Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

STUDIO DI MODELLI DI ILLUMINAZIONE E TEXTURE IN AMBIENTE WEBGL

Relatore:
Chiar.mo Prof.
CASCIOLA Giulio

Presentata da:
STEFANINI Francesco

Sessione III
Anno Accademico 2013/14

Introduzione

Il mondo di Internet ha vissuto un radicale e inarrestabile processo di rinnovamento nel corso dell'ultimo decennio. Nel giro di pochi anni, i siti che popolano il World Wide Web si sono evoluti divenendo vere e proprie applicazioni in grado di fornire un livello di interattività e di coinvolgimento fino ad allora impensabile. Il mondo del Web è mutato, e con esso quello dei browser, i quali assumono sempre più le conformazioni di "sistemi operativi nei sistemi operativi": si sono tramutati in complesse piattaforme di sviluppo in grado di fornire a programmatori e web designer potenti librerie e API relative a qualsiasi ambito, nonché avanzati strumenti di *debugging*.

Numerosi standard che governano l'ecosistema di Internet hanno raggiunto la maturità in questo contesto: fra tutti HTML5, il quale ha arricchito enormemente le potenzialità di un browser introducendo nuovi strumenti orientati alla multimedialità e alla classificazione semantica delle risorse.

Altri standard altrettanto importanti hanno visto la luce in questi anni, affermandosi e conquistando, nel giro di pochissimi anni, l'interesse di un'ampia platea di sviluppatori. È il caso di WebGL, una potente e flessibile libreria grafica derivata dal mondo di OpenGL che ha aperto le porte al rendering di scene tridimensionali all'interno di un qualsiasi browser moderno.

WebGL ha rappresentato un punto di svolta abbattendo un'ulteriore barriera tra il mondo del web che vive all'interno di un browser e la dimensione delle applicazioni native che popolano un sistema operativo, consolidando il già affermato concetto di *web app* che lentamente sta seppellendo l'idea di "sito" così come era stato concepito all'inizio del nuovo millennio.

Scopo di questo elaborato è quello di fornire una panoramica delle princi-

pali funzionalità offerte dalla libreria WebGL (con una particolare attenzione per il supporto *cross browser*) e di analizzare le possibilità che essa offre, studiando e implementando i principali modelli di illuminazione e le tecniche di applicazione texture per definire un ambiente tridimensionale esplorabile e il più possibile realistico all'interno della dimensione del web.

Indice

Elenco delle figure

Capitolo 1

La libreria WebGL

WebGL è una libreria grafica (più formalmente, una Application Programming Interface o API) orientata al mondo del web che permette di effettuare il rendering di scene tridimensionali all'interno dei più recenti browser. WebGL si pone come un subset di OpenGL (in particolar modo, di OpenGL ES - la versione sviluppata per le piattaforme mobile) e le sue specifiche, analogamente a quanto accade per la libreria madre, sono stilate e mantenute dal consorzio di aziende no profit Khronos Group.

La prima release ufficiale di WebGL ha visto la luce il 3 marzo 2011, frutto di un processo di sviluppo durato oltre due anni che ha portato il gruppo di lavoro a recepire e traslare in ambito web le specifiche di OpenGL ES 2.0. Al momento in cui si scrive, la libreria ha raggiunto il numero di versione 1.0.3; inoltre è in corso la stesura della prossima major release, targata 2.0, la quale recepirà le novità introdotte in ambiente mobile da OpenGL ES 3.0.

Le specifiche di WebGL affondano le proprie radici nel lavoro di Vladimir Vukićević, sviluppatore del progetto Mozilla da sempre attivo in ambito grafico, il quale nel 2006 concepì un nuovo modo per fare interagire la libreria OpenGL con le pagine HTML mediante l'oggetto `canvas` previsto dal nascente standard HTML5, servendosi esclusivamente di codice Javascript e senza l'ausilio di alcun plug-in esterno. Il progetto, denominato *Canvas 3D*, fu il punto di partenza del working-group *Accelerated 3D on the Web*, nato in seno al Khronos Group agli inizi del 2009 con il contributo di aziende del

calibro di Apple, Google, Opera nonché Mozilla stessa.

1.1 Panoramica

WebGL è una libreria grafica 3D di basso livello operante in *immediate mode* (le informazioni sulla scena da renderizzare devono essere inoltrate a ogni frame, a differenza di quanto accade in modalità *retained*), implementata a livello di browser e resa accessibile agli sviluppatori mediante il linguaggio di scripting Javascript. Essendo mutuata da OpenGL ES 2.0, la libreria non fornisce alcuna funzione di tipo *fixed* (tipiche delle prime incarnazioni di OpenGL) sposando in pieno l'approccio orientato agli shader, vale a dire programmi scritti in un opportuno linguaggio (GLSL) i quali vengono compilati, caricati all'interno della GPU e successivamente eseguiti in determinati stadi della pipeline grafica realizzando una vasta gamma di effetti ed elaborazioni sugli oggetti in fase di rendering (dalle trasformazioni geometriche sino all'implementazione di modelli di illuminazione o all'applicazione di texture).

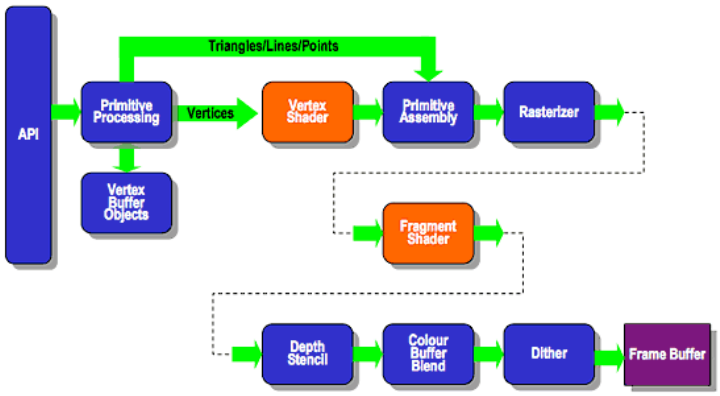

Figura 1.1: Pipeline grafica di WebGL 1.0

L'API, per poter funzionare, richiede la presenza di una scheda grafica ad architettura programmabile, opportunamente affiancata dai driver più aggiornati forniti dalla casa produttrice, driver specifici per il sistema operativo

in uso.

```
void main(void) {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Esempio di Fragment Shader scritto in GLSL

1.1.1 L'elemento canvas

In aggiunta ai requisiti verso il basso (ovvero a livello di hardware e driver), WebGL richiede che il browser supporti le specifiche HTML5 e, in particolar modo, che preveda l'elemento `canvas`.

Un canvas definisce un'area di dimensione rettangolare all'interno di una pagina HTML in cui è possibile generare in maniera procedurale forme e oggetti sia bidimensionali che tridimensionali, oltre che immagini bitmap. Il canvas può essere pilotato mediante codice Javascript utilizzando un opportuno contesto di rendering (*rendering context*) capace di popolare dinamicamente la superficie di disegno.

```
var canvas = document.getElementById('myCanvas');
var context = canvas.getContext('2d');
context.fillStyle = 'red';
context.fillRect(30, 30, 50, 50);
```

Esempio di codice Javascript che utilizza il contesto 2D

Le specifiche HTML5 definiscono esclusivamente il contesto per la grafica bidimensionale. WebGL arricchisce le potenzialità di un browser introducendo un secondo contesto (`WebGLRenderingContext`) dedicato alla grafica tridimensionale il quale offre allo sviluppatore la possibilità di effettuare chiamate di basso livello alla GPU e di visualizzare l'output della pipeline grafica (il contenuto del *framebuffer*, ovvero le informazioni a livello di pixel del frame renderizzato) all'interno dell'area delimitata dal canvas.

