the point of observation anywhere you want and look in any direction. Determining the viewing transformation is like placing and pointing a camera at the scene.

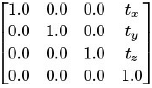
You must apply the viewing transformation before any other modeling transformation.

The reason is that it appears to move the current working coordinate system with respect to the eye coordinate system. All subsequent transformation then accur based on the newly modified coordinate system.

The transform that moves coordinates from world space to view space is sometimes called **world-view transformation.**

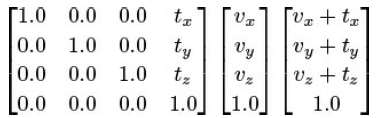
Concatenating the model-world and world-view transform matrices by multiplying them together yields the **model-view transformation** (i.e. the matrix that takes coordinates from model to view space).

### Translation Matrix



Examining the structure of the translation matrix reveals one of the reasons why we need

to use four-dimensional homogeneous coordinates to represent positions in 3D graphics.



Had the *w* component of *v* not been 1.0, then using this matrix for translation

would have resulted in *tx*, *ty*, and *tz* being scaled by that value, affecting the output of the

transformation. In practice, position vectors are almost always encoded using four

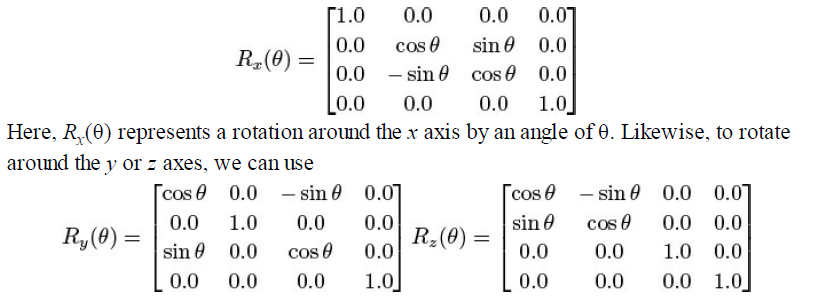
components with *w* (the last) being 1.0, whereas direction vectors are encoded either

simply using three components, or as four components with *w* being zero. Thus,

multiplying a four-component direction vector by a translation matrix doesn’t change it

at all.

### Rotation Matrix



### Quaternion

A quaternion is basically a 4 dimensions vector which base is {1,i,j,k} , I,j,k being the complex numbers.

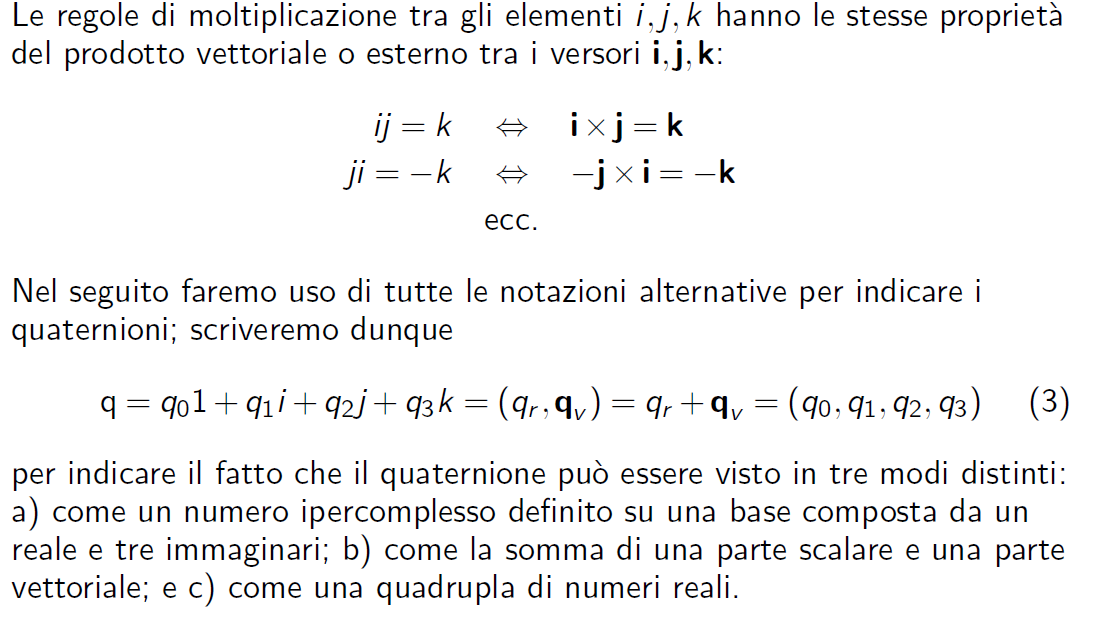
Hence we can always write a quaternion as follow :

Q = 1q0+iq1+jq2+kq3 being q0,q1,q2,q3 Real coefficients.

The imaginary part of a quaternion can be identified with a vetor, hence we can also write a quaternion as follow:

Q = (q0, **q**); being **q = I**q1 **+j**q2**+k**q3 ; being **I,j,k**  unit vector of Cartesian frame of reference

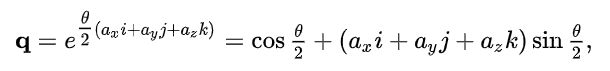
It can be shown that same rules apply for product between complex numbers and I,j,k versor .



#### Quaternion and Rotation Matrix

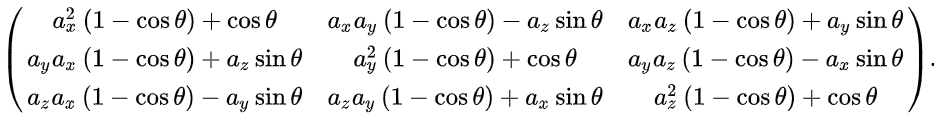
It can be shown that there is a 1:1 map between the space of quaternions and that of the rotation matrix.

Specifically any quaternion q can be written in eulerian form as follow :



Where theta is the rotation angle and (ax,ay,az) is the rotation axis veror.

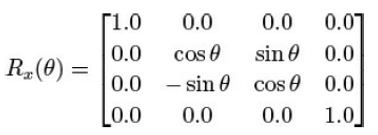
From here it can be show that such quaternion maps the following rotation matrix :



You will find that calculating the rotation matrix mapped to the following quaternion :

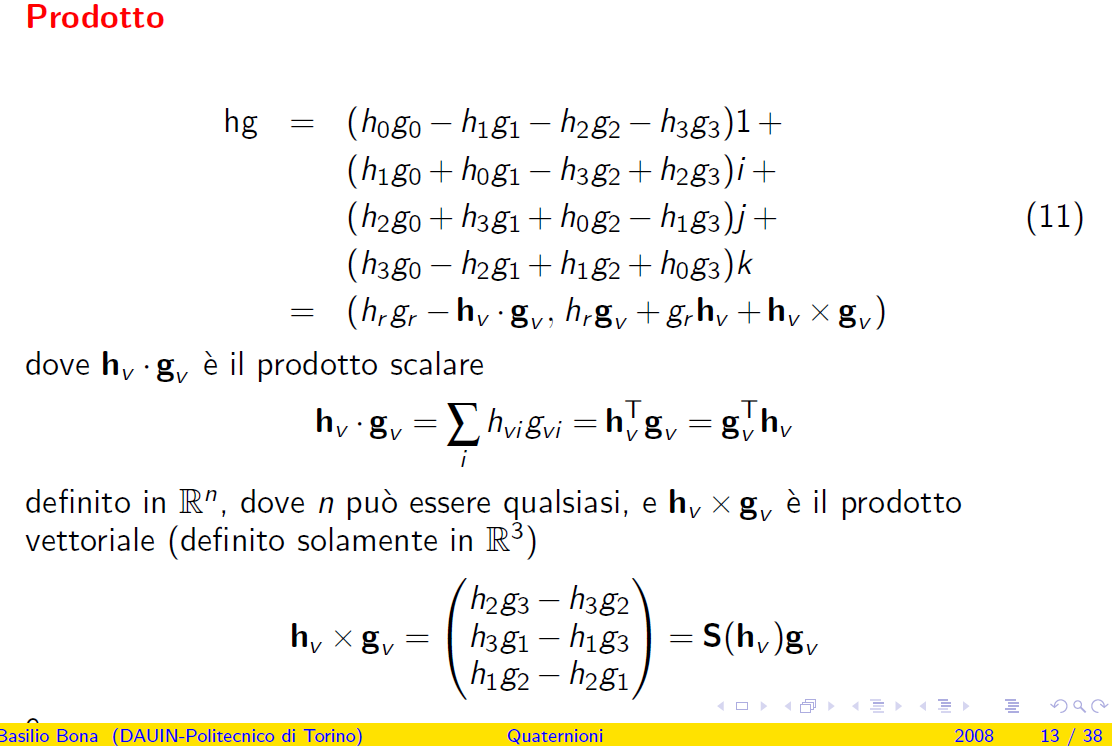
Qx = cos(theta/2) + isin(theta/2) (quaternion where ax = 1, ay = az = 0)

Is exactly the rotation matrix of an angle theta, about the x axis :



#### Product of Quaternions

So if you have 2 quaternions, say h and g, in product (that would rapresent concatenation of two rotation), here is shown the product, both in complex and vector form :



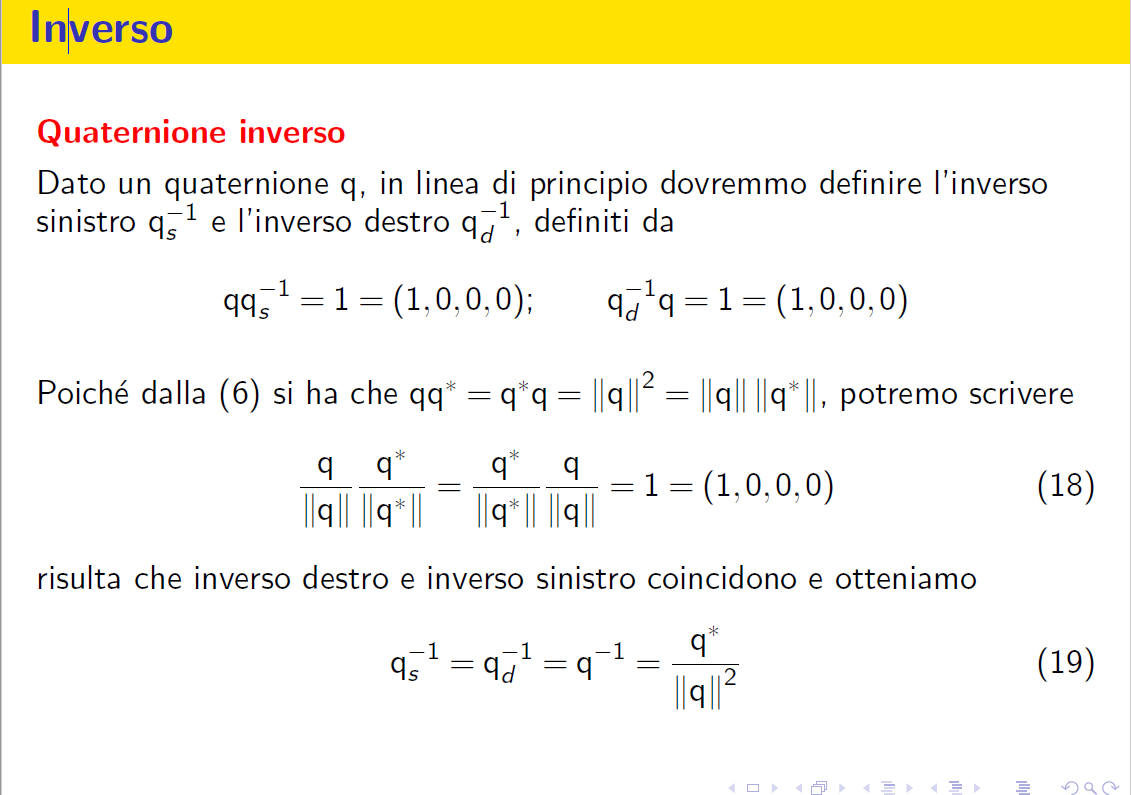
#### Unitary quaternion

From the above definition of product between quaternion, it s easy to see the following product:

(1,0,0,0)\*(w,x,y,z) = (w,x,y,z).

Hence (1,0,0,0) is the unitary quaternion.

#### Inverse Quaternion



### Concatenazione di Trasformazioni

Per concatenare tra loro più trasformazioni basta moltiplicare le rispettive matrici di trasformazione.

Ricorda , mai sommare le matrici di trasformazione : si potrebbe pensare che ad esempio nel caso delle traslazioni sia corretto sommare le matrici, in quanto questo porterebbe alla somma delle traslazioni.

(Matrice di traslazione).

