REPORT PER LAB_03

Matteo giri

Introduzione e comandi

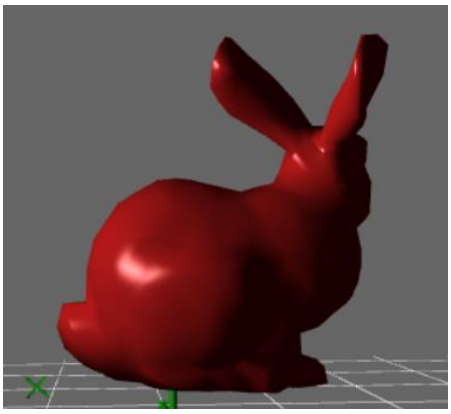
Questo è il report per l'esercizio di laboratorio **LAB_03**. Non c'è nessun comando particolare che merita spiegazione in quanto tutti i comandi utilizzabili sono già evidenziati nell'esercizio.

Dettagli realizzativi

STEP 1.a: Calcolo e memorizzazione delle normali ai vertici per i modelli mesh poligonali. Visualizzazione in modalità normali alle facce e normali ai vertici.

Il calcolo e la memorizzazione delle normali ai vertici sono già implementati di base. Infatti, se le normali ai vertici non sono già fornite dal file caricato queste vengono calcolate come il cross product tra due lati del triangolo. Per quanto riguarda la memorizzazione basta decommentare le righe di codice relative all'uno o all'altro tipo di memorizzazione nella funzione *loadObjFile(). Nella memorizzazione delle normali ai vertici vengono salvate le normali per ogni vertice, mentre in quella alle facce viene salvata solo una normale a faccia.*

Esempio di visualizzazione in modalità normali a vertici con Phong:



Esempio di visualizzazione in modalità normali a facce con Phong:

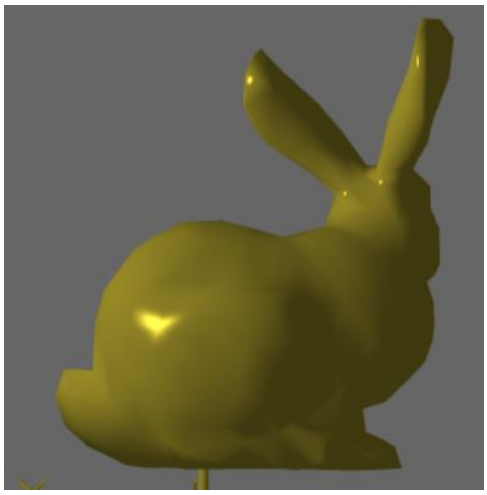
STEP 1.b: Provare a creare un materiale diverso da quelli forniti

Ho scelto di creare un materiale “gold” che rispecchi l’oro. Per prima cosa ho creato le variabili per i parametri ambiente, diffusivo, speculare, e lucentezza:

```
glm::vec3 gold_ambient = { 0.24725,0.2245,0.0645 },
    gold_diffuse = { 0.34615,0.3143,0.0903 },
    gold_specular{ 0.797357,0.723991,0.208006 };
GLfloat gold_shininess = 83.2f;
```

Per poi inserirli all’interno dell’array *materials*, e nel menù dei materiali visualizzato durante l’esecuzione.

Esempio:



STEP 1.d: Wave motion

Per creare l’effetto ad onda con gli shader ho realizzato uno shader che usa come base lo shader Gouraud e ho modificato i valori della posizione dei vertici nel vertex shader come suggerito dal testo dell’esercizio. In particolare, il fragment shader *f_wave.glsl* è esattamente uguale a quello di Gouraud, in quanto non ci interessa andare a lavorare con i pixel ma direttamente con i vertici nello spazio 3D. Il vertex shader *v_wave.glsl* è uguale a quello di Gouraud ma con l’aggiunta della funzione sinusoidale per cambiare la posizione dei vertici in base al parametro t che viene passato come input del vertex shader insieme agli altri parametri quale posizione e caratteristiche del materiale:

```
float a = 0.1;
float w = 0.001;
vec4 v = vec4(aPos,1.0);
v.y = a*sin(w*t + 10*v.x)*sin(w*t + 10*v.z),
gl_Position = P * V * M * v;
```

