REPORT PER LAB_07

Matteo giri

Introduzione e comandi

Questo è il report per l'esercizio di laboratorio **LAB_07**. E' possibile muoversi a destra e a sinistra su di una curva di Bèzier CHIUSA preimpostata centrata sull'oggetto al momento selezionato con i tasti 'k' e 'l'. Per vedere l'effetto trasparenza dello step 9 si può cambiare il parametro obj.blended degli oggetti trasparenti a true (è preimpostato a false per le finestre perché non ho pensato troppo a rendere il codice efficiente e quindi nel mio laptop lagga se li lascio a true).

Dettagli realizzativi

STEP 1: Permettere il movimento della camera virtuale lungo un percorso (curva) in modalità look at (mediante curva chiusa di Bèzier) in modo che la camera si muova lungo un percorso con centro di interesse un oggetto in scena

Per realizzare un movimento fluido su una curva di Bèzier chiusa centrata su un oggetto in scena ho per prima cosa riadattato le funzioni per generare una curva di Bèzier uniforme del LAB_01 semplicemente aggiungendogli una componente per la terza dimensione (in modo da renderla una curva 3D):

```
//function that applies deCasteljau method to create a Bezier curve
void deCasteljau(float tempArray[MaxNumPts][3], float t, float(&result)[3], float NumPts) {
    float pi0[MaxNumPts][3]; //output array
    int n = NumPts - 1; //curve degree

    for (int i = 0; i < MaxNumPts; i++) { //copy the array
        pi0[i][0] = tempArray[i][0];
        pi0[i][1] = tempArray[i][1];
        pi0[i][2] = tempArray[i][2];
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= n - i; j++) {
            pi0[j][0] = (1 - t)*pi0[j][0] + t * pi0[j + 1][0];
            pi0[j][1] = (1 - t)*pi0[j][1] + t * pi0[j + 1][1];
            pi0[j][2] = (1 - t)*pi0[j][2] + t * pi0[j + 1][2];
        }
    }
    result[0] = pi0[0][0];
    result[1] = pi0[0][1];
    result[2] = pi0[0][2];
}

//function to calculate the bezier curve
void uniformBezierCurve() {
    float result[3]; //result point array

    // for each step call deCasteljau algorithm
    for (int i = 0; i <= Steps; i++) {
        deCasteljau(PointArray, (GLfloat)i / Steps, result, NumPts);

        //put the results in the array
        CurveArray[i][0] = result[0];
        CurveArray[i][1] = result[1];
        CurveArray[i][2] = result[2];
    }
}
```

In questo caso ho però scelto di creare la curva nell'init (richiamando la funzione *uniformbezierCurve()*) con valori preimpostati per i control points. In particolare, ho creato 9 control points dove il primo è anche l'ultimo e in modo che formano una specie di cerchio che va

ad onda sull'asse Y. Il risultato della funzione viene inserito all'interno di una variabile *CurveArray,* che dopo l'init conterrà tutti i punti con cui la curva è stata approssimata.

Fatto ciò mi è bastato aggiungere alla funzione *keyboardDown()* due entry: 'k' per muoversi in una direzione della curva e 'l' per muoversi nell'altra:

```
case 'k': //move camera left along bezier curve
    curveStep = (curveStep - 1 + Steps) % Steps;
    ViewSetup.target = objects[movables[selected_obj]].M * glm::vec4(0.f, 0.f, 0.f, 1.f);
    ViewSetup.position.x =CurveArray[curveStep][0];
    ViewSetup.position.y = CurveArray[curveStep][1];
    ViewSetup.position.z = CurveArray[curveStep][2];
    break;
case 'l': //move camera right along bezier curve
    curveStep = (curveStep + 1) % Steps;
    ViewSetup.target = objects[movables[selected_obj]].M * glm::vec4(0.f, 0.f, 0.f, 1.f);
    ViewSetup.target = objects[movables[selected_obj]].M * glm::vec4(0.f, 0.f, 0.f, 1.f);
    ViewSetup.position.x = CurveArray[curveStep][0];
    ViewSetup.position.y = CurveArray[curveStep][1];
    ViewSetup.position.z = CurveArray[curveStep][2];
    break;
default:
    break;
}
glutPostRedisplay();
```

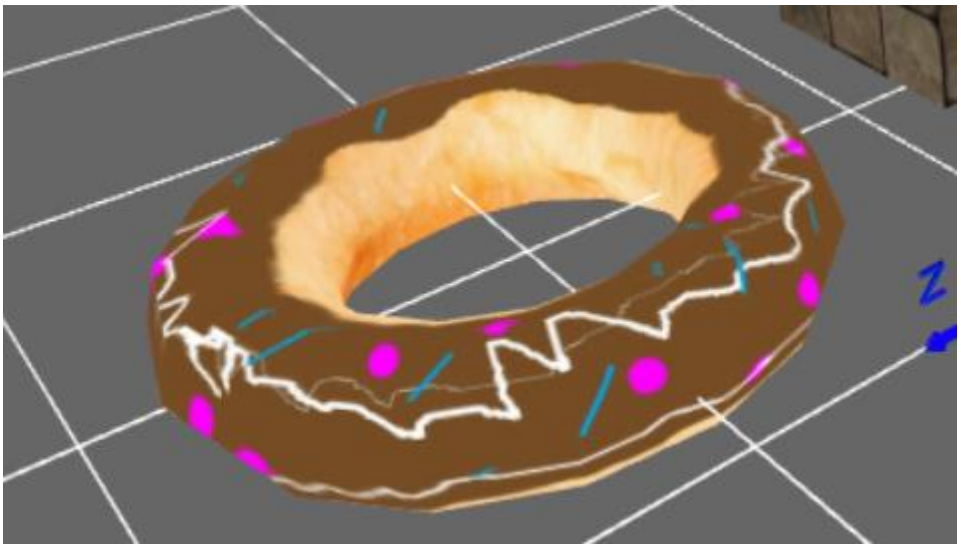
In particolare, tenendo premuto il tasto, si incrementa o decrementa una variabile *curveStep* in modo che sia sempre all'interno del range (0, dimCurva-1), in modo tale che, se decremento quando la variabile è 0 vado a finire a dimCurva-1, e se incremento quando la variabile è dimCurva-1 vado a finire a 0. Poi uso questa variabile per cambiare la posizione della camera con la posizione del punto con indice *curveStep* nell'array della curva. Aggiorno anche il target vector con la posizione dell'oggetto che è al momento selezionato, cosicché la camera guardi sempre l'oggetto anche se viene spostata.

STEP 2: Texture mapping 2D del toro

Il texture mapping 2D del toro (associare le coordinate parametriche del toro a quelle della texture) è già implementato nella funzione *computeTorusVertex()*, nella quale dopo aver costruito il vertice del toro tramite la funzione parametrica (theta e phi), queste vengono anche utilizzate per definire le texture coordinates. In questa funzione ho solo invertito le posizioni di theta e phi e ho diviso per 2 pi greco invece che per pi greco.

Mi è poi bastato caricare la giusta texture e cambiare lo shading in *init_torus()* a TEXTURE_ONLY, in quanto gli shaders *f_texture.glsl* e *v_texture.glsl* sono già realizzati di base.

Risultato:

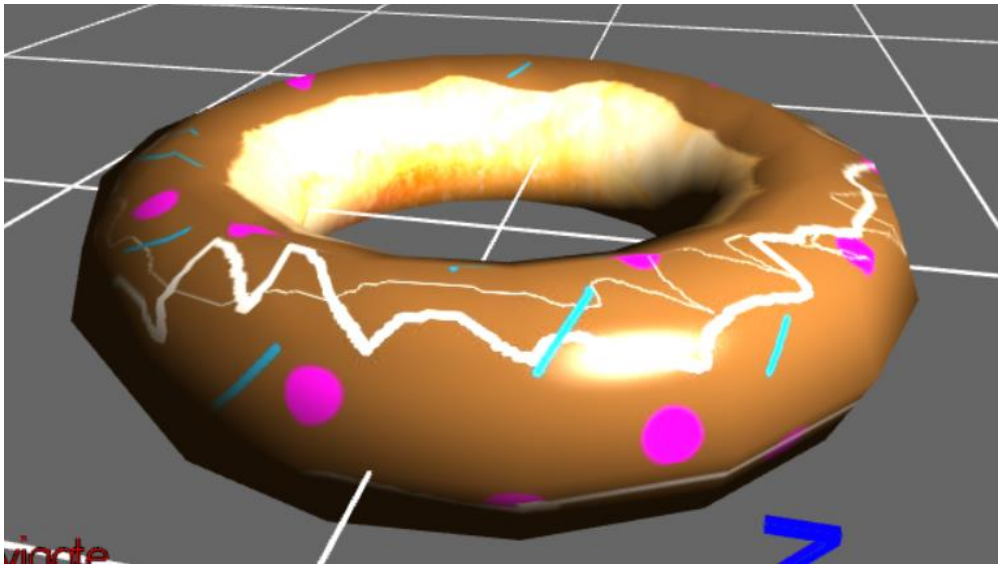


STEP 3: Texture mapping 2D + Shading

Per realizzare gli shaders *v_texture_phong.glsl* e *f_texture_phong .glsl* sono partito dagli shaders *v_phong.glsl* e *f_phong.glsl* del LAB_03 e gli ho aggiunto le componenti dello STEP 2 per il texture mapping. In particolare, nel fragment shader ho utilizzato il colore ottenuto come risultato del texture mapping come componente diffusiva del modello di illuminazione (ho preso le prime tre componenti del vec4 che si ottiene come risultato della funzione *texture()*), al posto della componente diffusiva del materiale utilizzato.

Nel programma principale mi è bastato sostituire lo shader del toro con TEXTURE_PHONG e decommentare il codice relativo alla creazione e loading dello shader (dato che era già implementato di base).

Risultato con materiale BRASS:



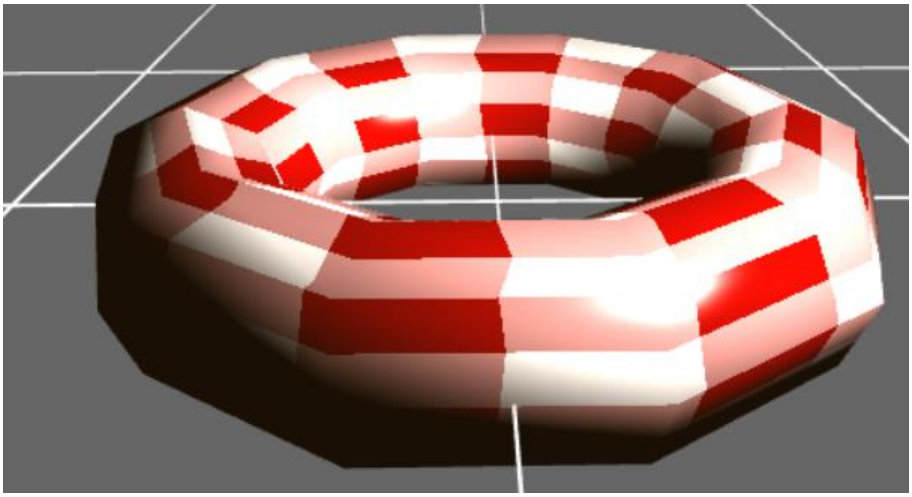
STEP4: Procedural mapping basato su un procedimento algoritmico a piacere sul toro

Ho scelto di creare come texture procedurale un pattern a scacchi colorati simile a una tovaglia. Per fare ciò ho creato la funzione *generateCheckTexture()* che tramite una funzione crea una texture bidimensionale. La funzione ritorna l'id della texture generata.

```
GLuint generateCheckTexture() {
    GLuint textureID;
    GLubyte image[64][64][3];
    int i, j, c;
    float ki = 1;
    float kj = 1;
    for (i = 0; i < 64; i++) {
        for (j = 0; j < 64; j++) {
            c = ((i & 0x4) == 0) * 125 + (((j & 0x4)) == 0) * 125;
            image[i][j][0] = (GLubyte)c*0.5 + 255 * 0.5;
            image[i][j][1] = (GLubyte)c;
            image[i][j][2] = (GLubyte)c;
        }
    }
    ////////////////////////////////////////
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D,  //the target
        0, // the mip map level we want to generate
        GL_RGB, // the format of the texture
        64, //texture_size, width
        64, //texture_size, heigth
        0,  // border, leave 0
        GL_RGB, // we assume is a RGB color image with 24 bit depth per pixel
        GL_UNSIGNED_BYTE, // the data type
        image);
    return textureID;
}
```

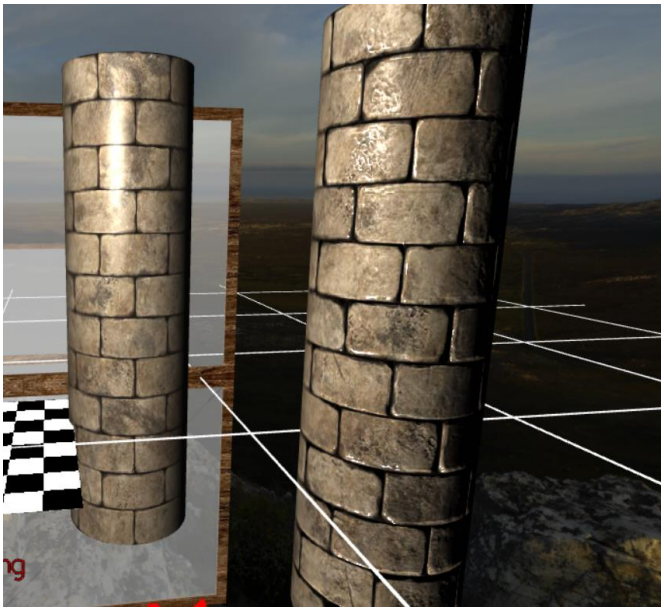
L'id della texture generata viene poi utilizzato come valore per il *diffuseTexID* del toro in *init_toro()*:

Risultato:



STEP 5: Normal mapping di un oggetto mesh (column.obj o sharprockfree.obj) con normal map e immagine texture lette da file

Lo step è già implementato di base. Ho solo sostituito lo shading TEXTURE_ONLY del muro senza normal mapping con TEXTURE_PHONG per andare a vedere la differenza tra semplice texture mapping e texture mapping con normal map:

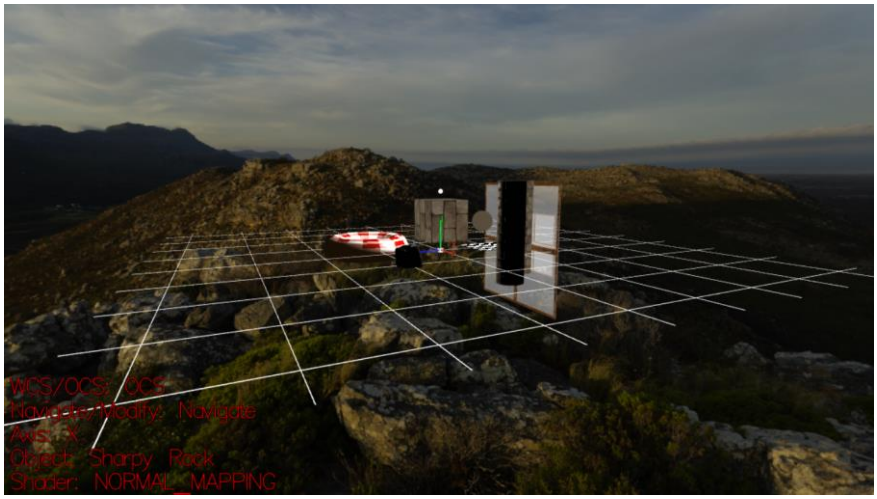


Dall'immagine si vede come il muro senza normal mapping sembra molto più piatto rispetto a quello con normal mapping, che dà illusione di profondità senza però modificare la mesh.

STEP 6: Environment cube mapping: skybox

Per rendere la skybox visibile ho scalato la skybox con un fattore di scala di 100 per ogni asse.

