Fondamenti di Computer Graphics M (8 cfu)

Lab 3 - Navigazione interattiva in scena con modelli mesh 3D

Michele Righi

Settembre 2023

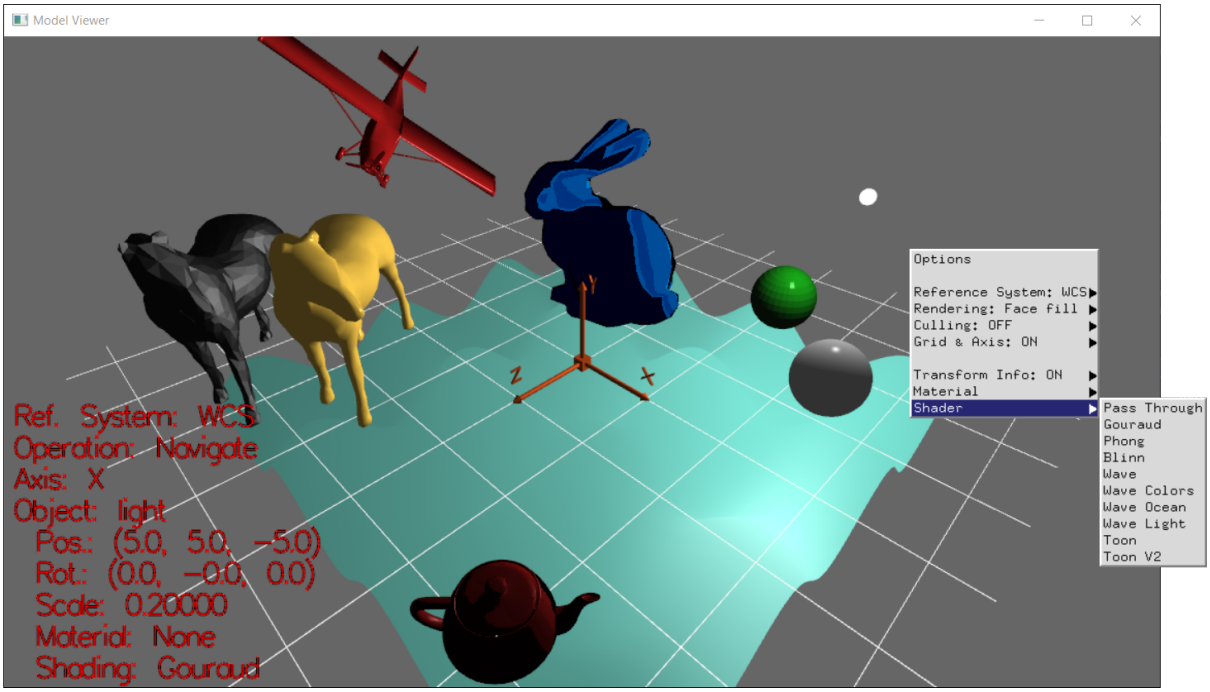
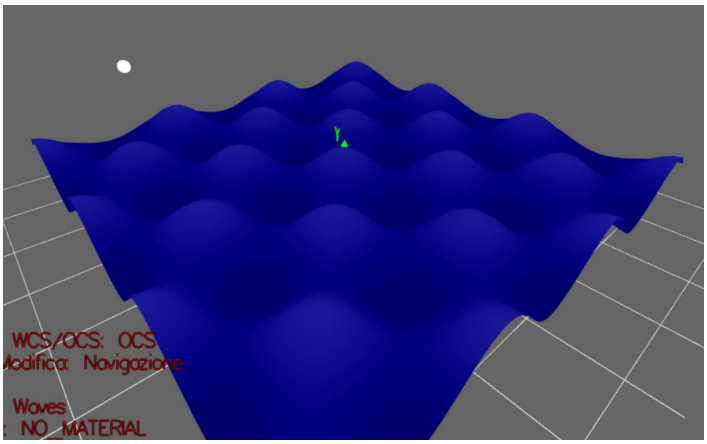
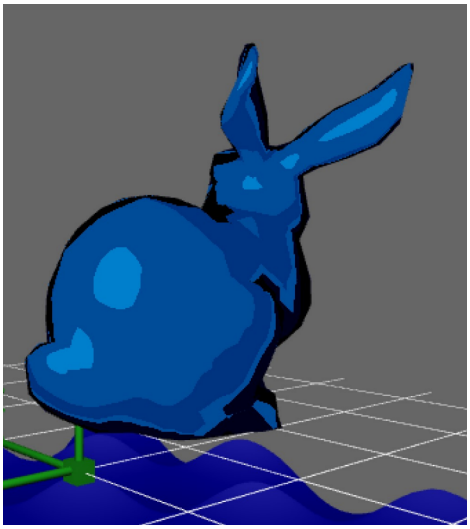


