Libreria Grafica

WebGL: esempi

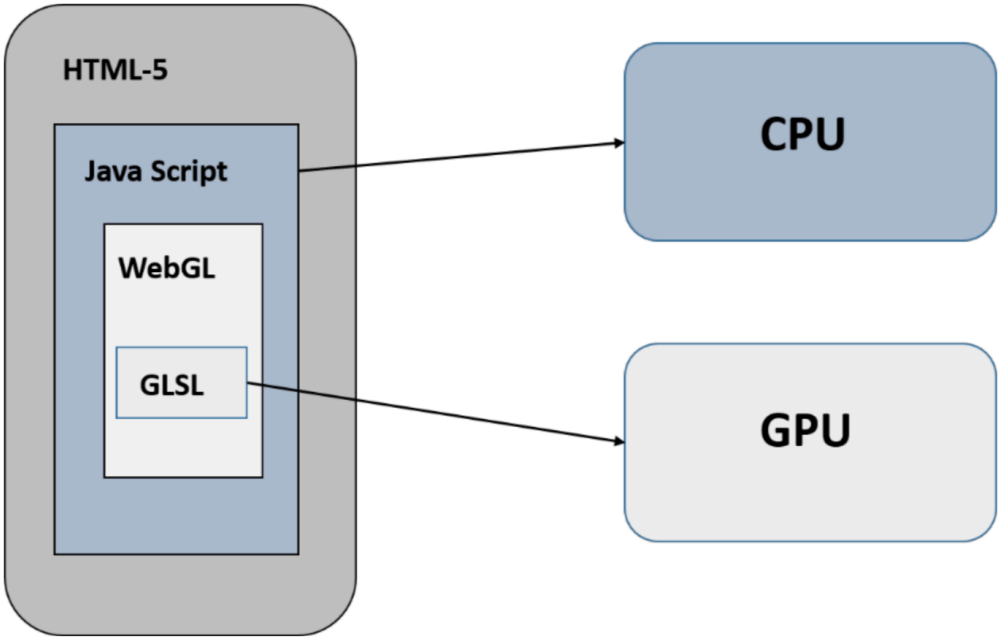
https://www.khronos.org/webgl/

Esempio di Applicazione WebGL

Vediamo insieme il seguente esempio per disegnare un triangolo

codice: HTML5_webgl_1/draw_triang_array.html

Il codice è un misto di JavaScript e OpenGL ES Shader Language:
-JavaScript serve per comunicare con la CPU
-OpenGL Shader Language serve per comunicare con la GPU



Esempio di Applicazione WebGL

Ci sono cinque step per disegnare un semplice triangolo:

Step 1: preparare l'area di disegno e ottenere il contesto di rendering WebGL. In questo passaggio, è necessario ottenere l'elemento canvas di HTML5 corrente, quindi il relativo contesto di rendering WebGL.

Step 2: definire la geometria e memorizzarla in un buffer object. In questo passaggio, prima di tutto, è necessario definire la geometria come vertici, indici, colore, ecc. e memorizzarli in array JavaScript. Successivamente, creare uno o più buffer object e passare gli array contenenti i dati al rispettivo buffer object. Nell'esempio, si sono memorizzati i vertici del triangolo in un array JavaScript e si è passato questo array a un VBO (Vertex Buffer Object).

Esempio di Applicazione WebGL

Step 3: creare e compilare i programmi Shader. In questo passaggio, è necessario scrivere i programmi vertex shader e fragment shader, compilarli e creare un unico programma collegando questi due programmi.

Step 4: associare i programmi shader ai buffer object. In questo passaggio, è necessario associare i buffer object al programma shader del passo precedente.

Step 5: disegnare l'oggetto richiesto (triangolo). Questo passaggio include operazioni come la definizione del colore di sfondo, il clear del frame buffer, l'abilitazione del test di profondità, l'impostazione della viewport di visualizzazione, ecc. Infine, è necessario richiamare le primitive di disegno utilizzando uno dei metodi: drawArrays( ) o drawElements( ).