Risultato:



STEP 7: Environment mapping: object REFLECTION

Per implementare la reflection per prima cosa ho realizzato gli shaders *v_reflection.glsl* e *f_reflection.glsl*. Nel vertex shader creo i vettori normale e posizione nel WCS del vertice e li passo al fragment shader:

```
// Vertex shader: Reflection shading
// ================
#version 450 core

// Input vertex data, different for all executions of this shader.
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out vec3 Normal;
out vec3 Position;

// Values that stay constant for the whole mesh.
uniform mat4 P;
uniform mat4 V;
uniform mat4 M;
uniform vec3 camera_position;

void main()
{
    Normal = mat3(transpose(inverse(M))) * aNormal;
    Position = vec3(M * vec4(aPos, 1.0)); //in WCS
    gl_Position = P * V * M * vec4(aPos,1.0);
}
```

Nel fragment shader creo il vettore direzione E e calcolo il raggio riflesso data la normale N, con la funzione *reflect()*. Il colore del fragment sarà il colore ottenuto dal punto dell'intersezione tra il raggio riflesso e la cube map, che ci viene data da R. Quindi tramite la funzione *texture* uso come texture coordinate R per definire il colore del fragment:

```
out vec4 FragColor;
in vec3 Normal;
in vec3 Position;
uniform vec3 camera_position;
uniform samplerCube cubemap;

void main()
{
    vec3 E = normalize(Position - camera_position);
    vec3 R = reflect(E, normalize(Normal));
    FragColor = texture(cubemap, R);
}
```

Infine ho caricato la texture nell' *initShader()* e nel *drawScene()*.

Risultato:

STEP 8: Environment mapping: object REFRACTION

Per implementare la refraction per prima cosa ho realizzato gli shaders *v_refraction.glsl* e *f_refraction.glsl*. Nel vertex shader creo i vettori normale e posizione nel WCS del vertice e li passo al fragment shader, come per la reflection.

Nel fragment shader calcolo il vettore direzione E, e poi calcolo il raggio rifratto con la funzione *reflect()* dando in input come ratio quello del vetro. Il punto in cui il raggio interseca la cube map è inserito in R, che viene poi utilizzato per decidere il colore del fragment:

```
out vec4 FragColor;
in vec3 Normal;
in vec3 Position;
uniform vec3 camera_position;
uniform samplerCube cubemap;

void main()
{
    vec3 E = normalize(Position - camera_position);
    float ratio = 1.00 / 1.52;
    vec3 R = refract(E, normalize(Normal), ratio);
    FragColor = texture(cubemap, R);
}
```

Risultato:

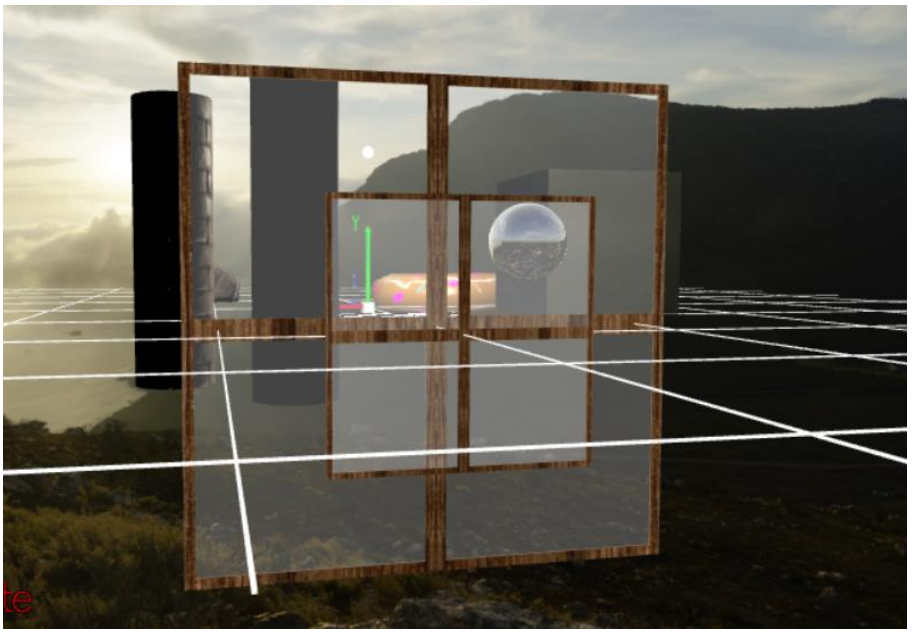
STEP 9: Oggetti semi-trasparenti: gestire in scena un paio di oggetti semi-trasparenti (es. window.obj) facendone la resa in drawScene() dal più lontano al più vicino dopo aver reso tutti gli oggetti opachi.