Tvec4<T>(1.0f, 0.0f, 0.0f, 0.0f),

Tvec4<T>(0.0f, 1.0f, 0.0f, 0.0f),

Tvec4<T>(0.0f, 0.0f, 1.0f, 0.0f),

Tvec4<T>(x, y, z, 1.0f));

Come si vede però se sommassimo tali matrici, otterremmo si la somma dei termini traslatori (x1+x2,y1+y2,z1+z2), ma otterremmo anche la somma degli elementi in diagonale il che è sbagliato!

Quando si concatenano più trasformazioni bisogna ricordare che l’ordine con cui si effettuano le trasformazioni non è indifferente : rotazione + traslazione porta ad una posizione finale dell’oggetto trasformato che è diversa rispetto a quella a cui porta la sequenza traslazione + rotazione.

Bisogna inoltre ricordare che l’ordine di lettura di un prodotto di matrici è opposto all’ordine con cui vengono eseguite le trasformazioni.

Pertanto se voglio trasformare un oggetto con una serie di trasformazioni T1,T2,T3… dovrò applicare ai suoi vertici la matrice T = …\*T3\*T2\*T1.

Allo stesso modo se volessi invertire tale trasformazione dovrei applicare la seguente matrice T^-1 = (…\*T3\*T2\*T1)^-1 =(T1^-1)\*(T2^-2)\*(T3^-3)\*…. Ovvero nell’inversione l’ordine si inverte nuovamente.

### Superbible VMATH library : how to use

#### Matrices

##### Identy Matrix

Definizione della matrice identità :

vmath::mat4 m2{ vmath::vec4(1.0f, 0.0f, 0.0f, 0.0f), // X Column

vmath::vec4(0.0f, 1.0f, 0.0f, 0.0f), // Y Column

vmath::vec4(0.0f, 0.0f, 1.0f, 0.0f), // Z Column

vmath::vec4(0.0f, 0.0f, 0.0f, 1.0f) }; // W Column

oppure

vmath::mat2 m2 = vmath::mat2::identity();

vmath::mat3 m3 = vmath::mat3::identity();

vmath::mat4 m4 = vmath::mat4::identity();

##### Translation Matrix

The vmath library contains two functions that will construct a 4 × 4 translation

matrix for you using either three separate components or a 3D vector:

**template** <**typename** T>

**static inline** Tmat4<T> translate(T x, T y, T z) { ... }

**template** <**typename** T>

**static inline** Tmat4<T> translate(**const** vecN<T,3>& v) { ... }

##### Rotation Matrix

Rotation around x, y, z axis :

**template** <**typename** T>

**static inline** Tmat4<T> rotate(T angle\_x, T angle\_y, T\_angle\_z);

Rotation around arbitrary axis

**template** <**typename** T>

**static inline** Tmat4<T> rotate(T angle, T x, T y, T z); //for example : rotate(45, 1.0,.0.45,0.0);

**template** <**typename** T>

**static inline** Tmat4<T> rotate(T angle, **const** vecN<T,3>& axis);

## **CAPITOLO 5 : DATA**

\*\*Attenzione : le funzioni presentate in questo capitolo potrebbero non essere compatibili con versioni di OGL inferiori alla 4.5\*\*

Fino ad ora abbiamo passato I dati agli shaders in due modi :

1) Inserendoli direttamente negli shaders come parametri.

2) Passandoli agli shader uno alla volta (con funzioni come glVertexAttrib\*)

Questo approccio è inefficiente.

Per permettere ai dati di essere acceduti e scritti da OGL esistono due forme di data storage: buffers e textures.

-buffers : blocchi lineari untyped, per generica allocazione di memoria.

-textures : blocchi multidimensionali di dati.

### **BUFFERS**

Sono allocazioni lineari di memoria.

##### **Step necessari per l’uso dei buffers**

1. riservare un nome, che è essenzialmente una "maniglia" per poter usare il buffer.

**Void glCreateBuffers(GLsizei n, GLuint\* buffers);**

**Void glGenBuffers(GLsizei n, GLuint \*buffers);**

(es : glGenBuffers(1,&buffer); // dove buffer : GLuint buffer;)

1. attaccare il buffer al contesto OpenGL**,** legandolo ad un “binding point” o “target”.

**Void glBindBuffer(GLenum target, Gluint buffer);**

Funzione che lega il buffer al contest OpenGL desiderato (ad es GL\_ARRAY\_BUFFER), (buffer è lo stesso argomento della funzione glCreateBuffers).

1. Creazione e Inizializzazione de Buffer

**void glBufferStorage(GLenum target, GLsizeiptr size, const void\* data, GLbitfield flags);**

**void glNamedBufferStorage(GLuint buffer, GLsizeiptr size , const void\* data , GLbitfield flags);**

glBufferStorage and glNamedBufferStorage create a new immutable data store. For glBufferStorage, the buffer object currently bound to *target* will be initialized:

come si vede queste due differiscono solo per il primo parametro :

La prima alloca la memoria sul **target** (eg sul contesto OpenGL al punto 2).

La seconda alloca la memoria direttamente sul **buffer**.

-**Size** specifica la grandezza della regione di memoria.

-**data** permette – passando un puntatore – di inizializzare il buffer con dei dati.

- **flags** dice ad OpenGL come useremo il buffer : in base a questo OpenGL crea una opportuna allocazione di memoria (vedi paragrafo successivo).

Lo spazio riservato da queste funzioni è considerato **immutable** : il contenuto può cambiare, ma non il size o il flag.

**Nota che l’ordine di 2 e 3 è interscambiabile.**

**Flags** :

**GL\_DYNAMIC\_STORAGE\_BIT**

Informa OGL che aggiorneremo il contenuto del buffer direttamente, così che venga riservata una memoria ad alta accessibilità.

Se non acceso la funzione **glBufferSubData()** non può essere usata.

**GL\_MAP\_READ\_BIT, GL\_MAP\_WRITE\_BIT, GL\_MAP\_PERSISTENT\_BIT, GL\_MAP\_COHERENT\_BIT**

Tutti questi metodi prevedono il mapping del buffer : la mappatura prevede l’uso di un puntatore per lavorare sul buffer…

##### **Delete del Buffer**

Per eliminare e liberare la memoria utilizzata dal buffer bisogna utilizzare la funzione : glDeleteBuffers(n, VBO); (GLuint \* VBO; VBO = new GLuint[n];

NB: eliminare il buffer utilizzando la funzione delete[] VBO; non funziona !

Così facendo infatti perderemo la maniglia sul VBO ma la sua memoria rimarrà occupata e non riutilizzabile! (memory leakage)

##### **Update del Buffer**

Se nella fase di inizializzazione si è usato il flag The contents of the data store may be updated after creation through calls to [glBufferSubData](https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glBufferSubData.xhtml).

#### DUE MODALITÀ DI INSERIMENTO DEI DATI NEL BUFFER

###### Inserimento diretto attraverso funzioni OGL apposite.

1. Direttamente al momento dell’inizializzazione, fornendo alle funzioni del punto 3 un puntatore ai dati da inserire.
2. Utilizzando le funzioni di scrittura diretta sul buffer :

**glBufferSubData(Glenum target, GLintptr offset, GLsizeiptr size, const GLvoid \* data)**

**glNamedBufferSubData(GLuint buffer, GLintptr offset, GLsizeiptr size, const GLvoid \* data)**

Utilizzabili avendo posto il flag GL\_DYNAMIC\_STORAGE\_BIT al momento dell’allocazione di memoria!

Le due funzioni differiscono solo per il primo parametro : la prima scrive sul buffer relativo al contesto OGL definito dal “target”, mentre alla seconda forniamo direttamente il buffer.

**Offset :** definisce l’offset che vogliamo dare ai dati rispetto all’inizio del buffer.

**Size :** la dimensione del blocco di dati che vogliamo inserire

**\*data :** Il puntatore ai dati da inserire.

###### Inserimento attraverso una mappatura dell'oggetto buffer

La mappatura consiste nello scrivere o leggere il buffer attraverso un puntatore : tale puntatore viene utilizzato poi in funzioni standard della libreria C++ come ad esempio memcpy(puntatoreBuffer, dati).

Funzioni di mappatura del buffer (funzioni che rilasciano un puntatore al buffer):

**-glMapNamedBuffer(GLuint buffer, GLenum usage);**

**-glMapBuffer(GLenum target, GLenum usage);**

**-glMapBufferRange(GLenum target, GLintptr offset, GLsizeiptr length, GLbitfield access)**

**-glMapNamedBufferRange(GLuint buffer,”,”,”);**

Mentre le prime due mappano l’intero buffer e non danno alcuna informazione circa il tipo di mapping da effettuare, le \*Range mappano solo il range tra **offset** e **offset +** **length,** e attraverso il flag **GLbitfield** dicono ad OGL come il mapping dovrebbe essere effettuato.

Questi GLbitfield non sono suggeritivi ma costrittivi: sbagliare bit significa commettere un errore.

Inoltre questi Bit devono andare d’accordo con quello delle \*BufferStorage: se voglio mappare con bitfield GL\_MAP\_READ\_BIT, devo avere lo stesso bit anche sulla \*BufferStorage.

esempio di utilizzo :

void \* ptr = **glMapNamedBuffer(NomeBuffer,GL\_WRITE\_ONLY);**

memcpy(ptr, data, sizeof(data));

glUnmapNamedBuffer(GL\_ARRAY\_BUFFER);

Funzioni per “smappare” il buffer:

**glUnmapBuffer()**

**glUnmapNamedBuffer();**

### Quando bindare VAO e VBO

 If you are using a VAO, then you should only bind the VAO when you set it up and when drawing the geometry. You only bind the VBO/IBO again when you need to update them.

After drawing or updating a buffer, you don't necessarily have to unbind it, though it might be a good idea to do so to avoid accidental writes to buffers that were left bound.

Now taking your fist sequence of operations, this is the overall order I would expect to see:

#### **On init:**

1. Create & bind a VAO. Any VBO and IBO that you bind in the sequence will be associated with the current VAO (this one).
2. Create & bind index buffer.
   * Fill index buffer with glBufferData/glMapBuffer.
3. Create & bind vertex buffer.
   * Fill vertex buffer glBufferData/glMapBuffer.
4. Set up vertex attributes with glEnableVertexAttribArray/glVertexAttribPointer, etc.
5. Optionally unbind everything to avoid accidental modification of the buffers and VAO. Remember to unbind the VAO first. E.g.: glBindVertexArray(0);

#### **On draw:**

1. **If only drawing the buffers**:
   * Bind the VAO;
   * Perform the draw call(s).
2. **If updating and drawing**:
   * Bind the VAO, VBO, IBO;
   * Update the buffers (updating vertex atributes is only necessary if the vertex format has changed);
   * Perform the draw call(s).
3. Optionally unbind to avoid accidental modification of the buffers and VAO.

That's as simple as that. This order of operations should function without problems.

#### **FUNZIONI DI RIEMPIMENTO E COPIA DEI BUFFER**

##### Riempimento

Prima di utilizzare il buffer bisogna inizializzarlo.

**-Void glClearBufferSubData(GLenum target, GLenum internalformat,GLintptr offset, GLsizeiptr size ,GLenum format, GLenum Type, const void\* data);**

**-Void glClearNamedBufferSubData(GLuint buffer, “, “ ,”,”,”,”)**

Questa è la funzione da utilizzare se si vogliono inserire valori costanti:

**data :** puntatore al dato da inserire

**internalformat** : Il formato a cui il dato da inserire viene convertito, prima di essere inserito

**offset, size** : Definiscono il range nel quale andrà il dato

**format, type** : Comunicano ad OGL il formato del dato a cui punta “data” : **format** specifica quante componenti ha il dato GL\_RED -> 1 componente , GL\_RG -> due componente … … GL\_RGBA -> 4 componenti. **Type** specifica il tipo delle componenti.

##### Copia

Se vogliamo copiare da Buffer1 a Buffer2 usiamo la funzione:

**-glCopyBufferSubData(GLenum readtarget,GLeum writetarget, GLintptr readoffset**

**, GLintptr writeoffset, GLsizeiptr size);**

**-glCopyNamedBufferSubData(GLuint readtarget,GLeum writetarget, GLintptr readoffset**

**, GLintptr writeoffset, GLsizeiptr size);**

con la solita distinzione tra la versione Named e la versione non-Named.

In particolare si noti che nel caso della non-Named, poiché un **binding point** (es GL\_ARRAY\_BUFFER) può avere solo un legame alla volta, non è possibile copiare tra buffer legati allo stesso binding point! Per questo motivo esistono i due buffer **GL\_COPY\_READ\_BUFFER** e **GL\_COPY\_WRITE\_BUFFER** : due binding points pensati proprio per risolvere questo genere di problema.