Step 1. WebGLRenderingContext

È l'interfaccia di WebGL. Rappresenta il contesto di disegno. Questo oggetto contiene tutti i metodi per agire sul buffer di disegno.

```
canvas.getContext(contextType, contextAttributes);
```

Il parametro WebGL contextAttributes è opzionale; definisce alcuni stati dei buffer di WebGL (lo riprenderemo più avanti).

```
var canvas = document.getElementById('canvas');
var gl = canvas.getContext('webgl');
```

Gli attributi di questa interfaccia sono:

- gl questo è un riferimento all'elemento canvas che ha creato questo contesto.
- drawingBufferWidth questo attributo rappresenta la larghezza effettiva del buffer/superficie di disegno (default = 300).
- drawingBufferHeight questo attributo rappresenta l'altezza effettiva del buffer/superficie di disegno (default = 150).

Questi ultimi attributi Width e Height possono differire da quelli di HTMLCanvasElement.

Step 2. Geometria WebGL

Dopo aver ottenuto il contesto WebGL, è necessario definire la geometria per la primitiva (oggetto mesh che si desidera disegnare) e memorizzarla.

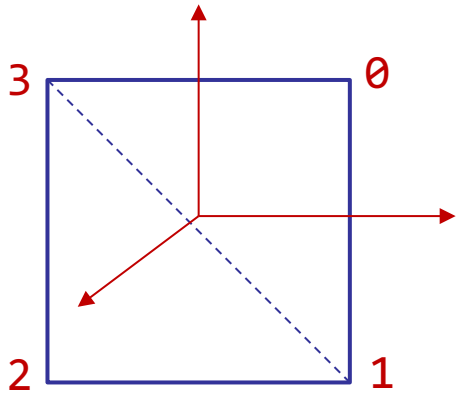
In WebGL, definiamo i dettagli di una geometria, ad esempio vertici, indici, colore della primitiva, utilizzando array JavaScript.

Per passare questi dettagli agli shader programs, dobbiamo creare dei buffer object e memorizzare gli array JavaScript contenenti i dati nei buffer object creati.

Geometria WebGL

Un oggetto mesh 3D per essere utilizzato in WebGL deve essere definito da facce di tipo poligonale composte da 3 o più vertici, ma deve essere scomposto e organizzato in facce triangolari.

```
var vertices = [ 0.5,0.5, 0.5,-0.5, -0.5,-0.5, -0.5,0.5];
var indices = [0,3,1,3,2,1];
```



Per disegnare primitive, WebGL fornisce i seguenti due metodi:

drawArrays() : fa riferimento ai vertici definiti in un array JavaScript

drawElements() : fa riferimento ai vertici definiti in un array JavaScript mediante i loro indici memorizzati in un altro array JavaScript.

Buffer Objects

Un buffer object è un'area di memoria allocata nella GPU. Nei buffer object è possibile memorizzare i dati del modello che si desidera disegnare (vertici, indici, colore, ecc.).

Utilizzando questi buffer object, è possibile passare più dati al programma vertex shader attraverso le sue variabili attributes.

Esistono due tipi di buffer object:

-Vertex buffer object (VBO): contiene i vertici (i dati sui vertici) della geometria che verrà renderizzata. I VBO servono per memorizzare ed elaborare i dati riguardanti i vertici come coordinate dei vertici, normali, colori e coordinate texture.

-Index buffer object (IBO): contiene gli indici (i dati sugli indici) della geometria che verrà renderizzata.

Buffer Objects

Per memorizzare i dati nei buffer object si deve nell'ordine:

- Creare un buffer vuoto
- Associare un oggetto array appropriato al buffer vuoto
- Passare i dati (vertici/indici) al buffer utilizzando un array tipizzato
- Disassociare il buffer (Opzionale)

Creare un Buffer Object

Per creare un buffer object vuoto, WebGL fornisce il metodo

createBuffer( )

Questo metodo restituisce un buffer object se la creazione ha avuto esito positivo; altrimenti, in caso di errore, restituisce un valore null.

WebGL funziona come una macchina a stati. Una volta creato un buffer, qualsiasi operazione successiva di buffer verrà eseguita sul buffer corrente fino a quando non lo si stacca/unbind.

```
var vertex_buffer = gl.createBuffer();
```

Nota: gl è ovviamente la variabile di riferimento al contesto WebGL corrente.

