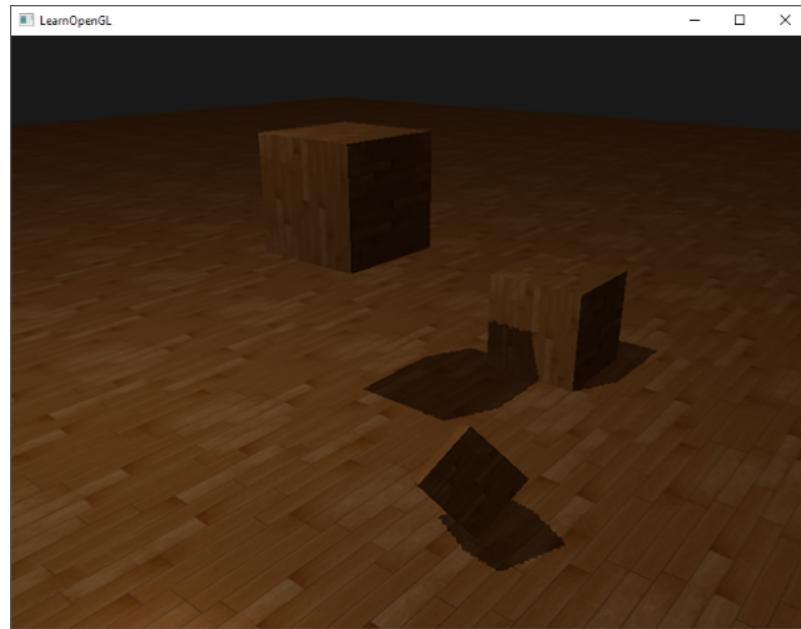


Shadow mapping

Generazione delle ombre in OpenGL



Giovanni Danieli
Laurea Magistrale in Ingegneria e Scienze Informatiche
VR463656

Università degli Studi di Verona
Anno Accademico : 2020/2021

Indice

	Pagina
1 Introduzione	2
2 Shadow mapping	3
3 Implementazione in OpenGL	5
3.1 Definizione della depth map	5
3.2 Impostare la trasformazione nello spazio luce	5
3.3 Eseguire il rendering sulla depth map	6
3.4 Renderizzare la scena con le ombre	6
4 Problemi	8
4.1 Shadow acne	8
4.2 Peter panning	9
4.3 Aliasing	10

Capitolo 1

Introduzione

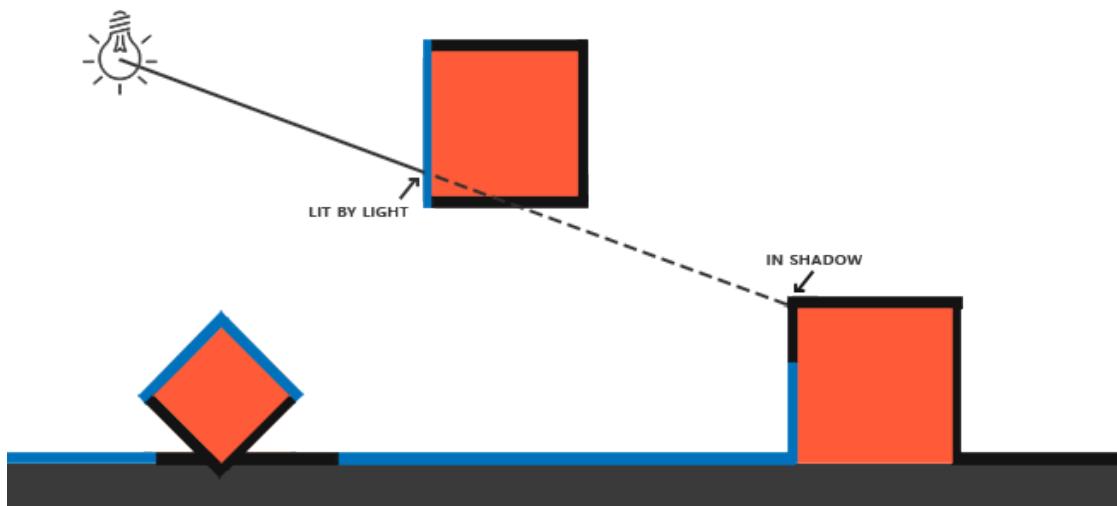
Le ombre, ovvero l'assenza di luce, non sono perfettamente implementabili a causa dell'assenza di algoritmi efficienti nel rendering in tempo reale. Tuttavia esistono molteplici tecniche che offrono buoni risultati per l'approssimazione delle ombre, ognuna però con i propri difetti e problemi che devono essere presi in considerazione.

