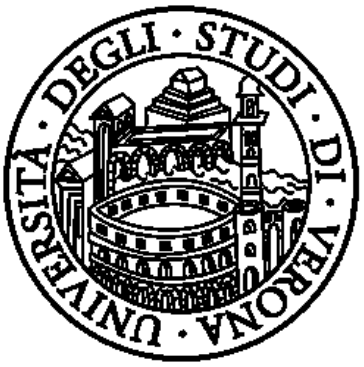


UNIVERSITÀ di VERONA

Università degli studi di Verona – Dipartimento di Ingegneria e scienze informatiche

SHADOW MAPPING

- Programmazione Grafica -

Prof. Referente
Umberto Castellani

A cura di
Davide Zanellato
13/09/24

INDICE

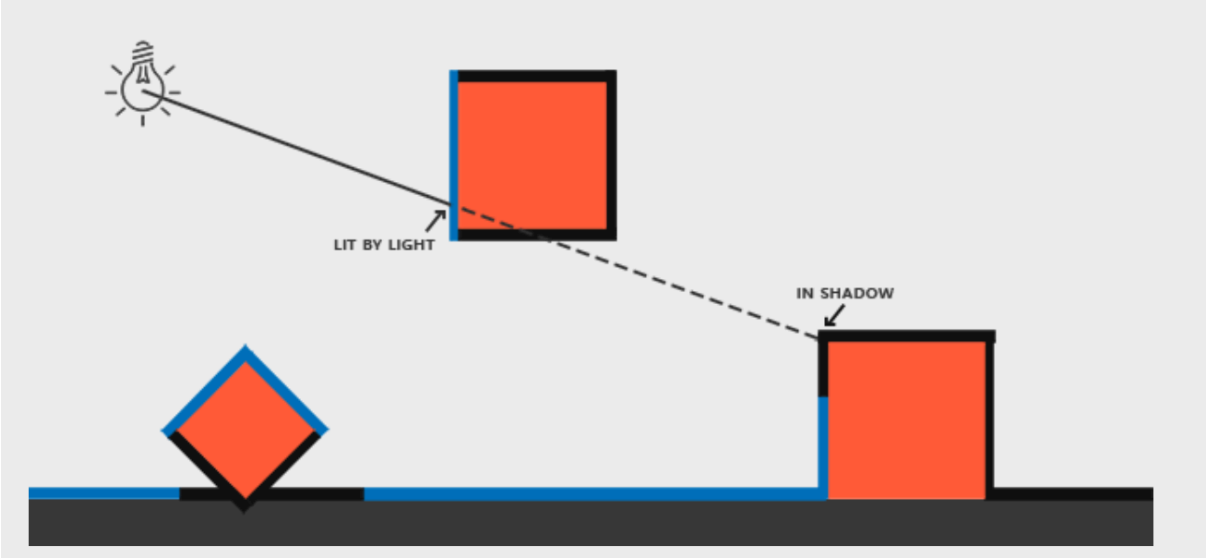
1. Introduzione

Quando i raggi di luce di una sorgente luminosa non colpiscono un oggetto perché viene occluso da un altro elemento, l'oggetto è in ombra. Le ombre forniscono maggiore senso di realismo a una scena illuminata e rendono più facile per un osservatore interpretare le relazioni spaziali tra gli oggetti, dando un maggiore senso di profondità alla scena.

Le ombre sono difficili da implementare, in quanto non è ancora stato sviluppato un algoritmo perfetto. Esistono tuttavia diverse buone tecniche di approssimazione delle ombre e una di queste è lo shadow mapping che viene utilizzata dalla maggior parte dei videogiochi e permette di ottenere buoni risultati.

2. Shadow Mapping

Questa tecnica si basa su un semplice principio: tutto ciò che vediamo dalla prospettiva della luce è illuminato, mentre tutto ciò che non possiamo vedere deve essere in ombra.