```
var canvas = document.getElementById('myCanvas');
var gl = canvas.getContext('webgl');
gl.clearColor(1.0, 0.0, 0.0, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT);
```

Esempio di codice Javascript che utilizza il contesto WebGL

1.2 L'approccio orientato agli shader

Gli shader sono - a tutti gli effetti - programmi che vengono eseguiti all'interno di una scheda grafica implementando, come il nome suggerisce, algoritmi di *shading*, vale a dire procedure per elaborare e modellare il colore, la luminosità e - in generale - tutte le proprietà visive di un modello tridimensionale.
Le potenzialità di questi strumenti vanno però ben oltre le tecniche di *shading*: ad oggi, tali programmi permettono di realizzare una vasta gamma di effetti e trasformazioni - a livello di immagini statiche così come di filmati animati - e offrono la possibilità, tra le altre cose, di generare nuove geometrie a partire da un dato insieme di vertici, di effettuare la tassellazione di superfici anche complesse nonché di svolgere operazioni di calcolo *general purpose*, non strettamente legate all'ambito grafico.

Gli shader sono progettati per essere eseguiti in parallelo direttamente dalla GPU utilizzando quante più unità di computazione vengono messe a disposizione dall'hardware grafico dell'elaboratore. Per quanto concerne la famiglia di librerie OpenGL, questi vengono implementati nel linguaggio di shading GLSL (OpenGL Shading Language), un linguaggio di alto livello sviluppato anch'esso in seno al Khronos Group che presenta numerose analogie sintattiche con C al fine di facilitare la transizione al nuovo approccio da parte degli sviluppatori più esperti.

La libreria OpenGL, nella sua ultima incarnazione (il ramo di sviluppo 4.x), definisce sei distinti tipi di shader: *vertex*, *geometry*, *tessellation control*, *tessellation evaluation*, *compute* e *fragment*. Di questi, solamente il primo (*vertex*) e l'ultimo (*fragment*) sono attualmente disponibili nelle spe-

cifiche di WebGL.
Ciascuno shader permette di programmare un determinato stadio della pipeline grafica, prendendo in input ed elaborando primitive grafiche e valori che in seguito vengono passati al modulo successivo, sino a raggiungere l'ultima fase in cui viene composto il framebuffer e la scena risulta così pronta per essere visualizzata.

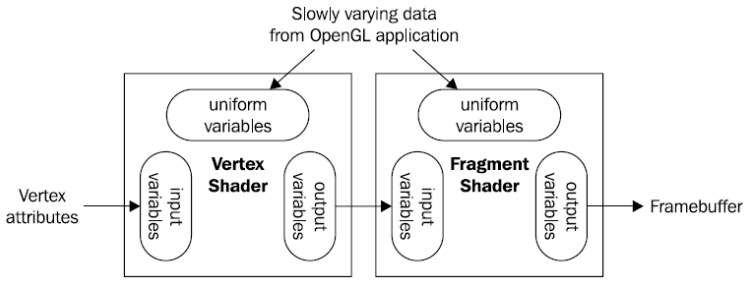

Figura 1.2: Relazioni tra i parametri *attribute* e *uniform*

I dati in input possono essere forniti agli shader mediante le due seguenti modalità:

- variabili *attribute* a livello di vertice: tutte le informazioni riguardanti un vertice (posizione, vettore normale, coordinate texture e quant'altro) vengono inviate al *vertex shader*, primo stadio della pipeline grafica, e a partire da esso possono essere propagate (anche in parte) a tutti i successivi componenti. La libreria effettua una mappatura di tutti gli *attribute* definiti a livello di shader dallo sviluppatore, fornendo a quest'ultimo gli opportuni riferimenti di memoria per poter trasmettere alla GPU, mediante l'ausilio di uno o più buffer, i valori di input;

- variabili *uniform*: tutti gli input che non sono soggetti a frequenti cambiamenti possono essere inviati come *uniform*, ovvero variabili in **sola lettura** accessibili da qualunque shader in qualsiasi momento. Vengono solitamente utilizzate per trasmettere dati quali le matrici

prospettiche, la posizione di una sorgente luminosa o il colore di una mesh.

Ciascuno shader, prima di poter essere utilizzato, necessita di essere compilato mediante il compilatore GLSL fornita dalla libreria grafica. In WebGL la fase di compilazione deve essere svolta a ogni avvio dell'applicativo; al contrario, la libreria madre OpenGL (a partire dalla versione 4.1) offre la possibilità di salvare in modo permanente gli shader pre-compilati, con un netto vantaggio in termini di prestazioni e di tempistiche.

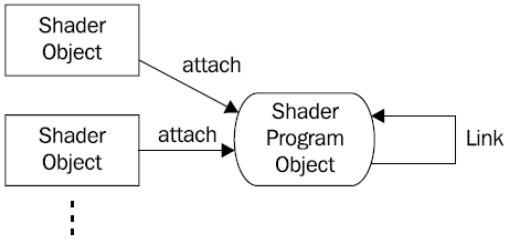

Figura 1.3: Processo di *linking* di shader GLSL

Analogamente a quanto accade nel processo di compilazione di una qualsiasi applicazione, gli shader non possono essere eseguiti singolarmente (possono essere paragonati ai file oggetti) e devono essere sottoposti a una fase di *linking*. Il linker GLSL, anch'esso incluso di default nella libreria, prende in input gli shader pre-compilati e restituisce un "programma shader" pronto per essere caricato e utilizzato all'interno di una scheda grafica. Il linker crea tutti i collegamenti necessari tra i parametri di input/output di ciascuno shader, nonché elabora i riferimenti di memoria che possono essere utilizzati all'interno di un programma OpenGL/WebGL per fornire i valori di input.

1.3 La pipeline programmabile

Come accennato in precedenza, le specifiche della prima release di WebGL offrono una limitata possibilità di personalizzare la pipeline grafica. In

particolare, è possibile definire esclusivamente i seguenti shader:

- *vertex shader*, eseguito una volta per ciascuno vertice passato in input, preferibilmente in parallelo. Definisce in modo immutabile la posizione di un vertice elaborando i parametri in input di tipo *attribute* e *uniform*, agendo sulla variabile di sistema `gl_Position`. Inoltre può calcolare e assegnare dei parametri in output che saranno successivamente utilizzati a livello di *fragment*;

- *fragment shader*, eseguito una volta per ciascun *pixel* (o *fragment*). Determina il colore di un pixel settando la variabile di sistema `gl_FragColor` prima che questo venga inviato al *framebuffer*; questa tipologia di shader fa uso delle informazioni ricevute dal *vertex shader*, nonché delle variabili *uniform* fornite dal programma WebGL.

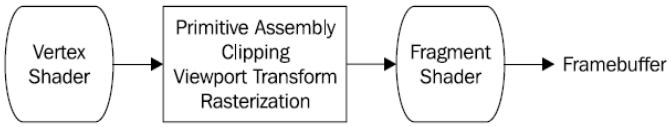

Figura 1.4: I passaggi intermedi tra l'esecuzione dei due shader

Tra l'esecuzione del *vertex* e *fragment shader*, la libreria grafica effettua le seguenti operazioni di *post-processing* a livello di vertice:

1. i vertici vengono assemblati in primitive. Le primitive disponibili in WebGL sono le **linee** (singole, continue oppure chiuse) e i **triangoli** (semplici, adiacenti oppure a ventaglio);

2. i vertici vengono sottoposti a un test di *clipping* e vengono scartati quelli che, essendo al di fuori dell'area rettangolare (*viewport*) definita dall'oggetto `canvas`, non è necessario visualizzare;

3. le coordinate tridimensionali dei vertici vengono convertite in coordinate bidimensionali: così facendo, la scena 3D viene mappata in un'immagine statica (*frame*);

4. il processo di *rasterizzazione* elabora i pixel che successivamente saranno sottoposti all'azione del *fragment shader*.

Infine, prima che i *fragment* raggiungano la fine della pipeline andando a popolare il *framebuffer*, la libreria svolge ulteriori test su di essi al fine di scartare i pixel superflui. Le principali operazioni che possono essere svolte sono:

- il *depth test* per determinare l'ordinamento dei pixel rispetto all'asse Z (ovvero verificare se un pixel è "davanti" oppure "dietro" un altro, rispetto a un punto di osservazione fissato);

- il *blending* per determinare (utilizzando una delle modalità disponibili a livello di API) il colore e la trasparenza di un pixel in base a quelli già presenti all'interno del *framebuffer*.

1.3.1 La *fixed-pipeline*: uno sguardo al passato

La libreria WebGL, nella sua prima e al momento unica iterazione, è stata concepita come una versione orientata al mondo del web di OpenGL ES 2.0; pertanto, fin dalle prime bozze, le specifiche non hanno mai contemplato l'esistenza di una pipeline *non* programmabile e di funzioni *hard-coded* (*fixed*) che implementino le più comuni operazioni e trasformazioni grafiche.

Il concetto di pipeline programmabile è stato introdotto in OpenGL nel 2004 con la versione 2.0 e la relativa introduzione del linguaggio di shading GLSL. La principale motivazione che ha spinto il board responsabile della libreria a una radicale e massiccia revisione dell'API (e che ha spinto i produttori, di conseguenza, a un altrettanto importante revisione dell'architettura interna delle schede grafiche) è stata la volontà di fornire agli sviluppatori un livello di flessibilità e potenza di calcolo esponenzialmente superiore al passato, semplificando il processo di stesura del codice e liberando i programmatori

dall'onere di servirsi di fantasiosi *workaround* per aggirare il basso grado di adattabilità delle precedenti funzioni *fixed* (ritenute a quel tempo, a detta di tutti, limitanti e ormai obsolete).

Le *fixed-function* sono ancora presenti all'interno di OpenGL ma, a partire dalla versione 3.0, sono state dichiarate **deprecate**: gli sviluppatori sono pertanto caldamente invitati a non utilizzarle più e a migrare all'approccio orientato agli shader, dal momento che queste non riceveranno più aggiornamenti da parte del board e, anzi, potrebbero essere totalmente eliminate in una versione futura della libreria.

WebGL non ammette (e si può tranquillamente affermare che mai ammetterà) la presenza di funzioni di tipo *fixed*. OpenGL ES, al contrario, ha visto la luce in un'epoca in cui gli shader erano una tecnologia ancora acerba e in fase di definizione; pertanto, nella sua prima versione, presentava il supporto a tali funzioni. Vista la stretta correlazione tra la libreria mobile di casa Khronos e la nuova arrivata orientata al mondo del web, la *fixed-function pipeline* di una ipotetica versione 0.x di WebGL potrebbe essere analoga a quella del ramo 1.x di OpenGL ES, ovvero modellata come in figura 1.5.

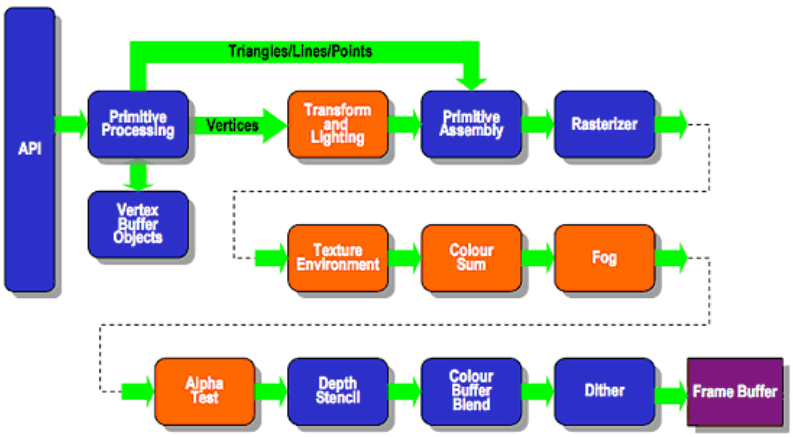

Figura 1.5: La *fixed-function pipeline* di OpenGL ES 1.0

1.4 Supporto nei browser

WebGL è attualmente supportato dai principali browser moderni (Google Chrome, Mozilla Firefox, Opera, Internet Explorer, Safari), sia su piattaforme desktop che su piattaforme mobile (ad esempio, Google Chrome su Android o Internet Explorer su Windows Phone).

Al fine di garantire una maggiore fluidità e stabilità, molto spesso in ambiente Windows i browser utilizzano un layer di astrazione grafico denominato **ANGLE** (**Almost Native Graphics Layer Engine**) pensato per tradurre le chiamate OpenGL in chiamate Direct3D mediante uno dei due backend disponibili (Direct3D 9.0c oppure Direct3D 11, a seconda di quanto è recente la scheda grafica). Chrome e Firefox adoperano ANGLE di default, pur sempre prevedendo la possibilità per l'utente di passare all'esecuzione nativa del codice OpenGL.

1.4.1 Il supporto hardware

Contrariamente a quanto visto lato software per i browser, le schede grafiche in grado di supportare l'esecuzione di un programma WebGL sono, in percentuale, un numero minore.

In linea di massima, tutte le GPU concepite e distribuite dopo il 2002 dispongono di un'architettura programmabile e permettono la manipolazione di almeno due stadi della pipeline grafica (il vertex e il fragment shader), garantendo così, almeno teoricamente, il supporto alla libreria.

Tuttavia nei fatti la situazione non appare così rosea; difatti, in presenza di determinate configurazioni, di sistemi operativi non più supportati o di versioni obsolete dei driver grafici, i browser possono disabilitare (parzialmente o totalmente) le funzionalità di WebGL. A tal proposito, essi dispongono di *whitelist* e *blacklist* stilate dagli sviluppatori che definiscono quali debbano essere le condizioni lato hardware e lato software affinché un programma WebGL possa essere eseguito correttamente, senza generare artefatti grafici e senza avere ricadute negative in termini prestazionali a livello di browser e di sistema operativo.

Ad esempio, Google Chrome nega l'abilitazione della libreria WebGL in presenza dei chipset grafici `Intel` serie `945 Express` e delle schede `S3 Trio`, nonché richiede, in ambiente Windows, che i driver grafici, indipendentemente dalla casa produttrice della GPU, non siano datati a prima del 1 gennaio 2009. Mozilla Firefox, invece, su piattaforma Windows esige che la versione minima del sistema operativo sia XP aggiornato all'ultimo service pack, così come non permette di utilizzare WebGL nelle versioni di Mac OS X inferiori a Snow Leopard (10.6). Tutte le restrizioni a livello hardware sono riportate nel wiki ufficiale del progetto WebGL.

In aggiunta alle *blacklist* e *whitelist*, i browser adottano numerosi workaround per porre rimedio ai bug grafici che possono intercorrere anche all'interno delle configurazioni hardware/software considerate attendibili.

1.5 Le estensioni

Analogamente a quanto accade per OpenGL, lo sviluppo di WebGL procede per estensioni. Un estensione è un insieme di modifiche alle specifiche ufficiali formulate da aziende, sviluppatori o gruppi di lavoro che possono essere poste al vaglio della comunità e, in un secondo momento, del board responsabile del mantenimento del progetto: se queste vengono valutate positivamente, l'estensione diventa parte integrante delle specifiche in ottica di una release futura.

Le software house sono ovviamente libere di integrare le estensioni all'interno dei propri browser prima che queste completino il canonico iter di approvazione; questo porta a definire degli standard de facto di caratteristiche e funzionalità che ogni programma concorrente dovrebbe avere, alimentando una nuova branchia nella pluridecennale guerra dei browser.

1.6 Le principali differenze con OpenGL

Al momento in cui si scrive, WebGL implementa un ristretto numero di funzionalità della libreria madre OpenGL e presenta numerose mancanze, so-

prattutto in termini di personalizzazione della pipeline grafica. Il linguaggio di shading utilizzato, inoltre, è un *subset* di GLSL denominato GLSL ES, anch'esso mutuato dalla versione mobile.

In aggiunta alla feature mancanti, alcune funzionalità presenti in entrambe le librerie presentano importanti differenze a livello implementativo e/o sono state introdotte solo in maniera parziale. Le più rilevanti discrepanze sono:

- il supporto alle texture **non-power-of-two** (**NPOT**) in WebGL è al momento limitato, a differenza di quanto accade in OpenGL che le supporta al 100%;
- le texture 3D non sono attualmente supportate in WebGL;
- l'*attribute* di indice 0 in OpenGL ha una semantica particolare: al contrario, in WebGL tutti i *vertex attribute* hanno la stessa semantica per garantire una maggiore uniformità;
- WebGL non supporta i tipi di dati in virgola mobile a **doppia** precisione;
- per quanto riguarda il linguaggio di shading, in GLSL ES il livello di precisione per i numeri in virgola mobile (`lowp`, `mediump` oppure `highp`) deve essere sempre definito, a differenza di quanto accade negli shader GLSL in ambiente desktop dove le variabili `float` vengono sempre trattate con la massima precisione possibile.

Capitolo 2

Modelli di illuminazione e texture

L'avvento di librerie grafiche di alto livello, in grado di fornire a sviluppatori, artisti e designer strumenti potenti e intuitivi per modellare un ambiente tridimensionale anche complesso all'interno dello spazio digitale di un elaboratore, ha immediatamente posto la sfida di ricercare procedure di calcolo convincenti (e, soprattutto, efficienti) per ricreare una vasta gamma di fenomeni fisici e naturali, nonché per simulare la presenza di diverse sostanze e materiali, al fine di raggiungere un soddisfacente livello di fotorealismo all'interno delle scene renderizzate.

La rinascita in chiave moderna della libreria OpenGL, avutasi con l'abbandono delle *fixed-function* e l'introduzione del concetto di shader, ha rappresentato un nuovo inizio nel processo di ricerca e definizione di tali algoritmi. Funzioni quali `glLight` o `glMaterial`, largamente utilizzate nei programmi serviti dalle prime istanze della libreria, sono oramai classificate come deprecate (e talvolta non sono neppure presenti, come nel caso di WebGL) in quanto considerate limitanti e poco flessibili per i livelli di qualità e di dettaglio che si desiderano raggiungere oggigiorno. Si è pertanto provveduto a implementare le funzionalità di *shading* preesistenti utilizzando il nuovo linguaggio GLSL, ponendo al contempo le basi per svilupparne di nuove più evolute oppure che non era possibile realizzare (anche solo parzialmente)

servendosi del vecchio approccio.

Scopo di questo capitolo è trattare i principali modelli di illuminazione (dal punto di vista dei principi teorici e delle modalità di implementazione) e le procedure di gestione e mappatura di texture realizzabili mediante la libreria WebGL, mantenendo uno sguardo sul passato e, in particolar modo, su quelle che erano le feature disponibili *out of the box* nelle prime versioni di OpenGL.

2.1 I principali modelli di illuminazione

2.1.1 Il modello *diffuse*

Il modello *diffuse* realizza il fenomeno della riflessione diffusa della luce: un raggio luminoso che incide su di una superficie viene propagato in tutte le direzioni indipendentemente dalla posizione della sorgente. Non è un fenomeno speculare e la luce incidente, prima di essere diffusa in maniera omnidirezionale, interagisce con la superficie assorbendo - parzialmente oppure totalmente - determinate lunghezze d'onda dello spettro visibile, rilasciandone altre.

Il modello *diffuse* conferisce a una superficie l'effetto tipico della pittura opaca, spenta e priva di alcun tipo di riflesso o brillantezza. Ai fini dell'implementazione si suppone che valgano le seguenti condizioni:

1. la superficie è da considerarsi ideale (o *lambertiana*) tale per cui la luce viene riflessa omogeneamente in tutte le direzioni, e non solamente in alcune;

2. la sorgente luminosa è unica e puntiforme;

3. il modello è applicato per vertice, e non per *fragment*.

Dal punto di vista matematico, la riflessione diffusa coinvolge due vettori: il vettore $\mathbf{s}$ che identifica la direzione da un punto della superficie alla sorgente luminosa e il vettore normale $\mathbf{n}$, anch'esso calcolato nel vertice preso in esame.

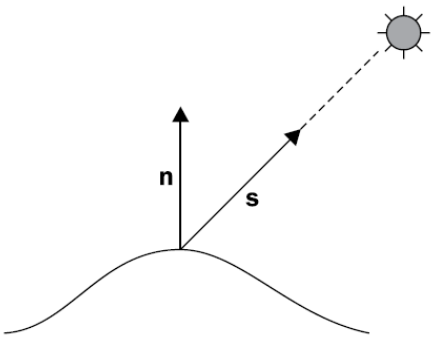

Figura 2.1: Diagramma del modello *diffuse*

La quantità di luce che raggiunge la superficie è strettamente dipendente dall'orientamento della stessa rispetto alla posizione della sorgente luminosa. Nello specifico, tale quantità di radiazioni è massima quando la sorgente è posizionata nella direzione indicata dal vettore normale ed è nulla quando il vettore direzione è perpendicolare alla normale del vertice. Ovvero, è direttamente proporzionale al coseno dell'angolo calcolato tra i vettori **s** e **n**.
Si consideri il prodotto scalare (*dot*) definito tra due vettori a, b come

$$a \cdot b := |a| * |b| * cos(\theta)$$

con θ l'angolo esistente tra di essi. Essendo direttamente proporzionale al coseno dell'angolo tra due vettori, è possibile esprimere la quantità di luce che irradia una superficie (**L**) come il prodotto tra l'intensità luminosa (L_d) e il prodotto scalare tra i vettori **s** e **n**. Ovvero:

$$L = L_d * (s \cdot n)$$

con **s**, **n** normalizzati e il prodotto scalare **non** negativo.
Dal momento che una frazione delle radiazioni in entrata viene assorbita dalla superficie, è necessario introdurre un fattore all'equazione che identifichi la quantità di luce che viene effettivamente diffusa. Tale fattore è conosciuto come **coefficiente di riflessione diffusa** (K_d) e permette di affinare la formula precedente come segue:

$$L = L_d * K_d * (s \cdot n)$$

In termini implementativi, L_d e K_d sono anch'essi vettori a tre componenti che definiscono un colore secondo lo standard RGB dove l'intervallo 0-255 è mappato in valori compresi tra 0.0 e 1.0, come da convenzione OpenGL. Se la luce in entrata è bianca (tutte le componenti di L_d sono pari a 1.0) e K_d definisce il colore di una superficie come sua proprietà intrinseca, il modello di illuminazione calcola così le zone di ombra e penombra che si vengono a formare all'interno della scena in base alla posizione della sorgente luminosa.