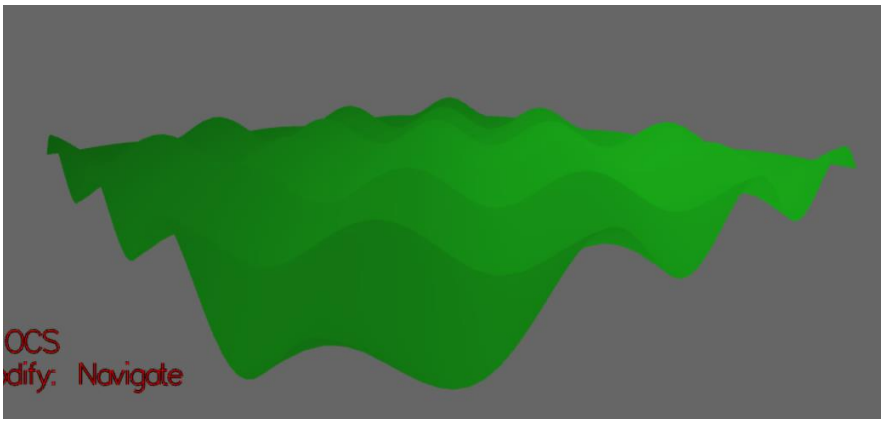
Nell’ *initShader()* ho inserito il loading dello shader da file andando a definire i vari parametri dello shader:

```
//Wave Shader Loading
shaders_IDs[WAVE] = createProgram(ShaderDir + "v_wave.glsl", ShaderDir + "f_wave.glsl");
base_unif.P_Matrix_pointer = glGetUniformLocation(shaders_IDs[WAVE], "P");
base_unif.V_Matrix_pointer = glGetUniformLocation(shaders_IDs[WAVE], "V");
base_unif.M_Matrix_pointer = glGetUniformLocation(shaders_IDs[WAVE], "M");
base_unif.time_delta_pointer = glGetUniformLocation(shaders_IDs[WAVE], "t");
base_uniforms[ShadingType::WAVE] = base_unif;
light_unif.material_ambient = glGetUniformLocation(shaders_IDs[WAVE], "material.ambient");
light_unif.material_diffuse = glGetUniformLocation(shaders_IDs[WAVE], "material.diffuse");
light_unif.material_specular = glGetUniformLocation(shaders_IDs[WAVE], "material.specular");
light_unif.material_shininess = glGetUniformLocation(shaders_IDs[WAVE], "material.shininess");
light_unif.light_position_pointer = glGetUniformLocation(shaders_IDs[WAVE], "light.position");
light_unif.light_color_pointer = glGetUniformLocation(shaders_IDs[WAVE], "light.color");
light_unif.light_power_pointer = glGetUniformLocation(shaders_IDs[WAVE], "light.power");
light_uniforms[ShadingType::WAVE] = light_unif;
//Rendiamo attivo lo shader
glUseProgram(shaders_IDs[WAVE]);
//Shader uniforms initialization
glUniform3f(light_uniforms[WAVE].light_position_pointer, light.position.x, light.position.y, light.position.z);
glUniform3f(light_uniforms[WAVE].light_color_pointer, light.color.r, light.color.g, light.color.b);
glUniform1f(light_uniforms[WAVE].light_power_pointer, light.power);
```

E nella *drawscene() sono andato a definire il caricamento dello shader e le variabili da passargli:*

```
case ShadingType::WAVE:
    glUseProgram(shaders_IDs[WAVE]);
    // Caricamento matrice trasformazione del modello
    glUniformMatrix4fv(base_uniforms[WAVE].M_Matrix_pointer, 1, GL_FALSE, value_ptr(objects[i].M));
    // Time setting
    glUniform1f(base_uniforms[WAVE].time_delta_pointer, clock());
    //Material loading
    glUniform3fv(light_uniforms[WAVE].material_ambient, 1, glm::value_ptr(materials[objects[i].material].ambient));
    glUniform3fv(light_uniforms[WAVE].material_diffuse, 1, glm::value_ptr(materials[objects[i].material].diffuse));
    glUniform3fv(light_uniforms[WAVE].material_specular, 1, glm::value_ptr(materials[objects[i].material].specular));
    glUniform1f(light_uniforms[WAVE].material_shininess, materials[objects[i].material].shininess);
    break;
```

Risultato:

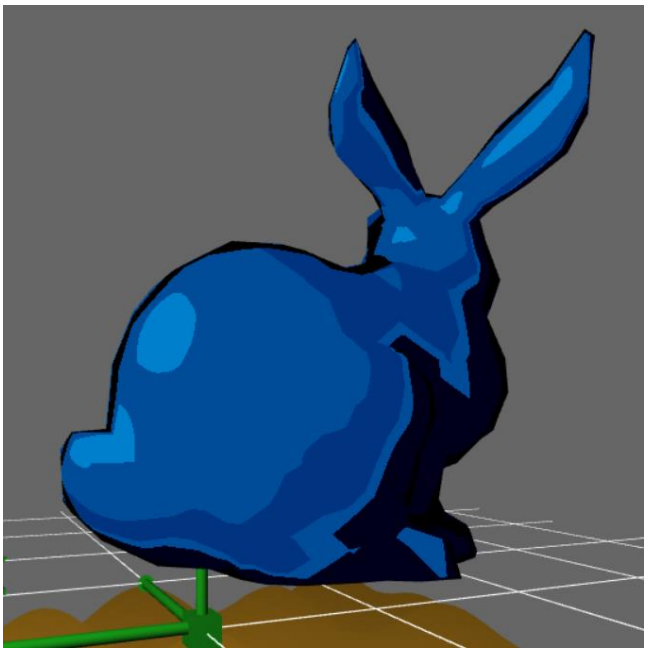


STEP 1.e: Toon shading

Per realizzare l'effetto cartone con gli shaders ho utilizzato come base lo shader di Phong. Il vertex shader *v_toon.glsl* è identico a *v_phong.glsl,* mentre è il fragment shader *f_toon.glsl* che realizza veramente l'effetto toon. Praticamente al suo interno vado a calcolare l'angolo tra la direzione con cui la luce colpisce il pixel e la normale del pixel e in base al suo valore utilizzo delle soglie per decidere il colore da utilizzare in quel range. Questo crea un cambio di colore distaccato (non sfumato). Infine, utilizzo l'angolo tra la direzione di vista e la normale del pixel per andare a colorare il pixel di nero se l'angolo è all'interno di un range (per creare l'effetto contorno).

Gli altri passaggi per collegare lo shader al programma principali sono pressocché identici a quelli visti nel punto 1.d.

Risultato:



STEP:2 Spostare la camera nello spazio

Per permettere lo spostamento della camera a destra e sinistra nello schermo ho semplicemente riarrangiato il codice già presente per spostare la camera su o giù (è bastato fare un prodotto tra vettori in meno in quanto lo slide_vector identifica già la direzione giusta):

```
void moveCameraLeft()
{
    glm::vec3 direction = ViewSetup.target - ViewSetup.position;
    glm::vec3 slide_vector = glm::normalize(glm::cross(direction, glm::vec3(ViewSetup.upVector)));
    glm::vec3 sideDirection = slide_vector * CAMERA_TRASLATION_SPEED * 10.0f;
    ViewSetup.position -= glm::vec4(sideDirection, 0.0);
    ViewSetup.target -= glm::vec4(sideDirection, 0.0);
}

void moveCameraRight()
{
    glm::vec3 direction = ViewSetup.target - ViewSetup.position;
    glm::vec3 slide_vector = glm::normalize(glm::cross(direction, glm::vec3(ViewSetup.upVector)));
    glm::vec3 sideDirection = slide_vector * CAMERA_TRASLATION_SPEED * 10.0f;
    ViewSetup.position += glm::vec4(sideDirection, 0.0);
    ViewSetup.target += glm::vec4(sideDirection, 0.0);
}
```

STEP 3: Trasformazione degli oggetti in scena

Per permettere di trasformare gli oggetti in scena scegliendo il tipo di trasformazione e il sistema di riferimento mi è bastato realizzare la funzione *modifyModelMatrix* (in quanto tutto il resto è gia pronto):

```
void modifyModelMatrix(glm::vec3 translation_vector, glm::vec3 rotation_vector, GLfloat angle, GLfloat scale_factor)
{
    Object object_to_modify = objects[selected_obj];
    glm::mat4 M_to_modify = object_to_modify.M;

    if (TransformMode == OCS) {
        M_to_modify = glm::translate(M_to_modify, translation_vector);
        M_to_modify = glm::rotate(M_to_modify, angle, rotation_vector);
        M_to_modify = glm::scale(M_to_modify, glm::vec3(scale_factor, scale_factor, scale_factor));
    }
    else {

        glm::mat4 translation_matrix = glm::translate(glm::mat4(1.0f), translation_vector);
        glm::mat4 rotation_matrix = glm::rotate(glm::mat4(1.0f), angle, rotation_vector);
        glm::mat4 scale_matrix = glm::scale(glm::mat4(1.0f), glm::vec3(scale_factor, scale_factor, scale_factor));
        M_to_modify = scale_matrix*rotation_matrix*translation_matrix * M_to_modify;
    }


    objects[selected_obj].M = M_to_modify;
}
```

La funzione prende la matrice M dell'oggetto selezionato e in base al sistema di riferimento le vengono applicati i vari vettori di traslazione, rotazione e scala. In particolare, se il sistema di riferimento scelto è quello dell'oggetto allora le variazioni dovranno essere relative all'oggetto stesso e quindi basta utilizzare le funzioni translate, rotate e scale di glm sulla matrice M dell'oggetto. Se il sistema di riferimento scelto è il WCS allora dovremo creare le tre matrici di traslazione, rotazione e scala andando ad applicare le funzioni di glm a delle matrici identità. Infine, si moltiplicano queste matrici alla matrice da modificare M.