Bind di un Buffer Object

Dopo aver creato un buffer object vuoto, è necessario associargli un array buffer (destinazione). WebGL fornisce il metodo

bindBuffer( )

```
void bindBuffer(enum target, Object buffer)
```

target: è un valore che rappresenta il tipo di buffer che vogliamo associare al buffer vuoto.
Sono disponibili due valori predefiniti per questo parametro:
ARRAY_BUFFER che rappresenta i dati dei vertice.
ELEMENT_ARRAY_BUFFER che rappresenta i dati degli indici.

buffer: è la variabile di riferimento al buffer object creato. La variabile può indicare un VBO o un IBO.

```
var vertex_buffer = gl.createBuffer();       //vertex buffer
gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);
var Index_Buffer = gl.createBuffer();        //index buffer
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, Index_Buffer);
```

Passare i dati nei buffer object

Dopo aver associato l'array object appropriato al buffer, è necessario passare i dati (vertici / indici), che devono essere in un array JavaScript

```
void bufferData(enum target, Object data, enum usage)
```

target: rappresenta il tipo di array buffer che abbiamo usato. Le possibilità sono:

ARRAY_BUFFER rappresenta vertex data

ELEMENT_ARRAY_BUFFER rappresenta index data

data: è il valore oggetto che contiene i dati che devono essere scritti nel buffer object; i dati vengono passati usando typed array.

usage: specifica come utilizzare i dati memorizzati nel buffer object e da disegnare. Le possibilità sono:

gl.STATIC_DRAW: i dati vengono specificati una volta e utilizzati più volte.

gl.STREAM_DRAW: i dati verranno specificati una volta e utilizzati alcune volte.

gl.DYNAMIC_DRAW: i dati verranno specificati più volte e utilizzati più volte.

Passare i dati nei buffer object

```
//vertex buffer
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);
//Index buffer
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,new Uint16Array(indices), gl.STATIC_DRAW);
```

Typed Arrays

WebGL fornisce un tipo speciale di array chiamato typed array per trasferire i dati sui vertici, indici e texture. Gli array tipizzati utilizzati da WebGL sono:
Int8Array, Uint8Array, Int16Array, Uint16Array, Int32Array, UInt32Array, Float32Array e Float64Array.

Nota: In generale, per la memorizzazione dei dati dei vertici, si utilizza Float32Array; e per memorizzare i dati indice, si usa Uint16Array.

È possibile creare array tipizzati proprio come gli array JavaScript utilizzando la parola chiave new.

Unbind dei buffer

Si suggerisce di disassociare i buffer dopo averli utilizzati. Basta passare un valore null al posto del buffer object,