```
vec3 diffuseModel(vec3 position, vec3 normal)
{
    vec3 s = normalize(uLightPosition - position);
    vec3 n = normal;
    vec3 Kd = uColor;

    return Ld * Kd * max(0.0, dot(s, n));
}

void main(void)
{
    vec3 position = vec3(uMatrix.mv * vec4(aPosition, 1.0));
    vec3 normal = normalize(uMatrix.n * aNormal);

    vFrontColor = diffuseModel(position, normal);
    vBackColor = diffuseModel(position, -normal);
}
```

Estratto dal *vertex shader* che implementa il modello *diffuse*

Il principale difetto del modello di illuminazione *diffuse* è quello di rendere le aree poco illuminate di una superficie eccesivamente scure, se non - in certi casi - completamente nere. Un problema questo a cui il seguente modello riesce a dare una soluzione.

2.1.2 Il modello Phong: *ambient*, *diffuse*, *specular*

Il modello Phong, altrimenti conosciuto come *ADS*, definisce la luminosità e il colore di una superficie come la somma di tre distinte componenti:

- **ambient**, ovvero la riflessione d'ambiente: rappresenta la luce che è stata riflessa talmente tante volte che appare come se fosse emanata uniformemente da qualsiasi direzione; permette di illuminare maggiormente le zone molto scure di una scena tridimensionale;
- **diffuse**, ovvero la riflessione diffusa (trattata in precedenza);
- **specular**, ovvero la riflessione speculare: definisce la brillantezza e la lucidità di una superficie e determina il riflesso visibile in un determinato punto di osservazione.

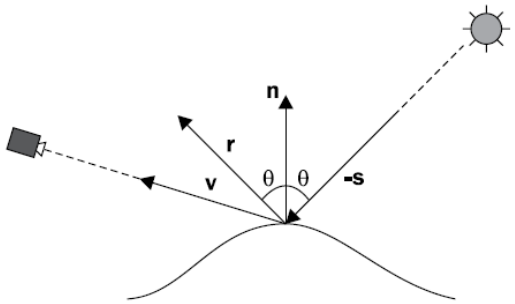

Figura 2.2: Diagramma del modello *ADS*

Il modello Phong, a differenza del precedente, conferisce a una superficie un effetto lucido e scintillante, simulando la presenza di una materiale di tipo metallico. Analogamente a *diffuse*, Phong - perlomeno in questa incarnazione - è calcolato a livello di vertice e non di fragment.

Dal punto di vista matematico, la componente ambientale, non dipendendo in alcun modo dalla direzione - in entrata così come in uscita - dei raggi emessi dalla sorgente, è definita come il prodotto tra l'intensità della luce ambientale (L_a) e un coefficiente di riflessione ambientale (K_a) proprio della superficie. Ovvero:

$$A = L_a * K_a$$

La componente *diffuse* è definita come nell'omonimo modello trattato in precedenza, ovvero come:

$$D = L_d * K_d * (s \cdot n)$$

La componente speculare, infine, definisce la lucentezza di una superficie. Per definizione, la luce riflessa è massima quando si ha una condizione di riflessione perfetta, ovvero quando l'angolo di incidenza dei raggi in entrata ha lo stesso valore dell'angolo di riflessione dei raggi in uscita ed entrambi i vettori direzionali sono coplanari rispetto al vettore normale. Si definisce il vettore di riflessione **r** come:

$$r = -s + 2 * (s \cdot n) * n$$

con **s** e **n** definiti come in precedenza, entrambi normalizzati.
Sia **v** il vettore che definisce la direzione dell'osservatore rispetto al vertice corrente e sia **f** un valore intero che rappresenta il **grado di lucentezza**. La componente speculare risulta così espressa:

$$S = L_s * K_s * (r \cdot v)^f$$

con L_s e K_s rispettivamente intensità e coefficiente di riflessione speculare.
Il fattore $(r \cdot v)^f$ formalizza il fatto che la riflessione speculare debba essere massima quando l'osservatore risulta allineato con il vettore **r** e che debba diminuire rapidamente allontanandosi da esso (maggiore l'esponente, maggiore la velocità con cui il coefficiente tende a zero all'aumentare dell'angolo tra i due vettori). Il grado di lucentezza **f** definisce inoltre l'ampiezza delle zone più luminose che si vengono a formare sulla superficie: tipicamente si utilizzano valori compresi tra 1 e 200.

```
vec3 phongModel(vec3 position, vec3 normal)
{
    vec3 s = normalize(uLightPosition - position);
    vec3 v = normalize(-position);
    vec3 n = normalize(normal);
    vec3 r = reflect(-s, n);
    vec3 Ka = uColor;
    vec3 Kd = uColor;

    vec3 ambient = La * Ka;
    vec3 diffuse = Ld * Kd * max(0.0, dot(s, n));
    vec3 specular = Ls * Ks * pow(max(0.0, dot(r, v)), f);

    return (ambient + diffuse + specular);
}
```

Estratto dal *vertex shader* che implementa il modello *ADS*

2.1.3 Un'ottimizzazione: Phong per *fragment*

I modelli di *shading* visti sinora sono stati progettati per essere elaborati a livello di *vertex shader*, calcolando e assegnando un colore a ciascun vertice di una superficie. Successivamente tali valori vengono sottoposti a un processo di interpolazione che permette di ottenere il colore interpolato da assegnare a un pixel al momento dell'esecuzione del *fragment shader*.
Questa tecnica di approssimazione risulta però eccessivamente dispendiosa in termini di (preziosi) cicli di esecuzione della GPU, nonché può portare alla formazione di artefatti indesiderati quali una maggiore enfasi sui vertici dei poligoni che compongono una mesh oppure un non corretto posizionamento della zona di riflessione speculare sulla superficie di un modello.

Al fine di ottenere una maggior grado di accuratezza, è possibile svolgere il calcolo delle equazioni di *shading* all'interno del *fragment shader*, interpolando la posizione e il vettore normale di un vertice invece che il valore del colore. Questa tecnica di approssimazione permette di generare scene molto

più precise e dettagliate dal punto di vista dell'illuminazione, pur non essendo totalmente immune alla formazione di artefatti e glitch grafici.

```
varying vec3 vPosition;
varying vec3 vNormal;

vec3 phongModel(vec3 position, vec3 normal) { ... }

void main(void)
{
    if (gl_FrontFacing) {
        gl_FragColor = vec4(phongModel(vPosition, vNormal), 1.0);
    } else {
        gl_FragColor = vec4(phongModel(vPosition, -vNormal), 1.0);
    }
}
```

Estratto dal *fragment shader* che implementa il modello Phong per fragment

I parametri in entrata `vPosition` e `vNormal` sono il frutto dell'interpolazione ottenuta a partire dalla posizione e dal vettore normale dei vertici che compongono la mesh. In questa implementazione il modello Phong è calcolato due volte al variare del segno del vettore normale, al fine di garantire un corretto e uniforme grado di colorazione ad ogni faccia della superficie, frontalmente così come posteriormente.

2.1.4 Il modello Blinn-Phong

Il modello Blinn-Phong, concepito dal ricercatore statunitense James F. Blinn, modifica parzialmente il lavoro sviluppato alcuni anni prima da Bui Phong intervenendo sul vettore **r** di riflessione speculare.

Il vettore **r** di riflessione speculare, così come da specifiche del modello Phong, è definito come segue:

$$r = -s + 2 * (s \cdot n) * n$$

Il calcolo di **r** prevede un'operazione di addizione, due moltiplicazioni e un prodotto scalare: è pertanto un procedimento lungo e dispendioso.

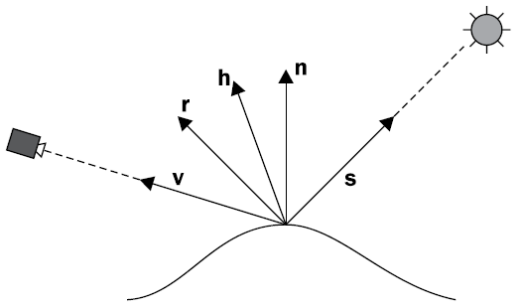

Figura 2.3: Diagramma del modello Phong-Blinn

Al contrario, Blinn definisce un vettore **h** (vettore **halfway**) posizionato a metà via tra **v** (la direzione verso l'osservatore) e **s** (la direzione verso la sorgente luminosa). Dal punto di vista matematico:

$$h = v + s$$

con **h** successivamente normalizzato.
La componente **S** è ora calcolata come il prodotto tra l'intensità speculare, il coefficiente di riflessione speculare e il prodotto scalare, elevato al grado di lucentezza, fra **h** e **n**. Ovvero:

$$S = L_s * K_s * (h \cdot n)^f$$

In termini qualitativi e di impatto visivo, i risultati ottenibili con il metodo di Blinn-Phong sono paragonabili a quelli apprezzabili con il modello Phong classico, con l'unica differenza che le aree di riflessione speculare appaiono molto più luminose a parità di parametri e valori in input. Pertanto è spesso necessario aumentare il valore dell'esponente **f** al fine di limitare il bagliore, spesso accecante e dirompente, che si viene a formare all'interno di una superficie.

```
varying vec3 vPosition;
varying vec3 vNormal;

vec3 blinnPhongModel(vec3 position, vec3 normal)
{
    vec3 s = normalize(uLightPosition - position);
    vec3 v = normalize(-position);
    vec3 n = normalize(normal);
    vec3 h = normalize(v + s);
    vec3 Ka = uColor;
    vec3 Kd = uColor;

    vec3 ambient = La * Ka;
    vec3 diffuse = Ld * Kd * max(0.0, dot(s, n));
    vec3 specular = Ls * Ks * pow(max(0.0, dot(h, n)), f);

    return (ambient + diffuse + specular);
}

void main(void) { ... }
```

Estratto dal *fragment shader* che implementa il modello Blinn-Phong

2.1.5 *Toon Shading*: un esempio di non-fotorealismo

Il *toon shading* (anche conosciuto come *cel shading*) è una tecnica non-fotorealistica in grado di far apparire gli oggetti tridimensionali realizzati con l'ausilio della computer grafica come se fossero stati disegnati a mano. Formalmente è una versione modificata del modello Phong in cui si considerano esclusivamente le componenti *ambient* e *diffuse* (quest'ultima in una veste diversa rispetto a quanto visto sinora).
Questo modello permette di definire, all'interno di una superficie, delle vaste aree di colore caratterizzate da un forte livello di contrasto e da transizioni nette tra l'una e l'altra. Così facendo è possibile simulare i tratti di matita

o pennello che un'artista effettuerebbe lavorando a mano.

Figura 2.4: Esempio di applicazione di *toon shading*

Dal punto di vista implementativo, il termine del prodotto scalare $(s \cdot n)$ viene *quantizzato* facendo in modo che assuma un numero fissato (definito a priori, i **livelli**) di valori possibili.