### **MODI DI ALIMENTARE I VERTEX SHADER**

1. Passare Vertex attribute attraverso funzioni tipo **glVertexAttrib\*() (vedi** [**Passare dati al Vertex Shader**](#_Passare_dati_al)**)**
2. Passare uniform vertex attribute attraverso funzioni tipo **glUniform\*() (vedi** [**Default Block Uniforms**](#_Default_Block_Uniforms)**)**
3. Attraverso i buffer (vedi sotto)

##### **1 - ALIMENTARE I VERTEX SHADER DAI BUFFERS**

Nel capitolo 2 abbiamo introdotto il **VAO (vertex array object)**, e spiegato come esso sia l’input al vertex shader. In ogni caso nel capitolo 2 il vertex shader non è stato alimentato da nessun input, e i dati desiderati sono stati scritti direttamente nello shader.

Nel capitolo 3 invece abbiamo spiegato il concetto di **VA (vertex attributes)**, ed utilizzato la funzione glVertexAttribute\* per modificare il valore **statico** dei VA.

Vediamo adesso come alimentare un VA attraverso un BO (buffer object) :

Ogni VA deve pescare i dati da un BO legato ad uno dei parecchi **vertex buffer bindings (VBB)** :

GLuint vao;

glCreateVertexArrays(1,&vao);

//do not call Create at each render iteration : calling at each iteration may result in a correct visualization but openGL is actually creating a new vertex array object per Iteration (in debug you would see the VAO’s value incrementing +1 at each iteration)

glBindVertexArray(vao);

**-void glVertexArrayAttribBinding(GLuint vao, GLuint attribIndex, GLuint bindingIndex);**

quando il **VAO** è legato ad un bindingPoint(vedi sopra), il Vertex Attribute all’indice **attribIndex**(vertex attribute index ( vai ) : l’attribute index viene specificato nello shader attraverso lo statement : *layout (location = indice)* ) deve prendere i dati dal BO legato all’indice **bindingIndex** ( vertex buffer binding index ( vbbo ), il quale a sua volta viene specificato dalla funzione :

**-void glVertexArrayVertexBuffer(GLuint vao, GLuint bindingIndex, GLuint buffer, GLintptr offset, GLsizei stride);**

La funzione specifica il buffer sorgente “buffer” e il range all’interno del quale si trovano i dati **(GLintptr offset** (The offset of the first element of the buffer. es: 0, è un numero) , **GLsizei stride** : “The distance between elements within the buffer (es: sizeof(double))”, inoltre associa al suddetto buffer uno specifico bindingIndex.

Riassumendo il vertex attribute index ( vai ) dipende dalla location che viene data al vertex attribute nello shader, mentre il vertex buffer binding index ( vbbi ) è definite in modo arbitrario nella funzione sopra.

Approfondimento su **stride**

Attenzione : Il passo (stride) si misura in base a cosa viene passato per disegnare il singolo vertice, ad esempio :

GLfloat x = 3.0f, y = 3.0f, z = 0, scale = 1.0;

static GLfloat vertices[] =

{

x,0.0,z,scale,

-x,0.0,z,scale,

x,y,z,scale

};

static GLuint attribIndex = glGetAttribLocation(shader\_prog, "vertices"),

bindingIndex = 0, offset = 0, stride = sizeof(vertices), num\_coord\_per\_vert = 4;

glNamedBufferSubData(buffer, 0, sizeof(vertices), vertices);

glCreateVertexArrays(1, &VAO);

glBindVertexArray(VAO);

glVertexArrayAttribBinding(VAO, attribIndex, bindingIndex);

glVertexArrayVertexBuffer(VAO, bindingIndex, buffer, offset, 4\*sizeof(GLfloat));

glVertexArrayAttribFormat(VAO, attribIndex, num\_coord\_per\_vert, GL\_FLOAT, GL\_FALSE, 0);

glEnableVertexArrayAttrib(VAO, attribIndex);

SHADERS :

#version 450 core

layout (location = 0 ) in vec4 vertices;

layout (location = 6) uniform mat4 transformMatrix;

void main(void){

gl\_Position = transformMatrix\*vertices;

}

**-void glVertexArrayAttribFormat(GLuint vao, GLuint attribIndex, Glint size, GLenum type, GLboolean normalized, GLuint relativeOffset);**

Funzione che descrive il layout e il formato dei dati.

Size è il numero di componenti che sono memorizzate nel buffer per ogni vertice

**-void glEnableVertexAttribArray(GLuint index);**

**-void glDisableVertexAttribArray(GLuint index);**

**Index** : Specifies the index of the generic vertex attribute to be enabled or disabled

Funzione che abilita/disabilita il riempimento automatico del vertex attribute, ovvero OGL viene informato di riempire il VA con i dati nel buffer e non con quelli passati dalla funzione **glVertexAttrib\*()**. Quando utilizzata OGL invia i dati al vertex Shader basandosi sul formato e sulla location specificata da **glVertexArrayVertexBuffer()**  e da **glVertexArrayAttribFormat().**

Queste funzioni vengono chiamate dopo tutte le altre !

###### Warning on VAO binding: The purpose of a vertex array object is to define the association between storage (stored in buffers) and vertex arrays that feed rendering operations.

Keep in mind that all draws of your application could be bound to the same vao (successively) and it would be a correct approach.

The thing you must be aware of, is that a binding error of the VAO may have unattended behavior not only in the specific drawing which the VAO is related, but also in other drawing that are apparently unrelated.

* It ‘s crucial to make sure that each draws is perfectly binded with the proper VAO at the moment of the call at glDrawArrays(…).

For this reason a good approach would be to formulate the render function for each object as follow:

Object::render()

{

glEnableVertexArrayAttrib(VAO, attribIndex);

// rendering code block

glDisableVertexArrayAttrib(VAO, attribIndex);

}

**OGL SuperBible : riprendi da pag 155**

**.**

**.**

**..**

**.**

**.**

**.**

**..**

**.**

**.**

**..**

**.**

**.**

**..**

**.**

**.**

**.**

**.**

**.**

###### **A “good” pattern when using Buffer**

An example of pattern when using buffers is [Pattern for Buffer Usage](#_Pattern_for_Buffer).

##### **2 - UNIFORMS STORAGE BLOCKS**

Abbiamo già visto come passare dati agli shader tramite Interface Blocks (capitolo 3) e Vertex Attributes.

Mentre abbiamo un Vertex Attribute per vertice, uno uniform è un’informazione “uniforme” per l’intera primitiva.

Ci sono due tipi di uniform : **default block uniforms** e **Named uniform blocks**.

###### Default Block Uniforms

Qualsiasi uniform dichiarato nel global scope di uno shader vive nel “Default Uniform Block”.

Esempio di dichiarazione di uniform in shader:

**uniform int uniName;**

Il valore di uno uniform può essere assegnato all’interno dello shader, solo al momento della dichiarazione (es : uniform int uniName = 13;).

In generale useremo funzioni OGL per assegnare il valore di uno uniform.

Per usare tale funzioni è conveniente specificare la posizione dello uniform nello shader come

segue:

es:

**layout (location = 17) uniform vec4 myUniform;**

Se non assegni una location, OGL ne assegnerà una che è possibile conoscere attraverso la funzione :

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

Per settare gli uniform si usano le funzioni **glUniform\*(uniformLocation,uniformValue).**

Possiamo anche settare Matrici uniform usando glUniformMatrix\*().

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

(è possibile usare assegnare stessa location a uniform di shaders differenti)

\*\*\*\*Attenzione : le funzioni glUniform\*\*\*() necessitano che sia stato già dichiarato l’uso dello shader : glUseProgram(“nome dello shader”);

Altrimenti non possono trovare l’indice dello uniform a cui legarsi.

###### Named Uniform Blocks

Quando si hanno molte variabili uniform settarle attraverso singole chiamate glUniform\*() diventa inefficiente.

OGL permette di raggruppare uniforms in blocchi, e immagazzinare interamente i blocchi in buffer (questa funzione è disponibile solo per i Named Uniform Blocks e non per i Default Uniform Blocks)

Uno Named Uniform Blocks si dichiara all’interno di uno shader come segue :

uniform TransformBlock

{

Float scale;

vec3 translation;

float rotation[3]

mat4 projectionMatrix

}transform;

TransformBlock è il nome della struttura (come se fosse il nome della classe)

Transform è il nome dell’istanza.

Dentro allo shader ci si può riferire ai membri del blocco come segue : transorm.translation, etc…

###### **Come si costruiscono Uniform Blocks nei Buffer ?**

Ci sono due metodi :

1)Si usa una disposizione standard per i dati (**standard layout**)

2) Si lascia OGL decidere la disposizione (**shared layout**) : questa è solitamente la soluzione più performante, ed è anche quella di default.

**Standard Layout:**

Il layout standard può essere usato in 3 modi, in ogni caso comunque è necessario specificare si tratta del layout standard:

**Modalità 1**: (non specifico gli offset)

layout(std140) uniform TransformBlock

{

Float scale;

vec3 translation;

float rotation[3];

mat4 projectionMatrix;

}transform;

Una volta dichiarato che il layout è standard i membri dello Uniform Block verranno disposti secondo delle regole per i cui dettagli rimandiamo al paragrafo “Regole di allineamento per Layout std140”

**Modalità 2**: (specifico gli **offset**, non è necessario dichiarare i membri nell’ordine)

layout(std140) uniform myBlock

{

Layout (offset = 32) vec4 foo; //viene disposto foo con un offset di 32 bytes

Layout (offset = 8) vec2 bar; // viene disposto bar con offset di 8 bytes

Layout (offset = 48) vec3 baz; // viene disposto baz con offset di 48 bytes

}blockInstance;

E’ possibile specificare direttamente l’offset dei membri del blocco Uniform, a patto di seguire le regole di allineamento richieste dal std140 : la differenza è che così facendo è possibile lasciare spazi tra i membri e dichiarare membri in modo non ordinato :

il primo membro nel blocco è bar, con un offset di 8 byte, questo offset soddisfa i requisiti di allineamento per una variabile di tipo vec2 (vedi spiegazione pag 161).

Anche il secondo e il terzo membro, (foo e baz rispettivamente) soddisfano i requisiti di allineamento.

**Modalità 3: (specifico layout ed offset)**

E’ anche possibile specificare il tipo di allineamento attraverso l’ **align** qualifier : funziona come offset ma spinge il membro al successivo multiplo (in memoria) dell’allineamento specificato.

**Align** può essere usato anche sull’intero blocco per forzare tutti i suoi membri ad essere allineati ad una certa posizione:

layout (std140, align = 16) uniform MyUniform

{

Layout (offset = 32) vec4 foo; //viene disposto foo con un offset di 32 bytes

Layout (offset = 8) vec2 bar; // viene disposto bar con offset di 8 bytes

Layout (offset = 48) vec3 baz; // viene disposto baz con offset di 48 bytes

}blockInstance;

Regole di allineamento per Layout std140

Quando un blocco Uniform è stato dichiarato come std140, ogni membro del blocco consuma una quantità di spazio predefinita da una serie di regole :

Ogni tipo occupante N bytes nel buffer inizia alla soglia di N byte all’interno di quel buffer : per es. i tipi GLSL int float e bool che sono tutti 4 byte iniziano su multipli di 4 bytes.

Un vettore di uno di questi tipi, di lunghezza 2 inizia sempre sulla soglia di 2N bytes, ad esempio vec2 che occupa 8 byte inizia alla soglia di 8 byte.

Vettori di 3 e 4 elementi iniziano sempre alla soglia di 4N byte, ad esempio vec3 e vec4 iniziano alla soglia di 16 byte (ricorda che N è il numero di byte del singolo elemento del vettore, quindi se vec3 è un vettore di int allora inizierà alla soglia di 4\*4, idem per vec4).

Ogni membro di un array di scalari o vettori (es array di int o di vec3) inizia sempre ad una soglia definita dalle stesse regole ma arrotondata all’allineamento di un vec4, perciò array di qualsiasi cosa tranne che di vec4 non sono impacchettati ma avranno gap tra ciascun elemento.

Le matrici sono trattate come array di vettori, e array di matrici sono trattate come array di vettori molto lunghi.

Infine strutture e array di strutture iniziano su soglie definite dai loro membri più grandi, arrotondati alla grandezza di un vec4(?).

Particolare attenzione bisogna porre al fatto che la disposizione std140 e le regole di disposizione seguite dal compilatore C++ sono diverse. Per questo non è possibile copiare dati da un array C ad uno Uniform Block a meno che i dati nell’array C non siano stati opportunamente arrangiati.

**Shared Layout**

Se si usa la disposizione automatica di OGL è necessario conoscere la disposizione dei membri.

