CORSO DI FONDAMENTI DI COMPUTER GRAPHICS

REPORT LABORATORI

September 2, 2019

Simone Berni 893034
Laurea Magistrale in Informatica
Alma Mater Studiorum

Introduzione

Tutte le seguenti esercitazioni sono state svolte utilizzando come sistema operativo Arch Linux e come ide CLion della IntelliJ. Se ci dovessero essere problemi di incompatibilità con i sistemi Windows, mi scuso, ma purtroppo tutte le mie macchine girano su sistemi Unix.

Esercitazione 1

`Consegne`

1. Compilare e far girare il programma. Provare i controlli da keyboard. Il left mouse button aggiunge un punto. I comandi 'f' e 'l' rimuovono il primo e l'ultimo punto dalla lista di punti, rispettivamente. Oltre ai 64 punti, i primi punti sono rimossi.

2. Osservare come il programma usa le OpenGL GLUT callback per catturare gli eventi click del mouse e determinare le posizioni (x, y) relative.

3. Provare a cambiare lo stile di punti e linee.

4. Disegnare la curva di Bézier a partire dai punti di controllo inseriti, utilizzando l'evaluator di OpenGL (glMap1f(), glMapGrid1f(), glEvalMesh1()). Ricordarsi di abilitare il disegno di curve con glEnable(GL_MAP1_VERTEX_3)

5. Sostituire alle routine di OpenGL il disegno della curva mediante algoritmo di de Casteljau.

6. Integrare nel programma in alternativa uno dei seguenti punti: (a) disegno di una curva di Bézier mediante algoritmo ottimizzato basato sulla suddivisione adattiva. (b) disegno interattivo di una curva di Bézier composta da tratti cubici, dove ogni tratto viene raccordato con il successivo con continuitá C 0 , C 1 , o G 1 a seconda della scelta utente da keyboard.

7. Permettere la modifica della posizione dei punti di controllo tramite trascinamento con il mouse

`Svolgimento`

1. Nulla da dover descrivere

2. Nulla da dover descrivere

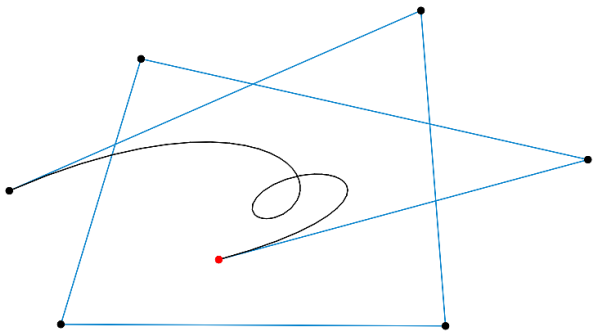
3. È possibile cambiare la dimensione della linea modificando il valore all'interno della funzione *glLineWidth*. La dimensione dei punti invece è modificabile attraverso la chiamata a *glPointSize*

4. Il codice dell'implementazione è verificabile nel sorgente sotto la funzione *drawBezierGL*, ma descrivendolo brevemente, è sufficiente utilizzare le funzioni di liberia *glBegin* per iniziare a disegnare i punti, i quali verranno calcolati attraverso la funzione *glEvalCoord1f*.

5. L'algoritmo di Casteljau è stato implementato nel seguente modo ed è verificabile nel sorgente sotto la funzione *castelAlgo*:
   Per ogni valore del parametro *t*, si effettuano una serie di LERP, e se i punti di controllo sono N, allora per ogni valore si effettuano
   $$\sum_{i=0}^{n} lerp$$
   , dove con lerp si intende l'interpolazione lineare, calcolata come Lerp(t,A,B)=(1-t)A+tB.
   A questo punto, è possibile disegnare la curva utilizzando una chiamata a *glVertex3f*.