```
vec3 toonModel(vec3 position, vec3 normal)
{
    vec3 s = normalize(uLightPosition - position);
    vec3 n = normalize(normal);
    float cosine = max(0.0, dot(s, n));
    vec3 Ka = uColor;
    vec3 Kd = uColor;

    vec3 ambient = La * Ka;
    vec3 diffuse = Ld * Kd * floor(cosine * float(levels)) * scaleFactor;

    return (ambient + diffuse);
}

void main(void) { ... }
```

Estratto dal *fragment shader* che implementa il *toon shading*

2.2 Le texture

In ambiente WebGL, una texture è un'immagine bidimensionale di tipo *bitmap* o *raster* utilizzata per aggiungere colore, dettagli e informazioni di varia natura a una superficie tridimensionale generata programmaticamente.

Figura 2.5: Esempio di applicazione di una texture alle facce di un cubo

Un texture può essere applicata a una mesh poligonale mappando le sue coordinate UV in quelle dei punti che compongono la superficie.
Le lettere U e V denotano gli assi del sistema di riferimento 2D delle texture: si utilizzando tali lettere in quanto i caratteri X,Y e Z sono già in uso per definire lo spazio tridimensionale. La caratteristica principale delle coordinate UV è che i valori di entrambe le componenti oscillano in un range compreso tra 0.0 e 1.0.

La mappatura avviene a livello di shader, più precisamente all'interno del *fragment shader*. Lo script GLSL accede al contenuto di memoria di una texture (precedentemente inizializzata dal programma WebGL) mediante un parametro *uniform* di tipo **sampler** il quale espone alla pipeline grafica i valori RGB dell'immagine. Dal momento che tale operazione ha luogo nello spazio dei *fragment*, i valori coloristici giungono interpolati secondo una delle funzioni di interpolazione (**filtri**) messe a disposizione dalla libreria. Attualmente WebGL permette di scegliere solamente fra i due seguenti filtri:

- `GL_NEAREST`: funzione che restituisce il pixel il più possibile prossimo alle coordinate UV in input;

- `GL_LINEAR`: funzione che restituisce la media pesata fra i quattro pixel che circondano il pixel il più possibile prossimo alle coordinate UV in input.

È possibile scegliere in maniera distinta quale filtro utilizzare in caso di ingrandimento o di rimpicciolimento della texture.
Il filtro `GL_LINEAR` permette di avere un'immagine più morbida e sfuocata, mentre `GL_NEAREST` determinata il classico effetto "pixelato" che talvolta è volutamente ricercato per ricreare una grafica simil 8-bit tipica della prima era videoludica.

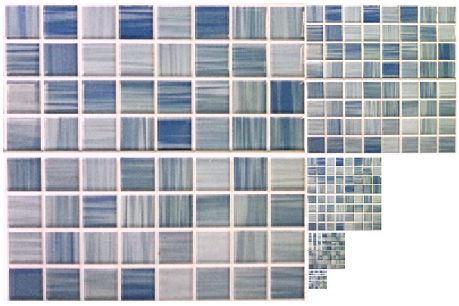

Figura 2.6: Esempio di *mipmap*

La libreria offre inoltre la possibilità di generare una *mipmap*, ovvero una raccolta ottimizzata di immagini ottenute a partire da una texture dove ciascuna di esse ha una risoluzione e un livello di dettaglio progressivamente minore. Una *mipmap* è in grado di alleggerire notevolmente la mole di calcolo che grava sulla GPU in fase di interpolazione dei punti di una texture, velocizzando le procedure di rendering.

Un valore RGB ricavato a partire da una texture può essere integrato a un livello di colore ottenuto mediante uno dei modelli di illuminazione trattati in precedenza. L'esempio successivo presenta un *fragment shader* che unisce il modello Phong con l'applicazione di una texture:

```
uniform sampler2D uTexture;
varying vec2 vTexCoord;

vec3 phongModel(vec3 position, vec3 normal) { ... }

void main(void)
{
    vec4 texColor = texture2D(uTexture, vTexCoord);

    if (gl_FrontFacing) {
        gl_FragColor = vec4(phongModel(vPosition, vNormal), 1.0)
                       * texColor;
    } else {
        gl_FragColor = vec4(phongModel(vPosition, -vNormal), 1.0)
                       * texColor;
    }
}
```

Da notare come le coordinate UV vengano inviate alla pipeline grafica come *attribute*, al pari della posizione e del vettore normale di un vertice. Esse vengono ricevute in input dal *vertex shader* e inoltrate *as is* al *fragment shader* come variabili di tipo `vec2` (vettori a due componenti).

2.2.1 *Normal mapping*: un uso non convenzionale

Le texture possono essere utilizzate per aggiungere numerosi dettagli a una mesh tridimensionali, fornendo informazioni che vanno al di là del semplice colore o del grado di opacità. Un esempio di utilizzo non convenzionale di texture è la tecnica di *normal mapping*.

La tecnica di *normal mapping* permette di realizzare variazioni e imperfezioni che non sono previste a livello di geometria di una mesh tridimensionale. Essa rende possibile produrre superfici ruvide che presentano protuberanze, ammaccature e increspature senza dover fornire i vertici (e le relative posizioni) che determinano tali deformazioni. La superficie, difatti, è perfettamente liscia e uniforme, ma appare irregolare in quanto si interviene sui vettori

normali dei vertici che la compongono, alterandoli in base alle informazioni contenute nella *normal map*.

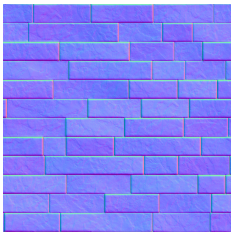

Figura 2.7: Una *normal map* che permette di simulare un muro di mattoni

Una *normal map* è una texture le cui informazioni a livello di pixel sono interpretate come le coordinate di un vettore normale e non come valori coloristici in formato RGB. Tipicamente la componente x è memorizzata nel canale *rosso*, la componente y nel canale *verde* e la componente z nel canale *blu*. I dati contenuti all'interno della *normal map* vengono pertanto utilizzati per alterare i vettori normali dei vertici e, di conseguenza, produrre dettagli e imperfezioni.

Questa tecnica permette di modellare superfici tridimensionali con un basso numero di poligoni, demandando a una *normal map* il compito di produrre tutte le deformazioni che altrimenti comporterebbero un aumento esponenziale del numero dei vertici da elaborare. Così facendo è possibile visualizzare scene complesse e dettagliate riducendo drasticamente la mole di calcoli da svolgere in fase di rendering.

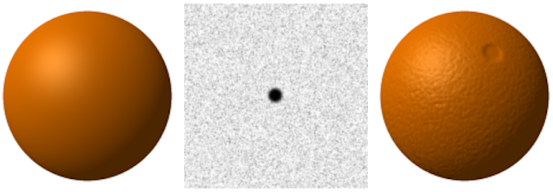

Figura 2.8: Una *bump map* applicata a una sfera

La tecnica di *normal mapping* rappresenta un'ottimizzazione del metodo di *bump mapping*. Quest'ultimo, invece di alterare i vettori normali di una superficie, modifica la geometria stessa di una mesh: una *bump map*, difatti, contiene le informazioni per manipolare la posizione dei vertici in fase di rendering (informazioni sempre codificate all'interno dello spettro RGB di un immagine bidimensionale).

Capitolo 3

Il progetto: architettura

Il naturale proseguimento dello studio degli argomenti sinora trattati ha portato alla produzione di un'applicazione web basata sulla libreria WebGL per sperimentare l'efficienza e la bontà, in termini di resa visiva, dei modelli di illuminazione introdotti nel capitolo precedente, nonché per provare sul campo le tecniche di applicazione di texture.

La fase di progettazione ha portato a definire tre vincoli nella scelta delle tecnologie e dei linguaggi da adottare a livello implementativo:

1. utilizzo preferenziale delle funzionalità previste dalle specifiche HTML5 per garantire un maggior supporto *cross-browser*;

2. utilizzo esclusivo del linguaggio Javascript limitando al massimo l'adozione di librerie di supporto, specie se di natura grafica, al fine di ottenere il controllo completo sull'esecuzione delle chiamate WebGL. Questo vincolo ha permesso di studiare a fondo la libreria, ottenendo al contempo dei vantaggi in termini prestazionali non avendo alcun intermediario responsabile di introdurre *overhead* nelle delicate operazioni di rendering;

3. utilizzo di "soggetti di test" sufficientemente dettagliati a livello di geometria per poter apprezzare meglio le differenze tra i vari modelli di illuminazione e per sperimentare l'applicazione di texture su superfici complesse.

A tal proposito, la scelta dei modelli di test è ricaduta sulle superfici NURBS, ottenute a partire da file in formato IGES.

3.1 Le superfici NURBS

Con l'acronimo NURBS (Non-Uniform Rational B-Spline) si identifica un modello matematico largamente utilizzato nel campo della computer grafica per rappresentare curve e superfici all'interno di uno spazio tridimensionale. Il modello NURBS garantisce un alto grado di precisione e flessibilità e permette di memorizzare le informazioni relative ad elementi geometrici (sia regolari che *free form*) in maniera compatta e ottimale.
Un oggetto NURBS consiste in una raccolta (limitata) di valori e parametri che necessitano di essere elaborati da un programma al fine di generare le coordinate dei vertici che lo compongono; questi ultimi, pertanto, non definiscono un'entità NURBS. In quest'ottica è possibile considerare i modelli NURBS al pari delle immagini vettoriali, le quali rappresentano una versione *parametrizzata* delle immagini *raster*.

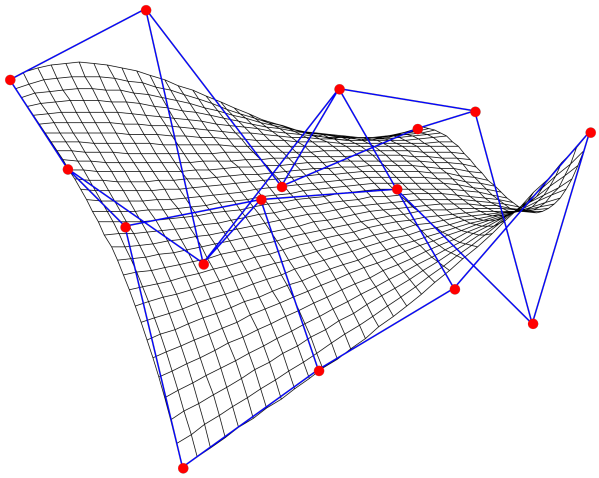

Figura 3.1: Esempio di superficie NURBS con i relativi punti di controllo

Una superficie NURBS è definita mediante le seguenti componenti:

1. i **gradi**, valori interi positivi e non nulli che definiscono l'elasticità di una superficie (due valori, uno per ogni direzione: maggiore il grado, maggiore la malleabilità);

2. i **punti di controllo**, una griglia di punti - memorizzati come vettori a quattro componenti - che determina la forma della superficie; per ciascuna direzione, il numero dei punti di controllo deve essere almeno pari al grado in quella direzione incrementato di un'unità. La quarta componente vettoriale è detta **peso** e rappresenta la capacità di un punto di attrarre a sè la curva;

3. i **nodi**, sequenze crescenti (ma non strettamente) di parametri non negativi che determinano l'influenza dei punti di controllo sulla curva; per ciascuna direzione, il numero dei nodi deve essere pari alla somma tra il grado e il numero di punti di controllo, somma incrementata di un'unità. All'interno di una sequenza, la *molteplicità* di un nodo (il numero di volte che si ripete) non deve essere superiore al grado in quella direzione.

Un superficie si dice **razionale** quando, in entrambe le direzioni, tutti i punti di controllo hanno lo stesso peso. Una superficie può altresì essere **uniforme** o **non uniforme**, come suggerisce l'acronimo del modello.
Si dice che un nodo ha *molteplicità piena* se la sua molteplicità è massima, ovvero pari al grado. Se entrambe le sequenze di nodi iniziano e terminano con un nodo a molteplicità piena, e nel mezzo vi sono nodi semplici ugualmente spaziati fra di loro, allora una superficie si dice **uniforme**.

I parametri di una superficie NURBS sono sottoposti a una fase di elaborazione mediante un algoritmo che implementa una opportuna funzione di valutazione. Tale algoritmo genera le coordinate dei punti che compongono la superficie con un livello di dettaglio personalizzabile, definibile dall'utente.

3.1.1 Il formato IGES

Ai fini del progetto si è scelto di adottare il formato IGES come tipo di file predefinito dal quale estrapolare le informazioni relative a una superficie

NURBS.

Lo standard IGES (acronimo di Initial Graphics Exchange Specification) nasce agli inizi degli anni '80 per volontà del National Bureau of Standards, ente del governo degli Stati Uniti d'America, ed è un formato in grado di consentire lo scambio di informazioni grafiche tra sistemi CAD (Computer-Aided Design). Ad oggi è ancora uno dei formati più utilizzati in ambito professionale per la realizzazione di progetti architetturali, schemi circuitistici e modelli tridimensionali di qualsiasi natura.

