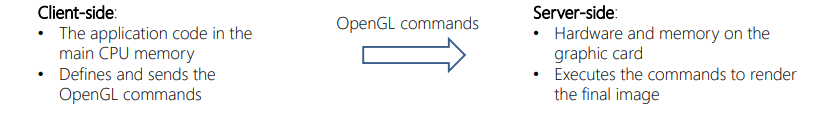
**WEBGL + GLSL 2D AND 3D GRAPHICS  
Manuale introduttivo**

OpenGL è una API per interfacciarsi con l’hardware dedicato al rendering della grafica 2D e 3D. Essendo una API non è un linguaggio di programmazione in sé; un programma ci si può appoggiare per definire o manipolare gli oggetti, applicare textures e luci, spostare oggetti all’interno di un ambiente tridimensionale e in generale utilizzare le funzioni da essa fornite.

A differenza di **Microsoft DirectX**, OpenGL è disponibile per una grande varietà di dispositivi, che comprendono smartphone, tablet e supercomputer. Questo la rende portatile, e addirittura può essere possibile implementarne le funzionalità anche senza un hardware grafico dedicato, ma appoggiandosi solamente alla CPU.

**Architettura OpenGL**

**OpenGL** è implementata come un sistema client-server:



Il suo funzionamento è quello di una grande **macchina a stati:**

* Una collezione di variabili definisce come OpenGL deve operare;
* Lo stato viene chiamato **OpenGL context;**
* Lo stato si modifica settando opzioni, manipolando buffers e infine renderizzando utilizzando il contesto corrente;
* Quando uno dei valori dello stato viene impostato, rimane tale finché altre funzioni non lo modoficano;
* **Il valore dello stato influenzerà tutti i passi successivi finché non viene modificato.**

La specifica di OpenGL definisce un’interfaccia **indipendente dall’hardware.** Questo significa che *non offre comandi per gestire l’input dell’utente* e *non offre comandi per visualizzare, mostrare e gestire le finestre.*

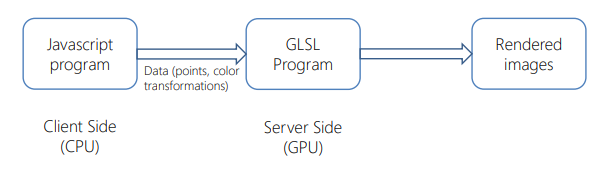
Quindi OpenGL ha bisogno di un sistema di finestre per effettuare il rendering sullo schermo: tale sistema di finestre dev’essere fornito dal sistema operativo della macchina su cui si vuole utilizzare OpenGL. Ogni sistema operativo a tal proposito espone APIs diverse. Esiste tuttavia un modo relativamente indipendente dal sistema operativo per effettuare il rendering di grafica con OpenGL, ed è l’utilizzo di un **web browser** come finestra all’interno della quale rappresentare ambienti 2D e 3D. Questo è reso possibile da **WebGL.**

**WEBGL: OpenGL-style rendering within web browsers**

WebGL è una API basata su OpenGL Embedded Systems (OpenGL ES) con l’obiettivo di appoggiarsi ai browser web per renderizzare grafica 3D. Essendo basata su una versione ridotta di OpenGL non ne implementa tutte le funzionalità, ma un sottoinsieme ridotto. Le funzionalità di OpenGL ES vengono chiamate usando il Javascript, mentre si usa HTML5 <canvas> e il WebGLRenderingContext per effettuare il rendering.

**WebGL rendering pipeline**

In WebGL è necessario usare il **GLSL** per specificare le operazioni di rendering. Esso è un linguaggio di programmazione simile al C/C++ che definisce come la GPU gestisce i dati e come l’immagine finale da renderizzare dev’essere calcolata. GLSL dice alla GPU come utilizzare i dati ricevuti dalla CPU, al fine di produrre l’immagine da renderizzare. Tale linguaggio segue le specifiche definite da OpenGL ES.

