## COMPUTER GRAPHICS

Introduzione

Prima di iniziare definiamo cosi è OpenGL, esso infatti è principalmente considerato un API (Application Programming Interface), ma in realtà è una interfaccia che permette l’accesso e il controllo del sistema grafico. Ne esistono due versioni: compatibility e core. La versione core profile è la più recente ed è progettata per essere più efficiente. OpenGL core profile consente di programmare utilizzando dei mini programmi ,che vengono eseguiti direttamente dalla scheda grafica, chiamati shaders. Le moderne GPU, infatti, sono costituite da un elevato numero di piccoli processori programmabili chiamati shader cores che possono eseguire gli shader.

GLFW è una libreria, scritta in c, principalmente per OpenGL, che fornisce i requisiti necessari per poter renderizzare al meglio.

GLFW

Per poter vedere qualsiasi cosa a video bisognerebbe prima di tutto creare una finestra invocando il metodo glfwcreateWindonw, ed inseguito inizializzare GLEW in modo da essere sicuro che di usare le moderne tecniche di gestione di OpenGL.

Per poter renderizzare a video qualcosa fin quando non decidiamo di chiuderlo bisogna impostare un ciclo, altrimenti GLFW mostrerebbe una sola volta cioè che vogliamo per poi chiudere immediatamente la finestra, esso verrà chiamato game loop il quale controllerà ad ogni frame se dovrebbe. Al suo interno vi saranno dei metodi fondamentali:

1. glfwPoolEvents: controlla se c’è qualche evento innescato, come la pressione di un tasto ecc…
2. glfwSwapBuffers: il quale scambierà il buffer che è stato riempito durante la visualizzazione di quello precedente, perché OpenGL utilizza due buffer per renderizzare uno che viene mostrato ed un altro che viene creato nel frattempo.
3. glClear: in modo da pulire ogni volta il buffer prima di effettuare un rendering.

Ed una volta usciti dal game loop dobbiamo ricordarci dii terminare correttamente GLFW in modo da poter liberare tutte le risorse richieste.

Per gestire gli input GLFW una funzione chiamata callback function.

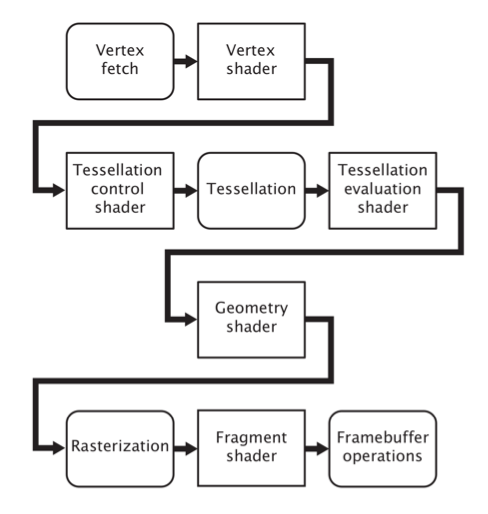
OpenGL lavora con oggetti 3d su schermi 2d, quindi una buona parte del lavoro di OpenGL è quello di trasformare le coordinate 3d in 2d attraverso una graphics pipeline, divisa in due grosse parti:

* trasformare le coordinate 3d in 2d
* colorazione effettiva del pixel

La pipeline è in realtà divisa in diversi step, ognuno lavora con l’output del precedente step, ed inoltre, ognuno di essi è specializzato in un determinato ambito fornendoci una esecuzione parallelizzata.

Gli step della pipeline vengono eseguiti da dei piccoli programmi chiamati Shader, i quali sono programmi elementari che vengono eseguiti direttamente nella GPU, scritti in OpenGL Shading Language (GLSL). Alcuni di essi posso essere configurati manualmente dallo sviluppatore che andranno a sostituire quelli di default tranne che per alcuni step.

Gli step della pipeline sono i seguenti:

* Vertex Shader: il suo obiettivo principale è quello di trasformare le coordinate 3d in altre coordinate 3d differenti (richiesta implementazione).
* Primitive Assembly: combina i precedenti vertici passati dal vertex shader unendo i punti.
* Geometry Assembly: esso genera delle forme primitive avendo come input la forma creata precedentemente.
* Tessellation Assembly: è capace di dividere ulteriormente le forme primitive del geometry in primitive piccolissime.
* Rasterization Stage: mappa il risultato ai corrispettivi pixel.
* Clipping: rigetta eventuali frammenti che si trovano al di fuori della vista per aumentare le performance.
* Fragment Shader: calcola il colore finale del pixel, ed è la fase dove si verificano tutti gli effetti di OpenGL avanzate.

Una volta creato l’oggetto con i corrispettivi colori esso verrà passo allo stage chiamato Alpha test and blending che controllerà se l’oggetto è di fronte o dietro altri oggetti ed anche il suo valore alpha (la trasparenza).

OpenGL lavora solo con coordinate comprese fra {1.0, -1.0} ed esse vengo chiamate normalized device coordinates. Una volta forniti i vertex data dovremmo anche dire ad OpenGL dove memorizzarli per poi mandarli come input della pipeline, ed esso è fatto mediante una memoria nella GPU che chiameremo vertex buffer objects (VBO). La comodità dei VBO sta nel fatto che non dovremmo mandare ogni volta i vertex data alla pipeline, visto che è un processo relativamente lento, poiché basterà memorizzali all’interno della GPU.