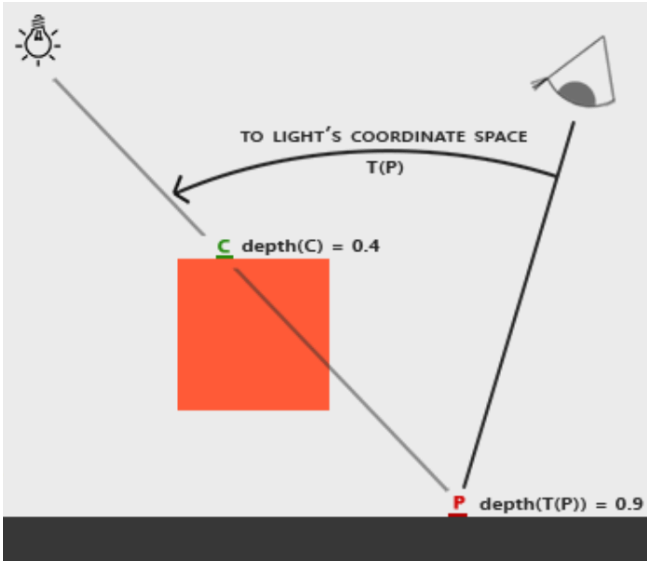


Considerando questa immagine vogliamo ottenere il punto sul raggio di luce che ha colpito per la prima volta un oggetto e confrontare questo punto più vicino con altri punti sul raggio. Si esegue quindi un test per verificare se la posizione di un punto di prova scelto arbitrariamente sul raggio di luce si trova più in basso rispetto al punto più vicino e, se è così, il punto di prova risulterà essere in ombra.

Per svolgere questo test viene utilizzato il buffer di profondità, i cui valori corrispondono alla profondità di un fragment compresi nell'intervallo [0, 1] dal punto di vista della telecamera.

La scena viene quindi renderizzata dalla prospettiva della luce e i valori di profondità risultanti vengono memorizzati in una texture chiamata depth map. Questi valori di profondità mostrano il primo fragment visibile dalla prospettiva della luce.

La depth map viene creata eseguendo il rendering della scena dalla prospettiva della luce utilizzando una matrice di proiezione e visualizzazione specifica per quella sorgente di luce. Queste due matrici di proiezione e visualizzazione, se utilizzate insieme, formano una trasformazione "T" che trasforma qualsiasi posizione 3D nello spazio delle coordinate della luce.



In questa immagine troviamo in alto a sinistra la sorgente luminosa e in alto a destra l'osservatore. L'obiettivo è determinare se il punto "P" si trova in una posizione in ombra. Per fare questo, trasformiamo prima il punto "P" nello spazio di coordinate della luce utilizzando la trasformazione "T" e in questo modo la sua coordinata "z" corrisponde alla sua profondità che in questo caso è 0.9. Utilizzando il punto "P" è possibile inoltre indicizzare la depth map per ottenere il valore di profondità più vicino visibile dalla prospettiva della luce, che è nel punto "C" con una profondità campionata di 0.4. Poiché l'indicizzazione della depth map restituisce una profondità inferiore alla profondità nel punto "P" possiamo concludere che è nascosto e quindi in ombra.

3. Depth Map

La depth map è la texture di profondità renderizzata dalla prospettiva della luce che viene utilizzata per eseguire i test relativi alle ombre. Poiché il risultato renderizzato di una scena viene memorizzato in una texture, è necessario utilizzare un framebuffer che archivia e permette di definire manualmente un color e un depth buffer.

```
// configure depth map FBO
// -----------------------
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024;
unsigned int depthMapFBO;
glGenFramebuffers(1, &depthMapFBO);
// create depth texture
unsigned int depthMap;
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
float borderColor[] = { 1.0, 1.0, 1.0, 1.0 };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
// attach depth texture as FBO's depth buffer
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Il codice presente rappresenta il procedimento seguito per la creazione della depth map:

- creazione di un oggetto framebuffer per il rendering della depth map;
- creazione di una texture 2D che viene utilizzata come buffer di profondità, poiché sono importanti solo i valori di profondità, i formati della texture sono specificati come GL_DEPTH_COMPONENT e viene impostata la sua larghezza e altezza a 1024;
- la texture di profondità generata viene allegata come buffer di profondità del framebuffer;
- non vi è la necessità di utilizzare un color buffer, tuttavia, un oggetto framebuffer non è completo senza la presenza color buffer, quindi è necessario dire esplicitamente a OpenGL che non verrà impostato alcun dato per quanto riguarda il colore, tutto questo viene eseguito impostando il buffer di lettura e di disegno su GL_NONE.

Per quanto riguarda la fase di rendering nel render loop, è importante impostare il “glViewport()” con le dimensioni della depth map, ovvero 1024, in quanto ha una risoluzione diversa dalla scena originale.

4. Trasformazione light space

Per effettuare il rendering della scena dalla prospettiva della luce è necessario implementare delle matrici “lightProjection” e “lightView” apposite.