```
S      1
1H,,1H;,,29H/media/disk/models/Lamp16.igs,                              G      1
38HXCMODEL ver.3.0 Universita' di Bologna,                              G      2
22Hformato IGES Nov. 2001,32,38,6,308,15,,1.000000E+00,2,2HMM,1,        G      3
1.000000E-02,13H090904.171450,1.000000E-02,1.713160E+01,,,10,0,         G      4
13H090904.171450;                                                       G      5
128       1       0       1       0       0       0            000000000D      1
128       0       1      24       0                                   0D      2
128      25       0       1       0       0       0            000000000D      3
128       0       1      24       0                                   0D      4
[...]
128,1,8,1,2,0,1,0,0,0,0.000000E+00,0.000000E+00,1.000000E+00,          1P      1
1.000000E+00,0.000000E+00,0.000000E+00,0.000000E+00,                    1P      2
2.500000E-01,2.500000E-01,5.000000E-01,5.000000E-01,                    1P      3
7.500000E-01,7.500000E-01,1.000000E+00,1.000000E+00,                    1P      4
1.000000E+00,1.000000E+00,1.000000E+00,7.071070E-01,                    1P      5
7.071070E-01,1.000000E+00,1.000000E+00,7.071070E-01,                    1P      6
[...]
```

Estratto dal file `Lamp.igs`, uno dei modelli disponibili all'interno del progetto

Un file in formato IGES, nella sua versione testuale, è composto da sequenze di 80 caratteri ASCII denominate **record**. La struttura dei record è un'eredità del periodo delle schede perforate, così come lo è il formato **Hollerith** utilizzato per rappresentare le stringhe di testo: tale formato codifica una stringa arbitraria `s` come:

```
length(s) + H + s
```

dove `length` è la funzione che restituisce il numero dei caratteri di `s`, `H` è il carattere ASCII di indice 72 e `+` è l'operatore di concatenazione fra stringhe.

All'interno di ogni file si identificano cinque sezioni: **start** (**S**), **global** (**G**), **directory entry** (**D**), **parameter data** (**P**) e **terminate** (**T**). La sezione di appartenenza di un record è memorizzata nel carattere in colonna 73.

Il formato IGES prevede oltre trenta diversi tipi di entità grafiche: le superfici NURBS sono quelle aventi id **128** e sono descritte come *Rational B-Spline Surface Entity*. Come accade per qualsiasi altra entità, i parametri relativi alla superficie sono memorizzati in record situati sia nella sezione **directory entry** sia in quella **parameter data**. La parte più cospicua delle informazioni è contenuta nella sezione **P**; in particolare, sono memorizzati - separati da virgola - i seguenti valori, nel seguente ordine:

1. il numero di punti di controllo (per entrambe le direzioni);
2. i gradi della superficie;
3. le sequenze di nodi;
4. la sequenza di pesi relativi ai seguenti punti di controllo;
5. i punti di controllo per entrambe le direzioni.

L'applicazione realizzata effettua il *parsing* in un file in formato IGES (con estensione `.igs`) considerando esclusivamente le entità relative alle superfici NURBS ed estrapolando a partire da esse le informazioni essenziali. Questi valori vengono successivamente riorganizzati come istanza di una classe `Surface` definita all'interno del codice Javascript.

3.2 Il linguaggio Javascript

Javascript è un linguaggio di scripting di alto livello debolmente tipizzato utilizzato all'interno dei browser per gestire le interazioni *client-side* con l'utente, per comunicare in maniera asincrona con i web server e, in generale, per realizzare complesse applicazioni in grado di vivere all'interno del

mondo del web. Il linguaggio implementa le specifiche ECMA-262 definite e standardizzate all'Ecma International e, al momento in cui si scrive, l'ultima versione di Javascript è targata 1.8.5, versione che recepisce la revisione 5.1 delle specifiche ECMAScript.

Javascript è un linguaggio multiparadigma che permette di adottare diversi stili di programmazione, che spaziano dall'approccio imperativo a quello funzionale, sino all'approccio orientato agli oggetti. Quest'ultimo è stato scelto come preponderante nell'organizzazione del progetto sperimentale in quanto permette di incapsulare in classi oggetti complessi quali le superfici NURBS o gli shader, nonché garantisce un ottimale grado di chiarezza e leggibilità all'interno del codice.

3.2.1 Le librerie: jQuery e glMatrix

Il successo di Javascript, in grandissima parte, è indubbiamente dovuto al numero di librerie che sono state sviluppate nel corso dell'ultimo decennio. Queste estensioni permettono di arricchire le funzionalità di un browser, semplificando al contempo il processo di stesura del codice.

Oggigiorno sono disponibili librerie relative a qualsiasi ambito, incluso quello grafico: tra le più importanti si ricordano *three.js* e *O3D* in grado di fornire al programmatore un framework pronto all'uso per sviluppare scene tridimensionali.

Queste librerie, come è giusto che sia, forniscono un elevato livello di astrazione nascondendo allo sviluppatore l'esecuzione delle chiamate WebGL e le modalità con cui le informazioni vengono inviate alla scheda grafica. In tal senso, essendo tali librerie poco "educative" e spesso eccessivamente complete e pesanti per le reali necessità di un progetto, ai fini della realizzazione dell'applicativo si è scelto di non adottarne nessuna e di gestire in prima persona aspetti quali le procedure di rendering e l'elaborazione delle matrici prospettiche.

Le uniche librerie Javascript che sono state utilizzate sono di carattere generale. In particolare, si è fatto uso di **jQuery** - per la manipolazione del Document Object Model, per il supporto AJAX nonché per la gestione

differita di determinate funzioni asincrone - e di **glMatrix**, come supporto matematico alle operazioni vettoriali e matriciali.

3.2.2 Il futuro di Javascript: Harmony

La prossima *major release* di Javascript, targata 2.0, recepirà le novità introdotte dalle specifiche ECMAScript 6, nome in codice *Harmony*. Tali specifiche, con molta probabilità, saranno ultimate nel corso della seconda metà del 2015.

Harmony rappresenta una profonda revisione del linguaggio, introducendo decine di nuove funzionalità tra cui il concetto di modularità, un rinnovato e migliorato supporto alla programmazione orientata agli oggetti, numerose strutture dati tra cui i `Set` e le `HashMap`, gli oggetti `Promise` per simulare il multi-threading e molto altro ancora.

Alcune di queste *feature* sono già state implementate all'interno dei principali browser. A tal proposito, l'applicazione utilizza le nuove strutture dati `Float32Array` e `Uint16Array`: queste permettono di creare array tipizziti in grado di contenere, rispettivamente, numeri in virgola mobile a 32 bit e numeri interi *unsigned* a 16 bit.
Gli array in tale formato sono stati adottati per inviare alla scheda grafica le coordinate dei vertici di una scena tridimensionale e l'ordine con cui questi debbono essere renderizzati.

3.3 HTML5

HTML5 è la quinta revisione del linguaggio di markup HTML, utilizzato per strutturare e visualizzare i contenuti all'interno del Web. Frutto di un processo di stesura durato oltre sei anni, le specifiche di HTML5 sono state ufficialmente pubblicate il 28 ottobre 2014 ricevendo lo stato di *reccomandation* da parte del World Wide Web Consortium (W3C), a 17 anni di distanza dal rilascio dell'ormai obsoleta versione 4.0.

Lo sviluppo di HTML5 (e di tutte le tecnologie ad esse collegate, CSS3 in primis) si è reso necessario per fornire a sviluppatori e web designer strumenti

moderni e potenti per realizzare in modo nativo applicazioni web complesse, interattive e multimediali, in grado di operare sul più ampio numero di dispositivi.

Le principali novità di HTML5 spaziano dall'introduzione di nuovi elementi per la riproduzione nativa di file audio e video al supporto per le formule matematiche, il *drag and drop*, il *web storage* per memorizzare informazioni in modo permanente all'interno di un browser nonché la definizione di decine di nuovi *tag* per arricchire la semantica di una pagina web.

L'applicazione che è stata realizzata a supporto di questo elaborato utilizza principalmente tre nuove funzionalità di HTML5. Le prime due sono l'elemento `canvas` e l'*animation frame*, necessari per il funzionamento della libreria WebGL, mentre la terza è la Pointer Lock API, adottata per gestire il movimento della camera mediante l'uso del mouse.

3.3.1 Canvas e *animation frame*

L'elemento `canvas` è già stato trattato nel corso del primo capitolo: definisce un'area di dimensione rettangolare all'interno di una pagina che può essere popolata dinamicamente con l'ausilio di una libreria grafica, ad esempio mediante WebGL.

L'*animation frame* è una feature anch'essa pensata appositamente per il mondo della grafica in real time. Essa permette di richiedere a un browser, mediante la chiamata `requestAnimationFrame`, di eseguire una funzione passata come parametro con un periodo di 60 pulsazioni al secondo. Così facendo, è possibile definire un ciclo all'interno del quale avvengono, tra le altre cose, le operazioni di rendering di una scena tridimensionale. Tale scena verrà così aggiornata e renderizzata 60 volte al secondo.
La cadenza prevista dalla `requestAnimationFrame` è un valore fissato, dettato dal fatto che il ciclo deve essere sincronizzato con la frequenza di aggiornamento del monitor (tipicamente, 60 Hz). Tale cadenza può essere ridotta automaticamente dal browser quando la pagina è in esecuzione, ma risulta in background.

Al momento in cui si scrive, l'*animation frame* è supportata a pieno da tutti i principali browser desktop e mobile.

3.3.2 Pointer Lock API

Pointer Lock è una tecnologia correlata alle specifiche di HTML5 in grado di *catturare* e gestire il movimento del puntatore nel mouse nel corso del tempo, ovvero permette di ottenere il delta di spostamento del cursore all'interno di un elemento (e non solamente la posizione assoluta, come previsto nelle prime versioni di HTML). Le specifiche dell'API sono attualmente una *candidate reccomandation* del W3C, ma sono già state ampiamente adottate dai principali browser in ambiente desktop.

Pointer Lock permette di catturare e nascondere il puntatore del mouse mediante la chiamata `requestPointerLock`, nonché di gestirne il movimento anche al di fuori dei confini dell'elemento che ha richiesto il controllo del cursore; tale controllo deve essere rilasciato esplicitamente dall'applicazione mediante il comando `exitPointerLock`.

Le specifiche dell'API arricchiscono l'oggetto Javascript `MouseEvent` (ottenibile *ascoltando* gli eventi relativi al mouse in una pagina, ad esempio `mousemove`) con le proprietà `movementX` e `movementY`, le quali rappresentano l'intervallo di spostamento del puntatore in entrambe le direzioni. Ciascuna proprietà può essere vista come la differenza tra le coordinate assolute del cursore in due eventi `mousemove` successivi; ad esempio, dati gli oggetti `eNow` e `ePrevious`, istanze della classe `MouseEvent`, si definisce `movementX` (rispettivamente `movementY`) come:

```
movementX = eNow.screenX - ePrevious.screenX
```

con `screenX` (`screenY`) posizione assoluta del puntatore lungo l'asse X (oppure Y).

All'interno dell'applicazione, Pointer Lock API ha permesso di implementare una camera con visuale in prima persona. Nei browser che ancora non supportano questa tecnologia (fra cui Internet Explorer) è comunque possibile utilizzare le frecce direzionali della tastiera per gestire l'orientamento dell'inquadratura al pari di quanto accade utilizzando il mouse.

Capitolo 4

Il progetto: implementazione

A supporto dell'elaborato è stata prodotta un'applicazione web con l'ausilio della libreria WebGL al fine di sperimentare sul campo i modelli di *shading* trattati in precedenza. Il progetto si compone di:

- due librerie Javascript di supporto (`glMatrix` e `jQuery`), situate all'interno della sottocartella `lib`;
- tre librerie Javascript di produzione propria per il *parsing* dei file IGES, la valutazione delle superfici NURBS e la gestione dei colori in formato HSV, anch'esse contenute all'interno di `lib`;
- sette file Javascript contenenti il corpo del programma organizzato in classi, presenti all'interno della sottocartella `js`;
- nove modelli in formato IGES con estensione `.igs` (provenienti dall'archivio NURBS di XCModel), situati in `assets/models`;
- otto coppie (vertex/fragment) di shader, di cui una dedicata al sottosistema tesuale (situati in `assets/shaders`);
- cinque texture in formato PNG, delle quali una contiene i caratteri previsti dalla codifica ASCII estesa (situate in `assets/textures`)
- un file JSON contenente la dimensione e la spaziatura di ciascun glifo della tabella ASCII di cui sopra, situato in `assets/fonts`;

- una coppia di file HTML/CSS che struttura la pagina all'interno della quale viene eseguito il progetto.

L'applicazione si avvia automaticamente al completamento del caricamento della pagina HTML. Si compone essenzialmente di tre parti:

1. una fase di riconoscimento e verifica della libreria WebGL;
2. una fase di inizializzazione in cui gli shader vengono caricati e compilati, le texture vengono predisposte e i modelli IGES vengono valutati e convertiti in griglie di punti e poligoni. Successivamente il sistema inizializza i renderer grafici, la camera e il sottosistema di testo e predispone l'ambiente HTML per ricevere input da tastiera e da mouse;
3. un loop all'interno del quale la scena tridimensionale viene aggiornata e renderizzata con una frequenza media di 60 frame per secondo.

4.1 Il riconoscimento della libreria WebGL

La fase di riconoscimento si compone di tre distinti stadi di identificazione che rispondono ai seguenti quesiti:

1. il browser implementa la libreria WebGL?
2. la libreria WebGL è abilitata?
3. il layer di astrazione grafico ANGLE è disponibile? Se sì, è abilitato?

Inizialmente lo script determina lo User Agent del browser in esecuzione e verifica se la libreria WebGL è implementata accertandosi che l'oggetto Javascript `window` preveda la proprietà `WebGLRenderingContext` (il contesto dedicato alla grafica tridimensionale).
L'esito del controllo, così come di quelli successivi, viene loggato all'interno della console del browser; se questo controllo risulta negativo, l'esecuzione termina con fallimento.

In caso contrario, lo script prosegue recuperando l'oggetto Javascript relativo al canvas avente id `#glCanvas` e tentando, a partire da esso, di ottenere il contesto WebGL.