6. E' stato scelto di integrare il punto A, utilizzare cioè la suddivisione adattiva per implementare la curva di Bezier. Anche in questo caso il sorgente è consultabile, sotto le funzioni *drawBezierAdaptive* e *adaptiveSubdivision*.
   In questo caso non viene dato il parametro t nel quale deve essere valutata la curva, come nelle fasi precedenti, bensì un valore di tolleranza. La prima parte consiste nel calcolare la distanza di ogni punto dalla corda tra il primo e l'ultimo punto e, se la distanza di ogni punto intermedio è minore della tolleranza, allora la curva può essere approssimata con una linea retta.
   Di questa parte si occupano le prime righe del codice, in particolare *pointLineDistance* che permette il calcolo della distanza tra ogni punto e la corda e, in caso positivo, cioè con distanza minore dalla tolleranza, si disegnano i punti con le primitive di OpenGL.
   In caso negativo invece, la curva viene splittata in due, aggiungendo i punti calcolati con la LERP, e il processo è iterato per le due sottocurve ricorsivamente, finchè non è possibile disegnare l'intera curva.

7. L'implementazione è stata effettuata attraverso due callback, chiamate *passive* e *motion*. La seconda viene invocata quando avviene un drag di un punto, e si occupa di modificare le coordinate dell'elemento e di invocare la funzione di display. In realtà è invocata ogni volta che il mouse si muove sullo schermo, ma effettua veramente del lavoro solo quando l'utente ha cliccato un punto, grazie alla variabile *isMoving*.
   Passive invece ha un compito diverso, quello di identificare che l'utente stia passando sopra ad un punto di controllo o no. In caso positivo, colora il punto di controllo in cui si trova di colore rosso, distinguendolo dagli altri. Per fare ciò si calcola la distanza della posizione dal mouse da tutti i punti sullo schermo e, in caso sia minore di una certa tolleranza, marca il punto e invoca la funzione di display, che si occuperà del colore.



Esercitazione 2

`Consegna`
Si richiede di realizzare una demo di animazione digitale 2D interattiva simile al materiale fornito.
Punti fondamentali del progetto:

- l'impegno artistico e l'impatto visivo della scena

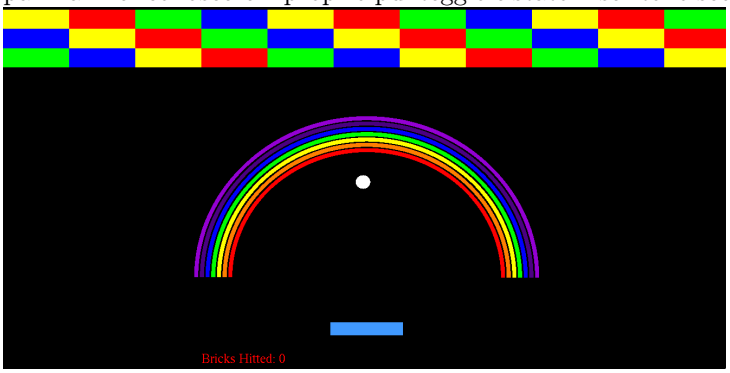
- la presenza e qualitá delle animazioni basate su simulazioni fisiche (es. uso leggi di cinematica/ dinamica)
- la presenza di elementi di gameplay con condizioni di vittoria/sconfitta per il giocatore
- l'utilizzo dei particellari nella scena

`Svolgimento`
Creazione di Arkanoid.
La demo creata consiste in un livello di Arkanoid, il cui scopo è colpire tutti i mattoncini con una palla. L'implementazione, in quanto doveva essere un breve gioco, consiste di un solo livello.
Analizzando ora tutti i punti fondamentali richiesti nel progetto:

- Le animazioni che sono state utilizzate all'interno del gioco sono tre: il movimento della pallina quando rimbalza su una superficie, il movimento del giocatore, e la scomparsa di un blocco quando esso viene colpito.

  - Per rendere la pallina controllabile, quando viene colpita dal giocatore, il nuovo angolo che avrà la pallina varia in base alla posizione di dove ha colpito il giocatore. È stata utilizzata la formula della interpolazione lineare per trovare il nuovo angolo. Per capire come è stata utilizzata la LERP per risolvere questo problema, è sufficiente consultare la funzione *playerBallAngle*
  - Il giocatore non ha accelerazione, ma si muove con velocità costante, altrimenti sarebbe incontrollabile e sarebbe impossibile colpire la pallina nel punto voluto
  - I blocchi una volta colpiti, non scompaiono veramente, ma il loro colore diventa uguale a quello dello sfondo e non vengono più considerati gli impatti con quel blocco

- Sinceramente, non trovo le particelle molto belle visivamente, ma data la richiesta, sono state emesse all'impatto della pallina con il giocatore.
- Ci sono due possibilità per il giocatore: vincere o perdere. In ogni caso, lo schermo finale sarà il medesimo, ma la prima possibilità la si ottiene colpendo tutti i blocchi, la seconda facendo cadere la pallina. Per conoscere il proprio punteggio è stato inserito lo score dei blocchi distrutti attualmente.



Esercitazione 3

`Consegna:`

1. Caricamento e visualizzazione modelli mesh poligonali
   (a) Caricamento e visualizzazione di più di un oggetto mesh con la possibilità di passare la selezione dall'uno all'altro tramite special key
   (b) Verifica della gestione della visualizzazione dei modelli poligonali tramite VAO/VBO
   (c) Calcolo e memorizzazione delle normali ai vertici per i modelli mesh poligonali. Visualizzazione con normali ai vertici in modalità smooth
   (d) Permettere il cambio di materiale dell'oggetto da pop-up menù
2. Navigazione interattiva in scena
   (a) Pan orizzontale camera
   (b) Pan verticale camera
   (c) Zoom camera
   (d) Culling
   (e) Smooth/Flat shading
   (f) Materials
   (g) Camera Motion
3. Trasformazione degli oggetti in scena
   (a) Traslazione, Rotazione, Scalatura WCS/OCS

`Svolgimento:`
Parto dal presupposto che ho dovuto commentare la generazione della sfera(luce) dal codice, perchè per una qualche ragione non ancora ben identificata, con quella matrice sul mio computer con OS Arch non visualizzo più nessun oggetto. Su Ubuntu invece, mi funzionava anche con quella riga di codice, sinceramente non ho capito la motivazione.
Inoltre, per poter testare, è necessario cambiare il path di **MeshDir**, in quanto la mia architettura richiedeva un path assoluto e non relativo.

1. Caricamento e visualizzazione modelli mesh poligonali
   (a) Ogni oggetto è, al momento della inizializzazione, caricato utilizzando la funzione *createMesh*, che inserisce l'oggetto all'interno di un array. Il controllo di quale elemento mostrare in ogni momento, è gestito grazie alla variabile **selectedObject** e dalla funzione *drawOne*
   (b) Nulla da dover descrivere
   (c) La richiesta è stata implementata aumentando le funzioni *generate_ and_ load_ buffers* e *loadObjFile*

(d) E' stata implementata la funzione *material_ menu_ function* che, dato il valore dell'opzione cliccato, modifica il tipo di materiale dell'oggetto in questo momento selezionato

2. Navigazione interattiva in scena

(a) Le funzioni che si occupano dello spostamento a destra e sinistra sono rispettivamente *moveCameraRight* e *moveCameraLeft*. Per risolvere la richiesta, è necessario modificare la posizione della camera e del target, di uno stesso valore. In particolare, il vettore che si occupa di quantificare lo spostamento è un vettore a 4 dimensioni con valori 1,0,-1,0. Per spostarsi a sinistra il vettore è stato sommato a quello che identifica la posizione del target e della camera, per lo spostamento a destra sono stati sottratti.

(b) Le funzioni che si occupano dello spostamento in alto e in baso sono rispettivamente *moveCameraUp* e *moveCameraDown*. Per risolvere la richiesta, è necessario modificare la posizione della camera e del target, di uno stesso valore. In particolare, il vettore che si occupa di quantificare lo spostamento è un vettore a 4 dimensioni con valori 1,0,-1,0. Per spostarsi in alto il vettore è stato sommato a quello che identifica la posizione del target e della camera, per lo spostamento in basso sono stati sottratti.

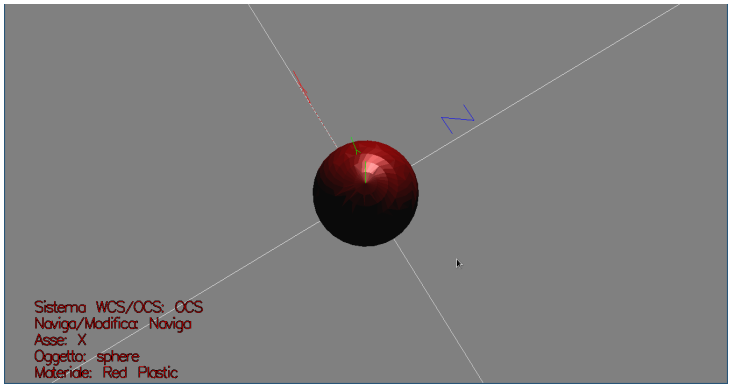
(c) Le funzioni che si occupano dello zoom in avanti e indietro sono rispettivamente *moveCameraForward* e *moveCameraBack*. L'idea delle due funzioni è la stessa, cambia solo la costante che indica il movimento, da 1 a -1. L'implementazione segue la regola della retta parametrica passante per un punto. Si prende il vettore che va dal punto target alla posizione attuale. Il valore delle componenti sarà positivo se ci si vuole avvicinare al target, negativo altrimenti. A questo punto si aggiorna la posizione sommando quella attuale con il vettore, moltiplicato per una costante che ne indica la quantità di spostamento.

(d) E' sufficiente invocare rispettivamente *glEnable* e *glDisable* del valore **GL_CULL_FACE** per abilitare o disabilitare il culling

(e) E' sufficiente invocare *gkShadeModel* con il valore **GL_FLAT GL_SMOOTH** per abilitare rispettivamente il flat shading o lo smooth shading

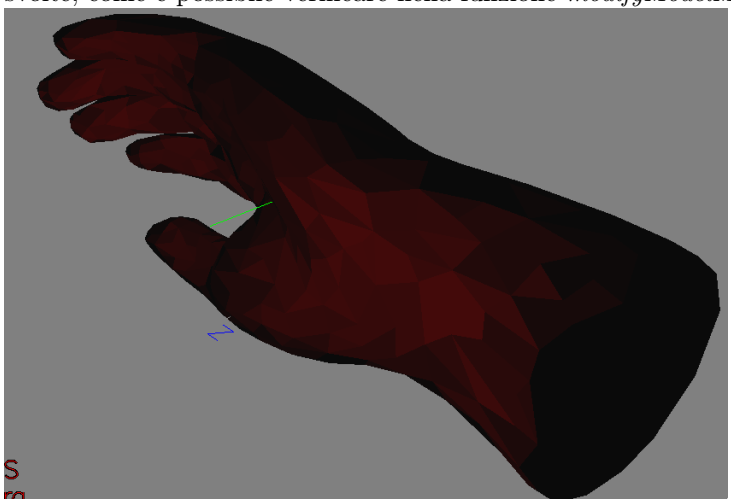
(f) La funzione è stata implementata per la consegna 1.4

(g) L'idea è avere n punti di controllo, tali che il primo e l'ultimo abbiano le stesse coordinate, e muovere la telecamera sulla curva di bezier da loro creata, utilizzando l'algoritmo creato nella esercitazione 1. Il codice è consultabile nella funzione *cameraMotion*. Per creare l'animazione, è stata utilizzata la funzione di libreria *glutIdleFunction*. Dato che i punti sono fissi, se si sposta l'oggetto dalla posizione 0,0,0, non ci si muoverà attorno all'oggetto, ma attorno alla sua posizione iniziale.



3. Trasformazione degli oggetti in scena

   (a) Per ottenere la posizione attuale dell'oggetto, si moltiplica l'identità per la matrice dell'oggetto, per effettuare la traslazione si può utilizzare la funzione *GlTransaltef*, per la rotazione la funzione *glRotatef* e infine per la scalatura *glScalef*. Per questa ultima funzione non ho capito se si dovesse scalare un asse alla volta, oppure no. Personalmente ho preferito implementare la scalatura dell'oggetto in sè, in quanto la consideravo più utile, ma se fosse stato necessario scalare un asse alla volta, è sufficiente modificare i parametri di *glScalef* controllando l'asse di lavoro. La differenza tra WCS e OCS consiste semplicemente nell'ordine delle operazioni svolte, come è possibile verificare nella funzione *modifyModelMatrix*



Esercitazione 6

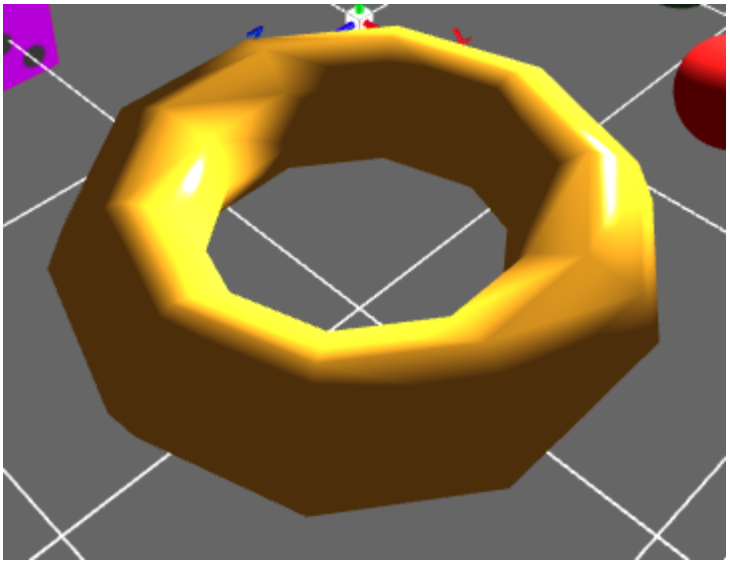
`Consegna:`

1. **Lighting**: permettere lo spostamento interattivo della luce posizionale/direzionale in scena
2. **Shading**: realizzare la modalità shading Phong realizzando gli shaders `v_phong.glsl` e `f_phong.glsl`
3. **Texture mapping 2D** del toro con immagine letta da file utilizzando gli shaders `v_texture.glsl` e `f_texture.glsl`.

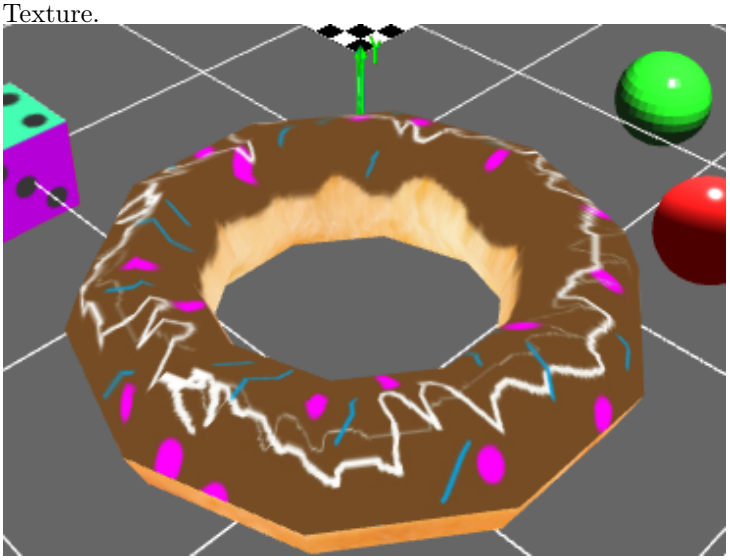
4. **Texture mapping 2D + Shading** Realizzare gli shaders `v_texture_phong.glsl` e `f_texture_phong.glsl` per combinare l'effetto shading Phong con la texture image sulla mesh toro
5. **Procedural mapping** basato su un procedimento algoritmico a piacere sul toro
6. **Wave Motion** creare l'animazione dell'oggetto *GridPlane* modificando la posizione dei vertici in un vertex shader `v_wave.glsl`. Utilizzando la variabile elapsed time t, passata da applicazione al vertex shader, per riprodurre il moto ondoso ottenuto con la sola modifica della coordinata y.
7. **Toon Shading**: realizzare gli shaders `v_toon.glsl` e `f_toon.glsl` per la resa non fotorealistica nota comunemente come Toon Shading

`Svolgimento:` Nuovamente è stato trovato un problema di compatibilità, in particolare la versione utilizzata negli shader. Per farli funzionare nella mia macchina, ho deciso di settarli a 320 es, ma questo ha provocato un ulteriore problema, la mancanza di precisione. Quest'ultimo punto è stato risolto inserendo la linea di codice *#precision mediump float*. La scelta del medium è per avere ulteriori problemi di compatibilità, in caso dovessi cambiare architettura hardware.
Nuovamente, sono stati inseriti dei path assoluti, precisamente **MeshDir**, **TextureDir** e **ShaderDir**, che devono essere modificati per testare il programma.

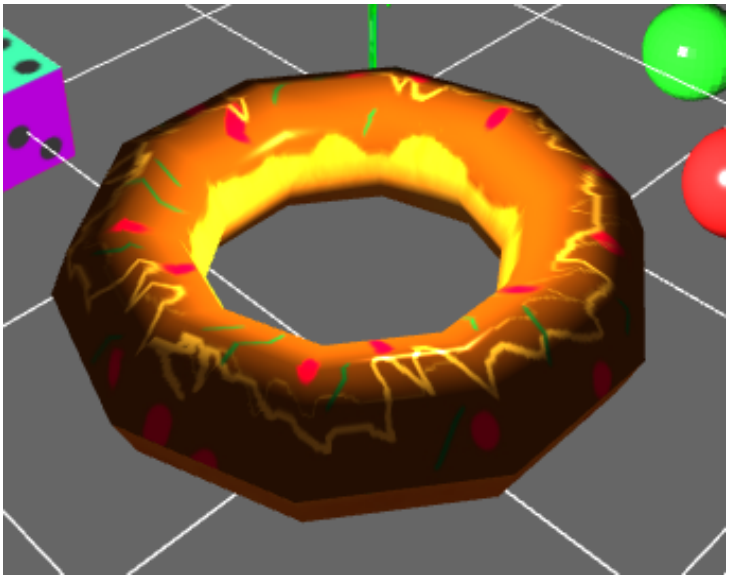
1. **Lighting**: Selezionando l'oggetto luce, e selezionando una *OperationMode* diversa dalla **Navigazione**, si arriverà ad invocare la funzione *modifyModelMatrix*. Il suo contenuto è lo stesso della esercitazione3, a differenza che in questo caso non abbiamo una matrice di float, ma un mat4, ma il ragionamento dietro è lo stesso. All'interno della funzione vi è stata aggiunta la parte per la gestione anche della luce vera e propria, modificandone la posizione grazie all'attributo `light.position`.
   Ma ciò non è sufficiente a notare, durante l'esecuzione, un cambiamento. Questo perché nella funzione *display* non venivano modificati, di volta in volta, il contenuto di **light_uniforms[SHADER]**, ma erano stati assegnati solo durante l'inizializzazione degli shaders. Aggiungendo quelle righe a tutti gli shader, è stato possibile completare la consegna.
2. **Shading**: l'algoritmo di Phong è stato implementato negli shader`v_phong.glsl` e `f_phong.glsl`, e per capire come programmare uno shader sono stati utilizzati gli shaders di Blinn e Gouraud come esempi.



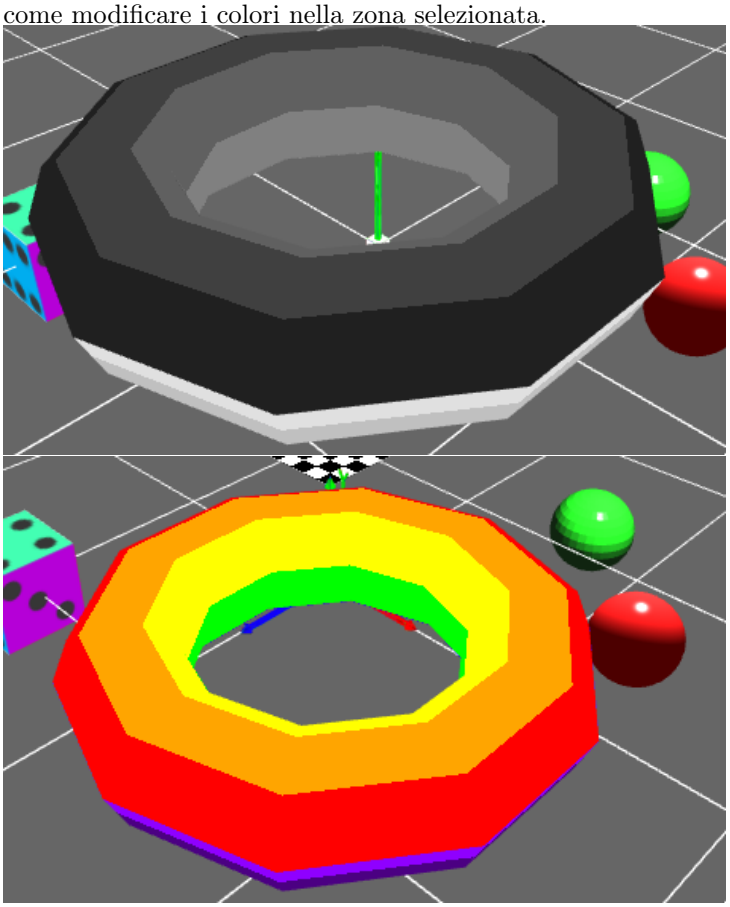
3. **Texture mapping 2D**: dopo aver modificato la funzione *init_ torus* con gli appositi parametri, mi sono reso conto che ciò non era sufficiente, in quanto la texture non veniva applicata correttamente. Per questo è stato aggiunto alla funzione *computeTorusVertex* l'inizializzazione delle coordinate Texture.

