# **Fondamenti di Computer Graphics LM**

Lab 6 – GLSL

Mattia Fucili

**Wave Motion**

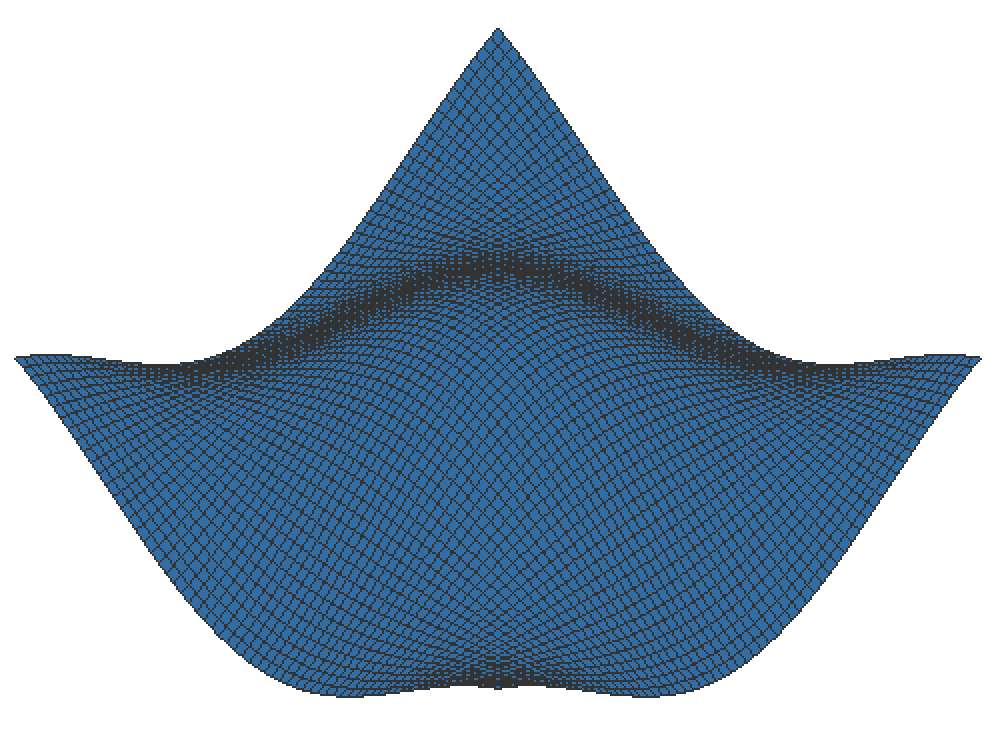
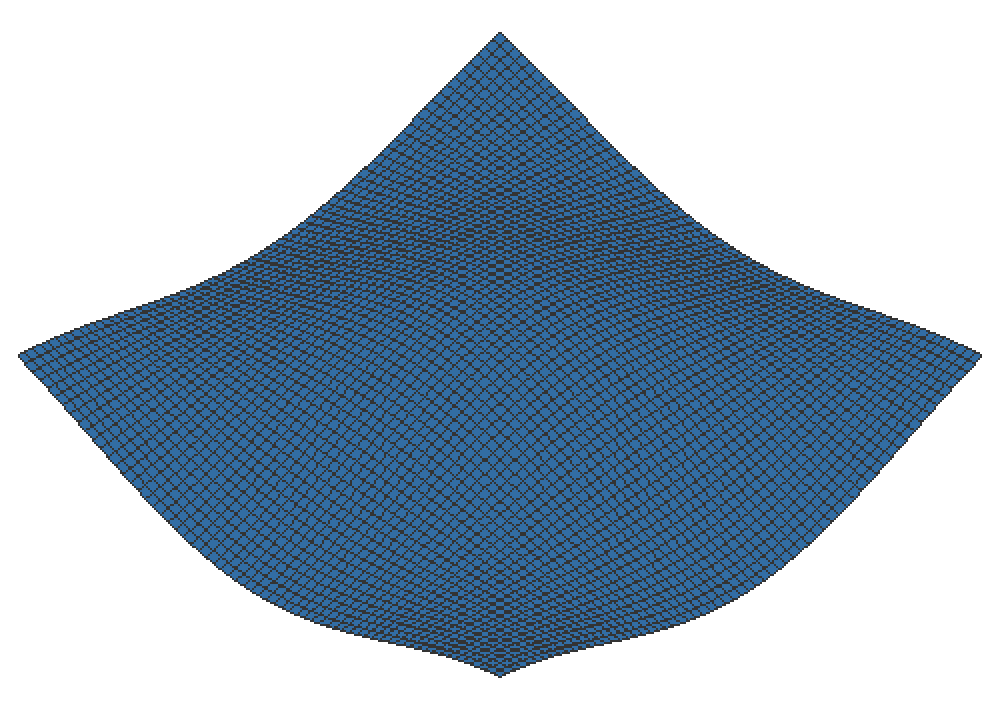
Questa prima parte dell’esercitazione aveva 3 obiettivi:

* generare un’alterazione della coordinata y di ogni vertice in funzione di tempo, ampiezza e frequenza applicati ad una certa formula;
* modificare il valore dell’ampiezza al click sinistro del mouse;
* modificare il valore della frequenza al click destro del mouse.

Il primo punto è stato realizzato nel file v.glsl, dove si è applicata la seguente formula:

considerando la posizione corrente del vertice (ottenibile con gl\_Vertex) e poi applicata al vertice tramite gl\_Position.

Per realizzare i seguenti due punti si è aggiunta la funzione mouse() al file wave.cpp per poter recepire i click dell’utente. Infine per passare i valori di ampiezza e frequenza al vertice si è usata la funzione glUniform1f(). Il risultato ottenuto è il seguente:



**Particle System**

Le richieste per questa parte erano:

* modificare le dimensioni della singola unità (particella) in base all’altezza;
* muovere le particelle anche lungo l’asse z.

La prima richiesta è stata soddisfatta abilitando la modifica della dimensione tramite le variabili GL\_POINT\_SPRITE e GL\_VERTEX\_PROGRAM\_POINT\_SIZE e poi andando a modificare gl\_PointSize nel file v.glsl.

Il secondo punto consisteva nel modificare il parametro vzParam in base ai valori all’interno del vettore velocity. È stato possibile ottenere questo parametro attraverso il metodo glGetAttribLocation() e, dopo averlo modificato, è stato possibile comunicarlo al vertex tramite glVertexAttrib1f(). Il risultato ottenuto è il seguente:



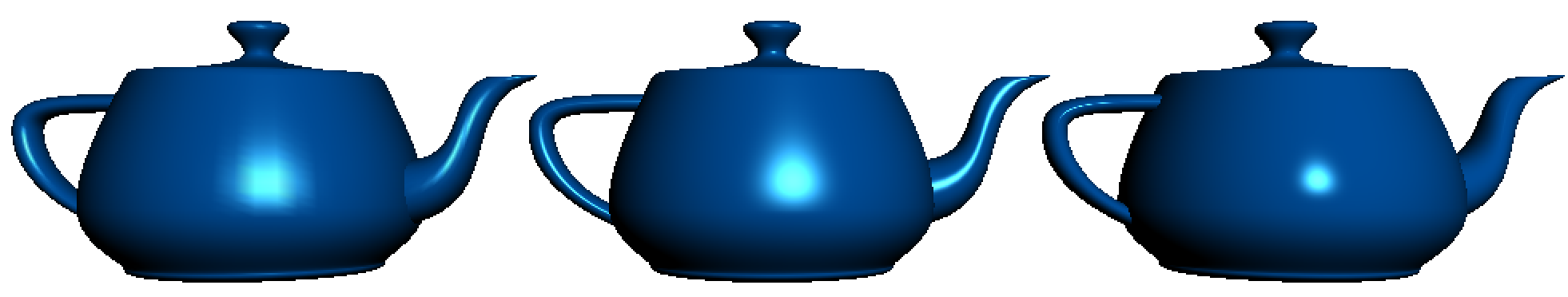
**Phong Lighting**

Nel terzo sotto progetto dell’esercitazione veniva richiesto di:

* aggiungere un terzo shader (a partire da uno già dato) che implementi la componente riflessiva di Phong e non quella di Blinn;
* mostrare 3 teiere, ognuna rappresentativa di un determinato shader.

Per prima cosa è stato realizzato il punto due modificando il codice del metodo draw() nel file phong.cpp. Sono state aggiunte altre due teiere, modificandone la collocazione nella scena e la grandezza in maniera tale da ottenere la scena che si vede infondo a questa sezione.