Ogni membro nello Uniform Block ha un suo indice che può essere recuperato con la seguente chiamata a funzione:

**void glGetUniformIndices(GLuint program, GLsizei uniformCount, const GLchar \*\* uniformNames, GLuint \* uniformIndices);**

Questa funzione permette di conoscere, con un’unica invocazione, gli indici degli uniform member specificati negli uniformNames anche se appartengono a differenti uniform Block: **uniformCount** è il numero di uniform member di cui vogliamo gli indici, **uniformNames** è un array di nomi di Uniform specificati come “BlockName.MemberName”, **uniformIndices** è l’array di interi dove vengono riposti gli indici degli uniform.

Una volta noti gli indici è possibile ricavare tutte le info sugli uniform block member con la seguente funzione:

**void glGetActiveUniformsiv(GLuint program, GLsizei uniformCount, const GLuint \* uniformIndices,**

**GLenum pname, GLint \* params);**

**UniformCount**  è il numero di uniform di quale si vuole conoscere il parametro **pname.**

La lista dei diversi parametric **pname** che possono essere conosciuti invocando la funzione suddetta, si trova a pagina 165.

**Params** è il vettore di uniformCount interi dove verrà memorizzata l’informazione richiesta (si, l’informazione restituita è sempre un intero, qualsiasi sia **pname**).

Esempio di utilizzo :

static const GLchar\* uniformNames[4] =

{ “BlockName1.UniformName2”, “BlockName2.UniformName2”, ….};

GLuint UniformIndices[4];

glGetUniformIndices(program,4, uniformNames, UniformIndices);

GLint uniformOffsets[4];

GLint arrayStrides[4];

GLint matrixStrides[4];

glGetActiveUniformsiv(program, 4 , uniformIndices, GL\_UNIFORM\_OFFSET, uniformOffset)

glGetActiveUniformsiv(program, 4, uniformIndices, GL\_UNIFORM\_ARRAY\_STRIDE, arrayStrides);

glGetActiveUniformsiv(program, 4, uniformIndices, GL\_UNIFORM\_MATRIX\_STRIDE, matrixStrides);

Per i tipi semplici come int, float, bool, e per i vettori tutto ciò che serve conoscere è l’offset.

Noto l’offset ci sono 2 modi per lo si può si usare nella funzione glBufferSubData() per caricare i dati nell’apposita locazione, sia usare l’offset direttamente nel codice per assemblare il buffer in memoria.

RIPRENDERE DA PAG 159

RIPRENDO DA PAG 168

Funzione che restituisce l’indice del blocco uniform :

**GLuint glGetUniformBlockIndex(GLuint program, const GLchar \* uniformBlockName);**

Esiste un insieme di “Buffer Binding Point” ai quali è possibile legare il buffer per fornire dati al blocco uniform.

Il processo con il quale si lega un Buffer ad un blocco Uniform è in due step:

1. Il blocco Uniform è assegnato ad un Binding Point
2. Il Buffer viene legato allo stesso Binding Point

Per assegnare un Binding Point ad un blocco Uniform si utilizza la funzione:

**void glUniformBlockBinding(GLuint program, GLuint uniformBlockIndex, GLuint uniformBlockBinding);**

**uniformBlockIndex** è l’indice del blocco Uniform.

**uniformBlockBinding** è l’indice del Binding Point che viene restituito dalla funzione.

E’ anche possibile specificare il Binding Index del blocco Uniform direttamente nello shader come segue:

layout (std140,binding = 2) uniform TransformBlock

{

….

}transform;

Una volta definito il Binding Point del blocco Uniform, leghiamo il buffer allo stesso Binding Point, in questo modo il blocco Uniform con i dati del Buffer :

**glBindBufferBase(GL\_UNIFORM\_BUFFER, index, buffer);**

GL\_UNIFORM\_BUFFER dichiara l’intenzione di legare il buffer ad un blocco uniform,

**Index** è l’indice del Binding Point e deve avere lo stesso valore di UniformBlockBinding restituito dalla funzione glUniformBlockBinding.

##### **3 - SHADER STORAGE BLOCKS**

Principali differenze con gli Unifom Blocks :

1)Uniform Blocks possono solo ricevere dati dal Buffer e non scrivere sul Buffer, mentre gli Shader Blocks possono fare entrambe le cose

2)Shader Blocks supportano anche il layout standard 430 che consente un impacchettamento stretto dei membri del blocco, a differenza del std140 che è l’unico supportato dagli Uniform.

3) Shader Blocks hanno capacità di storage illimitata, a differenza degli Uniform Block dove c’è un upper limit.

Fatte queste premesse perché dovremmo usare gli Uniform o i Vertex Attributes ? In realtà la maggiore flessibilità data dagli Shader Block la si paga con una maggiore difficoltà di ottimizzazione da parte di OGL, pertanto in alcune implementazioni gli Uniform o i VA potrebbero essere più performanti.

Esempio dichiarazione di uno Shader Blocks (in nero) .

version 450 core

struct my\_structure

{

int pea;

int carrot;

vec4 potato;

};

layout (binding = 0, std430) **buffer** my\_storage\_block

{

vec4 foo;

vec3 bar;

int baz[24];

my\_structure veggies;

};

come si vede la dichiarazione è analoga a quella di uno uniform, ma qui si usa la keyword **buffer.**

###### Leggere e Scrivere da/in uno Shader Block

1)Per leggere il contenuto dei membri di uno Shader Storage Block è sufficiente usarli come variabili.

2)Per scrivere in essi, basta fare un normale assegnamento.

Nel primo caso la sorgente del dato sarà il Buffer.

Nel secondo caso il Buffer sarà il sito depositario del dato.

Per far si che il Buffer possa fungere da sorgente useremo la funzione solita **glBufferData()**.

Per far si che il Buffer possa essere usato per la scrittura da shader a buffer useremo la funzione

**glMapBufferRange()** con flag di accesso : **GL\_MAP\_READ\_BIT** (o **GL\_MAP\_WRITE\_BIT**) è possibile leggere il dato prodotto dallo shader.

**Atomic Operations**

Operazioni di lettura/scrittura, performabili sui membri di uno Storage Block, che “blinda” la location di memoria su cui opera, ovvero non permette a nessun’ altra operazione sulla stessa location di memoria di essere eseguita, prima che il **ciclo di lettura-modifica-scrittura** non sia completato.

Esempio:

Si consideri il caso di due invocazioni a shader performano il calcolo *m = m+1* , usando la stessa locazione di memoria rappresentata da m . Se le due invocazioni avvengono in contemporanea il risultato finale sarà errato.

* Se l’operazione è resa atomica, le 2 invocazioni avvengono sequenzialmente, ed il risultato sarà corretto.

E’ possibile invocare operazioni Atomiche sui membri degli Shader Blocks utilizzando le seguenti funzioni :

atomicAdd(mem,data) : legge da *mem*, somma a *data*, mette il risultato in *mem*, ritorna il valore di contenuto inizialmente in *mem*.

AtomicAnd(mem,data) : legge da mem, opera l’AND logico con data, salva il risultato in mem, ritorna il contenuto originale di mem.

AtomicOr(mem,data) : vedi sopra

AtomicXor(mem,data): vedi sopra (XOR come OR ma V V => F)

AtomicMin(mem,data) : legge da mem, compara con data, mette il minimo in mem, ritorna il valore originale di mem.

AtomicMax(mem,data) : vedi sopra

AtomicExchange(mem,data) : legge mem, scrive data in mem, ritorna il valore di mem originale.

atomicCompSwap(mem,comp,data) : legge mem, confronta con comp, se mem = comp => scrivo data in mem. Ritorna sempre il valore originale in mem.

Tutte le funzioni hanno una versione int e una uint :

**versione int : dichiarazioni :**

inout int mem;

int comp;

int data;

tutte le funzioni che prendono questi valori ritornano int.

**versione uint : dichiarazioni:**

inout uint mem;

uint comp;

uint data;

tutte le funzioni che prendono questi valori ritornano uint.

Nota che non esistono operazioni atomiche per floating point, vettori o matrici o interi maggiori di 32-bit.

Quando più operazioni atomiche vengono invocate allo stesso istante sulla stessa location in memoria queste vengono eseguite a turni, quindi non si sa a priori quale sia il risultato (parziale) di una specifica operazione atomica.

**Barriere per la sincronizzazione degli accessi alla memoria**

Quando si scrive su buffer, sia che lo si faccia attraverso operazioni atomiche che attraverso la scrittura su variabili di Shader Blocks, bisogna prestare attenzione a 3 casi in particolare :

**RAW, WAW , WAR** ovvero , Read after Write, Write after Write, Write after Read .

Una qualsiasi di queste sequenze di operazioni sulla stessa location in memoria è rischiosa poichè talvolta, anche in base all’architettaura del Sistema, potrebbero essere invertiti gli ordini , ovvero ad esempio una operazione RAW effettua prima la scrittura e dopo la lettura …

Chiaramente questo genere di comportamento errato, porta a gravi conseguenze sull’applicazione.

OpenGL include dei meccanismi per alleviare e controllare questo genere di rischi.

Le barriere di memoria funzionano obbligando tutte le operazioni che stanno dopo la dichiarazione della barriera ad avvenire dopo tutte le operazioni che stanno prima la barriera.

Le barriere possono essere inserite sia nell’applicazione che negli shaders.

**Barriere in OpenGL**

La funzione che applica le barriere di memoria in OGL è :

**void glMemoryBarrier(GLbitfield barriers);**

**GLbitfield** specifica quale sottosistema di memoria di OpenGL deve obbedire alla barriera e quali invece no (vedi anche link Sincronizzazione dell’accesso ai contatori atomici), di seguito si elencano alcuni dei molti tipi di flag possibili :

**GL\_ALL\_BARRIER\_BITS** : Se si vuole sincronizzare tutti i processi

**GL\_SHADER\_STORAGE\_BARRIER\_BIT :**

..

..operazioni pre barriera..

..

glMemoryBarrier(GL\_SHADER\_STORAGE\_BARRIER\_BIT);

..

..operazioni post barriera..

Operations dopo la barriera possono partire se solo se operations prima della barrier ha completato.

Quindi se per esempio in uno shader scrivo su uno shader storage buffer, e metto questa barriera, garantisco che gli shaders lanciati dopo questa barriera vedranno il dato che ho scritto.

**GL\_UNIFORM\_BARRIER\_BIT :**

garantisce la chiusura di qualsiasi operazione di scrittura su buffer, prima di un eventuale utilizzo del buffer come Uniform Buffer.

Lo puoi usare se hai scritto in un buffer attraverso uno Shader Storage Block, e vuoi utilizzare quel buffer come sorgente per uno uniform.

**GL\_VERTEX\_ATTRIB\_ARRAY\_BARRIER\_BIT:**

garantisce la chiusura di qualsiasi operazione di scrittura su buffer, prima di un eventuale utilizzo del buffer come sorgente per un Vertex Attribute.

Barriere in GLSL

Esistono molteplici funzioni GLSL per applicare barriere di memoria :

**memoryBarrier() :** qualsiasi operazione di lettura e scrittura in corso sarà completata prima che questa funzione ritorni.

**memoryBarrierBuffer() :** ordina le transazioni in lettura e scrive sul buffer.

Ne esistono molte altre che vedremo più avanti.

#### ATOMIC COUNTERS

I contatori atomici sono variabili speciali che rappresentano una memoria condivisa tra più invocazioni di uno shader. Questa memoria è sostenuta da un buffer, che viene incrementato/decrementato da **opportune funzioni GLSL**, queste funzioni sono atomiche : proprio come per le operazioni atomiche applicabili ai membri di uno Shader Block, queste funzioni ritornano sempre il valore della variabile in memoria prima dell’ultima operazione , proprio come le operazioni atomiche, se lo shader viene invocato più volte contemporaneamente, queste funzioni vengono sequenzializzate : in questo modo la prima funzione operante vede il valore originale del contatore, la seconda vedrà il contatore incrementato di +1 e così via…

Dichiarazione di un contatore atomico in uno shader :

**layout (binding = 0) uniform atomic\_uint my\_variable;**

OpenGL fornisce una serie di punti di legame ai quali si possono attaccare i buffers dove vengono immagazzinati i valori dei contatori atomici, inoltre ogni contatore atomico è memorizzato nel buffer con un certo offset.