Figure 1: Demo Applicazione



(a) Wave shader



(b) Toon shader

Traccia

Estendere il programma secondo le seguenti specifiche:

1. **Caricamento e visualizzazione oggetti**
   (a) Calcolo e memorizzazione delle normali ai vertici per i modelli mesh poligonali. Visualizzazione in modalità normali alle facce (flat shading) e normali ai vertici (sia GOURAUD shading sia PHONG shading).
   (b) Provare a creare un materiale diverso da quelli forniti.
   (c) Prendete visione di come sono gestiti gli shader GOURAUD PHONG BLINN per l'illuminazione e shading.
   (d) **Wave motion**: Si crei l'animazione di un height field mesh (oggetto mesh *GridPlane.obj* contenuto nella directory Mesh) modificando la posizione dei vertici in un vertex shader *v_wave.glsl* (effetto da ottenere mostrato in Fig. 2a). Utilizzare la variabile elapsed time t, passata da applicazione al vertex shader, per riprodurre il moto ondoso ottenuto con la sola modifica della coordinata y mediante la formula:

$$v_y = asin(\omega t + 10v_x)sin(\omega t + 10v_z),$$

   dove l'ampiezza dell'oscillazione a e la frequenza ω siano scelte a piacere, es. $a = 0.1$ e $\omega = 0.001$.
   (e) **Toon shading**: Realizzare gli shaders *v_toon.glsl* e *f_toon.glsl* per la resa non-fotorealistica nota comunemente come "Toon shading". Un esempio è illustrato in Fig. 2b.
2. **Navigazione interattiva in scena** Implementare le seguenti funzionalità per il sistema interattivo:
3. **Trasformazione degli oggetti in scena**
   Si richiede di gestire opportunamente nella funzione modifyModelMatrix() le trasformazioni di traslazione, rotazione e scalatura dei **singoli** oggetti in scena rispetto al sistema di riferimento WCS o OCS (World e Object Coordinate Systems).

Funzionalità	***Descrizione***	***Tasti***
Pan oriz. camera (sx/dx)	modifica di una stessa quantità sia la posizione della camera (C) che del punto di riferimento in scena (A)	(CTRL+ scroll wheel up/down)

Table 1: Tabella delle funzionalità da sviluppare (indicate in grassetto) nell'applicazione model viewer

Permettere anche lo spostamento (traslazione) dell'oggetto LUCE.

La selezione del sistema di riferimento (WCS o OCS) rispetto al quale eseguire tale trasformazione avviene mediante menu pop-up.

I singoli oggetti devono essere selezionabili tramite i tasti frecce dx/sx.

Gli assi x,y,z rispetto ai quali effettuare la trasformazione sono selezionabili da keyboard.

La selezione del tipo di trasformazione avviene da keyboard tramite i tasti:

- 'g' modalità traslazione
- 'r' modalità rotazione
- 's' modalità scalatura

La quantità di spostamento è incrementale/decrementale e controllata da WHEEL up(+)/ down(-).

1 Introduzione

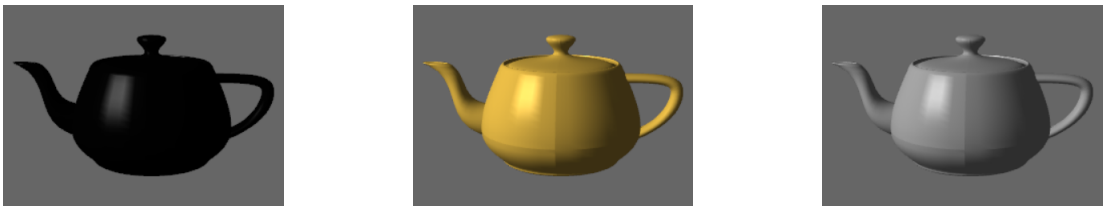
Per questa soluzione ho deciso di fare un refactoring generale del codice, separando ciascun componente in un modulo dedicato. Di maggiore interesse sono i seguenti file:

- `03_main.cpp` contiene il main, le funzioni di inizializzazione e quelle per disegnare i vari oggetti di scena;
- `mesh.h` fornisce una funzione per caricare una mesh poligonale da un file **.obj**, salvandola in una struttura dati `Mesh`. Permette di specificare il materiale, lo shader, i valori di transform e se porre le normali ai vertici o alle facce. Questa funzione restituisce anche l'esito, così da poter continuare l'esecuzione, se si prova a caricare mesh non valide (file assenti/formattati male);
- `maetrial.h` fornisce una funzione per inizializzare la lista dei materiali, che vengono poi salvati in una mappa che li distingue in base a un ID;
- `shader.h` fornisce una funzione per inizializzare la lista degli shader, salvati in una mappa che li distingue in base a un ID (che è diverso da quello assegnato da OpenGL);
- `menu.h` fornisce una funzione per inizializzare il menù dell'applicazione.

2 Soluzione

2.1 Materiali Aggiuntivi

In aggiunta ai materiali forniti, ne ho creati alcuni nuovi:



(a) `BLACK_PLASTIC` (b) `GOLD` (c) `SILVER`

(d) `BRONZE` (e) `COPPER` (f) `TURQUOISE`

Figure 3: Nuovi materiali con Gouraud shading

2.2 Pan Orizzontale

Per implementare il pan orizzontale della camera, dobbiamo prima ottenere il vettore di traslazione orizzontale (vettore $\vec{u}$ in Fig. 4).

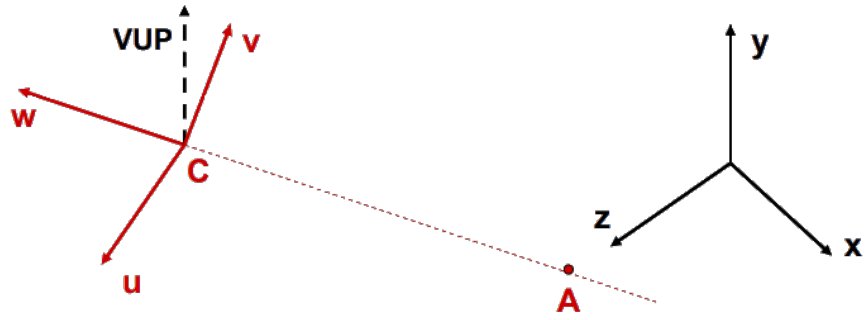


Figure 4: Camera Frame in WCS

Per ottenerlo possiamo sfruttare il vettore $\overrightarrow{VUP}$ (View Up Vector) ed il vettore che partendo dalla camera punta nella direzione verso cui questa è rivolta (vettore $\overrightarrow{CA}$ in Fig. 4, dove C è la posizione della camera, ed A il suo target). Facendo il cross product tra questi due vettori, otteniamo `rightDirection`, ovvero il vettore (orientato verso destra) lungo cui la camera si può spostare orizzontalmente.

```
void moveCameraLeft() {
    glm::vec3 cameraDirection = ViewSetup.target - ViewSetup.position;
    glm::vec3 rightDirection = glm::cross(cameraDirection, glm::vec3(ViewSetup.upVector)) *
        CAMERA_TRASLATION_SPEED;
    ViewSetup.position -= glm::vec4(rightDirection, 0.0);
    ViewSetup.target -= glm::vec4(rightDirection, 0.0);
}

void moveCameraRight() {
    glm::vec3 cameraDirection = ViewSetup.target - ViewSetup.position;
    glm::vec3 rightDirection = glm::cross(cameraDirection, glm::vec3(ViewSetup.upVector)) *
        CAMERA_TRASLATION_SPEED;
    ViewSetup.position += glm::vec4(rightDirection, 0.0);
    ViewSetup.target += glm::vec4(rightDirection, 0.0);
}
```

2.3 Shader

2.3.1 Wave

Lo shader che implementa l'effetto a onda sfrutta la funzione seno per assegnare il componente y dei vertici, in base a: la loro posizione sul piano xz, il tempo, la frequenza e una height scale.

```
// Vertex Shader
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
out vec3 Color;

uniform mat4 P;
uniform mat4 V;
uniform mat4 M; // = position * rotation * scaling
uniform float time; // in milliseconds
uniform float xFreq = 0.001, zFreq = 0.001;
uniform float heightScale = 0.1;

void main() {
    vec4 v = vec4(aPos.x, aPos.y, aPos.z, 1.0);

    v.y = aPos.y + heightScale * sin(time * xFreq + 5 * aPos.x) +
        heightScale * sin(time * zFreq + 5 * aPos.z);

    gl_Position = P * V * M * v;
}
```

Il fragment shader è semplicemente un pass-through che trasmette il colore in ingresso, direttamente in uscita:

```
// Fragment Shader
in vec3 Color;
out vec4 FragColor;

void main() {
   FragColor = vec4(Color, 1.0);
}
```

Poiché il vertex shader non passa alcun colore, gli oggetti con questa shader vengono visualizzati completamente neri (vedi Fig. 5).