Per quanto riguarda il punto uno occorre specificare che la formula di Blinn (utilizzata in v2.glsl e f2.glsl) è una ottimizzazione rispetto a quella di Phong in quanto il raggio riflesso che viene considerato ha un costo computazionale minore (essendo un’approssimazione). Per realizzare questo punto è stato modificato il file v3.glsl calcolando il raggio riflesso tra il raggio della luce che colpisce il vertice considerato e la sua normale. Questo valore poi è stato usato nel fragment shader (f3.glsl) per calcolare il contributo del della formula di Phong. I risultati a confronto sono mostrati nella figura seguente:



v1.glsl v2.glsl v3.glsl

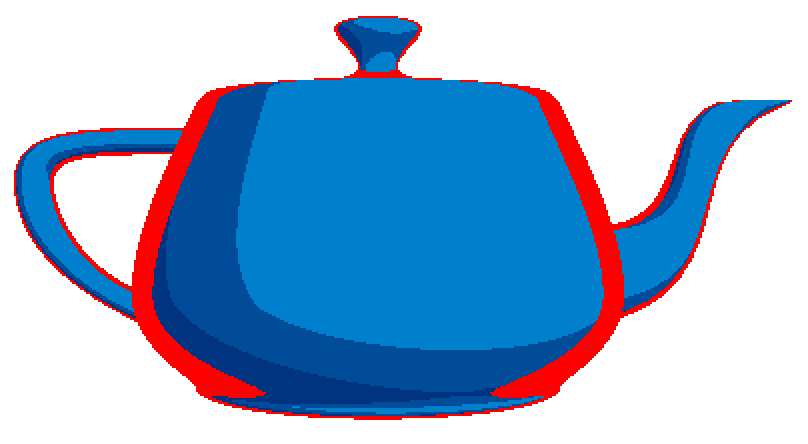
Nota: la prima teiera usa v1.glsl e f1.glsl che applica lo stesso calcolo che si ha nella seconda teiera (quindi la formula di Blinn) ma solo a livello di vertex shader.

**Toon Shading**

In questo punto dell’esercitazione c’era una sola richiesta:

* aggiungere una sagoma alla teiera (stile cartone).

Per implementare questa funzionalità è necessario dapprima (nel vertex shader) calcolare il vettore E che parte dalla camera ed arriva al vertice che si sta considerando sulla scena. Poi nel fragment shader si è calcolato il prodotto scalare tra la normale (N) al vertice e il vettore E. Questo risultato è stato utilizzato per applicare l’effetto silhouette rossa che si può vedere nell’immagine sottostante