```
canvas = document.getElementById('myCanvas');
var isEnabled = ((gl = getWebGLContext(canvas)) != null);
```

La procedura locale `getWebGLContext` invoca il metodo `getContext` previsto dall'oggetto `canvas` su tutte le keyword conosciute relative a WebGL fintanto che il contesto richiesto non viene restituito. Le keyword utilizzate sono le seguenti:

- `webgl`, il valore standard previsto dalle specifiche;
- `experimental-webgl`, il valore utilizzato dai browser nel periodo in cui le specifiche erano ancora in fase di definizione;
- `webkit-3d`, il valore utilizzato in passato dai browser basati su motore WebKit e derivati;
- `moz-webgl`, il valore utilizzato in passato da Mozilla Firefox.

Se `getWebGLContext` restituisce valore nullo si deduce che la libreria è correttamente implementata ma non abilitata: pertanto l'esecuzione termina con fallimento.

L'ultimo stadio verifica se è presente il layer di astrazione grafico ANGLE e, in caso affermativo, verifica se questo è al momento abilitato. La funzione locale `getANGLEStatus` restituisce il valore `NOT_IMPLEMENTED` (`-1`) se il sistema operativo non appartiene alla famiglia Windows oppure se il browser in esecuzione è Internet Explorer; in caso contrario, verifica se il software è abilitato o meno esaminando il range in ampiezza delle linee non soggette ad *anti-aliasing* (una proprietà, questa, di WebGL). Se tale range oscilla tra un valore minimo di 1 e un valore massimo di 1 allora è possibile concludere che ANGLE è abilitato.

```
var aliasedLineWidthRange =
gl.getParameter(gl.ALIASED_LINE_WIDTH_RANGE);
```

Il range è ottenuto mediante il metodo `getParameter` previsto dal contesto WebGL. Lo script infine comunica il numero di estensioni WebGL attualmente abilitate e successivamente termina, concludendo la fase di riconoscimento.

4.2 La fase di inizializzazione

Una volta conclusasi con successo la fase di riconoscimento della libreria WebGL, l'applicazione carica mediante chiamate AJAX tutti i file Javascript e le librerie necessarie per procedere con l'inizializzazione di ciascuna classe e componente del programma.

4.2.1 Shader

I programmi **shader** sono incapsulati in una omonima classe `Shader` che, in fase di istanziazione, prevede come unico parametro una stringa di testo che identifica il nome del programma.

Invocando il metodo `loadAndCompile` di un oggetto `Shader`, l'applicazione recupera mediante chiamante AJAX il codice sorgente sia del *vertex* sia del *fragment shader* corrispondenti al nome indicato. La routine `create`, lanciata successivamente, si occupa di compilare ciascuno shader (chiamata `compile`) e di eseguire l'operazione di *linking* (chiamata `link`) ottenendo così un programma shader pronto per essere utilizzato a livello di hardware grafico.

```
Shader.prototype.loadAndCompile = function()
{
    var shaderRootUrl = SHADER_PATH + "/" + this.name;
    var shader = this;

    return $.when(
        $.ajax(shaderRootUrl + ".vertex.glsl"),
        $.ajax(shaderRootUrl + ".fragment.glsl")
    ).then(function(vertexShader, fragmentShader)
    {
        var vertexShaderSrc = vertexShader[0];
        var fragmentShaderSrc = fragmentShader[0];

        shader.create(vertexShaderSrc, fragmentShaderSrc);
    });
};
```

Il metodo `loadAndCompile` della classe `Shader`

Il codice sorgente degli shader risiede all'interno del percorso `assets/shaders` e ciascun file è nella forma `<nome>.{vertex;fragment}.glsl`. Il metodo `create` inoltre scorre e mappa la lista di parametri di tipo *attribute* e *uniform* definiti all'interno degli shader per facilitare successive le operazioni di invio di dati alla scheda grafica.

4.2.2 Texture

Le **texture** vengono anch'esse incapsulate in oggetti di tipo `Texture` (identificati da un nome) i quali prevedono un unico metodo denominato `load`.
Questo metodo recupera mediante chiamata AJAX il file immagine in formato `.png` presente all'interno della cartella `assets/textures` (o, se specificato, in una delle sue sottocartelle); successivamente inizializza la texture a livello di libreria WebGL, specificando che si tratta di una texture 2D in formato RGBA, impostando i dovuti filtri per le operazioni di *scaling* e generando la *mipmap*.

```
gl.bindTexture(gl.TEXTURE_2D, texture.texture);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, minFilter);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, maxFilter);
gl.generateMipmap(gl.TEXTURE_2D);
gl.bindTexture(gl.TEXTURE_2D, null);
```

L'inizializzazione a livello WebGL di una texture
all'interno del metodo `load` della classe `Texture`

4.2.3 Modelli NURBS

All'interno dell'applicazione, i modelli IGES sono istanze della classe `Model` i cui attributi sono una stringa di testo (l'identificativo del modello) e una collezione di oggetti di tipo `Surface` i quali rappresentano le superfici NURBS.

Ciascun modello dispone di un metodo `loadAndParse` responsabile di recuperare il corrispondente file `.igs` e di sottoporlo all'azione del *parser* IGES il quale, a partire dal contenuto testuale dell'elemento, estrapola i parametri relativi a ciascun superficie restituendo un array di oggetti di tipo `Surface`.

```
Model.prototype.loadAndParse = function()
{
    var model = this;
    var fileName = this.name + "." + Iges.EXTENSION;
    var modelUrl = MODEL_PATH + "/" + fileName;

    return $.ajax(modelUrl).then(function(src) {
        model.surfaces = Iges.parse(src);
    });
};
```

Il metodo `loadAndParse` della classe `Model`:
`model.surfaces` è l'elenco di superfici che compongono il modello

All'interno di `loadAndParse` l'applicazione fa uso della libreria `Iges` (definita nel file `Iges.js` in `lib`) la quale considera esclusivamente le entità con id 128, scartando eventuali altre tipologie. Il *parser* è parzialmente derivato da una libreria Javascript prodotta da Takahito Tejima, sviluppatore dello studio di animazione Pixar.

La classe `Surface` prevede un importante metodo denominato `update` il quale, presi in input due parametri che definiscono l'ampiezza della griglia di punti che si desidera ottenere, elabora i parametri NURBS della superficie generando, tra le altre cose, i segmenti, le facce triangolari, i vettori normali e le coordinate UV.

```
Surface.prototype.update = function(stacks, slices)
{
    var nurbs = Nurbs.evaluate(this, stacks, slices);

    var points = nurbs.points;
    var lines = nurbs.lines;
    var triangles = nurbs.triangles;
    var normals = nurbs.normals;
    var uvCoords = nurbs.uvCoords;
    [...]
}
```

L'inizio del metodo `update` della classe `Surface`

La valutazione di una superficie NURBS è presa in carica dalla libreria `Nurbs` (definita nell'omonimo file in `lib`) la quale implementa le procedure di calcolo utilizzate all'interno della suite XCModel realizzata dal prof. Giulio Casciola del dipartimento di Matematica dell'Università di Bologna.

Il metodo `evalute`, invocato all'interno di `update`, è una variazione della funzione `griglia_psp` presente in XCModel; rispetto alla procedura originale, `evalute` identifica e debella le primitive degeneri al momento della loro creazione, nonché provvede a calcolare i vettori normali (a livello di faccia così come di vertice) e le coordinate UV per la mappatura texture. Inoltre, al fine di ridurre il carico di informazioni da inviare alla GPU, i segmenti e i triangoli sono definiti come insieme di indici e non di coordinate.

Tutti i valori sono organizzati come array tipizzati (`Float32Array` oppure `Uint16Array`) e pertanto pronti per essere utilizzati da un oggetto *renderer*.

4.2.4 Camera

L'applicazione implementa una camera di tipo *free-flying* con visuale in prima persona e due gradi di libertà, ovvero capace di ruotare attorno al proprio asse X e Y. Essa è definita attraverso la classe `Camera` ed è controllata mediante gli input congiunti di mouse e tastiera (oppure esclusivamente tramite la tastiera nei browser che non supportano la Pointer Lock API).

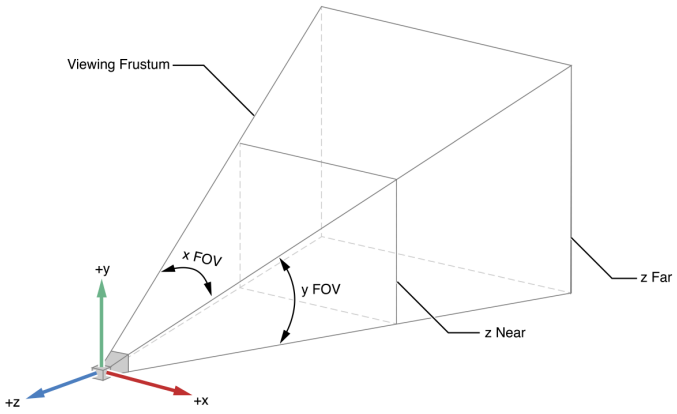

Figura 4.1: La camera nello spazio tridimensionale

La camera è responsabile di generare e aggiornare - a ogni spostamento o rotazione - le matrici che definiscono la sezione attualmente visibile della scena, sezione rappresentabile come un tronco di cono prospettico all'interno dello spazio tridimensionale (come esplicitato in figura 4.1).
Una camera, pertanto, è caratterizzata da un angolo di visione (in WebGL si considera il *field of view* lungo l'asse Y) e da due parametri (`zNear` e `zFar`, detti valori di *clipping*) i quali definiscono le soglie oltre le quali gli oggetti non vengono renderizzati in quanto troppo vicini (rispettivamente, troppo lontani) rispetto alla posizione dell'osservatore.

Le matrici definite dall'oggetto di tipo `Camera` permettono, a livello di vertex shader, di trasformare i vertici che compongono un modello, predisponendoli per essere sottoposti ai test (*depth test* in primis) previsti dalla pipeline grafica e, successivamente, visualizzandoli all'interno del canvas.

4.2.5 Gli oggetti *renderer*

L'applicazione prevede due classi, `ModelRenderer` e `TextRenderer`, incaricate di svolgere le operazioni di inoltro alla scheda grafica di valori quali le posizioni dei vertici, i vettori normali, le coordinate UV, le matrici prospettiche, il colore di una superficie o la posizione della sorgente luminosa.
Entrambi le classi sono figlie di una classe genitrice `Renderer` la quale definisce due metodi essenziali: `prepare` e `render`.

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, renderer.indexBuffer);
var length;

switch (renderMode)
{
    case gl.LINES:
    {
        length = surface.glLineIndices.length;
        gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
                      surface.glLineIndices, gl.STATIC_DRAW);
    } break;

    case gl.TRIANGLES:
    {
        length = surface.glTriangleIndices.length;
        gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
                      surface.glTriangleIndices, gl.STATIC_DRAW);
    } break;
}

gl.drawElements(renderMode, length, gl.UNSIGNED_SHORT, 0);
```

Estratto dal metodo `render` della classe `ModelRenderer`

La classe `ModelRenderer`, come suggerisce il nome, effettua le operazioni di disegno di un modello NURBS definito come insieme di superficie.
Il metodo `ModelRenderer.prepare` imposta il programma shader (ed eventualmente la texture) da utilizzare per le chiamate WebGL successive, nonché invia alla GPU le matrici prospettiche e la posizione della sorgente luminosa come parametri di tipo *uniform*. Il metodo `ModelRenderer.render`, invece, prende in input un oggetto di tipo `Model` e ne effettua il rendering in una delle due modalità previste dall'applicazione:

- *wireframe*, ovvero viene tracciata la griglia di punti che compongono il modello mediante i segmenti calcolati in fase di valutazione NURBS;
- *solid*, ovvero viene visualizzato il modello per intero tramite i triangoli, i vettori normali e le coordinate UV; in questa modalità è possibile apprezzare l'effetto dei modelli di illuminazione e applicare una texture alla superficie.

Le informazioni a livello di vertice (posizione, vettore normale e coordinate UV) sono inviate alla scheda grafica in modalità *interleaved*, ovvero come record di 32 byte (8 valori *float* da 4 byte l'uno) dove:

- i valori di indice 0, 1 e 2 sono le coordinate XYZ della posizione del vertice;
- i valori di indice 3, 4, 5 sono le componenti XYZ del vettore normale calcolato nel vertice;
- i valori di indice 6 e 7 sono le coordinate UV.

