LAB 6 – GLSL

Fondamenti di Computer Graphics M

Erika Gardini – Ingegneria Informatica A.A. 2016/2017

L'esercitazione richiedeva realizzare una serie di funzionalità sfruttando il vertex shader ed il fragment shader (shader programmabili che girano sulla GPU aumentando l'efficienza del programma). Per programmare gli shader è necessario utilizzare il linguaggio GLSL, il quale si basa su C.

Parte 1: Wave motion

Il primo esercizio consisteva nell'estendere la WAVE application già fornita. Caricando l'applicazione ed eseguendo, il risultato di partenza era il seguente:

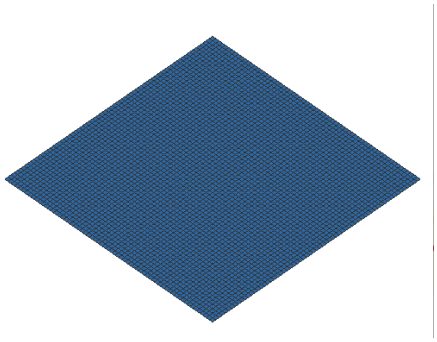

Figura 1: WAWE application di partenza

La prima funzionalità da realizzare consisteva nel modificare l'altezza del vertice v (componente y) utilizzando la formula data:

$$v_y = a \cdot \sin(\omega t + 5v_x) \cdot \sin(\omega t + 5v_z)$$

Per realizzare la funzionalità è stato necessario eseguire i seguenti comandi:

```
vec4 v = vec4(gl_Vertex);
v.y = A*sin(omega*time+5.0*gl_Vertex.x)*sin(omega*time+5.0*gl_Vertex.z);
gl_Position = gl_ModelViewProjectionMatrix * v;
```

Il primo comando consente di ottenere le quattro componenti del vertice v. Il secondo comando modifica la posizione y come richiesto dall'esercitazione. Il terzo comando applica la modifica alla posizione.
A questo punto è stato necessario assegnare dei valori alla frequenza, all'ampiezza ed al tempo. In particolare, per assegnare un valore al tempo, si utilizza la seguente funzionalità:

```
glutGet(GLUT_ELAPSED_TIME)
```

che restituisce il numero di millisecondi trascorsi da quando la `glutInit` è stata invocata.

Eseguendo si ottiene il seguente risultato:

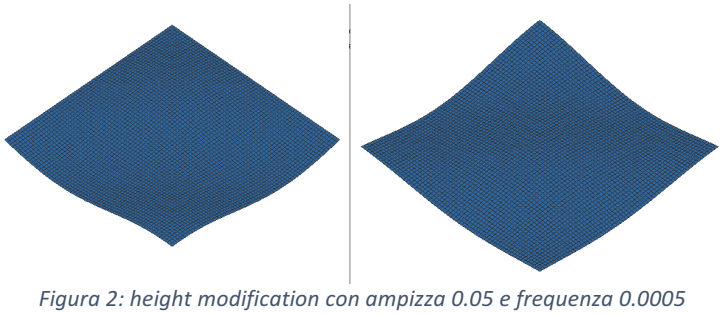

Figura 2: height modification con ampizza 0.05 e frequenza 0.0005

La seconda e la terza funzionalità da realizzare, invece, richiedevano di modificare l'ampiezza e la frequenza al click del mouse, utilizzando per la prima i valori [0.05, 0.1, 0.2] e per la seconda i valori [0.0005, 0.001, 0.002].
La logica della funzionalità è stata inserita nel metodo `mouse` e sono stati utilizzati i comandi `GLUT_LEFT_BUTTON` e `GLUT_RIGHT_BUTTON` e lo stato `GLUT_DOWN`.