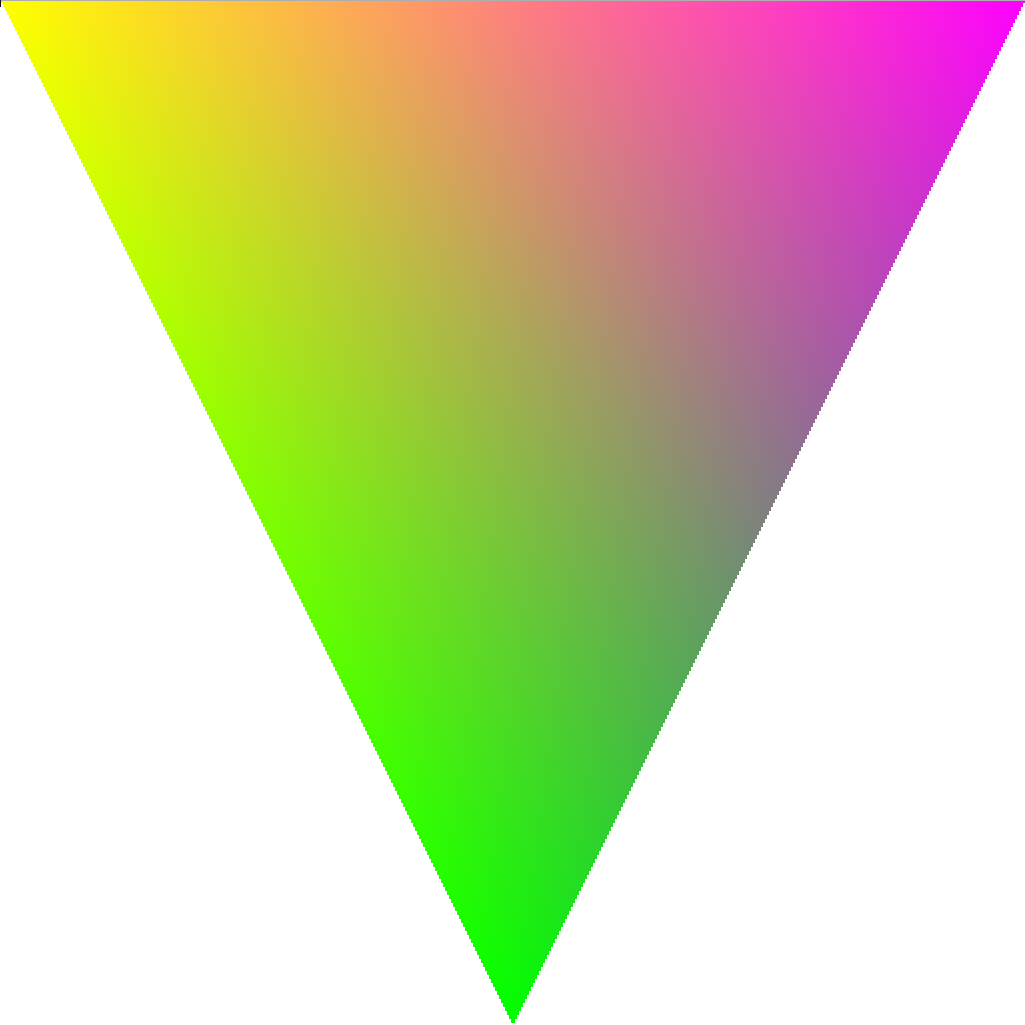
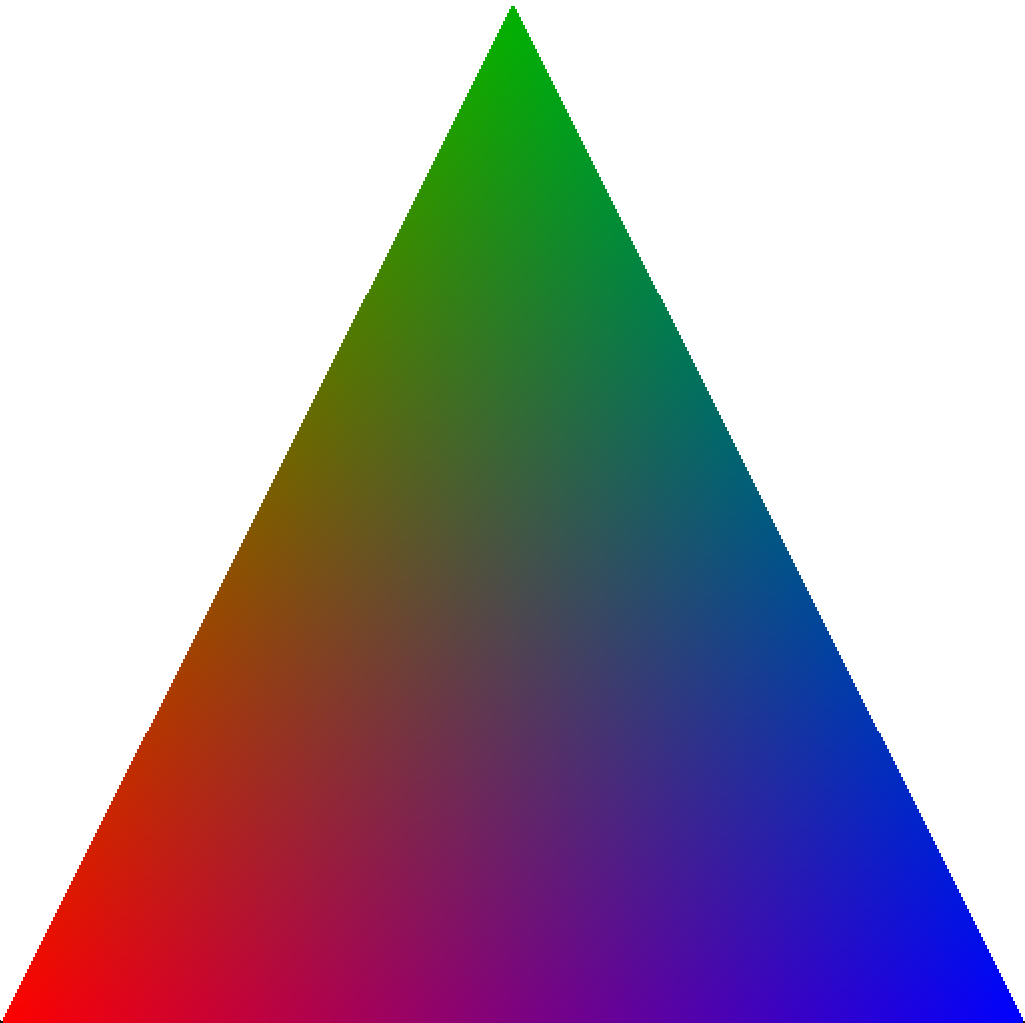
teiera di partenza teiera con effetto “Toon”

**Morphing**

In questa parte le richieste erano due:

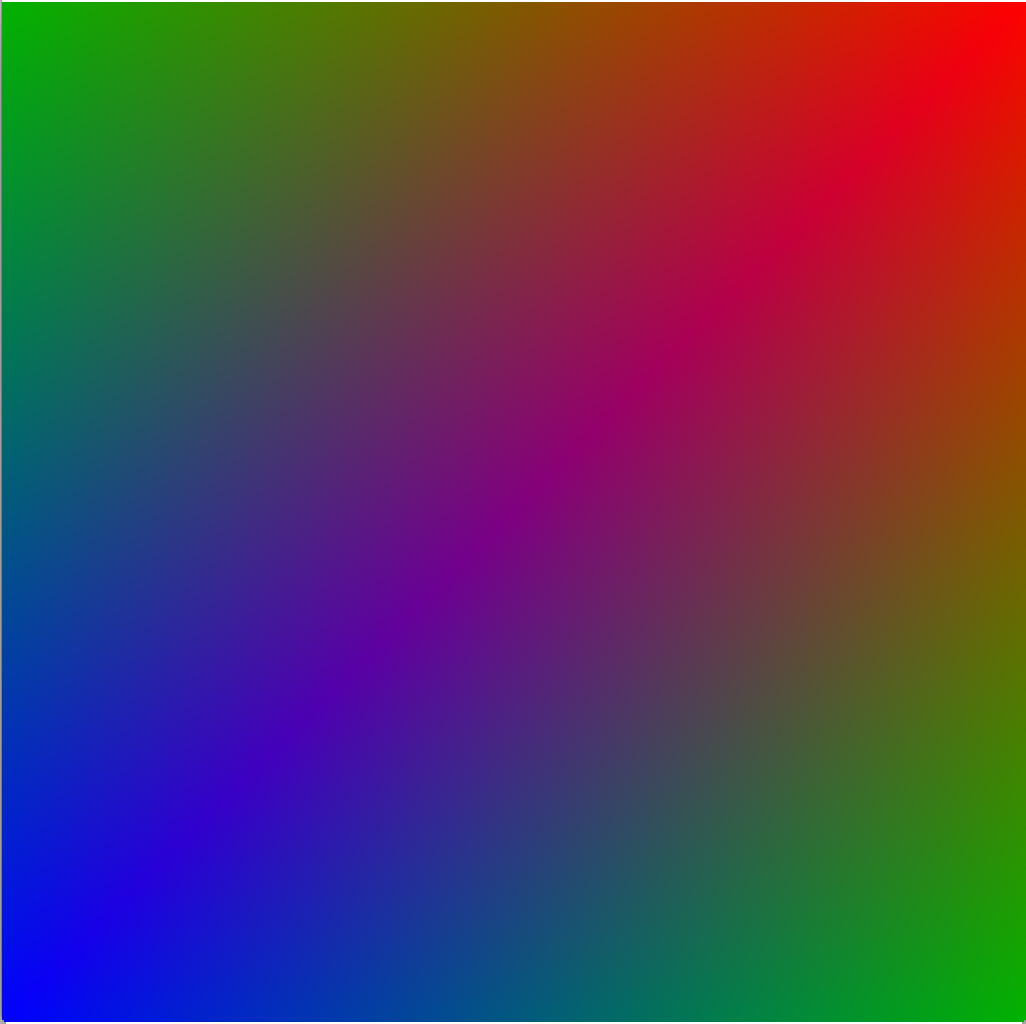
* riempire il triangolo dato interpolando i colori dei suoi vertici e modificarli durante il tempo;
* modificare il triangolo in un’altra forma, in questo caso quadrato.

Per soddisfare la prima richiesta è stata abilitata la modalità di disegno dei triangoli tramite l’istruzione glBegin(GL\_TRIANGLES). Dopodiché sono stati passati i vertici con i rispettivi colori del triangolo dritto (con la punta in alto) e i vertici con i rispettivi colori del triangolo ribaltato (con la punta in basso). Sia il colore che la posizione dei vertici sono stati modificati nel vertex shader tramite il metodo mix() di GLSL; che permette di interpolare da un inizio ad una fine secondo una funzione interna al metodo stesso. Il risultato è il seguente:



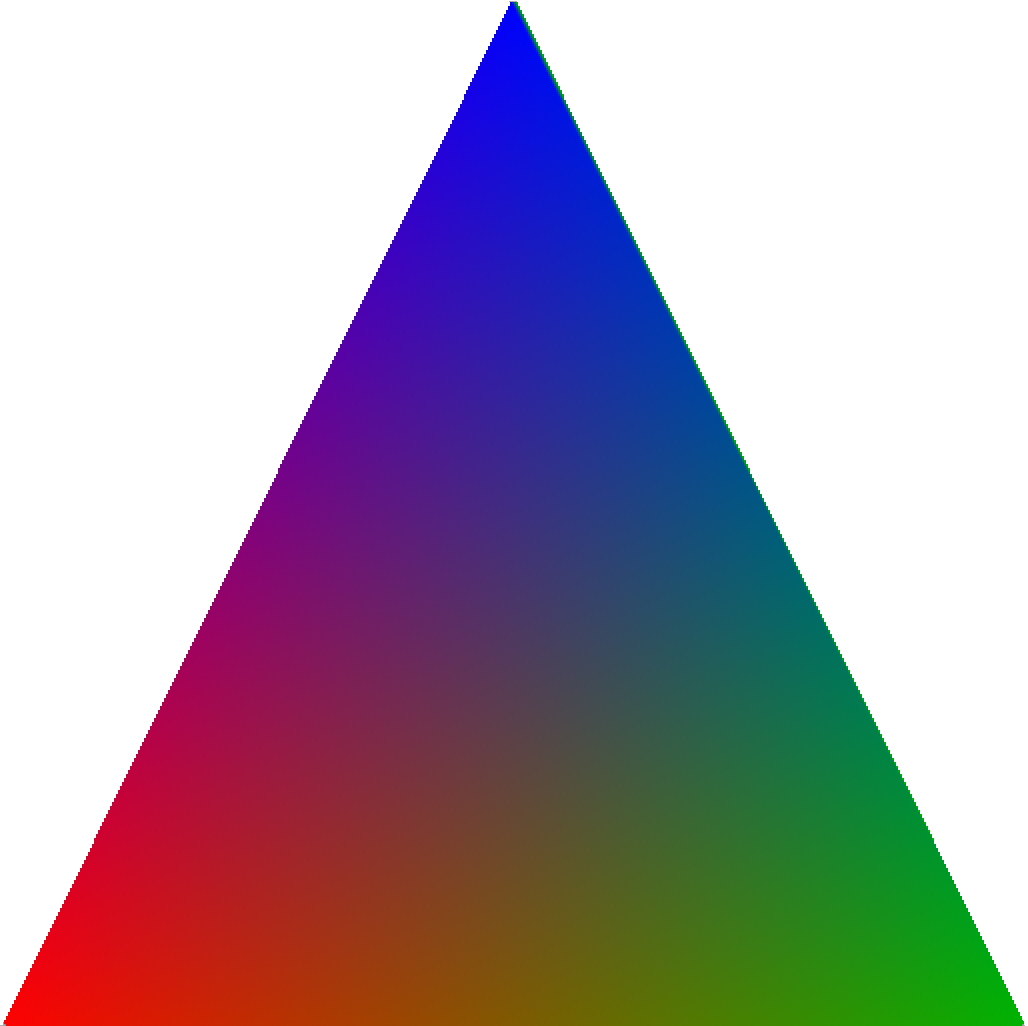
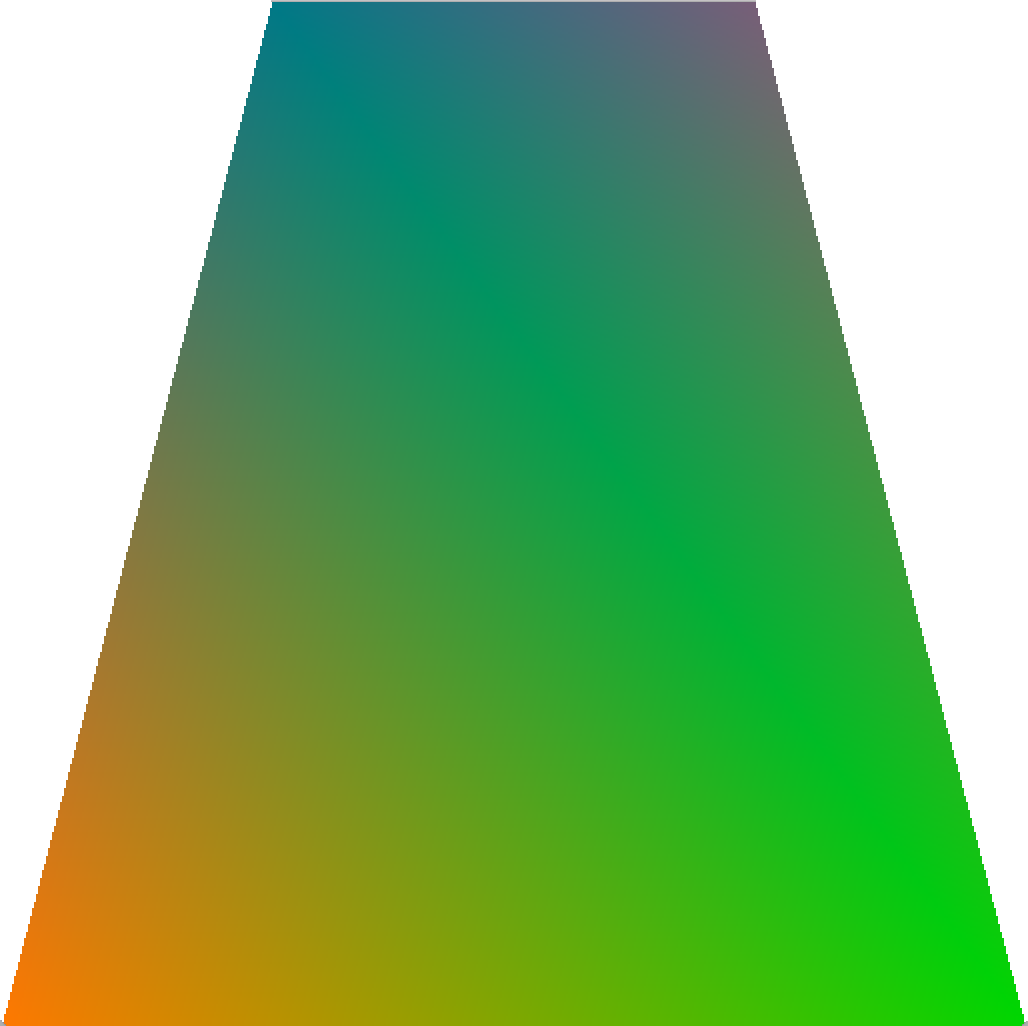