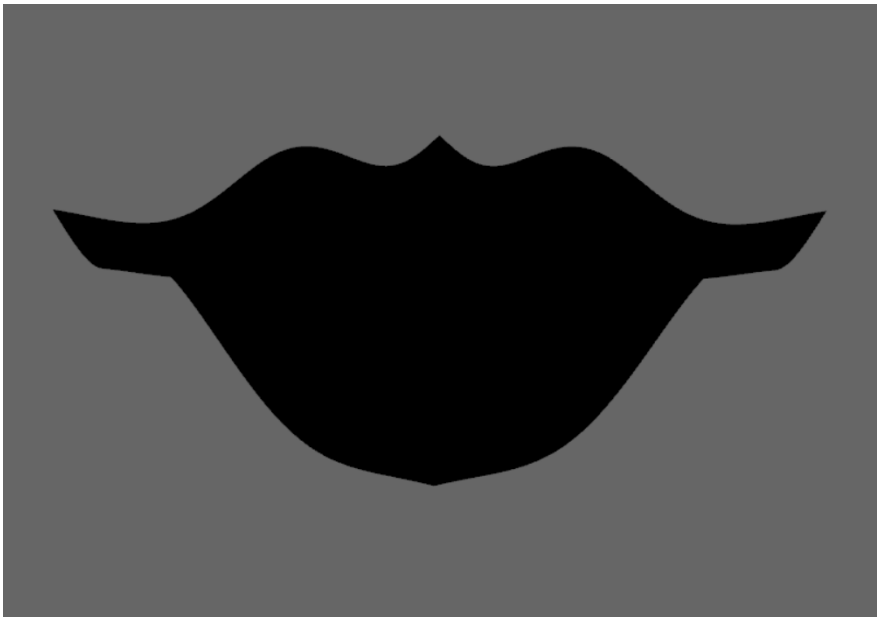


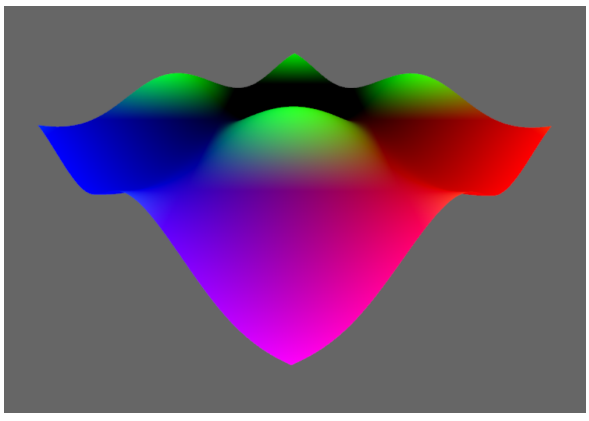
Figure 5: Shader `WAVE`

Wave Colorata Ho creato le seguenti 2 shader per fare delle prove di visualizzazione dei colori. A differenza della precedente, il vertex shader trasmette alla fase di rasterizzazione anche un **colore**, che è dato dalle componenti dei vertici. In Fig. 6a viene utilizzato il vertex shader (`v_wave_colors.glsl`) con il parametro Color che varia in base alle coordinate dei vertici:

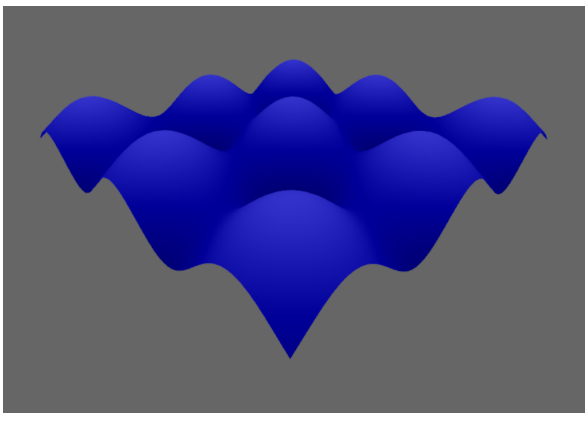
```
1 Color = vec3(v.x, v.y * 5, v.z);
```