Una tecnica molto utilizzata nei videogiochi che offre buoni risultati ed è semplice da implementare e non impatta più di tanto la performance è lo **shadow mapping**; inoltre può evolvere in algoritmi più avanzati come lo **Omnidirectional Shadow Maps** e lo **Cascaded Shadow Maps**.

Capitolo 2

Shadow mapping

L'idea dietro lo Shadow mapping è molto semplice: renderizzare la scena dal punto di vista della luce; tutto quello visibile dal punto di vista della luce è illuminato mentre quello non visibile è in ombra.

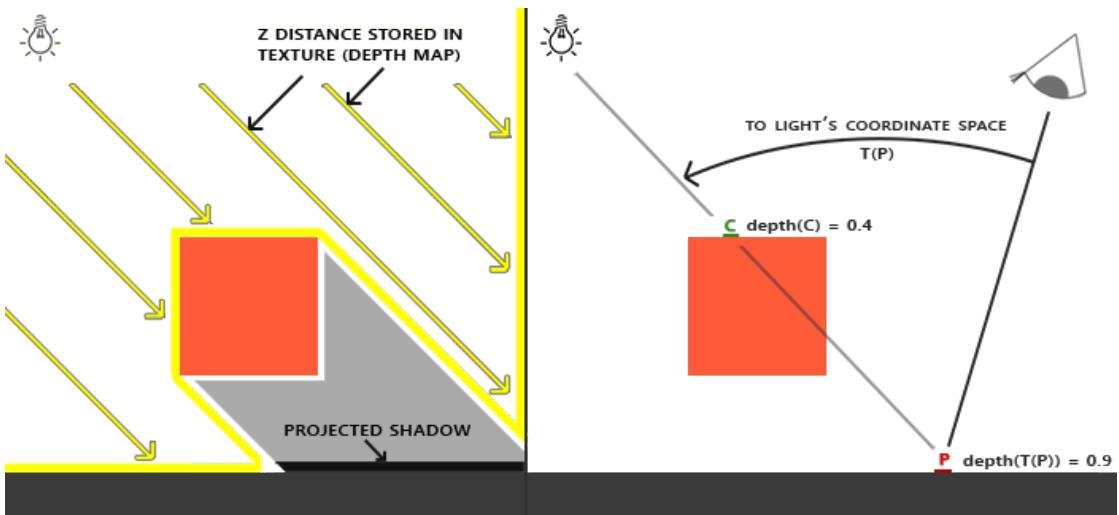


Nell'immagine le linee blu rappresentano i frammenti visibili alla fonte di luce, mentre le linee nere sono nascoste e rappresentano l'ombra. Se tracciamo una retta dalla fonte di luce al cubo più a destra il raggio di luce colpisce prima il cubo fluttuante. Come risultato il frammento del cubo fluttuante è illuminato mentre quello del contenitore più a destra non lo è, risultando in ombra.

Vogliamo ottenere il punto sul raggio dove è avvenuto il primo contatto con un oggetto e confrontarlo con altri punti su questo raggio. Quindi eseguiamo un test di base per vedere se la posizione del raggio di un punto di prova è più lontano dalla fonte rispetto al punto più vicino e, in tal caso, il punto di prova è in ombra. Iterare su possibili migliaia di raggi di luce è un approccio estremamente inefficiente e non si presta bene per il rendering Real Time.

Un approccio migliore e più efficiente è renderizzare la scena dal punto di vista della luce e salvare i valori della profondità all'interno di una texture; i valori memorizzati all'interno di questa texture rappresentano i valori di profondità più

vicini alla fonte di luce e la texture prende il nome di **shadow map** o **depth map**.