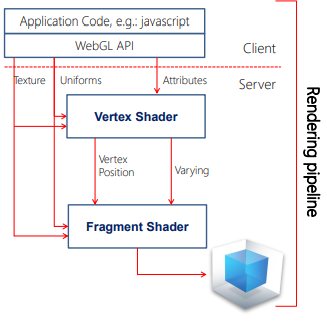


Quello che nello schema è chiamato GLSL Program è composto da due parti: **Vertex Shader** e **Fragment Shader**. Il Vertex Shader definisce la posizione delle primitive in 3D nel sistema di coordinate spaziali di WebGL e altri attributi dei vertici dei poligoni. Il Fragment Shader definisce invece il colore dei pixel dove le primitive sono renderizzate. Il Vertex quindi gestisce la struttura del modello, mentre il Fragment ne gestisce i colori.

La **rendering pipeline** in WebGL è quindi una sequenza di passi che vanno dalla definizione della scena in 3 dimensioni all’output finale in 2 dimensioni che viene visualizzato sullo schermo. Gli **shaders,** in generale, sono i programmi eseguiti dall’hardware grafico per:

* Processare i vertici
* Definire come rasterizzare le primitive
* Definire il colore di ogni pixel.

Per scrivere uno shader, WebGL si appoggia sempre al GLSL Shading Language. Vediamo la struttura della rendering pipeline:



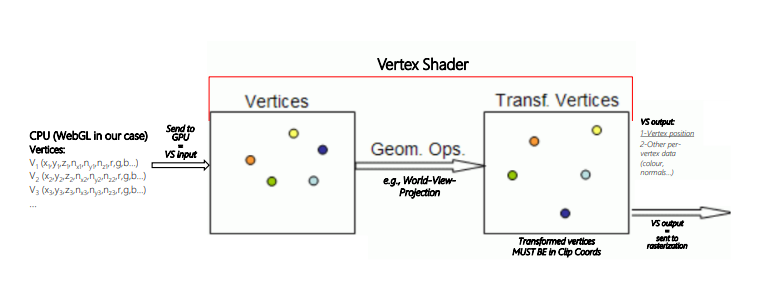
Le linee rosse rappresentano i dati che vengono scambiati tra i due shader.

**Vertex Shader**

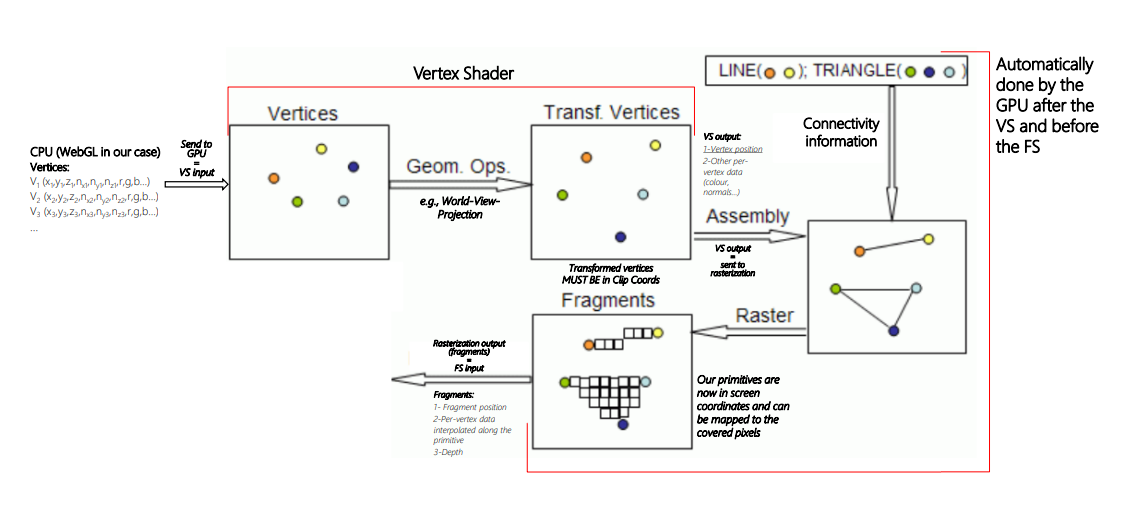
Essi contengono il codice sorgente per le operazioni che devono avvenire **su ogni vertice** che viene processato. Le operazioni per singolo vertice tipicamente includono:

* World-View Projection computation;
* Vertex colour computation;
* Light colour (solamente se si usa il Gouraud method)

Per il rendering di un triangolo con tre vertici, ad esempio, il vertex shader viene eseguito per tre volte, una per ogni vertice.



Dopo essere stati processati dal Vertex Shader, i vertici vengono assemblati a seconda della modalità scelta dal comando per disegnare (triangle, points, lines, line strips…). Qui vengono applicate funzioni come il **clipping** delle primitive, la **perspective divide** per le Normalized Device Coordinates, e la **Viewport transform**, invisibile al client. Infine, l’immagine viene **rasterizzata,** ovvero ogni primitiva viene convertita in un’immagine 2D. Ogni punto di questa immagine contiene informazioni come il **colore** e la **profondità**.



Dopo questo processo, le informazioni sono pronte per essere elaborate dal Fragment Shader.

Un vertex shader possiede la variabile **vec4 gl\_Position**, la variabile in cui salvare il risultato della computazione della posizione dei vertici della primitiva. Per esempio, se hai una variabile che contiene una matrice di trasformazione, sai che il tuo shader viene applicato ad ogni vertice. Quindi, applichi la matrice di trasformazione all’input del vertex shader e salvi il risultato in gl\_Position per ottenere il vertice trasformato.

**Fragment Shaders**

Un fragment shader contiene il codice sorgente per le operazioni che devono essere svolte su ogni singolo fragment, ovvero un quadratino di un’immagine 2D su una griglia che ha determinati parametri di profondità e colore. Alcune operazioni possono essere svolte in automatico, basta abilitarle, come per esempio:

* Anti aliasing
* Depth test
* Color blending
* Dithering

Mentre altre invece devono essere programmate esplicitamente, come per esempio:

* Texturing
* Colors and Light computation (se si usa il Phong model).

Alla fine di questa pipeline, l’immagine è completa e pronta per essere mostrata sullo schermo. L’**output del FS è l’immagine renderizzata finale.**

Un FS possiede la variabile **vec4 gl\_FragColor,** che contiene il colore finale del singolo fragment in RGBA.

Come si possono passare i dati attraverso la pipeline? Ci sono vari modi a seconda della funzione del dato che viene passato: attributi, uniformi, textures e varying.

**Attributes**

Gli attributes sono **valori che cambiano per ogni vertice.** Un esempio lampante sono le coordinate (x,y,z) del vertice, ma attributi più complessi possono essere definiti. I dati che noi decidiamo di includere per descrivere un vertice determinano il **vertex format:**

Immagine che contiene testo

Descrizione generata automaticamente

Essi possono essere passati solamente al vertex shader.

**Uniforms**

Le uniforms sono variabili tipiche del programma che **rimangono costanti durante l’esecuzione.** Di solito sono le transfomation matrices. Esse possono essere passate sia al vertex che al fragment shader. Una tipologia particolare di uniforms sono le textures.

**Textures**

Le textures sono categorie particolari di uniforms che sono utilizzate per identificare il texture object che viene utilizzato per ogni texture lookup.

**Varyings**

Le varyings sono variabili che contengono i risultati dell’esecuzione del vertex shader da essere utilizzati più avanti nella pipeline. Questi valori di solito sono poi interpolati per il rendering della primitiva. Se vuoi passare una particolare informazione da un Vertex Shader ad un Fragment Shader puoi utilizzare una variabile di questo tipo.

**Integrazione con Javascript e HTML**

Ipotizziamo di avere già a disposizione il programma GLSL. Come avviene l’integrazione con HTML e il Javascript?

Immagine che contiene testo

Descrizione generata automaticamente

WebGL sfrutta l’elemento <canvas> fornito da HTML. Esso permette di renderizzare elementi grafici all’interno del browser grazie al supporto di alcune librerie (tra cui ovviamente WebGL). L’area all’interno di un elemento canvas può essere manipolata tramite il Javascript.

Il canvas inoltre fornisce il **frameBuffer,** una regione della memoria fisica nella GPU che viene usata per conservare temporaneamente un’immagine per il rendering.

Una volta dichiarato un canvas in HTML, la prima cosa che un programma deve creare al fine di produrre grafica è un **context.** Esso:

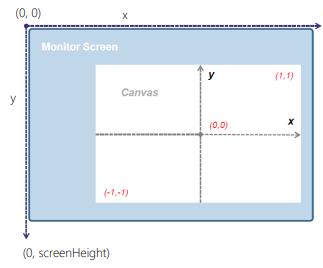
* Fornisce una struttura dati interna per tenere traccia delle impostazioni dello stato e delle operazioni
* Può essere richiesto chiamando la funzione .getContext sul canvas

Il contesto espone l’API che vogliamo utilizzare per disegnare sul canvas. Tramite esso quindi possiamo accedere alle funzioni di WebGL che ci servono.

**WebGL – sistema di coordinate**

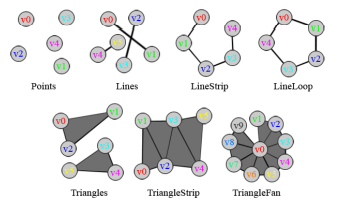
Le coordinate del drawing space in WebGL iniziano nel bel mezzo del canvas e si estendono nelle quattro direzioni definite dagli assi cartesiani. Tutti i punti definiti oltre il range [-1,1] non vengono disegnati, anche se la loro definizione è comunque possibile.

Le coordinate dello schermo sono invece gestite dal sistema operativo, e il punto 0,0 si posiziona in alto a sinistra. Tutti i valori sullo schermo sono quindi variazioni positive di x e y.



**WebGL – primitive**

Una primitiva in WebGL non è altro che una collezione di **vertici**, che possono essere assemblati in diversi modi. Tutte le primitive in WebGL possono essere assemblate come:



* POINTS
* LINES
* LINE\_STRIP
* LINE\_LOOP
* TRIANGLES
* TRIANGLE\_STRIP
* TRIANGLE\_FAN

Tutto ciò che WebGL fa è visualizzare le primitive secondo la modalità scelta e applicarci colori e textures. Queste informazioni devono essergli passate dalla CPU, che in questo caso esegue codice Javascript.

**Vertex Buffer Objects (VBO) e Vertex Array Objects (VAO)**

Prima di disegnare qualsiasi cosa sullo schermo, quindi, è necessario passare tutti i dati relativi ai vertici dal programma Javascript al programma GLSL.

Abbiamo già visto come i dati relativi ad un vertice, come la sua posizione, sono trattati da GLSL all’interno degli **attributes.** Per passare degli attributi al GLSL il JavaScript si appoggia ai **Vertex Buffer Objects (VBO).** Essi conterranno le posizioni, le normali, le coordinate UV per le textures ecce cc…

Vediamo come si dichiara un VBO e come vi si inseriscono i dati necessari in Javascript.

Immagine che contiene testo

Descrizione generata automaticamente

Ora abbiamo semplicemente ottenuto l’accesso al contesto di WebGL. Cominciamo con la definizione delle posizioni dei vertici e con la creazione di un buffer per conservarli.

Immagine che contiene testo

Descrizione generata automaticamente

Con queste istruzioni abbiamo creato il buffer e lo abbiamo impostato come **buffer attivo** indicando, tramite gl.ARRAY\_BUFFER, che esso contiene dei vertici. Con l’ultima linea di codice invece le posizioni dei vertici vengono effettivamente inserite all’interno del buffer. WebGL implicitamente usa il buffer che è bindato al momento come buffer di input da cui leggere i dati.

Ipotizziamo ora che il codice GLSL che vogliamo utilizzare prenda come input una variabile chiamata vec4 a\_position, dove ci sono le posizioni dei vertici delle primitive. Con le prossime istruzioni vedremo come collegare il buffer appena creato con tale variabile, in modo tale da avere a disposizione i dati relativi ai vertici all’interno degli shader in GLSL.

Immagine che contiene testo

Descrizione generata automaticamente

Con la prima delle due istruzioni abbiamo recuperato la posizione nella memoria della GPU dove il programma GLSL si aspetta di trovare i vertici, e con la seconda istruzione abbiamo attivato tale locazione di memoria. Adesso dobbiamo passare altre informazioni al programma GLSL, in particolare dobbiamo dirgli come interpretare i vertici.