Tale approccio permette di ottimizzare e velocizzare notevolmente le operazioni di rendering. Mediante la chiamata `vertexAttribPointer` prevista dalla libreria WebGL è possibile "istruire" l'hardware grafico affinché estrapoli correttamente i parametri *attribute* dai record, come mostrato di seguito.

```
gl.bindBuffer(gl.ARRAY_BUFFER, renderer.vertexBuffer);
gl.bufferData(gl.ARRAY_BUFFER, surface.glVertexData, gl.STATIC_DRAW);
renderer.shader.vertexAttribPointer("aPosition",
                VEC3_LENGTH, VERTEX_DATA_LENGTH, 0);
renderer.shader.vertexAttribPointer("aNormal",
                VEC3_LENGTH, VERTEX_DATA_LENGTH, VEC3_LENGTH);
renderer.shader.vertexAttribPointer("aTexCoord",
                VEC2_LENGTH, VERTEX_DATA_LENGTH, 2 * VEC3_LENGTH);
```

Il sottosistema testuale

Come accennato in precedenza, l'applicazione prevede un secondo oggetto *renderer* denominato `TextRenderer` il quale implementa un sottosistema testuale in grado di visualizzare all'interno del canvas, in sovrimpressione, qualsiasi stringa definita mediante la codifica ASCII estesa.

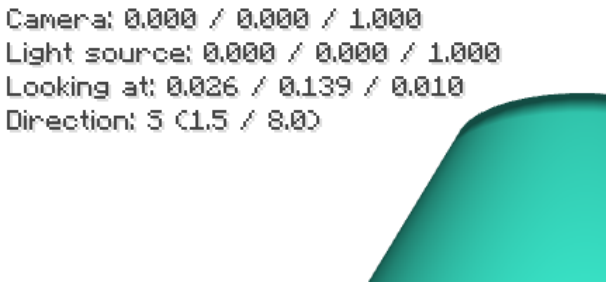

Figura 4.2: Il *TextRenderer* in azione

L'alfabeto è contenuto in una texture di dimensione 128x128 pixel situata nella sottocartella `fonts` di `assets/textures`; il `TextRenderer` viene inizializzato dall'applicazione con tale texture. La classe prevede inoltre il metodo `loadGlyphSizes` che permette di recuperare, mediante chiamata AJAX, un file JSON contenente le informazioni di spaziatura relative a ciascun carattere.

Il metodo `TextRenderer.render` prende in input una stringa e un vettore a due componenti il quale rappresenta la posizione XY all'interno del canvas (quindi non a livello di spazio tridimensionale) in cui si vuole visua-

lizzare il contenuto testuale. La funzione si appoggia al metodo `renderChar`, anch'esso previsto dalla classe `TextRenderer`, il quale, per ciascun carattere che compone la stringa, restituisce il glifo corrispondente estrapolato dalla texture.

```
TextRenderer.prototype.render = function(
                        text, x, y, color, dropShadow)
{
    [...]
    var position = vec3.fromValues(x, y, 0.0);

    for (var i = 0; i < text.length; i++)
    {
        var codeUnit = text.charCodeAt(i);
        position[0] += this.renderChar(codeUnit, position,
                                       color, dropShadow);

    }
};
```

Il metodo `render` della classe `TextRenderer`

4.3 Il loop grafico

Terminata la fase di inizializzazione, il ciclo di vita dell'applicazione è governato dalla funzione `loop` invocata mediante la chiamata `requestAnimation Frame` prevista dalle specifiche HTML5. Ogni iterazione si compone di due operazioni distinte, il `tick` e il `render`:

- il `tick` interviene sulla logica del programma, aggiornando lo stato e le proprietà degli oggetti presenti all'interno della scena; ad esempio, a ogni iterazione viene shiftata la tonalità del colore delle facce di ciascuna superficie in fase di rendering. La fase `tick` è inoltre quella che permette di assimilare gli input da mouse e tastiera;

- il `render` compone e visualizza la scena, mediante gli oggetti `ModelRenderer` e `TextRenderer`.

L'applicazione è impostata per svolgere 60 operazioni di `tick` al secondo, ma la frequenza del clock può essere modificata arbitrariamente in modo tale che il numero di aggiornamenti per secondo della scena sia superiore o inferiore (in ogni caso, svincolato) al numero di frame per secondo.

4.4 L'ambiente di sviluppo

L'applicazione è stata realizzata in ambiente Windows con l'ausilio della piattaforma di sviluppo **JetBrains WebStorm**, rilasciata in licenza *educational* per un anno agli studenti delle principali università scientifiche.

Per quanto concerne il versionamento del codice sorgente, è stato inizializzato un repository pubblico sulla piattaforma **Bitbucket** gestita da **Atlassian**; la gestione dei *commit* e tutte le operazioni GIT sono state effettuate mediante l'applicativo **SourceTree**, anch'esso distribuito da Atlassian.
Il repository è reperibile all'indirizzo `https://bitbucket.org/fstefani/progetto/`

Capitolo 5

Il progetto: sperimentazione

Il progetto è stato testato su una piattaforma dotata di scheda grafica dedicata AMD Radeon HD 4850 (con supporto a OpenGL 3.3) in presenza di due sistemi operativi, Windows 8.1 e Ubuntu 14.04 LTS, entrambi a 64 bit. I browser utilizzati sono stati Google Chrome 40, Mozilla Firefox 36, Internet Explorer 11 e Opera 27.

5.1 Fase di riconoscimento

In fase di riconoscimento della libreria WebGL si sono ottenuti i seguenti risultati:

- **Google Chrome**: WegGL implementato alla versione 1.0 e abilitato di default, ANGLE presente e abilitato di default;
- **Mozilla Firefox**: WegGL implementato alla versione 1.0 e abilitato di default, ANGLE presente e abilitato di default;
- **Internet Explorer** (solo su piattaforma Windows): WegGL parzialmente implementato alla versione 0.94 e abilitato di default, ANGLE non presente in quanto utilizza un proprio layer di astrazione grafico non disabilitabile;

- **Opera**: risultati analoghi a quelli riscontrati con Google Chrome, essendo anch'esso basato (a partire dalla versione 15) sul progetto Chromium.

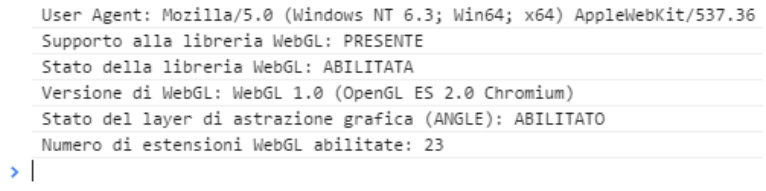

Figura 5.1: L'output della fase di riconoscimento

Lo script di riconoscimento ha rilevato correttamente lo stato di WebGL anche quando questo è stato forzatamente disabilitato a livello di browser. La disattivazione è stata effettuata come segue:

- su **Google Chrome** e **Opera**, abilitando l'*esperimento* all'indirizzo `chrome://flags/#disable-webgl` e riavviando il browser;.
- su **Mozilla Firefox**, ponendo `true` il valore `webgl.disabled` nel registro raggiungibile all'indirizzo `about:config`.

Non risulta al momento possibile disabilitare la libreria WebGL all'interno di Internet Explorer. Lo script ha inoltre rilevato correttamente lo stato del layer di astrazione grafico ANGLE quando questo è stato forzatamente disabilitato a livello di browser. La disattivazione è stata effettuata come segue:

- per quanto riguarda **Google Chrome** e **Opera**, lanciando il browser con il flag `--use-gl=desktop`;
- su **Mozilla Firefox**, ponendo `true` i valori `webgl.disable-angle` e `webgl.force-enabled` nel registro raggiungibile all'indirizzo `about:config`.

L'intera procedura di riconoscimento è stata testata anche su altre configurazioni hardware desktop, oltre che su piattaforme mobili.

5.2 Elaborazione grafica

La sperimentazione della fase di elaborazione grafica dell'applicativo ha preso in esame principalmente i seguenti aspetti:

1. la bontà, in termini di resa visiva, dei modelli di illuminazione e la precisione, in termini di corretta mappatura, nelle procedure di applicazione texture;
2. i tempi di riconoscimento della libreria WebGL, di inizializzazione e di elaborazione NURBS, nonché il numero medio di frame per secondo, per quanto riguarda l'ambito prestazionale.

I seguenti modelli NURBS in formato IGES sono stati scelti come soggetti di test:

- `Lamp16`, dotato di 10 superfici e 136 punti di controllo;
- `Wazowsky`, dotato di 35 superfici e 560 punti di controllo;
- `Stuka`, dotato di 48 superfici e 644 punti di controllo;
- `Vespa`, dotato di 110 superfici e 1493 punti di controllo;

I modelli sono stati selezionati con un numero crescente di superfici e punti di controllo dall'archivio NURBS di XCModel. Inoltre, i test sono stati condotti interagendo con un'istanza remota dell'applicativo raggiungibile all'URL `http://fstefani.web.cs.unibo.it/tesi/`; si è preferito tale approccio a quello in locale per considerare anche l'impatto delle chiamate asincrone AJAX in fase di inizializzazione.

5.2.1 Analisi qualitativa

Per quanto riguarda l'aspetto qualitativo, in fase di testing non si sono rilevate anomalie visive o artefatti grafici. I modelli appaiono solidi, omogenei e non sono apprezzabili imprecisioni nei punti di contatto fra le superfici che li compongono. I vettori normali calcolati a livello di vertice in fase di

elaborazione risultano validi, permettendo una corretta applicazione dei modelli di illuminazione.

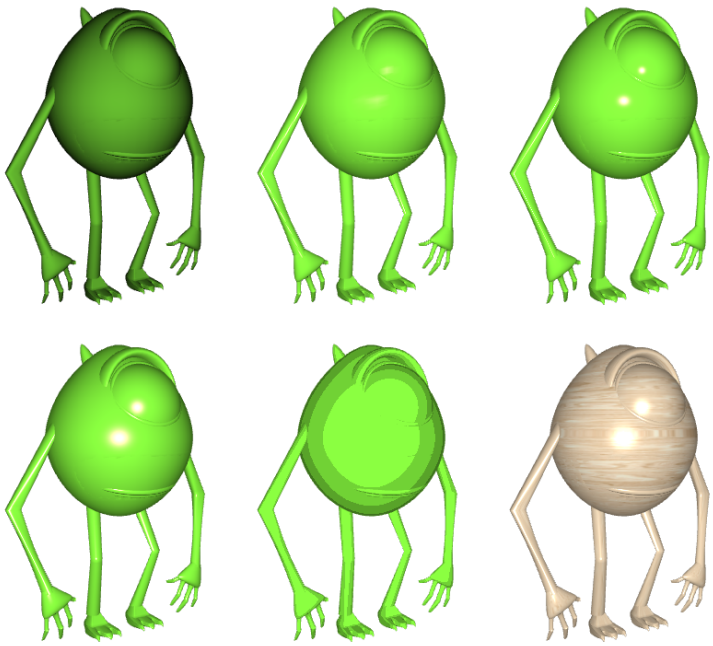

Figura 5.2: L'applicazione di shader e texture al modello *Wazowsky*

Come osservabile in figura 5.2, al variare dello shader applicato al modello in fase di rendering è possibile notare, ad esempio, come il riflesso speculare appaia maggiormente definito nel passare dal modello Blinn applicato a livello di vertice (riquadro centrale, prima riga) al medesimo modello applicato a livello di fragment (riquadro di destra, prima riga), o ancora come lo stesso riflesso speculare irrompa maggiormente sulla superficie adottando il modello Blinn-Phong (riquadro di sinistra, seconda riga).

È inoltre possibile apprezzare le differenze che intercorrono passando da un modello all'altro quando si fissa la sorgente luminosa. Ad esempio, in figura 5.3 è lampante lo scarto che si ha nel passare dal modello *diffuse* a uno qualsiasi dei modelli della famiglia Blinn per quanto concerne il grado di luminosità delle zone in ombra (che, nel primo caso, appaiono quasi com-

pletamente nere).

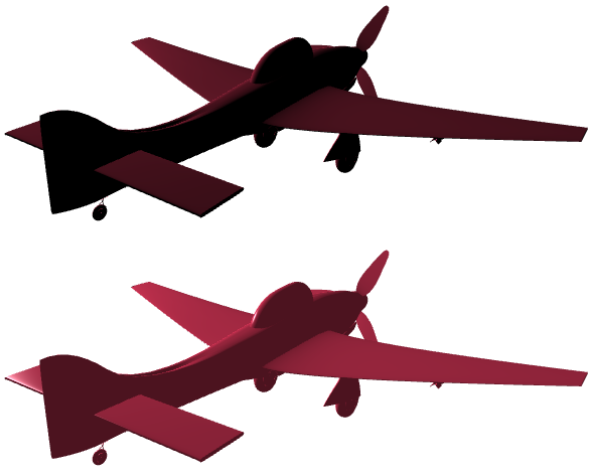

Figura 5.3: I modelli *diffuse* e ADS applicati all'oggetto *Stuka*

Anche la procedura di mappatura texture non ha portato alcuna anomalia e le coordinate UV calcolate in fase di elaborazione risultano anch'esse corrette. Le texture sono applicate su tutta l'area di ciascuna superficie che compone un modello e gestite in fase di *scaling* con un filtro di tipo lineare.

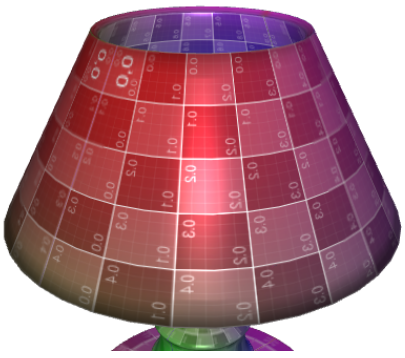

Figura 5.4: La texture di riferimento applicata all'oggetto *Lamp16*

5.2.2 Analisi prestazionale

Dal punto di vista delle prestazioni e dell'efficienza, i test svolti hanno monitorato la durata della fase di inizializzazione dell'applicativo e il valore medio di frame per secondo in presenza di varie configurazioni e mediante l'utilizzo dei principali browser.