In Fig. 6b viene utilizzato il vertex shader (`v_wave_ocean.glsl`) con il parametro Color che varia in base alla coordinata Y dei vertici, in modo da simulare l'acqua e la schiuma dell'oceano:

```
1 Color = vec3(v.y, v.y, v.y + 0.6);
```



(a) Shader `WAVE_COLORS`



(b) Shader `WAVE_OCEAN` con materiale `TURQUOISE`

Wave con Luce Il vertex shader `v_wave_ligh.glsl` implementa l'effetto delle onde colpite da una fonte luminosa, sfruttando le normali ai vertici. Ai calcoli eseguiti dalla shader `WAVE` di base, vengono aggiunte le seguenti operazioni:

```
// Transform vertex  position into eye (VCS) coordinates
vec4 eyePosition = V * M * vec4(aPos, 1.0);
// Transform Light  position into eye (VCS) coordinates
vec4 eyeLightPos = V * vec4(light.position, 1.0);
// Transform vertex normal into VCS
vec3 N = normalize(transpose(inverse(mat3(V * M))) * aNormal);

// Compute vectors E, L, R in VCS
vec3 E = normalize(-eyePosition.xyz);
vec3 L = normalize((eyeLightPos - eyePosition).xyz);
vec3 R = reflect(-L, N);

// Ambient
vec3 ambient = light.power * material.ambient;

// Diffuse
float diff = max(dot(L, N), 0.0);
vec3 diffuse = light.power * light.color * diff * material.diffuse;

// Specular
float spec = pow(max(dot(E, R), 0.0), material.shininess);
vec3 specular =  light.power * light.color * spec * material.specular;

Color = ambient + diffuse + specular;
```

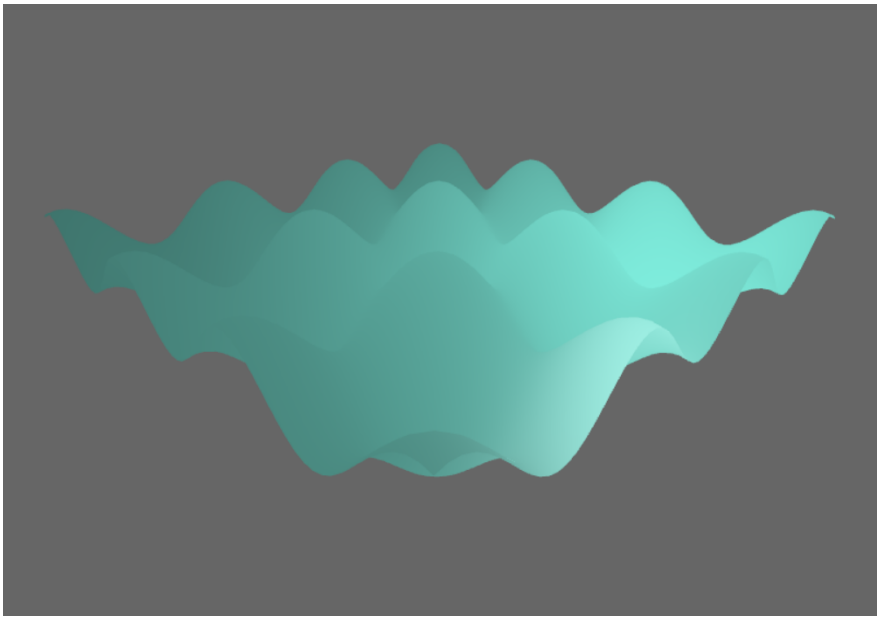


Figure 7: Shader `WAVE_LIGHT`

2.3.2 Toon

Di seguito il codice del vertex shader `v_toon.glsl`:

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

struct PointLight {
    vec3 position;
    vec3 color;
    float power;
 };
uniform PointLight light;
uniform mat4 P, V, M;

out vec3 E, N, L;

void main() {
    gl_Position = P * V * M * vec4(aPos, 1.0);

    vec4 eyePosition = V * M * vec4(aPos, 1.0);
    vec4 eyeLightPos = V * vec4(light.position, 1.0);