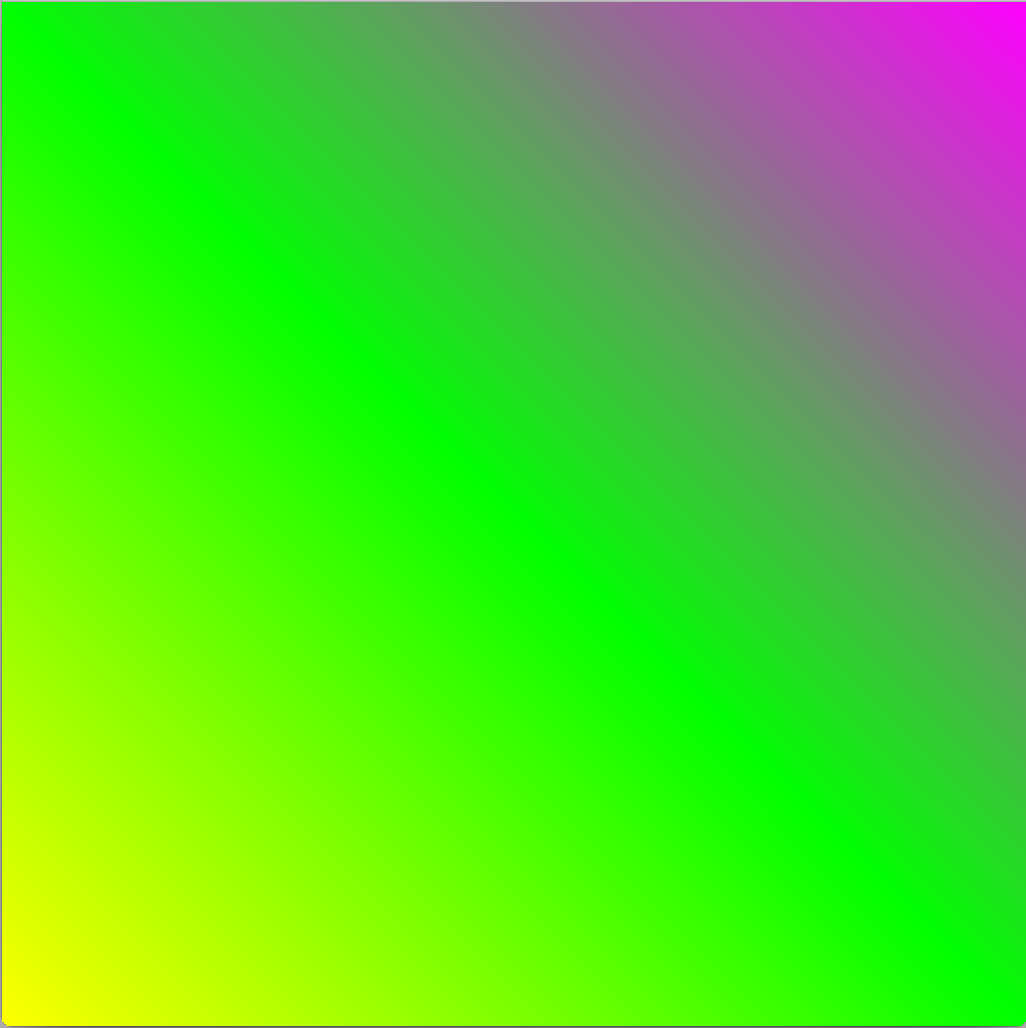
faccia frontale faccia ribaltata