I modelli NURBS sono stati elaborati con un crescente livello di dettaglio, utilizzando griglie parametriche di dimensione 16x16, 32x32, 64x64 e 128x128. Per ciascuna configurazione e per ciascun browser, i test sono stati ripetuti dieci volte ed è stata riportata la media aritmetica eliminando eventuali rilevazioni anomale.

I valori relativi alla fase di inizializzazione sono da considerarsi non comprensivi del tempo di riconoscimento della libreria WebGL, un valore questo pressoché stabile nell'arco di tutti i test effettuati e che oscilla tra un minimo di 5 ms e un massimo di 15 ms.

	16x16	**32x32**	**64x64**	**128x128**
Google Chrome	519	718	1664	6612
Mozilla Firefox	545	693	1242	3413
Internet Explorer	750	1366	3647	13578
Opera	487	654	1535	5189

Tabella 5.1: Fase di inizializzazione (in millisecondi) per il modello *Lamp*

I test svolti mostrano un incremento pressoché esponenziale dei tempi di elaborazione all'aumentare del grado di definizione di un modello NURBS. Mozilla Firefox si è dimostrato essere il browser più performante, mentre Internet Explorer il più lento sin dalle prime iterazioni (un risultato questo che non può essere imputabile alla parziale implementazione di WebGL in quanto le operazioni svolte in questa fase non coinvolgono ancora la libreria grafica).

	16x16	32x32	64x64	128x128
Google Chrome	60	58	39	28
Mozilla Firefox	60	60	41	32
Internet Explorer	33	21	14	9
Opera	60	59	35	24

Tabella 5.2: Numero di frame per secondo per il modello *Lamp*

Per quanto concerne il numero medio di frame per secondo, Google Chrome ha ottenuto il primato in tutti i test svolti, mentre il browser peggiore è stato nuovamente Internet Explorer.

L'andamento rilevato nei test svolti per il modello `Lamp16`, i cui risultati sono contenuti nelle tabelle precedenti, si è confermato valido anche con i successivi (e più complessi) modelli. Nello specifico, Mozilla Firefox ha consolidato la sua leadership come browser più prestante a livello di operazioni di calcolo Javascript, mentre Google Chrome ha continuato a fornire i valori FPS più alti, dimostrando di disporre della più efficiente implementazione della libreria WebGL.

	16x16	32x32	64x64	128x128
Google Chrome	766	1465	6322	27418
Mozilla Firefox	739	1387	4005	15979
Internet Explorer	1658	4382	15258	60193
Opera	710	1452	5444	23482

Tabella 5.3: Fase di inizializzazione (in millisecondi) per il modello *Stuka*

I test sono stati svolti sia in ambiente Windows sia in ambiente Linux, in presenza della medesima configurazione hardware. Per quanto riguarda le tempistiche in fase di inizializzazione non si segnalano differenze degne di nota, a testimonianza del fatto che un incremento prestazionale si può

avere agendo a livello di browser (e di motore Javascript) e non di sistema operativo. Diverso il discorso relativo al numero di frame per secondo. Su piattaforma Linux, la quale non dispone delle librerie DirectX e pertanto i browser non si avvalgono del layer di astrazione grafico ANGLE, l'applicazione ha rilevato un numero di FPS inferiore a quello ottenuto sul medesimo browser in ambiente Windows.

	16x16	**32x32**	**64x64**	**128x128**
Google Chrome	59	55	31	22
Mozilla Firefox	58	53	28	20
Opera	58	52	30	22

Tabella 5.4: Numero di frame per secondo per il modello *Lamp*, in ambiente Linux o comunque senza l'intervento del layer di astrazione grafico ANGLE

Questo risultato, verificabile anche in ambiente Windows disabilitando ANGLE come descritto nella sezione 5.1, testimonia quanto sia vantaggioso convertire le chiamate WebGL nelle equivalenti DirectX, se disponibili, e giustifica l'adozione di un layer di astrazione grafico per sopperire, si suppone, a una lacunosa implementazione di OpenGL all'interno del sistema operativo di casa Microsoft.

Conclusioni

WebGL è un tecnologia che ha visto una rapida ascesa nel corso dell'ultimo quinquennio. Complice un elevato grado di maturità raggiunto sin dalla prima release e un approccio tradizionale che non si distacca da quello delle sorelle maggiori, la più giovane delle librerie di casa Khronos ha saputo conquistare fin da subito l'interesse di sviluppatori, web designer, artisti e game maker, come testimonia il numero impressionante di progetti visionabili nella piattaforma `Chrome Experiments` gestita da Google.

Ad oggi la libreria è stata sposata da tutti i maggiori browser (Microsoft ha dichiarato di voler migliorare il supporto a WebGL nel futuro browser Spartan che vedrà la luce con Windows 10), sia in ambiente desktop che in ambiente mobile, e alcuni team di sviluppo sono già al lavoro per implementare le prime bozze delle specifiche della release 2.0, la quale dovrebbe debuttare nel corso del 2015.

Nel complesso, WebGL ha dimostrato di saper essere una libreria potente, flessibile e affidabile e ha permesso, senza alcuna difficoltà, l'implementazione di tutti i modelli di illuminazione e le tecniche di applicazione texture descritte all'interno del presente elaborato. Le carenze di gioventù, a livello di funzionalità mancanti rispetto alla libreria madre OpenGL, non hanno compresso in alcun modo il processo di sviluppo e, quando necessario, sopperire a tali mancanze non ha comportato sforzi di notevole rilevanza.

Alla luce dei risultati ottenuti in fase di sperimentazione, l'applicazione realizzata risulta essere stabile e visivamente attendibile, non generando alcun tipo di artefatto in fase di *rendering* e reagendo correttamente agli input ricevuti.

Il principale difetto riscontrabile è un eccessivo tempo di caricamento dovuto alle dispendiose operazioni di valutazione delle superficie NURBS in fase di inizializzazione. Tale problema potrà verosimilmente essere arginato in futuro quando la libreria permetterà di delegare alla scheda grafica operazioni delicate quali la tassellazione o la generazione di geometrie, agendo su appositi stadi della pipeline di rendering che attualmente non è possibile programmare mediante WebGL.

In aggiunta a ciò, è plausibile affermare che le future versioni di Javascript e la continua evoluzione dei relativi motori di *scripting* integrati all'interno dei browser porteranno un incremento prestazionale consistente, specie con l'adozione di tecnologie quali *asm.js* che renderanno più appetibile la conversione di applicazioni OpenGL in *web app* basate su WebGL.

In attesa che tutte queste novità vedano la luce, l'unica strada percorribile prevede un'ottimizzazione maniacale e ragionata delle chiamate grafiche, dell'ordine con cui queste vengono eseguite e delle modalità di trasmissione delle informazioni alla GPU; una strada realizzabile principalmente agendo in prima persona sulla libreria WebGL, senza alcun tipo di framework o di intermediario, per avvalersi a pieno della potenza di calcolo sprigionabile dall'hardware grafico

Se tale approccio non risulta sufficientemente efficace, è possibile valutare la realizzazione di un'architettura di tipo client-server, delegando a un secondo attore le procedure di valutazione di una superficie NURBS e la generazione di tutte le primitive grafiche. Una soluzione di questo tipo, però, necessita di accurati studi per valutare i tempi di latenza e la dimensione dei dati (in particolar modo, dal server al client) in fase di trasmissione.

Bibliografia

[1] Khronos Group, *WebGL: OpenGL ES 2.0 for the Web*, sito ufficiale, ultima visita il 23 febbraio 2015, `https://www.khronos.org/webgl/`

[2] Khronos Group, *WebGL: Public Wiki*, documentazione, ultima visita il 23 febbraio 2015, `https://www.khronos.org/webgl/wiki/Main_Page`

[3] Khronos Group, *WebGL Specification, Version 1.0.3*, specifiche ufficiali, 2014, `https://www.khronos.org/registry/webgl/specs/1.0/`

[4] Khronos Group, *WebGL 2 Specification*, bozza di specifiche, 2015 `https://www.khronos.org/registry/webgl/specs/latest/2.0/`

[5] Vladimir Vukićević, *Canvas 3D: GL power, web-style*, blog, 2007, `http://web.archive.org/web/20140222194911/http://blog.vlad1.com/2007/11/26/canvas-3d-gl-power-web-style/`

[6] Khronos Group, *Khronos Launches Initiative to Create Open Royalty Free Standard for Accelerated 3D on the Web*, rassegna stampa, 2009, `https://www.khronos.org/news/press/khronos-launches-initiative-for-free-standard-for-accelerated-3d-on-web`

[7] Khronos Group, *The OpenGL ES Shading Language*, specifiche ufficiali, 2009, `https://www.khronos.org/files/opengles_shading_language.pdf`

[8] World Wide Web Consortium, *HTML5: a vocabulary and associated APIs for HTML and XHTML*, reccomandation, 2014, `http://www.w3.org/TR/html5/`

[9] World Wide Web Consortium, *HTML Canvas 2D Context*, candidate reccomandation, 2014, `http://www.w3.org/TR/2dcontext/`

[10] David Wolff, *OpenGL 4.0 Shading Cookbook*, manuale, 2011

[11] D. Shreiner, G. Sellers, J. Kessenich, B. Licea-Kane, *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, manuale, 2013

[12] Khronos Group, *OpenGL ES 1.0: The Standard for Embedded Accelerated 3D Graphics*, sito ufficiale, ultima visita il 24 febbraio 2015, `https://www.khronos.org/opengles/1_X`

[13] Khronos Group, *Blacklists and whitelists*, documento ufficiale, ultima visita il 24 febbraio 2015, `https://www.khronos.org/webgl/wiki/BlacklistsAndWhitelists`

[14] Google Inc, *ANGLE: Almost Native Graphics Layer Engine*, sito ufficiale, ultima visita il 24 febbraio 2015, `https://code.google.com/p/angleproject/`

[15] Khronos Group, *WebGL Extension Registry*, specifiche ufficiali, ultima visita il 23 febbraio 2015, `https://www.khronos.org/registry/webgl/extensions/`

[16] K. Dempski, E. Viale, *Advanced Lighting and Materials With Shaders*, monografia, 2004

[17] L. Piegl, W. Tiller, *The NURBS Book*, monografia, 1997

[18] R. Nagel, W. Braithwaite, P. Kennicott, *Initial Graphics Exchange Specification IGES*, specifiche ufficiali, 1980

[19] Nick Desaulniers, *Raw WebGL*, presentazione, ultima visita il 26 febbraio 2015, `http://nickdesaulniers.github.io/RawWebGL/`

[20] Autori vari, *three.js: Javascript 3D library*, sito ufficiale, ultima visita il 26 febbraio 2015, `http://threejs.org/`

[21] Mozilla Developer Network, *JavaScript*, documentazione, ultima visita il 26 febbraio 2015, `https://developer.mozilla.org/en-US/docs/Web/JavaScript`

[22] ECMA International, *ECMA-262 6th Edition*, bozza di specifiche ufficiali, 2015, `http://people.mozilla.org/~jorendorff/es6-draft.html`

[23] The jQuery Foundation, *jQuery*, sito ufficiale, ultima visita il 26 febbraio 2015, `http://jquery.com/`

[24] B. Jones, C. MacKenzie, *glMatrix: Javascript Matrix and Vector library for High Performance WebGL apps*, sito ufficiale, ultima visita il 26 febbraio 2015, `http://glmatrix.net/`

[25] World Wide Web Consortium, *Pointer Lock*, reccomandation, 2013, `http://www.w3.org/TR/pointerlock/`

[26] Analytical Graphics Inc, *WebGL Report*, codice sorgente, 2014, `https://github.com/AnalyticalGraphicsInc/webglreport`

[27] Takahito Tejima, *Javascript IGES Parser*, codice sorgente, 2012, `https://github.com/takahito-tejima/webgl`

[28] Giulio Casciola, *XCModel: NURBS-based system to model free form curves and surfaces*, sito ufficiale, ultima visita il 27 febbraio 2015, `http://www.dm.unibo.it/~casciola/html/xcmodel.html`

[29] Riccardo di Tosto, *Una applicazione 3D con HTML5 e WebGL: progetto e realizzazione di un'applicazione web per la visualizzazione di NURBS*, tesi di laurea, 2014, `http://amslaurea.unibo.it/8002/1/ditosto_riccardo_tesi.pdf`

[30] JetBrains, *WebStorm*, sito ufficiale, ultima visita il 28 febbraio 2015, `https://www.jetbrains.com/webstorm/`

[31] Atlassian, *Bitbucket*, sito ufficiale, ultima visita il 28 febbraio 2015, `https://bitbucket.org/`

[32] Atlassian, *SourceTree*, sito ufficiale, ultima visita il 28 febbraio 2015, `https://www.atlassian.com/software/sourcetree/overview`

[33] Google Inc, *Chrome Experiments*, raccolta di progetti WebGL, ultima visita il 27 febbraio 2015, `https://www.chromeexperiments.com/webgl`

[34] Internet Explorer Developer Relations team via Twitter, 2015, `https://twitter.com/IEDevChat/status/560172727059755008`

[35] Mozilla Organization, *WebGL 2*, documento ufficiale, ultima visita il 27 febbraio 2015, `https://wiki.mozilla.org/Platform/GFX/WebGL2`

[36] Mozilla Organization, *asm.js: an extraordinarily optimizable, low-level subset of JavaScript*, sito ufficiale, ultima visita il 28 febbario 2015, `http://asmjs.org/`