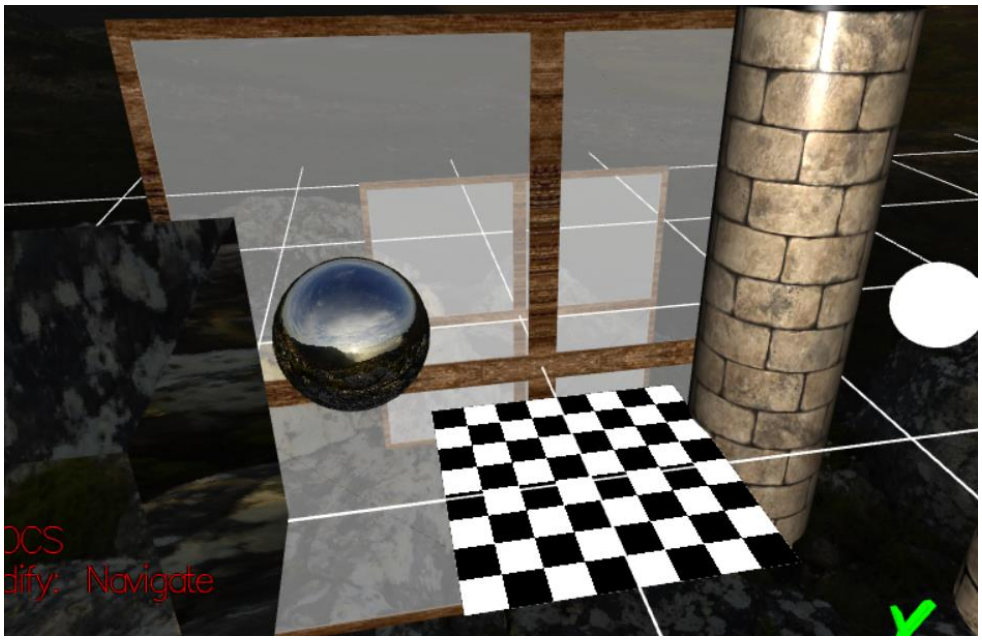
Per renderizzare gli oggetti trasparenti per prima cosa ho messo a true il parametro “blended” delle due finestre. Poi ho creato 3 funzioni: *calculateDistanceFromCamera()* che prende in ingresso la posizione della camera e la matrice di modello dell’oggetto e ritorna la distanza dell’oggetto dalla camera; *compareObjectsDistance()* che prende in ingresso due oggetti e ritorna true se l’oggetto 1 è più distante dell’oggetto 2; *sortObjectsByDistance()* che prende in ingresso un vettore di oggetti e lo sorta tramite la funzione std::sort che usa come comparing function la funzione compareObjectDistance creata appositamente:

```cpp
//Calculate the distance of an object from the camera
float calculateDistanceFromCamera(const glm::vec4& cameraPosition, const glm::mat4& objectModelMatrix) {
    // Extract the translation part of the object's model matrix
    glm::vec3 objectPosition(objectModelMatrix[3]);

    // Calculate the distance between the camera and the object
    float distance = glm::length(glm::vec3(cameraPosition) - objectPosition);

    return distance;
}

//Compare the distance of two objects from the camera
bool compareObjectsDistance(const Object& obj1, const Object& obj2) {
    float distanceObj1 = calculateDistanceFromCamera(ViewSetup.position, obj1.M);
    float distanceObj2 = calculateDistanceFromCamera(ViewSetup.position, obj2.M);

    // Check if the first object is further than the second one
    return distanceObj1 > distanceObj2;
}

// Function to sort the array of objects based on distance
void sortObjectsByDistance(std::vector<Object>& objects) {
    std::sort(objects.begin(), objects.end(), [&](const Object& obj1, const Object& obj2) {
        return compareObjectsDistance(obj1, obj2);
    });
}
```

Nella draw scene ho poi reso prima tutti gli oggetti opachi e messo in un vettore a parte gli oggetti semi-trasparenti, ho chiamato la funzione di sorting con il vettore degli oggetti semi-trasparenti, e ho poi reso il vettore di questi oggetti.

Risultato (ho messo due immagini per dimostrare che gli oggetti vengono resi nel giusto ordine):




OCS
dify: Navigate