Tale offset può essere specificato come nell’esempio seguente:

**layout (binding = 0, offset = 8) uniform atomic\_uint my\_variable;**

Per fornire un buffer di memoria per questo contatore atomico procediamo come segue :

GLuint buf;

glGenBuffers(1, &buf);

glBindBuffer(GL\_ATOMIC\_COUNTER\_BUFFER, buf);

glBufferData(GL\_ATOMIC\_COUNTER\_BUFFER, 16 \* sizeof(GLuint),

NULL, GL\_DYNAMIC\_COPY);

glBindBufferBase(GL\_ATOMIC\_COUNTER\_BUFFER, 3, buf);

1)Generate a buffer name

2)Bind it to the generic GL\_ATOMIC\_COUNTER\_BUFFER target and

3)initialize its storage

4)Now bind it to the fourth indexed atomic counter buffer target

Prima di usare un Atomic Counter è bene resettarlo :

Si possono usare a questo fine, differenti approcci :

1. glBufferSubData(); //si passa l’indirizzo di una variabile contenente il valore al quale si vuole inizializzare il contatore
2. glMapBufferRange(); //si mappa il buffer e si scrive il valore direttamente
3. glClearBufferSubData();

Esempio dell’approccio 3 :

Bind our buffer to the generic atomic counter buffer binding point

**glBindBuffer(GL\_ATOMIC\_COUNTER\_BUFFER, buf);**

// Method 1 - use glBufferSubData to reset an atomic counter

**const GLuint zero = 0;**

**glBufferSubData(GL\_ATOMIC\_COUNTER\_BUFFER, 2 \* sizeof(GLuint),**

**sizeof(GLuint), &zero);**

// Method 2 - Map the buffer and write the value directly into it

**GLuint \* data =**

**(GLuint \*)glMapBufferRange(GL\_ATOMIC\_COUNTER\_BUFFER,**

**0, 16 \* sizeof(GLuint),**

**GL\_MAP\_WRITE\_BIT |**

**GL\_MAP\_INVALIDATE\_RANGE\_BIT);**

**data[2] = 0;**

**glUnmapBuffer(GL\_ATOMIC\_COUNTER\_BUFFER);**

// Method 3 - use glClearBufferSubData

**glClearBufferSubData(GL\_ATOMIC\_COUNTER\_BUFFER,**

**GL\_R32UI,**

**2 \* sizeof(GLuint),**

**sizeof(GLuint),**

**GL\_RED\_INTEGER, GL\_UNSIGNED\_INT,**

**&zero);**

###### Funzioni per Atomic Counters

La seguente funzione ritorna il valore del contatore atomico.

**uint atomicCounter(atomic\_uint c);**

Funzioni di incremento

**Uint atomicCounterIncrement (atomic\_uint c) ;**

Questa funzione legge l’attuale valore dell’atomic counter, lo incrementa di +1 e sovrascrive l’atomico counter, infine ritorna il valore originariamente letto dal buffer dell’atomic counter.

Come detto le operazioni atomiche hanno la funzione di prendere eventuale N invocazioni contemporanee e sequenzializzarle in modo casuale(definire un ordine con cui verranno effettuate le invocazioni, ad esempio supponendo il caso di 5 invocazioni : a,b,c,d,e => c ->1, b->2 , e ->3 , a->4 , d ->5 , l’ordine dato è casuale) , poiché tale ordine è casuale, se invoco due volte uno shader con la chiamata atomicCounterIncrement(atomic\_uint c) ; non otterrò necessariamente due valori consecutivi.

**Uint atomicCounterDecrement(atomic\_uint c);**

a differenza della funzione di incremento questa ritorna il nuovo valore memorizzato, eg il valore decrementato di -1.

Pertanto se in uno shader abbiamo in successione la funzione incrementale e decrementale, il valore ritornato dalle due sarà lo stesso.

In pratica però di solito avremo invocazioni molteplici e contemporanee dello shader, quindi difficilmente le due funzioni incrementale e decrementale ritorneranno lo stesso valore, perché verranno sequenzializzate randomicamente .

Ovvero se indichiamo con 11 l’invocazione della funzione incrementale all’interno dell’invocazione 1 dello shader.

Con 12 l’invocazione della funzione decrementale all’interno dell’invocazione 1 dello shader

Con 21 l’invocazione della funzione incrementale nell’invocazione 2 dello shader

Con 22 l’invocazione della funzione decrementale nell’invocazione 2 dello shader

Con 31 …..

…

E così via.

Si capisce che se ho 4 invocazioni allo shader , le funzioni verranno sequenzializzate a caso , ad esempio potremmo avere :

32,12,22,41,42,11,.... e così via...

Si vede chiaramente che non sempre una x1 e seguita da una y2, motivo per cui è improbabile.

Barriere per la sincronizzazione dell’accesso ai contatori atomici

Durante l’esecuzione di uno shader il valore dei contatori atomici risiede in una memoria speciale nel processore grafico, per questo le operazioni sui contatori atomici sono più rapide di generiche operazioni atomiche sui membri degli Shader Blocks.

Quando l’esecuzione dello shader termina però il valore dei contatori verrà scritto in memoria, e pertanto l’operazione di incremento/decremento dei contatori atomici è considerata alla stregua di una operazione di memoria e quindi suscettibile dei rischi tipici di questo tipo di operazioni (vedi anche **Barriere per la sincronizzazione degli accessi alla memoria**).

Barriere in OpenGL

Per questo motivo la funzione glMamoryBarrier() supporta un bit specifico per sincronizzare gli accessi ai buffer degli atomic counters.

Chiamare :

**glMemoryBarrier(GL\_ATOMIC\_COUNTER\_BARRIER\_BIT);**

assicura che ogni accesso ad un atomic counter sarà sincronizzato in modo tale che qualsiasi operazione di lettura dal buffer del contatore veda sempre i dati aggiornati.

In generale il bit che va all’interno di glMemoryBarrier() è quello relativo all’operazione per cui quel buffer sarà usato, ovvero, se ho incrementato il contatore atomico e lo uso dopo per leggere il valore ed incrementare un contatore atomico allora utilizzerò GL\_ATOMIC\_COUNTER\_BARRIER, se invece devo usare il valore nel buffer del contatore per scriverlo in qualche membro di uno Shader Block allora dovrò usare GL\_SHADER\_STORAGE\_BARRIER\_BIT.

Barriere in GLSL

La funzione GLSL corrispondente all’invocazione di glMemoryBarrier(GL\_ATOMIC\_COUNTER\_BARRIER\_BIT) è **memoryBarrierAtomicCounter().**

### TEXTURES

Sono allocazioni di memoria strutturate, che possono essere rese accessibili dagli shaders sia per la lettura che per la scrittura. La struttura più comune per una memoria di tipo texture è 2D, ma può essere anche 1D o 3D.

Creare una texture in OGL:

// The type used for names in OpenGL is GLuint

GLuint texture;

// Create a new 2D texture object

glCreateTextures(GL\_TEXTURE\_2D, 1, &texture);

// Specify the amount of storage we want to use for the texture

glTextureStorage2D(texture, // Texture object

1, // 1 mipmap level

GL\_RGBA32F, // 32-bit floating-point RGBA data

256, 256); //width and height in texel

// Now bind it to the context using the GL\_TEXTURE\_2D binding point

glBindTexture(GL\_TEXTURE\_2D, texture);

Si noti la similitudine con **Step necessari per l’uso dei buffers :**

**glTextureStorage2D()** prende come parametro il nome della texture da allocare per lo storage, il numero di livelli usati per il mipmapping (vedi in seguito), il formato interno (GL\_RGBA32F è un formato floating point a 4 canali).

Di seguito vediamo come riempire di dati la texture usando la funzione :

void **glTexSubImage2D**(GLenum target, GLint level, GLint xoffset, GLint yoffset, GLsizei width, GLsizei height, GLenum format, GLenum type, const GLvoid \* pixels);

o alternativamente

|  |  |
| --- | --- |
| void **glTextureSubImage2D**( | GLuint *texture*, |
|  | GLint *level*, |
|  | GLint *xoffset*, |
|  | GLint *yoffset*, |
|  | GLsizei *width*, |
|  | GLsizei *height*, |
|  | GLenum *format*, |
|  | GLenum *type*, |
|  | const void \**pixels*); |

// Define some data to upload into the texture

float \* data = new float[256 \* 256 \* 4];

// generate\_texture() is a SuperBible function that fills memory with image data

generate\_texture(data, 256, 256); //link -> **Generate\_texture function (code snippets)**

// Assume that "texture" is a 2D texture that we created earlier

glTextureSubImage2D(texture, // Texture object

0, // Level 0

0, 0, // Offset 0, 0

256, 256, // 256 x 256 texels, replace entire image

GL\_RGBA, // Four-channel data

GL\_FLOAT, // Floating-point data

data); // Pointer to data

// Free the memory we allocated before - OpenGL now has our data

delete [] data;

Dopo questo passaggio si va a **Leggere le texture negli Shaders**

Se si vuole invece inizializzare una texture ad un valore fissato si può usare la funzione :

**void glClearTexSubImage(GLuint texture,**

**GLint level,**

**GLint xoffset,**

**GLint yoffset,**

**GLint zoffset,**

**GLsizei width,**

**GLsizei height,**

**GLsizei depth,**

**GLenum format,**

**GLenum type,**

**const void \* data);**

La dimensione della texture viene dedotta dall’oggetto passato.

**Level** specifica il livello mipmap su cui si applica la texture (diciamo per comodità “da texturare”).

**Xoffset , yoffset**, **zoffset** forniscono gli offset delle regioni da “texturare”.

**Width, height, depth** specificano le dimensioni della regione da texturare.

**Format, type** hanno lo stesso significato che hanno nella funzione glTexSubImage2d()

**Data** si assume sia un dato valido per un singolo **texel(texture element)** che viene replicato per l’intera texture.

#### Texture Target e Tipi

Il **texture target** è sostanzialmente il modello di texture che vogliamo creare. Una volta che leghiamo un **oggetto texture** al texture target, l’oggetto texture diventa una texture del **tipo** specificato dal texture target.

|  |
| --- |
| Texture target :  GL\_TEXTURE\_\* |
| 1D texture 1-dimensionale  2D texture 2-dimensionale  3D texture 3-dimensionale  RECTANGLE texture rettangolo  1D\_ARRAY array di texture 1D  2D\_ARRAY array di texture 2D  CUBE\_MAP texture a mappa cubica  CUBE\_MAP\_ARRAY array di texture a mappa cubica  BUFFER buffer texture ?  2D\_MULTISAMPLE texture multicampione 2D  2D\_MULTISAMPLE\_ARRAY array di texture multicampione 2D |

GL\_TEXTURE\_2D è il texture target che useremo di più. GL\_TEXTURE\_1D è una texture 2D di altezza fissata a 1. GL\_TEXTURE\_3D può essere usata per rappresentare un volume ed ha effettivamente 3 coordinate.

GL\_TEXTURE\_RECTANGLE sono texture 2D speciali che sono state introdotte quando gli hardware non supportavano texture di dimensioni pari ad una potenza di 2 : ad oggi sono pressochè inutilizzate.

Tutte quelle con “\_ARRAY” sono semplicemente array dei tipi di texture suddetti.

GL\_TEXTURE\_BUFFER sono un tipo speciale di texture 1D ad eccezione del fatto che la loro memoria è un buffer e ha dimensione massima molto maggiore di quella delle 1D. Inoltre differiscono dalle texture 1D perchè mancano di alcune importanti caratteristiche come il filtraggio e il mipmap.

GL\_TEXTURE\_2D\_MULTISAMPLE e GL\_TEXTURE\_2D\_MULTISAMPLE\_ARRAY sono usate per l’antialiasing multicampione.

#### Leggere le texture negli Shaders - Sampler

Le texture sono rappresentate negli shaders attraverso le variabili campionatrici (sampler variables), e vengono agganciate al mondo esterno dichiarando variabili uniform con tipi campionatori .

Così come ci sono diversi tipi di texture, ci sono anche diverse variabili campionatrici :

**sampler2D** è la variabile associata alle texture 2D.

**texelFetch(sampler variable, fragment coordinates, mipmapping level)** è la funzione GLSL built-in per la lettura delle texture.

*gl\_FragCoord comincia dal vertice a sx in basso (0,0) . Pertanto la texture in s viene applicata a partire da quel punto.*

#version 450 core

uniform sampler2D s;

out vec4 color;

void main(void)

{

color = texelFetch(s, ivec2(gl\_FragCoord.xy), 0);

}

Il **mipmapping level** è invece posto a zero poiché in questo esempio vogliamo un solo livello.

5.37 : Reading from a texture in GLSL (fragment shader)

Nell’esempio leggiamo dalla variabile campionatrice **s** usando la coordinata texture derivata dalla variabile built-in **gl\_FragCoord.**

Questa variabile è un input al fragment shader che contiene le coordinate floating point del frammento processato in coordinate di finestra.