    E = normalize(-eyePosition.xyz);
    N = normalize(transpose(inverse(mat3(V * M))) * aNormal);
    L = normalize((eyeLightPos - eyePosition).xyz);
}
```

Da notare che, a differenza delle altre shader, questa passa in uscita i vettori $\vec{E}$, $\vec{N}$, $\vec{L}$. Infatti, questi (normalizzati) ci servono per calcolare:

- dot product fra $\vec{L}$ ed $\vec{N}$ per l'**intensità**, dunque in quale "fascia" ci troviamo (in Fig. 8 blu scuro, blu, blu chiaro, azzurro);
- dot product fra $\vec{E}$ ed $\vec{N}$ per aggiungere il **bordo nero** attorno alla sagoma della mesh.

Figure 8: Shader `TOON`

Toon V2 Ho creato questa shader per applicare l'effetto cartoon, sfruttando il materiale dell'oggetto, a differenza del precedente, che era mono-tonalità. Ho creato anche una shader `TOON_V2` che applicasse l'effetto *cartone* al materiale dell'oggetto:

```
const float a = 0.8, b = 0.6, c = 0.5, d = 0.6, e = 0.1;

in vec3 Color;
in vec3 N, E, L;

out vec4 fragColor;

void main() {
    vec4 color = vec4(Color, 1.0);

    float intensity = dot(normalize(L),normalize(N));
    if (intensity > 0.95)
        color *= a;
    else if (intensity > 0.5)
        color *= b;
    else if (intensity > 0.25)
        color *= c;
    else if (intensity > 0.15)
        color *= d;
    else
        color *= e;

    float aa = dot(normalize(E), normalize(N));
    if(aa >= 0.0 && aa < 0.30)
        color = vec4(0.0, 0.0, 0.0, 1.0);

    fragColor = color;
}
```

(a) Shader `TOON_V2` con materiale `BRONZE`

(b) Shader `TOON_V2` con materiale `EMERALD`

2.4 Trasformazione degli Oggetti in Scena

Per eseguire le operazioni di traslazione, rotazione e scalatura, ho utilizzato le funzioni fornite dalla libreria GLM, sulla matrice di modellazione dell'oggetto correntemente selezionato:

```
void modifyModelMatrix(
    glm::vec3 translation_vector,
    glm::vec3 rotation_vector,
    GLfloat angle,
    GLfloat scale_factor)
{
    glm::mat4 newModelMatrix = objects.at(selected_obj).M;

    // Translation
    newModelMatrix = glm::translate(newModelMatrix, translation_vector);
    // Rotation
    if (transformMode == OCS) {
        // Rotation with pivot on the object's center
        newModelMatrix = glm::rotate(newModelMatrix, glm::radians(angle), rotation_vector);
    }
    else if (transformMode == WCS) {
        // Rotation with pivot on the scene center
        glm::vec3 objectPosition = newModelMatrix[3];
        glm::vec3 origin = glm::vec3(0, 0, 0);

        newModelMatrix = glm::translate(newModelMatrix, origin - objectPosition);
        newModelMatrix = glm::rotate(newModelMatrix, glm::radians(angle), rotation_vector);
        newModelMatrix = glm::translate(newModelMatrix, objectPosition - origin);
    }
    // Scaling
    newModelMatrix = glm::scale(newModelMatrix, glm::vec3(scale_factor, scale_factor, scale_factor));
    objects.at(selected_obj).M = newModelMatrix;
}
```

In base alla modalità che l'utente sceglie (Translate, Rotate e Scale) e all'asse di riferimento, l'oggetto selezionato viene trasformato. Lo scaling viene effettuato mantenendo la dimensione identica per ciascun asse. La rotazione viene eseguita in base al sistema di riferimento:

- in **OCS** (Object Coordinate System), l'oggetto ruota attorno al proprio centro;
- in **WCS** (World Coordinate System), l'oggetto ruota attorno all'origine del sistema di riferimento della scena.

2.4.1 Trasformazioni Luce

Lo spostamento della luce si comporta in modo diverso: poiché gli shader, per creare l'effetto della luminosità, necessitano di conoscere la posizione della luce, quando trasliamo questo oggetto, bisogna aggiornare anche il valore negli shader (uniform).

Per farlo, quando viene effettuata un'operazione di transform, controlliamo se il nome dell'oggetto corrisponde con quello della fonte luminosa e, in caso affermativo, aggiorniamo la posizione e lo

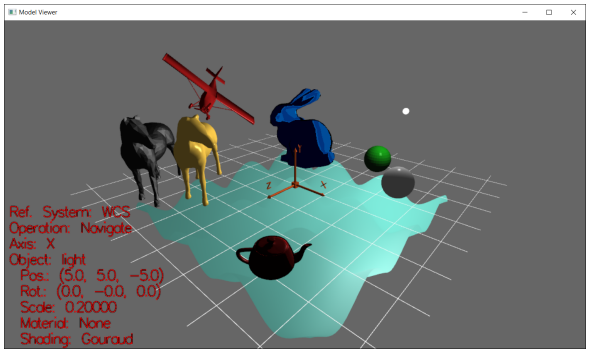
scaling (la rotazione non è interessante, in quanto la fonte luminosa è puntiforme):