L’immagine a sinistra mostra una fonte di luce direzionale (i raggi sono paralleli tra di loro) proiettare l’ombra del cubo sulla superficie sotto. Usando i valori memorizzati nella depth map troviamo i punti più vicini alla fonte e li utilizziamo per determinare se i frammenti della superficie sotto sono in ombra. Creiamo la depth map renderizzando la scena utilizzando una view matrix e una projection matrix specifiche per quella fonte di luce. Queste matrici insieme formano la matrice T che trasformano qualsiasi posizione 3D nello spazio coordinate della fonte di luce.

Nell’immagine a destra vediamo la stessa luce direzionale e la camera. Renderizziamo un frammento nel punto P il quale dobbiamo determinare se si trova in ombra o meno. Per prima cosa dobbiamo trasformare il punto P nello spazio delle coordinate della luce usando la trasformazione T . Ora che il punto P è visto dalla prospettiva della luce, la sua coordinata z corrisponde alla sua profondità. Usando il punto P possiamo indicizzare la depth map per ottenere la profondità più vicina visibile dalla prospettiva della luce, ovvero il punto C . Dato che la depth map ritorna una profondità minore rispetto a quello del punto P , possiamo concludere che il punto P è occluso e quindi in ombra.

Lo shadow mapping quindi consiste in due passaggi: prima renderizziamo la depth map, successivamente renderizziamo la scena normalmente e usiamo la depth map generata per determinare quali frammenti sono in ombra.

Capitolo 3

Implementazione in OpenGL

3.1 Definizione della depth map

Come prima cosa generiamo la depth map. Siccome vogliamo solo i valori della profondità specifichiamo il formato della texture come GL_DEPTH_COMPONENT. Inoltre diamo alla texture una risoluzione di 1024 x 1024 pixel. Infine attacchiamo la texture al relativo framebuffer.

Dato che non abbiamo un color buffer ed abbiamo disabilitato i draw e read buffer, i fragments risultanti non richiedono alcun tipo di elaborazione, quindi usiamo un fragment shader vuoto.

3.2 Impostare la trasformazione nello spazio luce

Una volta definita la depth map dobbiamo impostare le view e projection matrix per renderizzare la scena dal punto di vista della luce.

Per la projection matrix, usando una luce direzionale dove tutti i raggi di luce sono paralleli, impostiamo una matrice di proiezione ortografica.

```
float near_plane = 1.0f, far_plane = 7.5f;
glm::mat4 lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f,
                                         near_plane, far_plane);
```

Per creare la view matrix utilizziamo la funzione `glm::lookAt` con la fonte di luce come posizione e il centro della scena come target.

```
glm::mat4 lightView = glm::lookAt(glm::vec3(-2.0f, 4.0f, -1.0f),
                                    glm::vec3(0.0f, 0.0f, 0.0f),
                                    glm::vec3(0.0f, 1.0f, 0.0f));
```

combinando le due matrici otteniamo la matrice di trasformazione.

```
glm::mat4 lightSpaceMatrix = lightProjection * lightView;
```

3.3 Eseguire il rendering sulla depth map

Definita la depth map, la matrice di trasformazione e i relativi shader possiamo finalmente renderizzare la shadow map. Il risultato è un buffer contenente tutti i valori delle profondità dei frammenti visibili alla luce.