#### La funzione **texelFetch**

**vec4** texelFetch(**sampler1D** s, **int** P, **int** lod);

**vec4** texelFetch(**sampler2D** s, **ivec2** P, **int** lod);

**ivec4** texelFetch(**isampler2D** s, **ivec2** P, **int** lod);

**uvec4** texelFetch(**usampler3D** s, **ivec3** P, **int** lod);

texelFetch performs a lookup of a single texel from texture coordinate *P* in the texture bound to ***sampler***. The array layer is specified in the last component of ***P*** for array forms. The ***lod*** parameter (if present) specifies the level-of-detail from which the texel will be fetched. The *sample* specifies which sample within the texel will be returned when reading from a multi-sample texure.

accetta coordinate intere che variano da (0,0) a (texture\_width, texture\_height), per questo motivo costruiamo un vettore di interi (**ivec)** a partire dalle coordinate x e y di gl\_Fragcoord.

Nota che tale funzione è presente in molte versioni (overloaded function), una per ogni tipo di texture, ad esempio abbiamo .

**s**  è la variabile sampler contenente la texture, **P** è la coordinata del punto in cui applichiamo la texture, **lod** è il livello di mipmap della texture.

Vedi un esempio di uso [qui](#_Leggere_le_texture)

#### Tipi di Sampler

Tipi di texture e relativi Sampler.

|  |  |
| --- | --- |
| Texture target :  GL\_TEXTURE\_\* | SAMPLER TYPE |
| 1D texture 1-dimensionale  2D texture 2-dimensionale  3D texture 3-dimensionale  RECTANGLE texture rettangolo  1D\_ARRAY array di texture 1D  2D\_ARRAY array di texture 2D  CUBE\_MAP texture a mappa cubica  CUBE\_MAP\_ARRAY array di texture a mappa cubica  BUFFER buffer texture ?  2D\_MULTISAMPLE texture multicampione 2D  2D\_MULTISAMPLE\_ARRAY array di texture multicampione 2D | Sampler1D  Sampler2D  Sampler3D  Sampler2DRect  Sampler1DArray  Sampler2DArray  samplerCube  samplerCubeArray  samplerBuffer  sampler2DMS  sampler2DMSArray |

I tipi GLSL sampler1D, sampler2D etc rappresentano dati floating point. Esistono anche i rispettivi tipi interi signed (**isampler2D**) e unsigned(**usampler2D**) etc

### La funzione **textureSize**

**int** textureSize(**sampler1D** sampler, **int** lod);

**ivec2** textureSize(**sampler2D** sampler, **int** lod);

**ivec3** textureSize(**gsampler3D** sampler, **int** lod);

Anche questa funzione prende il parametron **lod** che specifica il livello mipmap della texture di cui vogliamo conoscere la dimensione.

### Caricare Textures da files

Ci sono due modi per utilizzare le textures:

1. convertire le texture in hard-coded arrays
2. caricarle direttamente da files.

Purtroppo ci sono veramente pochi formati di immagini che possono immagazzinare in modo proprio tutti i formati supportati da OGL.

Uno di questi formati è il **.KTX** (Khronos TeXture format), che è stato appositamente pensato per immagazzinare tutto ciò che può essere rappresentato come OGL Texture.

Infatti il **.KTX** include la maggior parte dei parametri che devono essere passati alle funzioni di texturing come glTextureStorage2D() e glTextureSubImage2D().

L’header di un file .KTX è il seguente.

**struct** header

{

**unsigned char** identifier[12];

**unsigned int** endianness;

**unsigned int** gltype;

**unsigned int** gltypesize;

**unsigned int** glformat;

**unsigned int** glinternalformat;

**unsigned int** glbaseinternalformat;

**unsigned int** pixelwidth;

**unsigned int** pixelheight;

**unsigned int** pixeldepth;

**unsigned int** arrayelements;

**unsigned int** faces;

**unsigned int** miplevels;

**unsigned int** keypairbytes;

};

Listing 5.38: The header of a .KTX file

(per dettagli sui parametri di questo header vedi pag 200 OGL Superbible)

Poichè il formato .KTX è stato pensato appositamente per l’uso in applicazioni basate su OGL, scrivere del codice per caricare un file .KTX è molto semplice.

Nonostante ciò OpenGL Superbible include una funzione apposita che permette il caricamento di immagini .KTX in texture :

// Generate a name for the texture

glGenTextures(1, &texture);

// Load texture from file

sb7::ktx::file::load("media/textures/icemoon.ktx", texture);

Se il file .KTX è riconosciuto il loader associerà direttamente la texture al target appropriato e la caricherà con i dati dal .KTX.

Se l’operazione va a buon fine il loader ritorna il nome della texture, in caso contrario ritorna 0.

E’ opportuno eliminare la texture ritornata dal loader dopo che è stata caricata chiamando la funzione **glDeleteTextures().**

## **CAPITOLO 6 : SHADERS AND PROGRAMS**

GLSL is a C like language, yet there are major differences :

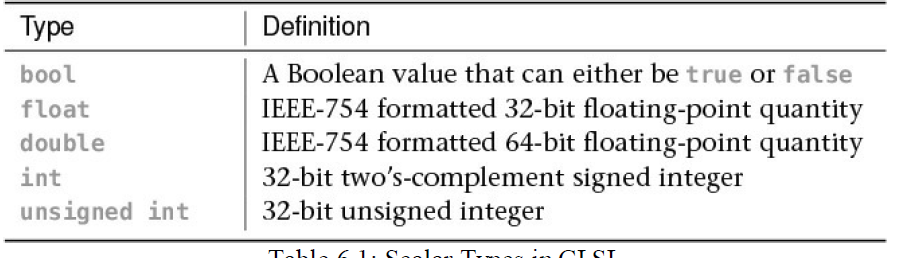
Matrix and vector types are built into the language.

Another major difference between GLSL and C is that GLSL is designed to run on massively parallel

implementations—most graphics processors will run thousands of copies (or

*invocations*) of your shaders at the same time.

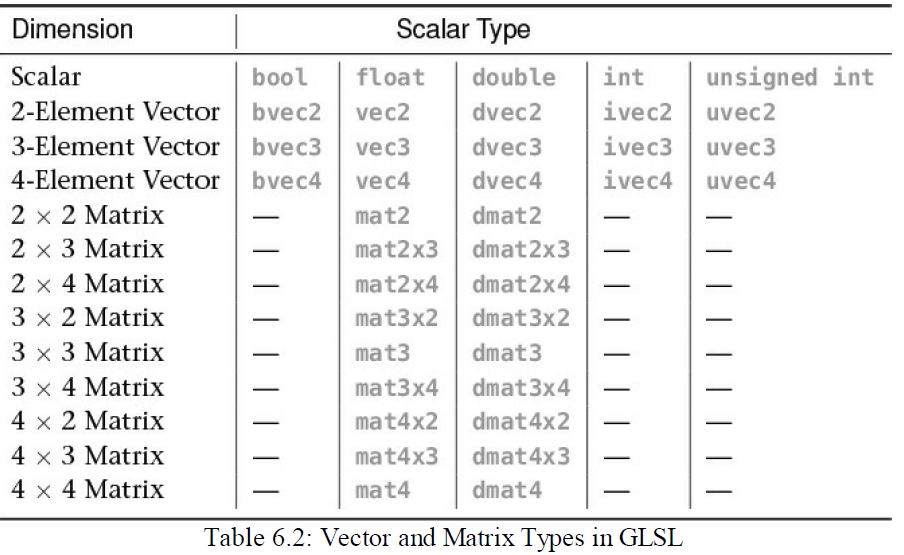
### The scalar types supported in GLSL are shown



Also, GLSL doesn’t support chars pointers or integer types larger than 32 bits.

Vector and matrix type names are decorated with their underlying scalar type’s name; the exceptions are floating-point vectors and matrices, which have no decoration.

### Vector and matrix



Vectors may be constructed from other vectors, from a single scalar, from sequences of

scalars, or from any combination of scalars and vectors of the appropriate type, so long

as there are enough fields in total to fill the destination.

The components of a vector may be accessed as if the vector were an array. That is, the

four components of **vec4** foo may be accessed through the classic c/c++ like subscriptor operator [] : so the first component will be foo[0], the second foo[1], and so on …

Alternatively, vectors may be accessed as if they were structures with fields

representing their components.

The **first** component can be accessed through the .**x, .s,or .r** field.

The **second** component is accessed through the **.y, .t, or .g** field.

The **third** is accessed through the **.z, .p, or .b** field.

The **fourth** component can be accessed through the **.w, .q, or .a** field.

This seems confusing, but *x*, *y*, *z*, and *w* are often used to denote **positions or directions**;

*r*, *g*, *b*, and *a* are often used to represent **colors**;

and *s*, *t*, *p*, and *q* are used to denote **texture coordinates**.

the first three components of foo (which is a **vec4**) could be extracted by writing foo.xyz (or foo.rgb or foo.stp). The powerful thing is that you can also specify these fields in any order you

wish, and you can repeat them. So, foo.zyx would produce a three-element vector

with the *x* and *z* fields of foo swapped, and foo.rrrr would produce a four-element

vector with the *r* component of foo in every field.

matrices can be treated like twodimensional arrays. For example, if we declare bar as a **mat4** type, then bar[0] is a vec4representing its first column, and bar[0][0] is the first component of that

vector (as is bar[0].x), bar[0][1] is the second component of the vector (which is equivalent to bar[0].y), and so on

### Built-in Matrix and Vector Functions

#### Comparator

If you need to compare two vectors to each other, a number of built-in functions will do

this for you in a component-by-component manner:

lessThan(), lessThanEqual(), greaterThan(), greaterThanEqual(), equal(), and notEqual().

bvec **lessThanEqual**(vec x, vec y);

returns a boolean vector in which each element *i* is computed as x[*i*] ≤ y[*i*].

bool **all**(bvec *x*);

all returns true if all elements of x are true and false otherwise.

Each of these functions takes two vectors of the same type and size,

applies the operation that its name suggests, and returns a Boolean vector of the same

size of the function’s parameters (that is, a **bvec2**, **bvec3**, or **bvec4**). Each

component of this Boolean vector contains the result of the comparison for the

corresponding components in the source parameters.

Given a Boolean vector, you can test it to see if any of its components are **true** using

the **any()** function; you can determine whether *all* of its components are **true** with the

**all()** function. You can also invert the value of a Boolean vector using the not()

function.

### Operators

Standard operators, such as + and -, are defined for vectors and matrices. The

multiplication operator (\*) is defined between two vectors to be component-wise, and

between two matrices or a matrix and a vector as a matrix–matrix or matrix–vector

multiplication operation. Division of vectors and matrices by scalars behaves as

expected. Division of vectors and matrices by other vectors and matrices is executed

component-wise, so it requires the two operands to be of the same dimension.

### Array declaration and Constructor

There are two ways to declare an array in GLSL. The first is similar to the syntax used

in C or C++, where the array size is appended to the variable name. The following are

examples of this type of declaration:

**float** foo[5];

**ivec2** bar[13];

**dmat3** baz[29];

The second syntax is to implicitly declare the type of the whole array by appending the

size to the *element type* rather than the variable name. Thus the above declaration could

equivalently be written as follows:

**float**[5] foo;

**ivec2**[13] bar;

**dmat3**[29] baz;

To a C programmer, this may seem odd. However, it’s actually a very powerful feature

because it allows types to be implicitly defined without the **typedef** keyword, which

GLSL lacks. One example use of this is to declare a function that returns an array:

**vec4**[4] functionThatReturnsArray()

{

**vec4**[4] foo = ...

**return** foo;

}

Declaring array types in this form also implicitly defines the **constructor** for the array.

This means that you can write:

**float**[6] var = **float**[6](1.0, 2.0, 3.0, 4.0, 5.0, 6.0);

However, in this case, recent versions of GLSL also allow the traditional, C-style

array initializer syntax to be used:

**float** var[6] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 };

### Appunti

* Una modifica agli shaders non richiede la ricompilazione del programma!

Lo shader program viene ricompilato a runtime, e non ha nulla a che vedere con la compilazione del programma.

Un prodotto il GLSL va sempre indicato con l’asterisco (ad esempio 2N è sbagliato mentre 2\*N è corretto).

### Vertex Shader di una circonferenza

#version 450 core

layout (location = 10) uniform mat4 transformMatrix;

void **main**(){

const int N = 100;

vec4 vertices;

float x,y,z;

float pi = 180.0;

float r = .5;

int i = 0;

float theta = 2\*pi/N\*gl\_VertexID;

vertices.x = r\***cos**(theta);

vertices.y = r\***sin**(theta);

vertices.z = 0.0;

vertices.w = 1.0;

gl\_Position = transformMatrix\*vertices;

}

### Vertex Shader di una Sfera

Sfera ottenuta facendo ruotare una circonferenza attorno all’asse x.

Il problema di questa sfera è che non posso utilizzare GL\_TRIANGLES come primitiva.