```
if (app.operationMode == MODE_TRASLATING ||
    app.operationMode == MODE_ROTATING ||
    app.operationMode == MODE_SCALING)
{
    if (objects.at(selectedObj).name == "light")
    {
        light.position = getPosition(objects.at(selectedObj).M);
        light.power = getScalingFactor(objects.at(selectedObj).M) * LIGHT_SCALE_FACTOR;
    }
}
```

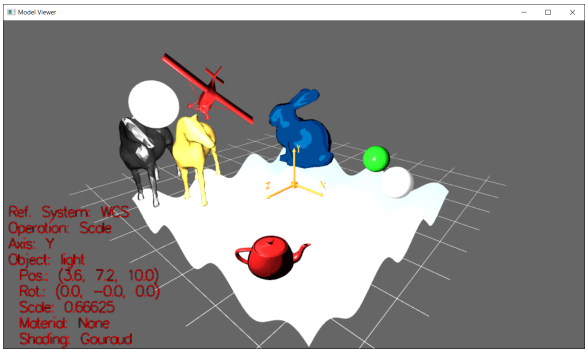
la funzione `getPosition()` ricava il vettore della posizione dalla quarta colonna della matrice di modellazione (i primi 3 elementi sono rispettivamente x, y, z); la funzione getScalingFactor, ricava il fattore di scaling e lo utilizza per aggiornare l'intensità della luce, moltiplicandolo per 5.

Dopodiché, nelle funzioni di draw, prima di eseguire il binding e la draw dei vertici, aggiorniamo i valori della luce, per ciascuno shader che utilizza la fonte luminosa:

```
// Update light position
glUniform3f(object.shader.lightUniform.light_position_pointer,
    light.position.x,
    light.position.y,
    light.position.z);
// Update light intensity
glUniform1f(object.shader.lightUniform.light_power_pointer,
    light.power);
```



(a) Scena con luce di default



(b) Scena con luce traslata e scalata

3 Extra

3.1 Menù

Ho fatto un refactoring anche del menù: per ogni impostazione selezionabile ho creato un submenu (anche quelle che presentavano solo i valori ON/OFF), e ho aggiunto la possibilità di mostrare/nascondere la griglia e gli assi del sistema di riferimento, le informazioni di transform dell'oggetto, e la lista degli shader selezionabili.

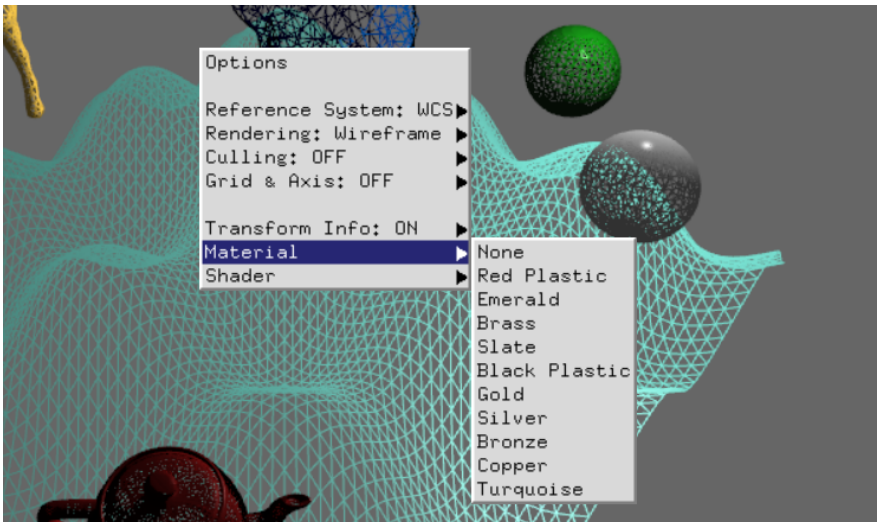


Figure 11: Menu

3.2 UI

Ho aggiunto alla UI il nome dello shader, ed i valori di trasformazione dell'oggetto selezionato (posizione, rotazione e scaling).

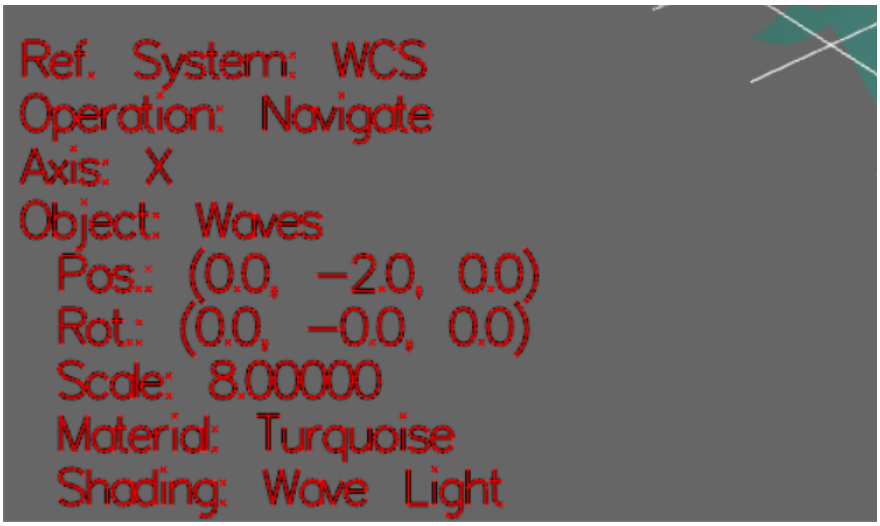


Figure 12: User Interface

3.3 Controlli

Oltre ai controlli di base, già forniti nel codice dell'applicazione iniziale, ho aggiunto i seguenti:

- `F1`, per resettare la scena;
- `F2`, `F3`, `F4`, `F5`, per impostare velocemente diverse configurazioni della luce;
- `C`, per resettare la camera;

3.4 Scena Rotta

Questa scena è stata caricata usando le funzioni di lettura senza specificare il numero di valori da leggere. Come risultato, si hanno delle mesh con vertici dai valori assurdi:

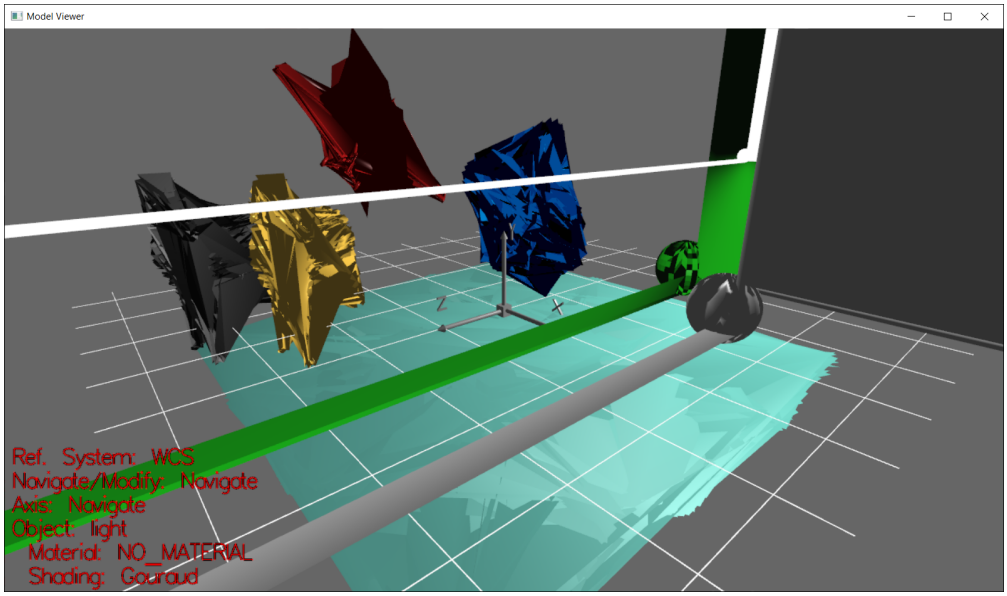


Figure 13: Scena Rotta