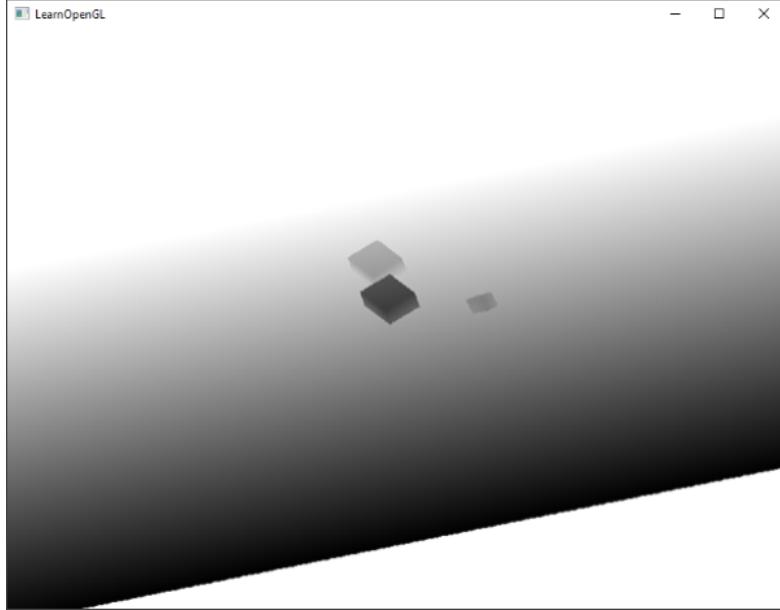


Figura 3.1: Visualizzazione della depth/shadow map memorizzata nel buffer

3.4 Renderizzare la scena con le ombre

Ottenuta la depth map possiamo passare a renderizzare la scena con le nostre ombre, apportando dei cambiamenti agli shader utilizzati finora per renderizzare la scena. Nel vertex shader aggiungiamo il calcolo del vettore FragPosLightSpace, ovvero il vettore della posizione dello spazio mondo trasformato nello spazio luce, il quale viene mandato in input al fragment shader.

```
vs_out.FragPosLightSpace = lightSpaceMatrix * vec4(vs_out.FragPos,  
1.0);
```

Nel fragment shader, dove utilizziamo il modello di lighting di Blinn-Phong, aggiungiamo il calcolo delle ombre e modifichiamo il calcolo del vettore *lightning*.

```
float shadow = ShadowCalculation(fs_in.FragPosLightSpace);
```

Il valore restituito dalla funzione *ShadowCalculation* (ne parleremo dopo) è un float compreso tra 0.0 (ovvero il frammento non è in ombra) e 1.0 (il frammento

è totalmente in ombra). Questo valore viene utilizzato nel calcolo del lighting dove viene moltiplicato con i componenti della luce diffuse e specular; l'ambient viene escluso in quanto le ombre non sono mai completamente nere a causa dello scattering della luce.

```
vec3 lighting = (ambient + (1.0 - shadow) * (diffuse + specular))
                * color;
```

La funzione ShadowCalculation prende la profondità del frammento corrente nello spazio luce e la confronta con closestDepth, ovvero la profondità del frammento più vicino sulla retta tra la fonte di luce e il frammento corrente campionata dalla shadow map; se la profondità del frammento corrente è maggiore di currentDepth, restituisce 1, altrimenti 0.

La funzione è così definita:

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    projCoords = projCoords * 0.5 + 0.5;
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    float currentDepth = projCoords.z;
    float shadow = currentDepth > closestDepth ? 1.0 : 0.0;