Il risultato ottenuto modificando l'ampiezza è il seguente:

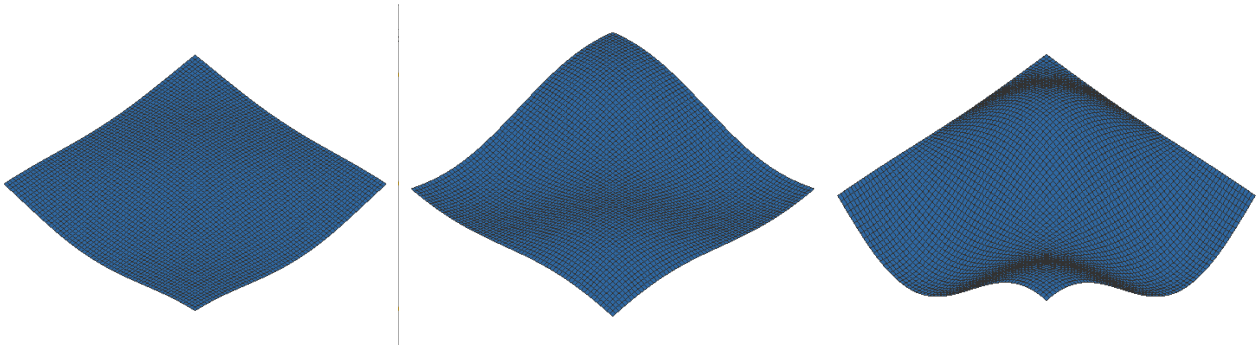

Figura 3: ampiezza da 0.005 a 0.01 a 0.02

Per i risultati sulla modifica della frequenza è necessario eseguire l'applicazione, in quanto non è possibile effettuare delle immagini significative.

Parte 2: Particle System
Il secondo esercizio richiedeva di estendere le funzionalità realizzate nella PARTICLE application.
Importando ed eseguendo il programma, si otteneva il seguente risultato:

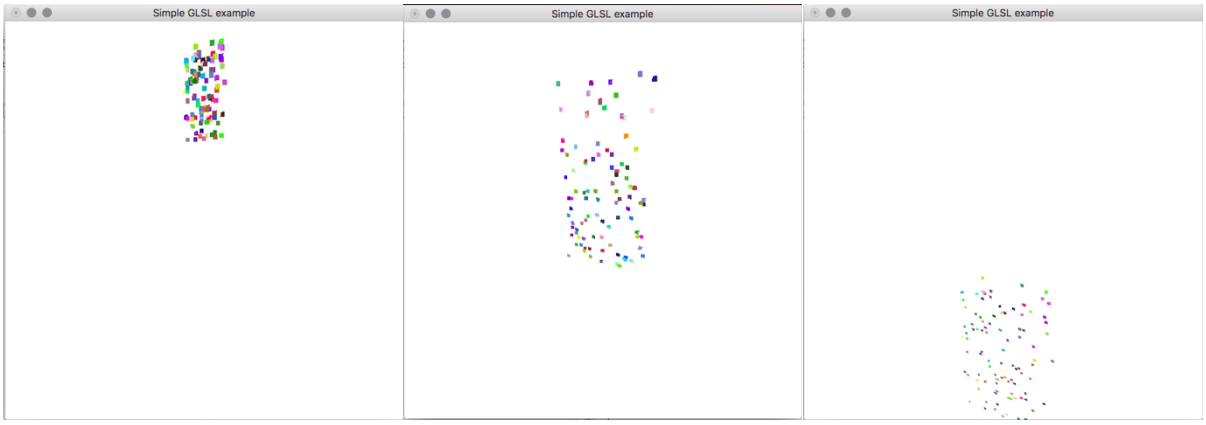

Figura 4: particle system default

La prima funzionalità da realizzare richiedeva di impostare la dimensione dei punti in relazione all'altezza.
Per realizzare questa funzionalità è stato necessario utilizzare uno dei seguenti comandi

```
gl_PointSize = min(t.y * t.z, 0.1);
```

```
gl_PointSize = max(t.y * t.z, 0.1);
```

i quali consideravano rispettivamente il più basso valore ed il più alto valore.
La seconda funzionalità, invece, richiedeva di muovere i puntini colorati anche rispetto alla direzione z.
Per realizzare questa funzionalità è stato necessario assegnare un valore alla z ed una velocità.