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, null);
```

WebGL fornisce i seguenti metodi per eseguire operazioni sui buffer:

void bindBuffer(enum target, Object buffer)
target: ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER

void bufferData(enum target, long size, enum usage)
target: ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER
usage: STATIC_DRAW, STREAM_DRAW, DYNAMIC_DRAW

void bufferData(enum target, Object data, enum usage)
target and usage: come prima

Unbind dei buffer

void bufferSubData(enum target, long offset, Object data)
target: ARRAY_BUFFER, ELEMENT_ARRAY_BUFFER

Object createBuffer()

void deleteBuffer(Object buffer)

any getBufferParameter(enum target, enum pname)
target: ARRAY_BUFFER, ELEMENT_ ARRAY_BUFFER
pname: BUFFER_SIZE, BUFFER_USAGE

bool isBuffer(Object buffer)

Step 3. Shaders

Gli shader sono i programmi eseguiti su GPU.

Gli shader sono scritti in OpenGL ES Shader Language (noto come ES SL), che è una combinazione di due linguaggi strettamente correlati che vengono utilizzati per creare i programmi vertex shader e fragment shader.

ES SL ha variabili proprie, tipi di dati, qualificatori, input e output integrati.

Shaders

Data Types

La tabella seguente elenca i tipi di dati di base forniti da OpenGL ES SL:

void	rappresenta un valore “vuoto”
bool	può essere true o false
int	tipo di dato intero con segno
float	tipo di dato scalare floating point
vec2, vec3, vec4	point vector floating point di lunghezza n=2,3,4
bvec2, bvec3, bvec4	boolean vector
ivec2, ivec3, ivec4	signed integer vector
mat2, mat3, mat4.	2x2, 3x3, 4x4 floating point matrix
sampler2D	puntatore ad un 2D texture
samplerCube	puntatore ad un cube mapped texture

Shaders e Qualifiers

Esistono tre principali qualificatori in OpenGL ES SL:

attribute Questo qualificatore funge da collegamento tra un vertex shader e OpenGL ES per i dati per vertice. Il valore di questo attributo cambia per ogni esecuzione del vertex shader.

uniform Questo qualificatore collega i programmi shader e l'applicazione WebGL. A differenza del qualificatore attribute, i valori uniforms non cambiano. Gli uniforms sono di sola lettura; si possono usare con qualsiasi tipo di dato, per dichiarare una variabile. Esempio: uniform vec4 lightPosition;

varying Questo qualificatore costituisce un collegamento tra un vertex shader e un fragment shader per dati interpolati. Può essere utilizzato con i seguenti tipi di dati: float, vec2, vec3, vec4, mat2, mat3, mat4 o array.

Vertex Shader

Vertex shader è un programma chiamato su ogni vertice. Applica delle trasformazioni alla geometria.

Gestisce i dati di ciascun vertice (dati per vertice) come le coordinate del vertice, le normali, i colori e le coordinate texture. Nel codice ES SL del vertex shader, i programmatori devono definire gli attribute per gestire i dati. Questi attributi puntano a un vertex buffer object.

Il vertex shader permette:

- Trasformazioni sui vertici
- Trasformazioni e normalizzazioni normali
- Generazione delle coordinate texture
- Trasformazione delle coordinate texture
- Illuminazione
- Applicazione di materiali

Vertex Shader

Variabili predefinite

OpenGL ES SL fornisce le seguenti variabili predefinite per il vertex shader:

highp vec4 gl_Position; Mantiene la posizione del vertice.

mediump float gl_PointSize; Mantiene le dimensioni del punto trasformato. Le unità per questa variabile sono pixel.

Esempio di vertex shader:

```
attribute vec2 coordinates;
   void main(void) {
       gl_Position = vec4(coordinates, 0.0, 1.0);
   };
```

Vertex Shader

Se si osserva attentamente il codice draw_triang_array.html, si è dichiarata una variabile attribute con il nome coordinates. Questa variabile viene associata al Vertex Buffer Object utilizzando il metodo getAttribLocation().

L'attribute coordinates viene passato come parametro a questo metodo insieme all'oggetto shader program.

Poi viene definita la variabile gl_position.
gl_position è una variabile predefinita disponibile solo nel vertex shader. Contiene la posizione del vertice.

Nel codice, l'attribute coordinates viene passato sotto forma di un array. Poiché il vertex shader è un'operazione per vertice, il valore gl_position viene calcolato per ciascun vertice.

continua

Vertex Shader

Al termine dell'elaborazione, il valore gl_position viene utilizzato nell'assemblaggio delle primitive utilizzate, nel clipping, nel culling e nelle altre operazioni a funzionalità fissa.

Si possono scrivere programmi vertex shader per tutte le possibili operazioni di vertex shader.

Fragment Shader

Una mesh è formata da più triangoli e la superficie di ciascun triangolo è conosciuta come fragment.

Un fragment shader è il codice che viene eseguito su ogni pixel di ciascun fragment. Questo viene scritto per calcolare e disegnare con un colore ogni singolo pixel.

Il fragment shader permette:

- Operazioni su valori interpolati
- Accesso alle texture
- Applicazione di texture
- Nebbia
- somma colori

Fragment Shader

Variabili predefinite

mediump vec4 gl_FragCoord;	fragment position nel frame buffer.
bool gl_FrontFacing;	fragment del front-facing primitive.
mediump vec2 gl_PointCoord;	fragment position di un punto (point rasterization only).
mediump vec4 gl_FragColor;	valore del fragment color mediump
vec4 gl_FragData[n]	fragment color per il colore relativo ad n.

Codice di esempio di un fragment shader per applicare un colore a ogni pixel di un triangolo.