    return shadow;
}
```

Il risultato del secondo rendering (ovvero della scena) è il seguente:

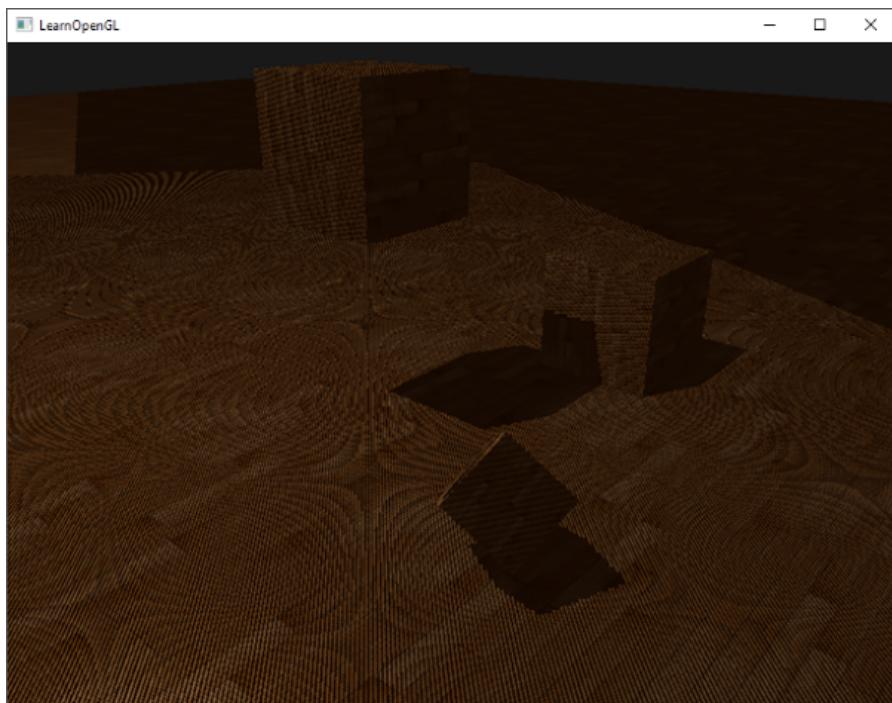


Figura 3.2: Risultato del secondo rendering

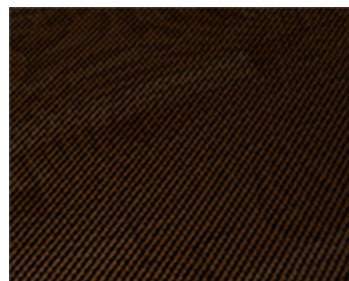
Capitolo 4

Problemi

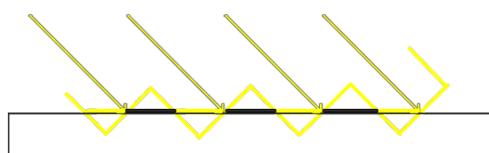
Nel risultato finale possiamo notare che le ombre sono state generate, ma sono presenti diversi problemi.

4.1 Shadow acne

Il primo problema (e quello più evidente) è un pattern di linee nere sul pavimento.



Questo artefatto è chiamato shadow acne e può essere spiegato con la seguente immagine:



Quando i raggi di una fonte di luce colpiscono una superficie con un'angolazione non perpendicolare, molti frammenti (quando sono relativamente distanti dalla fonte di luce) accedono allo stesso texel di profondità che potrebbe risultare sopra per alcuni o sotto alla superficie per altri; questo porta ad una discrepanza nell'ombra.

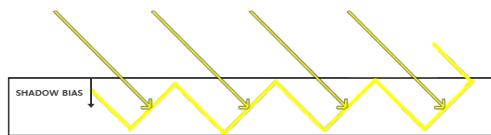


Figura 4.1: applicando il bias

La soluzione consiste nell'aggiungere un bias alla profondità corrente (o alla shadow map) affinchè i frammenti non siano considerati sopra la superficie. Questo metodo è chiamato **shadow bias**

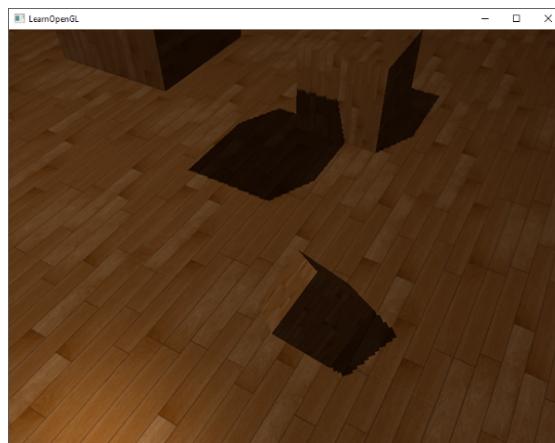


Figura 4.2: Applicando il bias al nostro esempio

4.2 Peter panning

Uno svantaggio dell'applicare lo shadow bias è l'applicazione di un offset alla profondità effettiva delle ombre, facendole apparire distaccate dagli oggetti (da qui il nome **peter panning**)

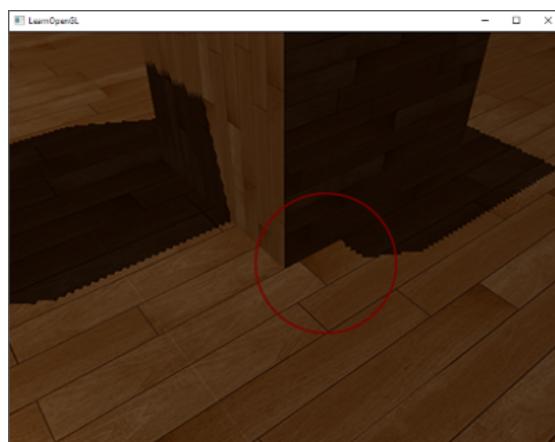
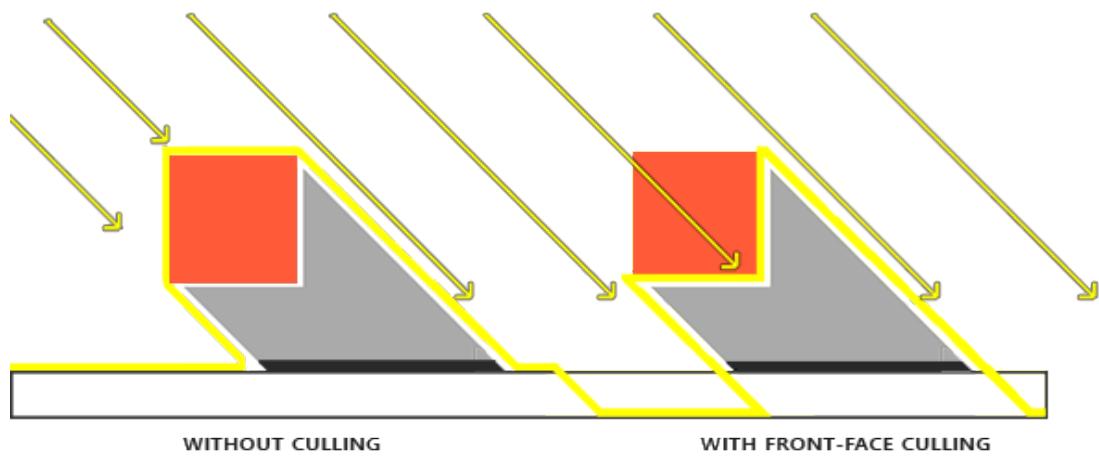


Figura 4.3: Esempio di peter panning applicando un bias esagerato

Per risolvere questo problema si genera la shadow map utilizzando il retro delle facce degli oggetti.