Il risultato finale ottenuto è il seguente:

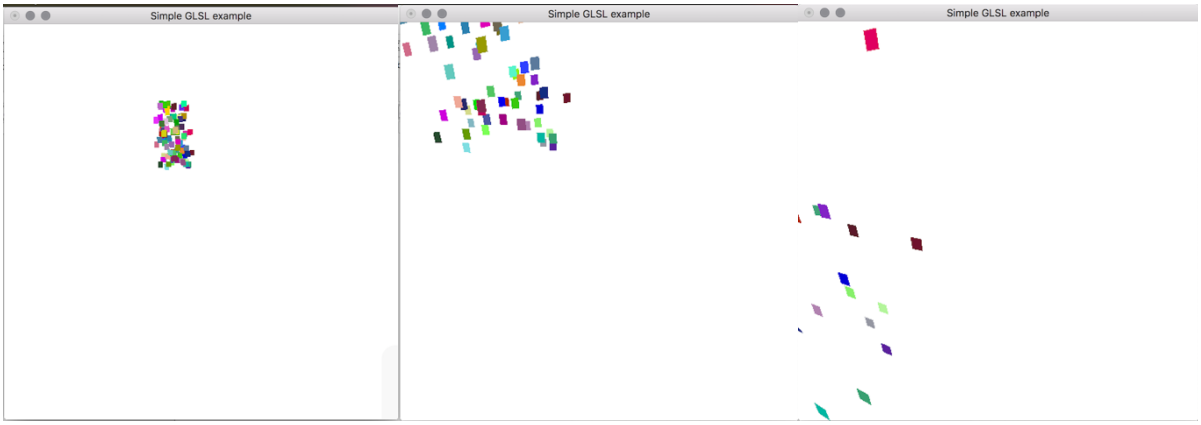

Figura 5: particle system anche rispetto a z

Parte 3: Phong Lighting

Il terzo esercizio richiedeva di aggiungere alcune funzionalità alla PHONG application. In particolare, il programma a default mostrava soltanto il seguente output:

Figura 6: PHONG application di default

Il risultato è stato ottenuto calcolando il modello di illuminazione di Phong senza far uso del fragment shader. Il secondo shader è già implementato e calcola la luce riflessa mediante un'approssimazione (halfway vector, più efficiente). Il terzo shader, invece, doveva essere realizzato in modo tale da considerare l'esatto raggio di luce riflessa.
Per far questo è stato calcolato il raggio di luce riflessa attraverso il seguente comando:

```
R = -reflect(L, N);
```

che calcola il raggio riflesso fra la luce e la normale al vertice.
Il raggio di luce riflessa è poi stato utilizzato nel fragment shader in sostituzione del raggio approssimato calcolato nel secondo shader.

Una volta realizzati tutti gli shader, veniva richiesto di mostrare le tre teiere, ciascuna con uno shader differente.
Per far questo è stato necessario modificare il metodo `draw` andando ad aggiungere le due teiere mancanti, ciascuna con lo shader corrispondente.
Per ottenere un buon risultato durante l'esecuzione è stata scelta la dimensione 1 per le teiere, mentre la dimensione della finestra è stata aumentata a 1256x512.

Il risultato ottenuto è il seguente:

Figura 7: illuminazione di PHONG in vertex shader, con approssimazione e con esatto raggio di luce riflessa

Parte 4: Toon Shading

Il quarto esercizio richiedeva di estendere le funzionalità della NONPHOTO application, la quale implementa un rendering non realistico, chiamato anche "Toon Shading".
Caricando l'applicazione ed eseguendola, si ottiene il seguente risultato:

Figura 8: Toon Shading

La funzionalità da realizzare richiedeva di aggiungere l'effetto outline/siluette alla teiera.
Per far questo è stato necessario calcolare il vettore fra la camera ed il punto in questione:

```
E = normalize(cam.xyz – eyePosition.xyz);
```

ed utilizzarlo per calcolare l'angolo con la normale dell'oggetto.

```
float angle = dot(normalize(E),normalize(N));
if(angle < 0.4)
      color = f;
```

Quando l'angolo è inferiore ad un angolo prefissato si assegna il colore rosso.
Il risultato ottenuto è il seguente:

Figura 9: Toon Shading con effetto outline/siluette

Figura 10: Toon shading con outline/siluette - altre angolazioni

Parte 5: Morphing

Il quinto esercizio richiedeva di estendere le funzionalità della MORPH application. Caricando ed eseguendo il programma si ottengono i seguenti risultati:

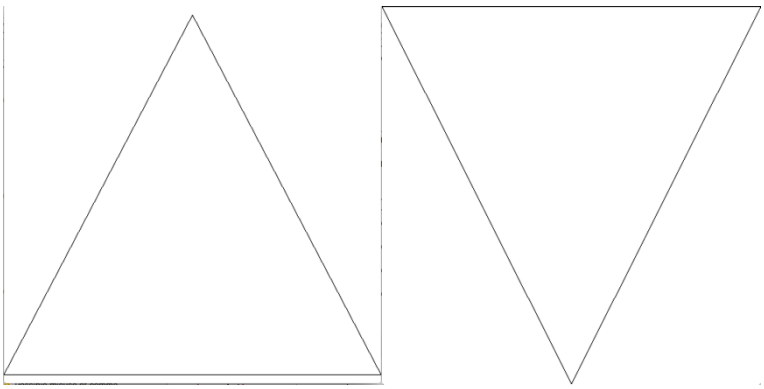

Figura 11: morphing di default

La prima funzionalità da implementare consisteva nell'assegnare un colore a ciascun vertice del triangolo e nel realizzare una color morphing/variation attraverso il vertex shader.
Per realizzare questa funzionalità è stato pertanto necessario assegnare un colore a ciascun vertice del triangolo di partenza ed un colore a ciascun vertice del triangolo di arrivo. L'interpolazione dei colori è stata poi realizzata mediante il seguente comando:

```
gl_FrontColor = mix(gl_Color, color2, s);
```

il quale, appunto, effettuava il "mix" del colore di partenza (gl_Color) con il colore di arrivo (color2) seguendo la funzione s, definita come segue:

```
float s =  0.5*(1.0+sin(0.001*time));
```

Il risultato ottenuto è il seguente:

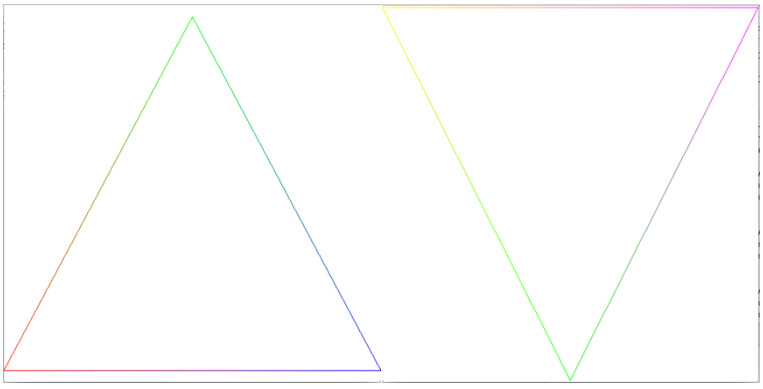

Figura 12: da rosso/verde/blu a giallo/verde/fucsia

Per far in modo che il colore sia distribuito in tutto il triangolo e non solo lungo i contorni è necessario modificare la modalità di disegno nel metodo `draw` da `GL_LINE_LOOP` a `GL_TRIANGLES`.

Il risultato ottenuto è il seguente:

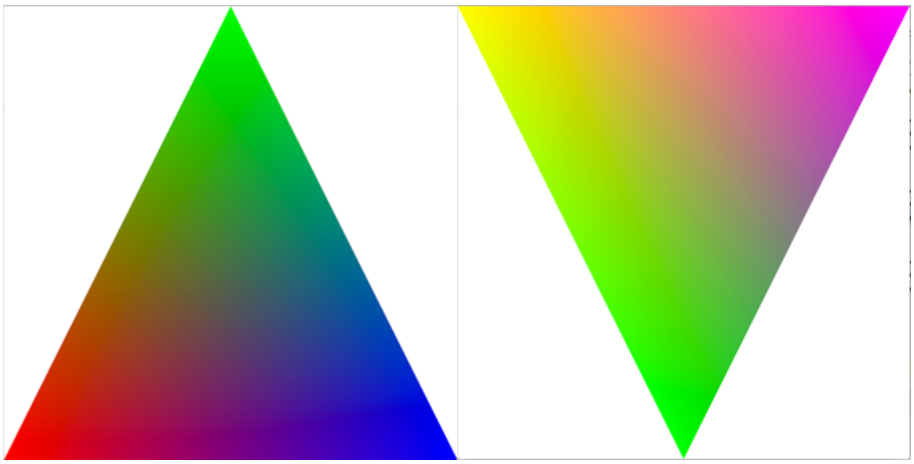

Figura 13: color morphing/variation

Per disegnare un quadrato, invece, è necessario spostare il vertice del triangolo che rappresenta la punta, aggiungerne un altro ed assegnare a quest'ultimo un colore. È inoltre necessario modificare la modalità di disegno nel metodo `draw` da `GL_TRIANGLES` a `GL_QUADS`.
Prima di eseguire è stata anche modifica la rotazione del quadrato, che non è più come quella del triangolo, ma è una rotazione a sinistra di 90°.

Il risultato ottenuto è il seguente:

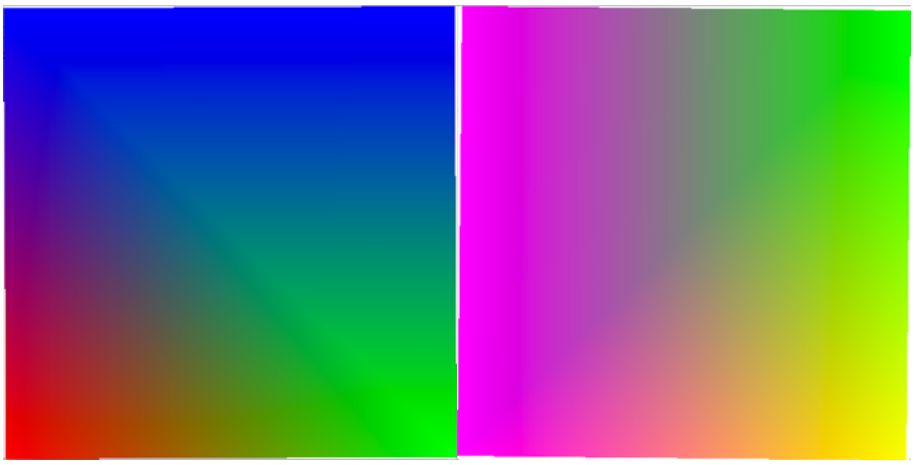

Figura 14: color morphing/variation e modifica della rotazione

La seconda funzionalità da implementare, invece, richiedeva di realizzare un passaggio da triangolo ad un'altra figura.
Per realizzare questa funzionalità si è deciso di passare da triangolo a quadrato. A tal scopo il triangolo di partenza è stato implementato come quadrato avente i due vertici superiori coincidenti in uno unico (la punta).
La funzione che effettuata il passaggio da triangolo a quadrato è la stessa che, nei punti precedenti, consentiva di passare da un triangolo all'altro o di ruotare il quadrato e viene riportata qui di seguito:

```
vec4 t = mix(gl_Vertex, vertices2, s);
```

In particolare, essa prende in input i vertici di partenza e quelli di arrivo e li sposta seguendo la stessa funzione s, riportata precedentemente.

Il risultato ottenuto è il seguente:

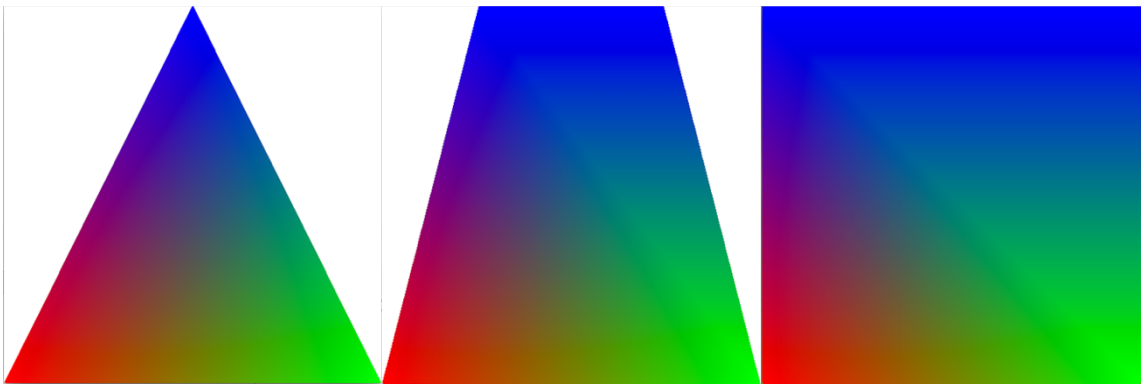

Figura 15: morphing fra triangolo e quadrato

Parte 6: Bump Mapping

Il sesto esercizio richiedeva di estendere le funzionalità della BUMPMAP application, la quale realizza il bump mapping texture. Caricando il programma fornito ed eseguendolo si ottiene il seguente risultato di partenza:

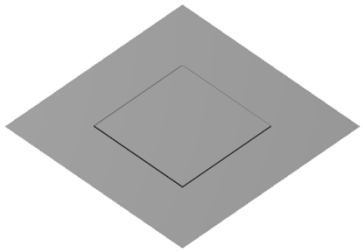

Figura 16: bump mapping texture di default

L'esercitazione richiedeva di modificare le altezze di input, in modo da ottenere diverse texture.
A tal scopo è stato necessario modificare l'array `data` nel `main`.

Effettuando vari esperimenti sono stati ottenuti i seguenti risultati:

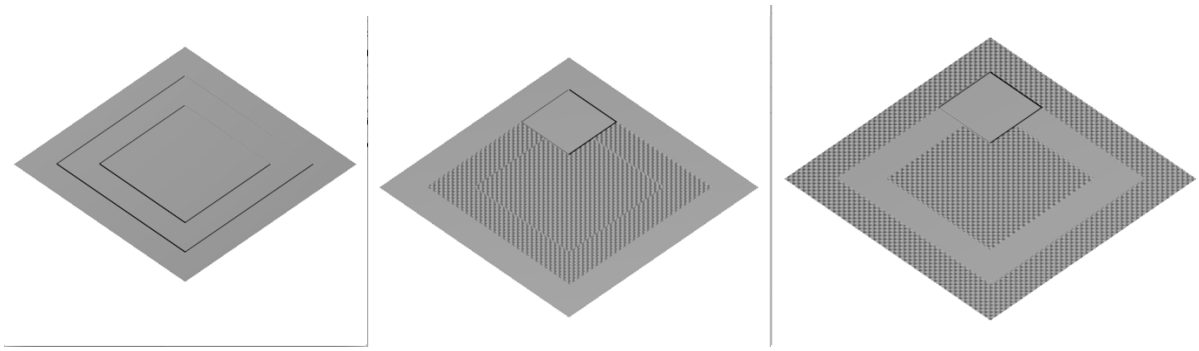

Figura 17: bump mapping finale

In particolare, nel primo risultato sono stati impostati degli alti valori (100 e 200) a $\frac{N}{4}$ e a $\frac{N}{8}$.
Il secondo risultato è stato ottenuto utilizzando seni e coseni, così come il terzo.

Parte 7: Cube environment mapping

L'esercizio 7 richiedeva di estendere le funzionalità della CUPEMAP application. A default, l'applicazione implementa un environmental mapping cubico su un oggetto riflessivo (la teiera), associando invece di 6 immagini 6 differenti colori. Compilando ed eseguendo l'applicazione di partenza si ottiene il seguente risultato:

Figura 18: environmental mapping cubico con colori

La funzionalità da realizzare consisteva nel caricare, al posto dei colori, delle immagini. I comandi utilizzati sono stati gli stessi sperimentati nell'esercitazione 5. In particolare, quindi, è stato necessario caricare le immagini da file (con il medesimo metodo utilizzato nel 5 laboratorio) ed inserire 6 comandi `glTexImage2D` ciascuno dei quali con un'immagine da caricare. Affinché il tutto funzionasse è stato necessario importare i file RgbImage.c ed RgbImage.h dall'esercitazione 5, in quanto essi fornivano dei metodi necessari alla realizzazione dell'environmental mapping cubico.

Il risultato ottenuto è stato il seguente:

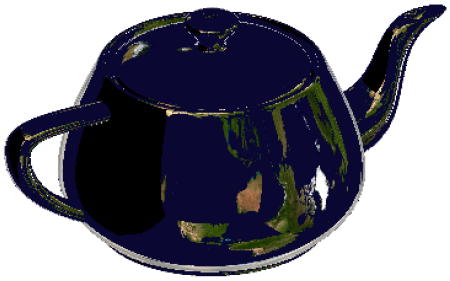

Figura 19: environmental mapping cubico con immagini