```
// 1. render depth of scene to texture (from light's perspective)
// --------------------------------------------------------------
glm::mat4 lightProjection, lightView;
glm::mat4 lightSpaceMatrix;
float near_plane = 1.0f, far_plane = 7.5f;
lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, near_plane, far_plane);
lightView = glm::lookAt(lightPos, glm::vec3(0.0f), glm::vec3(0.0, 1.0, 0.0));
lightSpaceMatrix = lightProjection * lightView;
// render scene from light's point of view
simpleDepthShader.use();
simpleDepthShader.setMat4("lightSpaceMatrix", lightSpaceMatrix);
```

Poiché viene modellata una sorgente di luce direzionale, tutti i suoi raggi di luce sono paralleli ed è possibile utilizzare una matrice di proiezione ortografica per la sorgente di luce in quando non vi è alcuna deformazione prospettica. Per la creazione di una matrice di visualizzazione, utilizzata per trasformare ogni oggetto in modo che sia visibile dal punto di vista della luce, si utilizza la funzione “lookAt()” dove la posizione della sorgente luminosa è rivolta verso il centro della scena. Infine, combinando queste due matrici, si ottiene una matrice di trasformazione dello spazio luce chiamata “lightSpaceMatrix” che trasforma ogni vettore dello spazio mondo nello spazio visibile dalla sorgente luminosa e successivamente viene inviata al “simpleDepthShader”.

Questo vertex shader riceve inoltre una matrice “model” e trasforma tutti i vertici nello spazio luce utilizzando la “lightSpaceMatrix”. Poiché non è presente alcun color buffer e i buffer di disegno e lettura sono stati disabilitati, i fragment risultanti non richiedono alcuna elaborazione e quindi viene utilizzato un fragment shader vuoto.

5. Rendering delle ombre

Nel vertex shader della scena, oltre a tutti i soliti calcoli, viene determinato un vettore di output extra chiamato “vs_out.FragPosLightSpace” che utilizza la matrice definita precedentemente “lightSpaceMatrix” per trasformare la posizione dei vertici dallo spazio mondo allo spazio luce per l'uso successivo nel fragment shader.

Nel fragment shader viene inviata la depth map e si implementa il modello di illuminazione di Blinn-Phong per la luce direzionale e puntiforme. Tramite la funzione “ShadowCalculation()” si calcola un valore ombra che è 1.0 quando il fragment è in ombra e 0.0 quando non è in ombra.

Le componenti diffuse e speculari risultanti vengono quindi moltiplicate per questo componente ombra e poiché è molto raro che le ombre siano completamente scure, a causa della dispersione della luce, il componente ambientale viene escluso dalla moltiplicazione delle ombre.

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // perform perspective divide
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.5;
    // get closest depth value from light's perspective (using [0,1] range fragPosLight as coords)
    float closestDepth = texture(shadowMap, projCoords.xy).r;
    // get depth of current fragment from light's perspective
    float currentDepth = projCoords.z;
    // calculate bias (based on depth map resolution and slope)
    vec3 normal = normalize(fs_in.Normal);
    vec3 lightDir = normalize(lightPos - fs_in.FragPos);
    float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
    // check whether current frag pos is in shadow
    // PCF
    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;
    // keep the shadow at 0.0 when outside the far_plane region of the light's frustum.
    if(projCoords.z > 1.0)
        shadow = 0.0;

    return shadow;
}
```

Nella funzione “ShadowCalculation()” è necessario verificare se un fragment è in ombra trasformando la sua posizione in coordinate normalizzate. Quando viene inviata al fragment shader la posizione di un vertice nel clip space tramite “gl_Position” presente nel vertex shader, OpenGL esegue automaticamente una divisione prospettica, dividendo le coordinate x, y e z per il componente del vettore w e, poiché “FragPosLightSpace” non viene inviato al fragment shader tramite gl_Position, è necessario eseguire esplicitamente questa operazione per ottenere la posizione del fragment nello spazio luce nell'intervallo [-1, 1]. Successivamente le coordinate normalizzate del fragment vengono trasformate nell’intervallo [0, 1], in quanto verranno utilizzate per campionare la depth map, i cui valori di profondità sono compresi nell’intervallo [0, 1]. Successivamente si calcola il valore di profondità più vicino dal punto di vista della luce e si ottiene la profondità corrente di un determinato fragment recuperando la coordinata z del vettore proiettato.