Stessa cosa è stata applicata ad un quadrato, ovviamente aggiungendo un vertice in più e modificando la modalità di disegno con glBegin(GL\_QUADS):

faccia frontale faccia ribaltata

Per quanto riguarda il secondo obiettivo è stato affrontato come nel caso precedente di creazione del quadrato, però a due dei quattro vertici di partenza sono stati specificate le stesse coordinate in modo tale da ottenere dapprima un triangolo e poi un quadrato. Il risultato è mostrato di sotto:

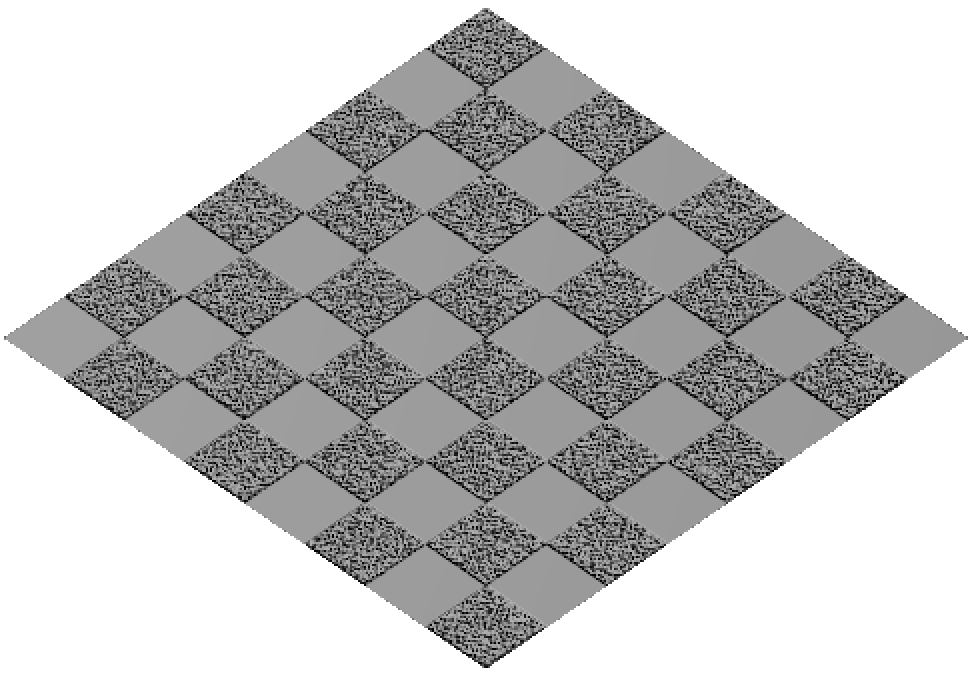
triangolo iniziale transizione quadrato finale

**Bump Mapping**

Per la parte relativa al Bump Mapping veniva richiesto di:

* modificare il motivo visualizzato in partenza.

Si è deciso di implementare un motivo a scacchiera suddividendo l’intero piano in una griglia e tramite un apposito algoritmo (nel file bumptex.cpp) si è scelto quale quadrato doveva subire un Bump Mapping o meno. Il risultato del procedimento è il seguente:



**Cube environment mapping**

L’ultimo punto dell’esercitazione richiedeva di:

* mappare sulla teiera le texture che derivano dalle facce di un cubo nel quale è immersa.

Per fare ciò è stato necessario utilizzare i file RgbImage.c e RgbImage.h, forniti già nell’esercitazione 5 per poter sfruttare le utility presenti per poter leggere le immagini .bmp. Una volta caricate in memoria le immagini sono state applicate al cubo tramite il metodo glTexImage2D(). Il risultato è il seguente:

