Esercitazione 6 - Shaders

La sesta esercitazione prevede l'implementazione di una serie di vertex shaders e fragment shaders personalizzati per modificare il comportamento a default della GPU nella pipeline di rendering.

WAVE

La prima parte dell'esercitazione prevede la realizzazione delle seguenti funzionalità:

- modifica dell'altezza del piano secondo la formula fornita (vertex shader)
- possibilità di modificare l'ampiezza dell'oscillazione
- possibilità di modificare la frequenza dell'oscillazione

Per realizzare la prima funzionalità si è modificato il vertex shader nel file *v.glsl* impostando la posizione y del vertice secondo la formula data e assegnando la nuova posizione al vertice.

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

Nel file wave.cpp si sono aggiunti i binding con le variabili utilizzate dallo shader.

```
timeParam = glGetUniformLocation(program, "time");
amplitudeParam = glGetUniformLocation(program, "A");
frequencyParam = glGetUniformLocation(program, "omega");
glUniform1f(timeParam, 0);
glUniform1f(amplitudeParam, 0.05);
glUniform1f(frequencyParam, 0.0005);
```

Per realizzare la seconda e la terza funzionalità si è modificata la funzione mouse() cambiando in maniera casuale l'ampiezza e l'oscillazione quando vengono premuti rispettivamente i tasti sinistro e destro del mouse.

```
float values[] = {0.05, 0.1, 0.2};
if (button == GLUT_LEFT_BUTTON) {
   int index = rand() % 3;
   glUniform1f(amplitudeParam, values[index]);
} else if (button == GLUT_RIGHT_BUTTON) {
   int index = rand() % 3;
   glUniform1f(frequencyParam, (float) values[index] / (float) 100);
}
```

Il risultato ottenuto è il seguente:

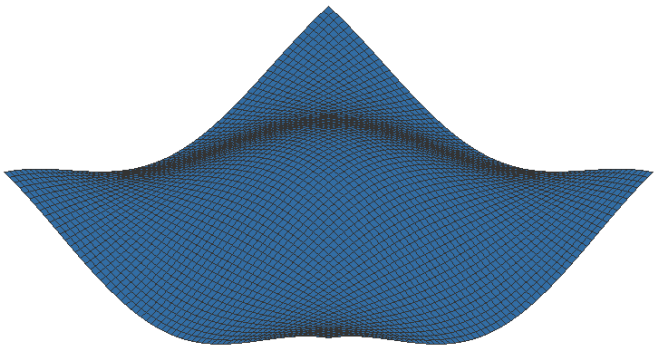

Figura 1 - Wave

PARTICLE SYSTEM

La seconda parte dell'esercitazione prevede la realizzazione delle seguenti funzionalità:

- dimensione dipendente dall'altezza
- permettere alle particelle di muoversi anche lungo l'asse z

Per implementare il primo punto si sono abilitate due funzioni di OpenGL che consentono di modificare la dimensione dei punti attraverso il vertex shader.

```
glEnable(GL_POINT_SPRITE);
glEnable(GL_VERTEX_PROGRAM_POINT_SIZE);
```

Si è quindi modificata la dimensione del punto in maniera dipendente dalla posizione del punto rispetto a y e z.

```
gl_PointSize = t.y;
```

Il secondo punto è stato realizzato aggiungendo la velocità di movimento delle particelle lungo l'asse z analogamente a quanto era già stato fatto per gli altri due assi.

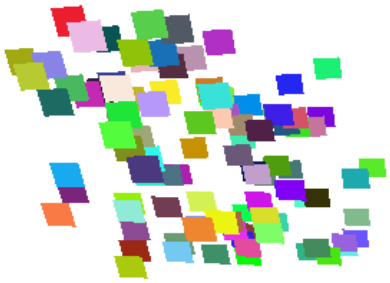

Figura 2 - Particle system

PHONG LIGHTING

La terza parte dell'esercitazione prevede la realizzazione delle seguenti funzionalità:

- creare un terzo shader che utilizzi il vero raggio riflesso per l'illuminazione
- mostrare le tre modalità di illuminazione contemporaneamente

Per il primo punto si è calcolato il raggio riflesso tramite la funzione reflect tra L (raggio luminoso) e N (normale alla superficie):

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

Nel fragment shader si è sostituito l'Half-way con il nuovo raggio riflesso calcolato:

```
vec3 Ray = normalize(R);
float Ks = pow(max(dot(Ray, Normal), 0.0), gl_FrontMaterial.shininess);
```

Per il secondo punto si sono utilizzate le funzioni glTranslatef() e glUseProgram() per consentire rispettivamente di traslare le diverse mesh e di utilizzare diversi vertex/fragment shader.

```
glUseProgram(program[0]);
glPushMatrix();
   glTranslatef(-3.0f, 0.0f, -5.0f);
   glRotatef(k*t, 1.0, 0.0, 0.0);
   glRotatef(k*t, 0.0, 1.0, 0.0);
   glutSolidTeapot(1.0);
glPopMatrix();
```

```
glUseProgram(program[1]);
glPushMatrix();
   glTranslatef(0.0f, 0.0f, -5.0f);
   glRotatef(k*t, 1.0, 0.0, 0.0);
   glRotatef(k*t, 0.0, 1.0, 0.0);
   glutSolidTeapot(1.0);
glPopMatrix();

glUseProgram(program[2]);
glPushMatrix();
   glTranslatef(3.0f, 0.0f, -5.0f);
   glRotatef(k*t, 1.0, 0.0, 0.0);
   glRotatef(k*t, 0.0, 1.0, 0.0);
   glutSolidTeapot(1.0);
glPopMatrix();
```

Figura 3 - Phong lighting

TOON SHADING

La quarta parte dell'esercitazione prevede la realizzazione di un outline sulla mesh (già colorata in stile cartone animato).

Per fare ciò a livello di vertex shader si sono calcolati i vettori N ed E:

```
N = normalize(gl_NormalMatrix * gl_Normal);
vec4 nullVector = vec4(0, 0, 0, 1);
E = normalize(nullVector.xyz - eyePosition.xyz);
```

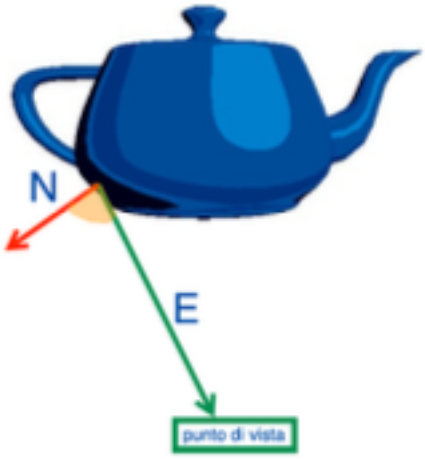

Figura 4 - Vettori normale e vista

A livello di fragment shader invece si è modificato il colore per i punti la cui *intensity* (coseno dell'angolo tra i due vettori) è inferiore ad una certa soglia, cioè quando il vettore normale è praticamente ortogonale a quello di vista.

```
intensity = dot(normalize(E), normalize(N));
if (intensity < 0.3)
   color = vec4(1.0,0.0,0.1,1.0); // red
```

Figura 5 - Toon shading

MORPHING

La quinta parte dell'esercitazione prevede di realizzare due funzionalità:

- disegno del triangolo *filled* con cambiamento del colore periodico nel tempo
- implementare il cambiamento di forma nel tempo tra triangolo e quadrato

Per implementare il primo punto è necessario modificare l'istruzione glBegin() nel seguente modo:

```
glBegin(GL_TRIANGLES);
```

e abilitare tramite la funzione mix l'interpolazione del colore (array colorOne e colorTwo) in base al parametro s:

```
gl_FrontColor = mix(gl_Color, colors2, s);
```

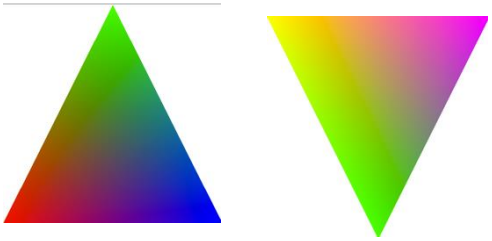

Figura 6 - Morphing triangolo

Per il secondo punto si procede in maniera analoga a quanto fatto per il triangolo, definendo vertici/colori di partenza e di arrivo del quadrato. Inoltre si modifica l'istruzione glBegin() nel seguente modo:

```
glBegin(GL_QUADS);
```

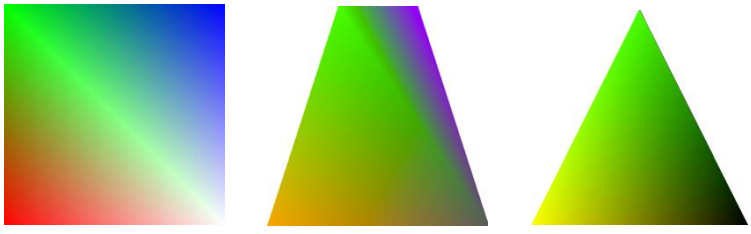

Figura 7 - Morphing triangolo/quadrato

BUMP MAPPING

La sesta parte dell'esercitazione prevede di modificare l'input per il calcolo delle normali che, a loro volta, definiscono il bump mapping. Il quadrato esterno presenta un mapping a scacchiera, mentre quello interno a cerchi concentrici.

```
float data[N+1][N+1];
for (i = 0; i < N + 1; i++) // Quadrato esterno
  for (j = 0; j < N + 1; j++)
    data[i][j] = pow(sin(i), 2) + pow(cos(j), 2);

for (i = N / 4; i < 3 * N / 4; i++) // Quadrato interno
  for (j = N / 4; j < 3 * N / 4; j++) {
    int value = pow(i - (N/2), 2) + pow(j - (N/2), 2);
    data[i][j] = value % 250;
  }
```

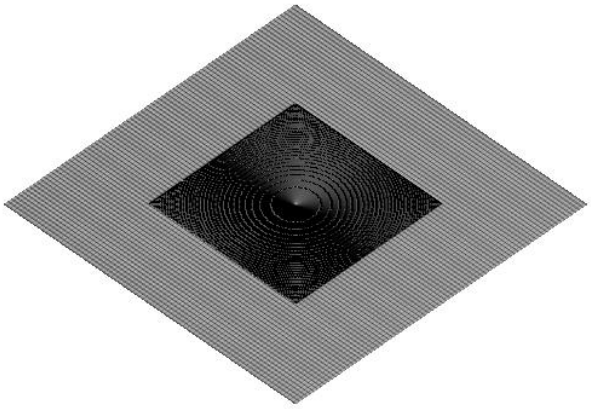

Figura 8 - Bump mapping

CUBE ENVIRONMENT MAPPING

La settima parte dell'esercitazione prevede di mappare delle immagini al posto dei colori (utilizzando ad esempio immagini prese da quelle dell'esercitazione 5). Per fare ciò si è semplicemente aggiunta la libreria per la lettura di immagini in formato .bmp ed eseguito il mapping sul cubo analogamente all'esercitazione 5.

```
RgbImage img1 = getTextureFromFile(cube[0]);
RgbImage img2 = getTextureFromFile(cube[1]);
RgbImage img3 = getTextureFromFile(cube[2]);
RgbImage img4 = getTextureFromFile(cube[3]);
RgbImage img5 = getTextureFromFile(cube[4]);
RgbImage img6 = getTextureFromFile(cube[5]);

glTexImage2D( GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0, GL_RGB, 256, 256, 0, GL_RGB,
GL_UNSIGNED_BYTE, ImageData(&img1));
glTexImage2D( GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGB, 256, 256, 0, GL_RGB, GL_UNSIGNED_BYTE,
ImageData(&img2));
glTexImage2D( GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, GL_RGB, 256, 256, 0, GL_RGB,
GL_UNSIGNED_BYTE, ImageData(&img3));
glTexImage2D( GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0, GL_RGB, 256, 256, 0, GL_RGB, GL_UNSIGNED_BYTE,
ImageData(&img4));
glTexImage2D( GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, GL_RGB, 256, 256, 0, GL_RGB,
GL_UNSIGNED_BYTE, ImageData(&img5));
glTexImage2D( GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0, GL_RGB, 256, 256, 0, GL_RGB, GL_UNSIGNED_BYTE,
ImageData(&img6));
```

Figura 9 - Cube environment mapping