#version 450 core

layout (location = 10) uniform mat4 transformMatrix;

void **main**(){

const int N = 100;

vec4 vertices;

float pi = 180.0;

float r = .5;

int i = 0;

float theta = 2\*pi/N\*gl\_VertexID;

vertices.x = r\***cos**(theta);

vertices.y = r\***sin**(theta);

vertices.z = 0.0;

vertices.w = 1.0;

for(int M = 0; M < 7; M++){

*//con 7 iterazioni (M<7) ottengo un giro completo di circonferenze equidistribuite nello spazio.*

if(gl\_VertexID > M\*N && gl\_VertexID <= (M+1)\*N){

float x\_angle = M\*pi/N;

mat4 rotation = **mat4**(

**vec4**(1.0,0.0,0.0,0.0),

**vec4**(0.0,**cos**(x\_angle),-**sin**(x\_angle),0.0),

**vec4**(0.0,**sin**(x\_angle),**cos**(x\_angle),0.0),

**vec4**(0.0,0.0,0.0,1.0));

vertices = vertices\*rotation;

gl\_Position = transformMatrix\*vertices;

}

}

}

## **APPENDIX A : VARIOUS TOPICS**

### Text Rendering

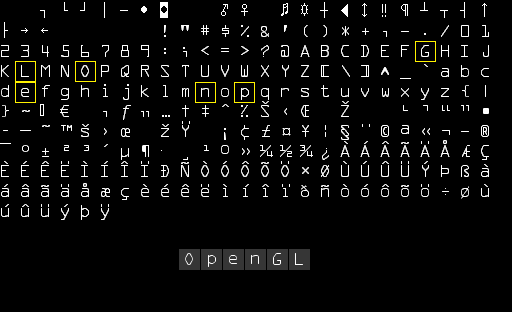
At some stage of your graphics adventures you will want to draw text in OpenGL. Contrary to what you might expect, getting a simple string to render on screen is quite difficult with a low-level library like OpenGL. If you don't care about rendering more than 128 different characters, then it's probably not too difficult. Things are getting more difficult as soon as each character has a different width, height and margin. Based on where you live, you might also need more than 128 characters, and what if you want to express special symbols for like mathematical expressions or sheet music symbols, and what about rendering text from top to bottom? Once you think about all these complicated matters of text, it wouldn't surprise you that this probably doesn't belong in a low-level API like OpenGL.

Since there is no support for any text capabilities within OpenGL it is up to us to define a system for rendering text to the screen. Since there are no graphical primitives for text characters, we have to get creative. Some example techniques are: drawing letter shapes via GL\_LINES, create 3D meshes of letters or render character textures to 2D quads in a 3D environment.

Most often developers choose to render character textures onto quads. Rendering these textured quads by itself shouldn't be too difficult, but getting the relevant character(s) onto a texture could prove challenging. In this tutorial we'll explore several methods and implement a more advanced, but flexible technique for rendering text using the FreeType library.

#### Classical text rendering: bitmap fonts

In the early days rendering text involved selecting a font (or create one yourself) you'd like for your application and extracting all relevant characters out of this font to paste them all within a single large texture. Such a texture, that we from now on call a bitmap font, contains all character symbols we want to use in predefined regions of the texture. These character symbols of the font are known as glyphs. Each glyph has a specific region of texture coordinates associated with them. Whenever you want to render a character, you select the corresponding glyph by rendering this section of the bitmap font to a 2D quad.



Here you can see how we would render the text 'OpenGL' by taking a bitmap font and sampling the corresponding glyphs from the texture (by carefully choosing the texture coordinates) that we render on top of several quads. By enabling [blending](https://learnopengl.com/#!Advanced-OpenGL/Blending) and keeping the background transparent, we will end up with just a string of characters rendered to the screen. This particular bitmap font was generated using Codehead's Bitmap [Font Generator](http://www.codehead.co.uk/cbfg/).

This approach has several advantages and disadvantages. First, it is relatively easy to implement and because bitmap fonts are pre-rasterized they're quite efficient. However, it is not particular flexible. When you want to use a different font, you need to recompile a complete new bitmap font and the system is limited to a single resolution; zooming will quickly show pixelated edges. Furthermore, it is often limited to a small character set so Extended or Unicode characters are often out of the question.

This approach was quite popular back in the day (and still is) since it is fast and works on any platform, but as of today more flexible approaches exist. One of these approaches is loading TrueType fonts using the FreeType library.

#### Modern text rendering: FreeType

FreeType is a software development library that is able to load fonts, render them to bitmaps and provide support for several font-related operations. It is a popular library used by Mac OS X, Java, PlayStation Consoles, Linux and Android to name a few. What makes FreeType particularly attractive is that it is able to load TrueType fonts.

A TrueType font is a collection of character glyphs not defined by pixels or any other non-scalable solution, but by mathematical equations (combinations of splines). Similar to vector images, the rasterized font images can be procedurally generated based on the preferred font height you'd like to obtain them in. By using TrueType fonts you can easily render character glyphs of various sizes without any loss of quality.

FreeType can be downloaded from their [website](http://www.freetype.org/). You can choose to compile the library yourself from their source code or use one of their precompiled libraries if your target platform is listed. Be sure to link to freetype.lib and make sure your compiler knows where to find the header files.

Then be sure to include the appropriate headers:

#include <ft2build.h>

#include FT\_FREETYPE\_H

Due to how FreeType is developed (at least at the time of this writing), you cannot put their header files in a new directory; they should be located at the root of your include directories. Including FreeType like #include <FreeType/ft2build.h> will likely cause several header conflicts.

What FreeType does is load these TrueType fonts and for each glyph generates a bitmap image and calculates several metrics. We can extract these bitmap images for generating textures and position each character glyph appropriately using the loaded metrics.

To load a font, all we have to do is initialize the FreeType library and load the font as a face as FreeType likes to call it. Here we load the arial.ttf TrueType font file that was copied from the Windows/Fonts directory.

FT\_Library ft;

**if** (FT\_Init\_FreeType(&ft))

std::cout << "ERROR::FREETYPE: Could not init FreeType Library" << std::endl;

FT\_Face face;

**if** (FT\_New\_Face(ft, "fonts/arial.ttf", 0, &face))

std::cout << "ERROR::FREETYPE: Failed to load font" << std::endl;

Each of these FreeType functions returns a non-zero integer whenever an error occurred.

Once we've loaded the face, we should define the font size we'd like to extract from this face:

FT\_Set\_Pixel\_Sizes(face, 0, 48);

The function sets the font's width and height parameters. Setting the width to 0 lets the face dynamically calculate the width based on the given height.

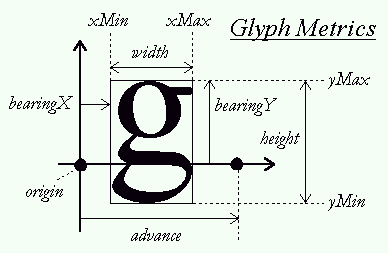
A FreeType face hosts a collection of glyphs. We can set one of those glyphs as the active glyph by calling FT\_Load\_Char. Here we choose to load the character glyph 'X':

**if** (FT\_Load\_Char(face, 'X', FT\_LOAD\_RENDER))

std::cout << "ERROR::FREETYTPE: Failed to load Glyph" << std::endl;

By setting FT\_LOAD\_RENDER as one of the loading flags, we tell FreeType to create an 8-bit grayscale bitmap image for us that we can access via face->glyph->bitmap.

Each of the glyphs we load with FreeType however do not have the same size (as we had with bitmap fonts). The bitmap image generated by FreeType is just large enough to contain the visible part of a character. For example, the bitmap image of the dot character '.' is much smaller than the bitmap image of the character 'X'. For this reason, FreeType also loads several metrics that specify how large each character should be and how to properly position them. Below is an image by FreeType that shows all of the metrics it calculates for each character glyph.



Each of the glyphs reside on a horizontal baseline (as depicted by the horizontal arrow) where some glyphs sit exactly on top of this baseline (like 'X') or some slightly below the baseline (like 'g' or 'p'). These metrics define exactly the offsets to properly position each glyph on the baseline, how large each glyph should be and how many pixels we need to advance to render the next glyph. Below is a small list of these properties we'll be needing.

* **width**: the width (in pixels) of the bitmap accessed via face->glyph->bitmap.width.
* **height**: the height (in pixels) of the bitmap accessed via face->glyph->bitmap.rows.
* **bearingX**: the horizontal bearing e.g. the horizontal position (in pixels) of the bitmap relative to the origin accessed via face->glyph->bitmap\_left.
* **bearingY**: the vertical bearing e.g. the vertical position (in pixels) of the bitmap relative to the baseline accessed via face->glyph->bitmap\_top.
* **advance**: the horizontal advance e.g. the horizontal distance (in 1/64th pixels) from the origin to the origin of the next glyph. Accessed via face->glyph->advance.x.

We could load a character glyph, retrieve its metrics and generate a texture each time we want to render a character to the screen, but it would be inefficient to do this each frame. We'd rather store the generated data somewhere in the application and query it whenever we want to render a character. We'll define a convenient struct that we'll store in a map.

**struct** Character {

**GLuint** TextureID; // ID handle of the glyph texture

glm::ivec2 Size; // Size of glyph

glm::ivec2 Bearing; // Offset from baseline to left/top of glyph

**GLuint** Advance; // Offset to advance to next glyph

};

std::map<**GLchar**, Character> Characters;

For this tutorial we'll keep things simple by restricting ourselves to the first 128 characters of the ASCII character set. For each character, we generate a texture and store its relevant data into a Character struct that we add to the Characters map. This way, all data required to render each character is stored for later use.

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1); // Disable byte-alignment restriction

**for** (**GLubyte** c = 0; c < 128; c++)

{

// Load character glyph

**if** (FT\_Load\_Char(face, c, FT\_LOAD\_RENDER))

{

std::cout << "ERROR::FREETYTPE: Failed to load Glyph" << std::endl;

**continue**;

}

// Generate texture

**GLuint** texture;

glGenTextures(1, &texture);

glBindTexture(GL\_TEXTURE\_2D, texture);

glTexImage2D(

GL\_TEXTURE\_2D,

0,

GL\_RED,

face->glyph->bitmap.width,

face->glyph->bitmap.rows,

0,

GL\_RED,

GL\_UNSIGNED\_BYTE,

face->glyph->bitmap.buffer

);

// Set texture options

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

// Now store character for later use

Character character = {

texture,

glm::ivec2(face->glyph->bitmap.width, face->glyph->bitmap.rows),

glm::ivec2(face->glyph->bitmap\_left, face->glyph->bitmap\_top),

face->glyph->advance.x

};

Characters.insert(std::pair<**GLchar**, Character>(c, character));

}

Within the for loop we list over all the 128 characters of the ASCII set and retrieve their corresponding character glyphs. For each character, we generate a texture, set its options and store its metrics. What is interesting to note here is that we use GL\_RED as the texture's internalFormat and format arguments. The bitmap generated from the glyph is a grayscale 8-bit image where each color is represented by a single byte. For this reason we'd like to store each byte of the bitmap buffer as a texture's color value. We accomplish this by creating a texture where each byte corresponds to the texture color's red component (first byte of its color vector). If we use a single byte to represent the colors of a texture we do need to take care of a restriction of OpenGL:

glPixelStorei(GL\_UNPACK\_ALIGNMENT, 1);

OpenGL requires that textures all have a 4-byte alignment e.g. their size is always a multiple of 4 bytes. Normally this won't be a problem since most textures have a width that is a multiple of 4 and/or use 4 bytes per pixel, but since we now only use a single byte per pixel they can have any possible width. By setting its unpack alignment equal to 1 we ensure there are no alignment issues (which could cause segmentation faults).

Also be sure to clear FreeType's resources once you're finished processing the glyphs:

FT\_Done\_Face(face);

FT\_Done\_FreeType(ft);

##### Shaders

To render the actual glyphs we'll be using the following vertex shader:

#version 330 core

**layout** (location = 0) **in** vec4 vertex; // <vec2 pos, vec2 tex>

**out** vec2 TexCoords;

**uniform** mat4 projection;

**void** main()

{

gl\_Position = projection \* vec4(vertex.xy, 0.0, 1.0);

TexCoords = vertex.zw;

}

We combine both the position and texture coordinate data into one vec4. The vertex shader multiplies the coordinates with a projection matrix and forwards the texture coordinates to the fragment shader:

#version 330 core

**in** vec2 TexCoords;

**out** vec4 color;

**uniform** sampler2D text;

**uniform** vec3 textColor;

**void** main()

{

vec4 sampled = vec4(1.0, 1.0, 1.0, texture(text, TexCoords).r);

color = vec4(textColor, 1.0) \* sampled;

}

###### The fragment shader

The fragment shader takes two uniforms: one is the mono-colored bitmap image of the glyph and the other is a color uniform for adjusting the text's final color. We first sample the color value of the bitmap texture. Because the texture's data is stored in just its red component, we sample the r component of the texture as the sampled alpha value. By varying the color's alpha value the resulting color will be transparent for all the glyph's background colors and non-transparent for the actual character pixels. We also multiply the RGB colors by the textColor uniform to vary the text color.