```
void main(void) {
   gl_FragColor = vec4(0, 0.8, 0, 1);
}
```

Il fragment shader passa l'output alla pipeline utilizzando variabili globali; gl_FragColor è una di queste ed è il colore da utilizzare per il pixel

Storing and Compiling Shader Programs

Poiché gli shader sono programmi indipendenti, possiamo scriverli come script separati e utilizzarli in una applicazione. In alternativa, è possibile memorizzarli direttamente nel formato stringa, come:

```
var vertCode =
    'attribute vec2 coordinates;' +
    'void main(void) {' +
    ' gl_Position = vec4(coordinates, 0.0, 1.0);' +
'}';
```

Compilare lo shader

Dopo aver scritto il codice, si deve compilare il programma; tre passi:

- Creazione dell'oggetto shader
- Collegamento del codice sorgente all'oggetto shader creato
- Compilazione dello shader

Creating Shader programs

Per creare uno shader vuoto, WebGL fornisce un metodo chiamato createShader(). Crea e restituisce l'oggetto shader. La sua sintassi è la seguente:

```
Object createShader(enum type)
```

Questo metodo accetta un valore enum predefinito come parametro. Si hanno due opzioni per questo:

- gl.VERTEX_SHADER per la creazione di vertex shader
- gl.FRAGMENT_SHADER per la creazione di fragmente shader

Creating Shader programs

È possibile collegare il codice sorgente all'oggetto shader creato utilizzando il metodo shaderSource(). La sintassi è:

```
void shaderSource(Object shader, string source)
```

Questo metodo accetta due parametri:

- shader: è necessario passare l'oggetto shader creato come un parametro.
- source: è necessario passare il codice del programma shader in formato stringa.

Creating Shader programs

Per compilare il programma, è necessario utilizzare il metodo compileShader(). La sintassi è:

```
compileShader(Object shader)
```

Questo metodo accetta l'oggetto shader program come parametro. Dopo aver creato un oggetto shader program, si collega ad esso il codice sorgente e si passa l'oggetto a questo metodo.

Il pezzo di codice seguente mostra come creare e compilare un vertex shader e uno fragment shader per disegnare un triangolo.

Creating Vertex e Fragment Shaders

```
// Vertex Shader
var vertCode =
     'attribute vec3 coordinates;' + 'void main(void) {' +
     ' gl_Position = vec4(coordinates, 1.0);' +
'}';
var vertShader = gl.createShader(gl.VERTEX_SHADER);
gl.shaderSource(vertShader, vertCode);
gl.compileShader(vertShader);

// Fragment Shader
var fragCode =
    'void main(void) {' +
' gl_FragColor = vec4(0, 0.8, 0, 1);' +
 '}';
var fragShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragShader, fragCode);
gl.compileShader(fragShader);
```

Combined Program

Dopo aver creato e compilato entrambi i programmi shader, è necessario creare un unico programma che li contiene entrambi. I passi per farlo sono:

- Creare un programma oggetto
- Collegare entrambi gli shader
- Linkare entrambi gli shader
- Utilizzare il programma

Si creare un programma oggetto utilizzando il metodo createProgram( ); questi restituirà un programma oggetto vuoto. La sintassi è:

```
createProgram();
```

Attach e Link Shaders

Gli shader si collegano al programma oggetto creato usando il metodo attachShader(). La sintassi è:

```
attachShader(Object program, Object shader);
```

-program passa il programma oggetto vuoto creato come un parametro

-shader passa uno dei programmi di shader compilati (vertex shader, fragment shader)

Nota: è necessario collegare entrambi gli shader

Linkare gli shader utilizzando il metodo linkProgram(), passando il programma oggetto a cui si sono collegati gli shader.

```
linkProgram(shaderProgram);
```

Use the Program

WebGL fornisce un metodo chiamato useProgram(). Si deve passare il programma linkato. La sintassi è:

```
useProgram(shaderProgram);
```

Il seguente esempio di codice mostra come creare, attaccare, linkare e utilizzare uno shader combinato in unico programma:

```
var shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertShader);
gl.attachShader(shaderProgram, fragShader);
gl.linkProgram(shaderProgram);
gl.useProgram(shaderProgram);
```

Step 4. Associare Attribute e Buffer Object

Ogni attribute nel programma vertex shader punta a un vertex buffer object (VBO). Dopo aver creato i VBO, si devono associare agli attribute del programma vertex shader.

Ogni attribute punta a un (solo) VBO da cui si estraggono i valori dei dati, e questi attribute vengono passati allo shader program.

Per associare i VBO agli attribute del programma vertex shader, è necessario:

- Ottenere la posizione dell'attribute
- Puntare l'attribute su un VBO
- Abilitare l'attribute

Associare Attribute e Buffer Object

WebGL fornisce un metodo chiamato getAttribLocation( ) che restituisce la posizione dell'attribute.