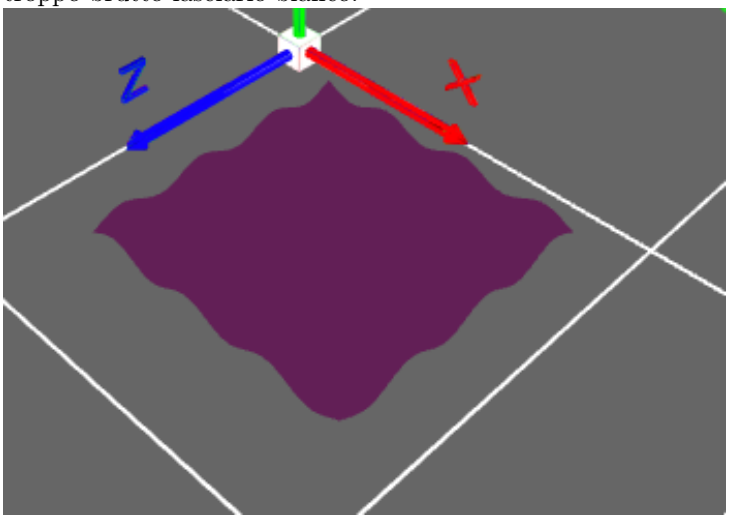


4. **Texture mapping 2D + Shading**: nuovamente sono stati presi come base due shaders per creare quello richiesto, **Texture_Only**, dato già nella esercitazione, e Phong, creato per un punto precedente.



5. **Procedural mapping** Utilizzando la funzione *generate_ texture*, che era già presente perla generazione della texture scacchiera su un piano, sono state create due funzioni, *generateShadesOfGray* e *generateRainbowMap*. La prima mi è servita per capire come lavorare sul toro, e come la texture veniva mappata sul toro. La seconda, esteticamente magari più piacevole, mi ha permesso di capire come modificare i colori nella zona selezionata.



6. **Wave Motion**: La costruzione prevede varie parti: la prima è l'inizializzazione della wave, effettuata nella funzione *init_ wave*. Successivamente c'è la parte di configurazione dello shader, simile agli altri, con solo la differenza che gli viene passato anche il parametro *t*, richiesto dall'esercizio e ottenuto attraverso *glutGet(GLUT_ ELAPSED_ TIME)*. Per poter attivare il movimento è stata aggiunta una nuova opzione nel menù, **Move waves**, che attiverà *glutIdleFunc* della mia callback, *moveWaves*. Questa funzione non fa altro che richiamare costantemente lo shader sull'oggetto, con il nuovo valore di **waveTime**.
   Lo shader è abbastanza lineare, come il vertex shader `v_passthrough.glsl` ottiene la posizione e, applicando la formula inserita nella richiesta dell'esercitazione, calcola la nuova posizione. Sono state aggiunte anche le righe di codice necessarie al passaggio di una texture, in quanto mi sembrava troppo brutto lasciarlo bianco.
   

7. **Toon Shading** L' idea per creare il toon shading è abbastanza semplice: si considerano intervalli fissi di luce, nel mio caso sono 4 intervalli limitati dai valori 30, 60 e 80 e ogni punto, a seconda della intensità della luce che lo colpisce, viene reso più luminoso moltiplicando il colore base per un valore costante a seconda dell'intervallo in cui cade. Il resto del codice di `v_toon.glsl` e `f_toon.glsl` è lo stesso degli altri shaders.