Immagine che contiene testo

Descrizione generata automaticamente

Con l’ultimo passaggio abbiamo collegato il VBO con dentro i vertici alla posizione di memoria nella GPU che deve contenere tali valori (attributo), passando inoltre altre informazioni fondamentali sulla gestione di tali dati. Ci manca solo da definire il tipo di primitiva da usare per collegare i vertici, e passare ulteriori informazioni come il numero di vertici da disegnare e da quale elemento partire:

Immagine che contiene testo

Descrizione generata automaticamente

**Disegnare con indexed primitives**

Per disegnare tramite indice, bisogna seguire gli stessi passaggi fin ora; l’unica differenza è che ci saranno due buffer attivi: uno per i vertici e uno per gli indici. Il codice è il seguente:

Immagine che contiene testo

Descrizione generata automaticamente

**Passare uniforms da Javascript a GLSL shaders**

Qui il passaggio è molto simile, ma meno verboso del precedente:

Immagine che contiene testo

Descrizione generata automaticamente

**Compilare e linkare gli shader da JavaScript**

Questo è il passo fondamentale per eseguire uno shader. Esso va linkato e compilato chiamando le funzioni apposta dal codice JavaScript, altrimenti non ci sarà nulla da eseguire e la pipeline non sarà completa. Innanzitutto, è necessario creare un oggetto di tipo Shader sia per i Vertex Shader sia per i Fragment Shader, con la funzione **gl.createShader.** Poi bisogna caricare il codice sorgente dello shader, che sarà preventivamente stato salvato in una stringa (nell’esempio qui è chiamato vertexShaderSource), con la funzione **gl.shaderSource**, e infine lo shader va compilato chiamando la funzione **gl.compileShader. ATTENZIONE:** se si verifica un errore di compilazione nello shader, il programma Javascript non dice nulla. Quindi è necessario stare molto attenti ed eventualmente preventivare un sistema di intercettazione degli errori di compilazione.

Alla fine della compilazione degli shader bisogna definire uno **shader program,** ovvero un oggetto che contiene la completa pipeline di WebGL. Questo si ottiene con la funzione **gl.createProgram**. In seguito, al programma bisogna allegare gli shader con **attachShader** e infine bisogna linkare il programma con **gl.linkProgram.** A questo punto la pipeline è completa e grazie a WebGL abbiamo potuto renderizzare un’immagine. Ecco le istruzioni appena descritte all’interno del codice:

Immagine che contiene testo

Descrizione generata automaticamente

**Trasformazioni e animazioni**

C’è una sequenza particolare di trasformazioni che viene applicata nella pipeline di OpenGL/WebGL. Innanzitutto si parte dalle coordinate locali, ovvero le posizioni x,y,z,w dei vertici di un modello relativi al centro del sistema di riferimento in cui è stato creato. La prima trasformazione che si applica ad un modello, o ad una serie di modelli, è il posizionamento nel mondo, che avviene per la moltiplicazione di una determinata **World Matrix.** Essa mette il modello in una certa posizione e in un certo orientamento nello spazio.

Una volta posizionati gli oggetti, è il momento di calcolare la **View Matrix,** ovvero le trasformazioni da applicare per posizionare la telecamera in un certo punto dello spazio.

Ora è il momento della **Projection Matrix,** ovvero la matrice che definisce la proiezione con la quale il mondo in 3 dimensioni viene percepito dalla telecamera.

L’operazione di divisione per w è eseguita in automatico daWebGL, e resta solamente da definire la viewport transform, che definisce la trasformazione per rappresentare la scena sul canvas in base alle sue dimensioni.

Immagine che contiene testo, screenshot, persona

Descrizione generata automaticamente

Le matrici per le trasformazioni vanno passate al programma GLSL sotto forma di **uniforms.**

**IMPORTANTE:** se ci si appoggia ad una libreria come utils.js per definire le matrici, bisogna ricordarsi di **trasporle sempre** prima di passarle al GLSL, poiché esso le interpreta nella versione trasposta. Se non effettuiamo questa operazione, tutte le matrici che utilizzeremo daranno risultati imprevedibili.