6. Shadow acne

Il problema di shadow acne si verifica a causa della risoluzione limitata della depth map e in questo caso vengono visualizzate sul pavimento evidenti linee nere alternate. Questo accade in quanto più fragment possono campionare lo stesso valore della depth map quando sono relativamente lontani dalla sorgente luminosa.

Sebbene questo sia generalmente accettabile, diventa un problema quando la sorgente luminosa è rivolta ad angolo verso la superficie, poiché in quel caso anche la depth map viene renderizzata da un angolo. Diversi fragment accedono quindi allo stesso texel di profondità inclinato, mentre alcuni sono sopra e altri sotto il pavimento; il risultato è una discrepanza di ombre in quanto alcuni fragment sono considerati in ombra e altri no. Per risolvere questo problema si utilizza uno shadow bias la cui funzione è compensare la profondità della superficie con una piccola quantità, in modo che i fragment non vengano erroneamente considerati sopra la superficie. Una volta applicato il bias, tutti i campioni ottengono una profondità inferiore alla profondità della superficie e quindi l'intera superficie è correttamente illuminata senza ombre.

Il bias viene implementato modificando il suo valore in base all'angolo tra la superficie e la sorgente luminosa, si calcola con il prodotto scalare tra la normale della superficie e la direzione della luce e può assumere un valore massimo di 0.05 e un valore minimo di 0.005. In questo modo il pavimento che è quasi perpendicolare alla sorgente luminosa ottiene un piccolo bias, mentre le facce laterali del cubo ottengono un bias molto più grande.

7. Sovracampionamento

Un ulteriore problema è rappresentato dalle regioni della scena presenti al di fuori del frustum. Queste infatti appaiono in ombra in quanto le coordinate proiettate al di fuori del frustum della luce sono più alte di 1.0 e quindi campioneranno la texture di profondità al di fuori del suo intervallo predefinito di [0, 1]. In questo caso il problema è rappresentato da una porzione abbondante di pavimento che risulta essere in ombra. Per risolvere questo problema e far sì che queste coordinate non vengano visualizzate in ombra, è necessario che tutte le coordinate al di fuori dell'intervallo della depth map abbiano una profondità di 1.0. Si imposta quindi un colore del bordo della texture e le opzioni di wrap della texture di profondità come GL CLAMP TO BORDER.

Tuttavia questo potrebbe non bastare, infatti, le coordinate esterne al piano lontano del frustum ortografico della luce vengono ancora visualizzate in ombra in quanto una coordinata di fragment proiettata nello spazio luce è più distante del piano lontano della luce quando la sua coordinata z è maggiore di 1.0. In quel caso il metodo di wrapping GL_CLAMP_TO_BORDER non funziona più e, confrontando la coordinata z con i valori della depth map, il valore ombra calcolato viene sempre impostato ad 1 per z maggiore di 1.0. Per risolvere tale problema è necessario implementare, nella funzione “ShadowCalculation()” del fragment shader, un controllo che imposta il valore dell’ombra a 0.0 quando la coordinata z del vettore proiettato è maggiore di 1.0.

8. PCF

Poiché la depth map ha una risoluzione fissa è molto frequente che più fragment campionino lo stesso valore di profondità dalla depth map e producano gli stessi risultati riguardo le ombre, il che si traduce in bordi frastagliati e squadrati.

Una soluzione è chiamata PCF, percentage-closer filtering, che consiste nel campionare più di una volta la depth map con coordinate di texture leggermente diverse. Per ogni singolo campione si controlla se risulta essere in ombra o meno e infine tutti i sotto-risultati vengono combinati e mediati. Questo metodo viene implementato nella funzione “ShadowCalculation()” del fragment shader.

La funzione “textureSize()” restituisce un vettore formato da due dimensioni che rappresentano la larghezza e l'altezza della texture depth map, il quale viene poi diviso per 1, ottenendo la dimensione di un singolo texel.

“pcfDepth” legge il valore di profondità dalla depth map in una posizione offset spostata rispetto alle coordinate proiettate “projCoords.xy” del fragment corrente. L'offset è calcolato come “vec2(x, y) * texelSize” e in questo modo si legge la profondità dei texel vicini.

“shadow” verifica se la profondità corrente “currentDepth” meno un piccolo bias è maggiore della profondità letta dalla shadow map “pcfDepth”, se questo è vero, il punto è in ombra e “shadow” viene incrementato di 1, in caso contrario il punto non è in ombra e alla variabile “shadow” non viene sommato alcun valore.

Infine il valore accumulato di shadow viene diviso per 9.0 per ottenere una media poiché ci sono 9 campioni.