```
ulong getAttribLocation(Object program, string name)
```

Questo metodo accetta l'oggetto vertex shader e i valori degli attribute del programma vertex shader.

```
var coordinatesVar = gl.getAttribLocation(shader_Program,
"coordinates");
```

shader_Program è il nome del programma vertex shader
coordinates è l'attribute del programma vertex shader

Associare Attribute e Buffer Object

Per assegnare il VBO alla variabile attribute, WebGL fornisce un metodo chiamato vertexAttribPointer ().

```
void vertexAttribPointer(location, int size, enum type, bool normalized, long stride, long offset)
```

Questo metodo accetta sei parametri.

- location: specifica la posizione di memoria di una variabile attribute. Si deve passare il valore restituito dal metodo getAttribLocation( ).
- size: specifica il numero di componenti per vertice nel VBO
- type: specifica il tipo di dati.
- normalized: questo è un valore booleano. Se true, i dati non-floating point sono normalizzati in [0, 1]; altrimenti, normalizzati in [-1, 1].
- stride: specifica il numero di byte tra i diversi elementi di dati del vertice o zero per il passo predefinito.
- offset: specifica l'offset (in byte) in un VBO per indicare da quale byte vengono memorizzati i dati del vertice. Se i dati sono memorizzati dall'inizio, l'offset è 0.

Associare Attribute e Buffer Object

Il seguente pezzo di codice mostra come usare vertexAttribPointer( ) in un programma:

```
gl.vertexAttribPointer(coordinatesVar, 3, gl.FLOAT, false, 0, 0);
```

Per accedere al VBO bisogna attivare l'attribute del vertex shader

```
enableVertexAttribArray()
```

Questo metodo accetta la posizione dell'attribute come parametro. Vediamo un esempio di utilizzo:

```
gl.enableVertexAttribArray(coordinatesVar);
```

Step 5. Drawing a Model

Dopo aver associato i buffer agli shader, si può disegnare. WebGL fornisce due metodi: drawArrays( ) e drawElements( )

drawArrays( ) è il metodo utilizzato per disegnare modelli mediante vertici:

```
void drawArrays(enum mode, int first, long count)
```

Questo metodo accetta i seguenti tre parametri:
-mode: si deve scegliere uno dei tipi primitivi forniti da WebGL:
gl.POINTS, gl.LINE_STRIP, gl.LINE_LOOP, gl.LINES,
gl.TRIANGLE_STRIP, gl.TRIANGLE_FAN e gl.TRIANGLES.

-first: questa opzione specifica l'elemento iniziale nell'array abilitato. Non può essere un valore negativo

-count: questa opzione specifica il numero di elementi da disegnare.

Drawing a Model

Se si disegna un modello utilizzando il metodo drawArrays(), WebGL, durante il rendering, considera la geometria nell'ordine in cui sono le coordinate del vertice definito.
Se per esempio si vuole disegnare un singolo triangolo, allora si devono passare i tre vertici e chiamare il metodo drawArrays (), come segue.

```
var vertices = [-0.5,-0.5, -0.25,0.5, 0.0,-0.5,];
gl.drawArrays(gl.TRIANGLES, 0, 3) ;
```

Supponiamo di voler disegnare due triangoli contigui, allora si devono passare i successivi tre vertici in ordine nel VBO e dire il numero di elementi che devono essere resi, cioè 6.

```
var vertices = [-0.5,-0.5, -0.25,0.5, 0.0,-0.5, 0.0,-0.5,
       0.25,0.5, 0.5,-0.5,];
gl.drawArrays(gl.TRIANGLES, 0, 6);
```

Drawing a Model

drawElements( ) è il metodo utilizzato per disegnare modelli utilizzando vertici e indici. La sua sintassi è la seguente:

```
void drawElements(enum mode, long count, enum type, long offset)
```

Questo metodo prende i seguenti 4 parametri:

-mode: si deve scegliere uno dei tipi primitivi forniti da WebGL: gl.POINTS, gl.LINE_STRIP, gl.LINE_LOOP, gl.LINES, gl.TRIANGLE_STRIP, gl.TRIANGLE_FAN e gl.TRIANGLES.

-count: questa opzione specifica il numero di elementi da renderizzare.

-type: questa opzione specifica il tipo di dati degli indici che devono essere UNSIGNED_BYTE o UNSIGNED_SHORT.

-offset: questa opzione specifica il punto di partenza per il rendering. Di solito è il primo elemento (0).

Drawing a Model

Se si disegna un modello utilizzando il metodo drawElements (), allora l'IBO dovrebbe essere creato insieme al VBO. Se si usa questo metodo, i dati dei vertice verranno elaborati una sola volta e utilizzati tutte le volte che ci si riferisce agli indici.

Se per esempio si vuole disegnare un singolo triangolo usando gli indici, si devono passare gli indici insieme ai vertici e chiamare il metodo drawElements( )

```
var vertices =[ -0.5,-0.5,0.0, -0.25,0.5,0.0, 0.0,-0.5,0.0 ];
var indices =[0,1,2];
gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_SHORT,0);
```

Required Operations

Prima di disegnare una primitiva, si devono fare alcune operazioni:

Clear the Canvas
Prima di tutto, si deve cancellare il Canvas, usando il metodo clearColor(). Si possono passare i valori RGBA di un colore.

Quindi WebGL cancella il canvas e lo riempie con il colore specificato. E' possibile utilizzare questo metodo per impostare il colore di sfondo.

```
gl.clearColor(0.5, 0.5, .5, 1);
```

Enable Depth Test
Abilita il Depth test utilizzando il metodo enable( ),

```
gl.enable(gl.DEPTH_TEST);
```

Required Operations

Clear Color Buffer Bit

Cancella sia il frame buffer che il depth buffer usando il metodo clear(),

```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

Set ViewPort

La viewport è l'area rettangolare che contiene i risultati del rendering del buffer di disegno. È possibile impostare le dimensioni della viewport utilizzando il metodo viewport( ).

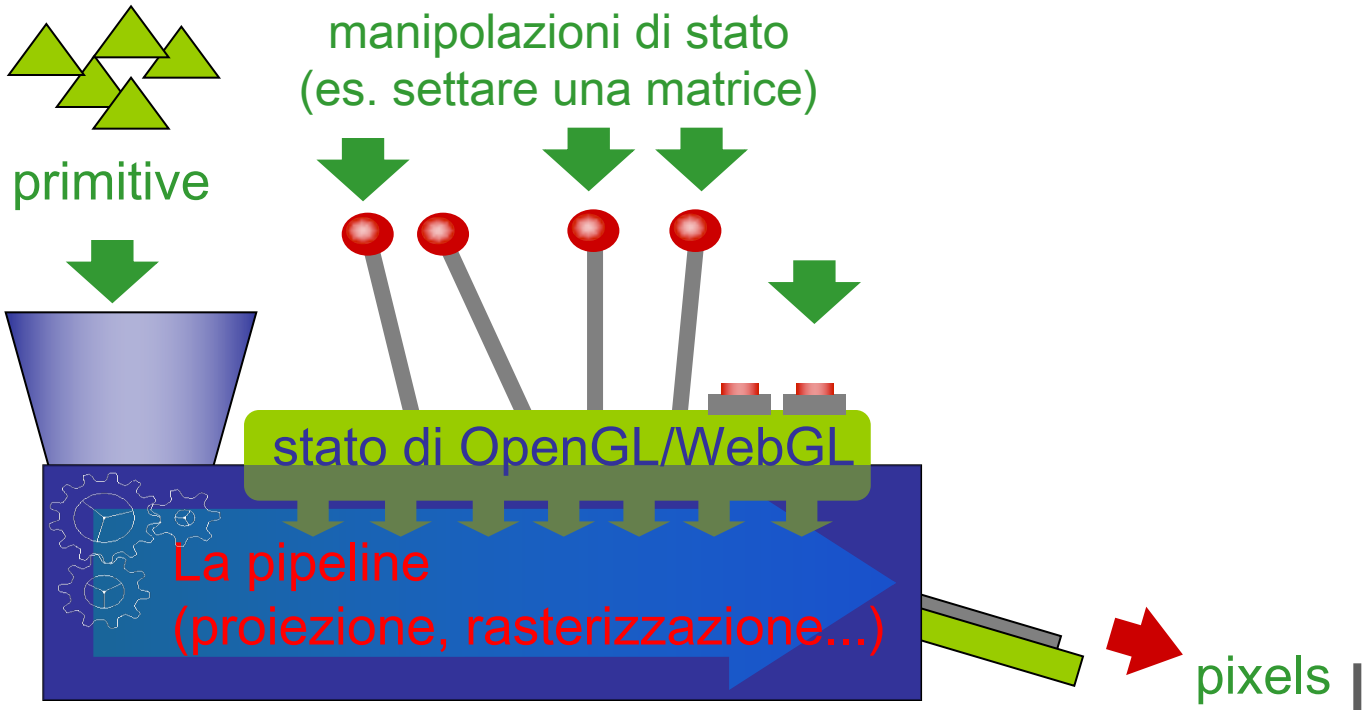
```
gl.viewport(0,0,canvas.width,canvas.height);
```

Nella chiamata si impostano le dimensioni della viewport come quelle dell'elemento canvas.

Una macchina a Stati

WebGL = macchina a STATI FINITI

l'input dall'applicazione altera gli stati e induce la macchina a produrre un output visibile.



WebGL pipeline

Vertici & loro attributi

computazioni per vertice

Vertici proiett & attributi computati

set-up

rasterizer punti

set-up

rasterizer triangoli

set-up

rasterizer segmenti

Frammenti & attributi interpolati

computazioni per frammento

pixel finali (nello screen-buffer)

associamo degli ***attribute*** ai vertici
es: colore RGB

gli attribute possono subire varie computazioni

gli attribute vengono interpolati

ogni fragment avrà una valore interpolato degli attribute per vertice

Esercitiamoci insieme

Si esamini ed esegua il codice draw_triangle_element.html.
A partire da questo codice, cioè utilizzando la
gl.drawElements(gl.TRIANGLES, indices.length, gl.UNSIGNED_SHORT,0);
lo si modifichi per:

1. disegnare il quadrato di vertici: [-0.5,0.5,0.0, -0.5,-0.5,0.0, 0.5,-0.5,0.0, 0.5,0.5,0.0];
2. disegnare lo stesso quadrato, ma con i seguenti colori per vertice: [0,0,1, 1,0,0, 0,1,0, 1,0,1];
3. disegnare il triangolo di vertici [-1,-1,-1, 0,-1,-1, 0,0,-1] e il suo traslato di [0.5,0.5,0.0];
4. disegnare il triangolo di vertici [-0.5,0.5,0.0, -0.5,-0.5,0.0, 0.5,0.5, 0.0] scalato di [0.5,1.5,1.0] utilizzando un matrice 4x4
5. disegnare il triangolo al punto 4. con un colore per ogni vertice e che ruoti intorno all'asse z (si usi matrice 4x4) per dar luogo ad una animazione; usare la window.requestAnimationFrame( );

Soluzioni Esercizi

Nella cartella HTML5_webgl_1 si possono trovare le soluzioni agli esercizi proposti:

1. draw_square_element.html
2. draw_square_element_color.html
3. draw_triangle_element_translate.html
4. draw_triang_element_scale.html
5. draw_triang_element_rot.html

Sitografia

Nella pagina Web del corso si consultino i siti WebGL:

https://www.khronos.org/webgl/ (sito ufficiale)

https://webglfundamentals.org/

https://webgl2fundamentals.org/

http://webglsamples.org/

https://experiments.withgoogle.com/search?q=WebGL

https://beginningwebgl.com/examples

https://www.tutorialspoint.com/webgl/

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Giulio Casciola

Dip. di Matematica

giulio.casciola at unibo.it