In OpenGL chiamiamo la funzione `GL_CULL_FACE()` con argomento `GL_FRONT` per abbattere le facce frontali per il calcolo della depth map, ripristinandole successivamente per il rendering della scena normale.

Questo metodo può essere applicato solo per oggetti solidi che non hanno aperture verso l'interno. Inoltre non funziona sui single plane (il nostro pavimento per esempio), poiché sarebbero abbattuti.

Oggetti troppo vicini alla fonte di luce potrebbero dare ancora risultati non corretti.

Tuttavia il fenomeno del peter panning è evitabile utilizzando bias piccoli.

4.3 Aliasing

Se zoomiamo sull'ombra possiamo vedere come questa è affetta da aliasing, ovvero sul bordo dell'ombra c'è una transizione netta tra ombra e luce, risultando sgradevole alla vista.



Ci sono varie soluzioni parziali all'aliasing, tra queste aumentare la risoluzione della texture dell'ombra e/o avvicinare il più possibile il tronco di luce alla scena.

Un'altra soluzione parziale al problema dell'aliasing si chiama **PCF**, ovvero percentage-closer filtering, un set di funzioni di filtraggio che producono ombre più morbide. L'idea è quella di campionare più volte la depth map ogni volta con piccole variazioni nelle coordinate della texture. Per ogni campione controlliamo se questo è in ombra o meno. Una volta terminato facciamo la media di tutti i campioni e otteniamo l'ombra.



(a) Prima



(b) Dopo

Una semplice implementazione in OpenGL è la seguente:

```
float shadow = 0.0;
vec2 texelSize = 1.0 / textureSize(shadowMap, 0);
for(int x = -1; x <= 1; ++x)
{
    for(int y = -1; y <= 1; ++y)
    {
        float pcfDepth = texture(shadowMap, projCoords.xy
                                + vec2(x, y) * texelSize).r;
        shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
    }
}
shadow /= 9.0;
```