We do need to enable [blending](https://learnopengl.com/#!Advanced-OpenGL/Blending) for this to work though:

glEnable(GL\_BLEND);

glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);

For the projection matrix we'll be using an orthographic projection matrix. For rendering text we (usually) do not need perspective and using an orthographic projection matrix also allows us to specify all vertex coordinates in screen coordinates if we set it up as follows:

glm::mat4 projection = glm::ortho(0.0f, 800.0f, 0.0f, 600.0f);

We set the projection matrix's bottom parameter to 0.0f and its top parameter equal to the window's height. The result is that we specify coordinates with y values ranging from the bottom part of the screen (0.0f) to the top part of the screen (600.0f). This means that the point (0.0, 0.0) now corresponds to the bottom-left corner.

Last up is creating a VBO and VAO for rendering the quads. For now we reserve enough memory when initiating the VBO so that we can later update the VBO's memory when rendering characters.

**GLuint** VAO, VBO;

glGenVertexArrays(1, &VAO);

glGenBuffers(1, &VBO);

glBindVertexArray(VAO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferData(GL\_ARRAY\_BUFFER, **sizeof**(**GLfloat**) \* 6 \* 4, NULL, GL\_DYNAMIC\_DRAW);

glEnableVertexAttribArray(0);

glVertexAttribPointer(0, 4, GL\_FLOAT, GL\_FALSE, 4 \* **sizeof**(**GLfloat**), 0);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindVertexArray(0);

The 2D quad requires 6 vertices of 4 floats each so we reserve 6 \* 4 floats of memory. Because we'll be updating the content of the VBO's memory quite often we'll allocate the memory with GL\_DYNAMIC\_DRAW.

##### Render line of text

To render a character we extract the corresponding Character struct of the Characters map and calculate the quad's dimensions using the character's metrics. With the quad's calculated dimensions we dynamically generate a set of 6 vertices that we use to update the content of the memory managed by the VBO using glBufferSubData.

We create a function called RenderText that renders a string of characters:

**void** RenderText(Shader &s, std::string text, **GLfloat** x, **GLfloat** y, **GLfloat** scale, glm::vec3 color)

{

// Activate corresponding render state

s.Use();

glUniform3f(glGetUniformLocation(s.Program, "textColor"), color.x, color.y, color.z);

glActiveTexture(GL\_TEXTURE0);

glBindVertexArray(VAO);

// Iterate through all characters

std::string::const\_iterator c;

**for** (c = text.begin(); c != text.end(); c++)

{

Character ch = Characters[\*c];

**GLfloat** xpos = x + ch.Bearing.x \* scale;

**GLfloat** ypos = y - (ch.Size.y - ch.Bearing.y) \* scale;

**GLfloat** w = ch.Size.x \* scale;

**GLfloat** h = ch.Size.y \* scale;

// Update VBO for each character

**GLfloat** vertices[6][4] = {

{ xpos, ypos + h, 0.0, 0.0 },

{ xpos, ypos, 0.0, 1.0 },

{ xpos + w, ypos, 1.0, 1.0 },

{ xpos, ypos + h, 0.0, 0.0 },

{ xpos + w, ypos, 1.0, 1.0 },

{ xpos + w, ypos + h, 1.0, 0.0 }

};

// Render glyph texture over quad

glBindTexture(GL\_TEXTURE\_2D, ch.textureID);

// Update content of VBO memory

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferSubData(GL\_ARRAY\_BUFFER, 0, **sizeof**(vertices), vertices);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

// Render quad

glDrawArrays(GL\_TRIANGLES, 0, 6);

// Now advance cursors for next glyph (note that advance is number of 1/64 pixels)

x += (ch.Advance >> 6) \* scale; // Bitshift by 6 to get value in pixels (2^6 = 64)

}

glBindVertexArray(0);

glBindTexture(GL\_TEXTURE\_2D, 0);

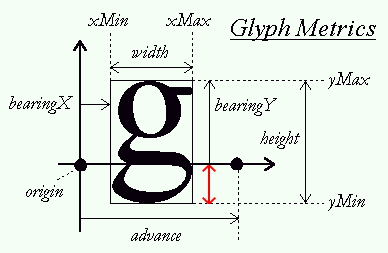
}

The content of the function should be relatively self-explanatory: we first calculate the origin position of the quad (as xpos and ypos) and the quad's size (as w and h) and generate a set of 6 vertices to form the 2D quad; note that we scale each metric by scale. We then update the content of the VBO and render the quad.

The following line of code requires some extra attention though:

**GLfloat** ypos = y - (ch.Size.y - ch.Bearing.y);

Some characters (like 'p' or 'g') are rendered slightly below the baseline so the quad should also be positioned slightly below RenderText's y value. The exact amount we need to offset ypos below the baseline can be figured out from the glyph metrics:



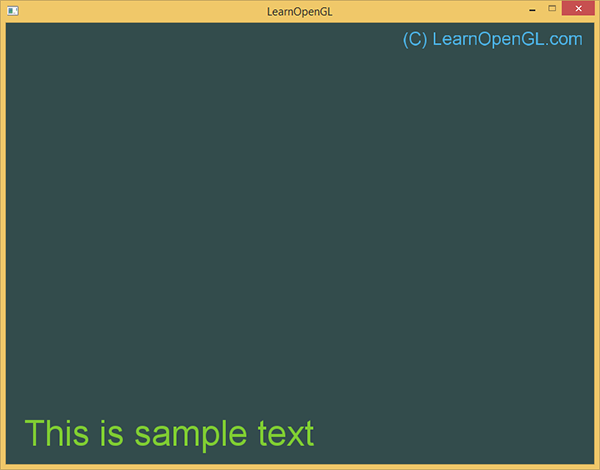
To calculate this distance e.g. offset we need to figure out the distance a glyph extends below the baseline; this distance is indicated by the red arrow. As you can see from the glyph metrics we can calculate the length of this vector by subtracting bearingY from the glyph's (bitmap) height. This value is then 0.0 for characters that rest on the baseline (like 'X') and positive for characters that reside slightly below the baseline (like 'g' or 'j').

If you did everything correct you should now be able to successfully render strings of text with the following statements:

RenderText(shader, "This is sample text", 25.0f, 25.0f, 1.0f, glm::vec3(0.5, 0.8f, 0.2f));

RenderText(shader, "(C) LearnOpenGL.com", 540.0f, 570.0f, 0.5f, glm::vec3(0.3, 0.7f, 0.9f));

This should then look similar to the following image:



You can find the code of this example [here](https://learnopengl.com/code_viewer.php?code=in-practice/text_rendering).

To give you a feel of how we calculated the quad's vertices we can disable blending to see what the actual rendered quads look like:



Here you can clearly see most quads resting on the (imaginary) baseline while the quads that corresponds to glyphs like 'p' or '(' are shifted downwards.

#### Going further

This tutorial demonstrated a text rendering technique with TrueType fonts using the FreeType library. The approach is flexible, scalable and works with many character encodings. However, this approach is likely going to be overkill for your application as we generate and render textures for each glyph. Performance-wise bitmap fonts are preferable as we only need one texture for all our glyphs. The best approach would be to combine the two approaches by dynamically generating a bitmap font texture featuring all characters glyphs as loaded with FreeType. This saves the renderer from a significant amount of texture switches and based on how tightly each glyph is packed could save quite some performance.

Another issue with FreeType fonts is that the glyph textures are stored with a fixed font size so a significant amount of scaling might be required that will introduce jagged edges. Furthermore, rotations applied to the glyphs will cause them to appear blurry. This can be mitigated by, instead of storing the actual rasterized pixel color, storing the distance to the closest glyph outline per pixel. This technique is called signed distance fields and Valve published a [paper](http://www.valvesoftware.com/publications/2007/SIGGRAPH2007_AlphaTestedMagnification.pdf) a few years ago about their implementation of this technique which works surprisingly well for 3D rendering applications.

#### Further reading

* [70+ Best Free Fonts for Designers](https://www.websiteplanet.com/blog/best-free-fonts/): summarized list of a large group of fonts to use in your project for personal or commercial use.

### GLSL

#### Texture built in function

uniform sampler2DRect tex0;

in vec2 texcoord;

....

void main() {

vec4 position = texture(tex0, texcoord);

...

}

The texture() function is accessing the texture in a given position and returning the color in normalised RGBA (this is important, the output color in GLSL ranges from 0 to 1, not 0 to 255). It’s a lookup function.

**sampler2DRect** means that the texture is in texel space. That is, it ranges from 0 to its size.

texcoord it’s the variation of that fragment.

So, it’s like using a getColor(x, y) function with pixels. Sort of.

# **OGL\_Application**

## **Pattern for Buffer Usage**

A “good” (because it works and it seems to be clean), pattern when using buffer is the following :

you have a create() function where you actually create and initialize the vertex buffer object and the vertex array object (VBO,VAO), like the following :

In the render function you keep the follwing fundamentals lines

void Triangle\_tb::render(fpcameraNS::Transformation& cam)

{

glUseProgram(shader\_prog);

static GLuint va\_location = glGetAttribLocation(shader\_prog, "triangle");

static GLuint transformMatrix\_Location = glGetUniformLocation(shader\_prog, "transformMatrix");

VAO BINDING, TEXTURE BINDING

glBindVertexArray(triangle\_vao);

glBindTexture(GL\_TEXTURE\_2D, texture);

glEnableVertexArrayAttrib(triangle\_vao, va\_location);

glUniformMatrix4fv(transformMatrix\_Location, 1, GL\_FALSE, cam.getPlayerCamera());

glDrawArrays(GL\_TRIANGLES, 0, 3);

glDisableVertexArrayAttrib(triangle\_vao, va\_location);

}

void Triangle\_tb::create() {

GLfloat x = 5.0f, y = 5.0f, z = 5.0, scale = 1.0;

static GLfloat triangle\_vertex[] =

{

x,0.0,z,scale,

-x,0.0,z,scale,

x,y,z,scale

};

BUFFER

glCreateBuffers(1, &triangle\_buffer);

//dico a openGL che sto creando un buffer per vertex data

glBindBuffer(GL\_ARRAY\_BUFFER, triangle\_buffer);

//alloco la memoria nel buffer

glNamedBufferStorage(triangle\_buffer, sizeof(triangle\_vertex), NULL, GL\_DYNAMIC\_STORAGE\_BIT);

//metto dati in buffer

glNamedBufferSubData(triangle\_buffer, 0, sizeof(triangle\_vertex), triangle\_vertex);

VERTEX SETTING

///////////SETTING UP VERTEX ARRAY OBJECT

glCreateVertexArrays(1, &triangle\_vao);

glVertexArrayAttribBinding(triangle\_vao, vai, vbbi);

glVertexArrayVertexBuffer(triangle\_vao, vbbi, triangle\_buffer, 0, sizeof(vmath::vec4));

glVertexArrayAttribFormat(triangle\_vao, vai, 4, GL\_FLOAT, GL\_FALSE, 0);

TEXTURE SETTING

/////////SETTING UP TEXTURE

glCreateTextures(GL\_TEXTURE\_2D, 1, &texture);

glTextureStorage2D(texture, 1, GL\_RGBA32F, 256, 256);

glBindTexture(GL\_TEXTURE\_2D, texture);

texture = SOIL\_load\_OGL\_texture

(

"C:/Users/aloforte/source/repos/OGL\_Application/root/OGL\_Application/DEPENDENCIES/SOIL2-master/SOIL2-master/bin/img\_test.bmp",

SOIL\_LOAD\_AUTO,

SOIL\_CREATE\_NEW\_ID,

SOIL\_FLAG\_MIPMAPS | SOIL\_FLAG\_INVERT\_Y | SOIL\_FLAG\_NTSC\_SAFE\_RGB | SOIL\_FLAG\_COMPRESS\_TO\_DXT

);

/\* check for an error during the load process \*/

if (0 == texture)

{

printf("SOIL loading error: '%s'\n", SOIL\_last\_result());

}

}

## **Pattern for including a new drawable object**

* + 1. Introduce in proper folders files : object.cpp, object.h, object.vert, object.frag
    2. Write code or easily copy-paste code from another existing drawable object

(If you copy –paste the code from another drawable object althogeter remember that class names and header guards in header must be changed or you will get error because compiler wont read your new class header).

* + 1. Add an instance of the new object in the main App class
    2. Setup the object in the app::startup method where all the drawable object are set, in particular, remeber to call the baseObject setShaders() method.
    3. Finally remember to call the baseObject render() function in the App::render() method.

# **GKM\_C++**



# **GKM\_PHYSICS**



# **GKM\_TOOLS**