Una volta creato il buffer dovremmo copiarlo nella GPU attraverso il metodo glBufferData, che prendo in input 4 parametri:

1. Il tipo (GL\_ARRAY\_BUFFER)
2. La dimensione in termini di byte
3. L’array vero e proprio
4. Il modo in cui la GPU dovrà gestire i dati, ed esso si divide in:

* GL\_STATIC\_DRAW i dati non cambieranno o lo faranno raramente.
* GL\_DYNAMIC\_DRAW i dati probabilmente cambieranno spesso
* GL\_STREAM\_DRAW i dati cambieranno ogni volta che verranno disegnati

Shaders

Come ben sappiamo per poter utilizzare OpenGL core profile dobbiamo scrivere almeno uno shader per il vertex ed uno per il fragment. Una volta scritti dovremmo collegare i nosti shaders col programma lato host memorizzandoli con id univoci. Ora protremmi eseguirli attraverso uno shaderProgram al quale attaccheremo i nostri shaders per poi eseguirli.

Vertex shader ci permette di specificare qualsiasi input in forma di attributes vertex ma così facendo dovremmo specificare noi stessi i dati che andranno nei attributes vertex. Gli shader prendono in input gli attributes vertex.

Un vertex array object (VAO) può essere associato come un VBO e qualsiasi

vertex attribute successivo chiamato da quel momento in poi verrà memorizzato all'interno del VAO.

Ma oltre ai VBOs ed ai VAOs c’è un altro tipo di buffer che potremmo usare, il element buffer objects (EBO) che lavora proprio come un vertex buffer objects che memorizza gli indici che OpenGL usa per decidere quale vertice disegnare, e lo fa attraverso il indexed drawing.

Gli shader comunicano fra di loro attraverso input ed ouput. Ogni shader devi ricevere qualcosa in input altrimenti è inutile. The fragment shader richiede in input un vec4 a differenza degli altri shaders.

Gli uniforms sono un altro modo per poter passare dati dalla nostra applicazione host agli shaders nella GPU, essi sono globali ovvero è una variabile unica per gli shader program e può essere vista da ogni shader.

Texture

Texture è una immagine 2d usata pe raggiungere dettagli ad un oggetto. Per poter mappare una texture ad un oggetto dobbiamo definire, per ogni vertice, un texture coordinate. Il range della texture coordinate è di {0, 1}. Il recupero del colore della texture usando una texture coordinare è chiamato sampling. Se dovessimo dare dei valori non contenute nel range, di default il comportamento di OpenGL è quello di ripete le immagini della texture, ma esistono anche altre opzioni disponibili:

* GL\_REPEAT comportamento di default
* GL\_MIRRORED\_REPEAT stesso comportamento del repeat ma eseguendo un mirror sull’immagine per ogni ripetizione
* GL\_CLAMP\_TO\_EDGE blocca le coordinate fra {0, 1} e le altre vengo allungate su di esse
* GL\_CLAMP\_TO\_BORDER uguale al clamp\_edge ma le coordinate al di fuori del range saranno senza colore

Texture coordiantes non dipendono dalla risoluzione ma può essere un qualsiasi valore in virgola mobile e quindi OpenGL deve capire quali pixel (texel) vanno mappati con la texture coordinate. Ci sono diversi tipi di texture filtering:

* GL\_NEAREST che è quello di default, in cui OpenGL seleziona i pixel più vicini al centro della texture coordiante.
* GL\_LINEAR assume un valore interpolato dalla trama coordinate del texel vicini, approssimando un colore tra i texel.

Mipmaps è praticamente una collezione di texture images dove ogni sotto sequenza di texture è due volte più piccolo di quello precedente. L’idea dietro il concetto di mipmaps è estremamente facile, dopo una certa distanza dalla “camera” OpenGL userà una differente mipmap texture che meglio si adatta alla distanza dell’oggetto.

Ad ogni texture viene assegnata una location univoca, conosciuta col nome di texture unit, ed essa permetta di avere più texture per uno shader.

Trasformazioni

I vettori, in una visione base, sono semplicemente delle direzioni, infatti esso ha una direzione ed una lunghezza.

Uno scalare è un valore singolo, oppure un vettore con un solo componente.

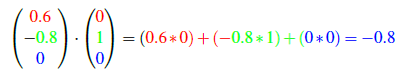
Per determinare la lunghezza di un vettore utilizzeremo il Teorema di Pitagora

C:\Users\Lorenzo\AppData\Local\Microsoft\Windows\INetCacheContent.Word\triangolorettangolo.png

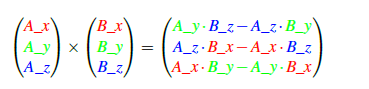
Vi è anche un altro vettore particolare, chiamato unit vector, che ha la peculiarità di avere come lunghezza un valore preciso, che è esattamente 1. Da ogni vettore è possibile ricavare uno unit vector, semplicemente dividendo ogni membro con la sua lunghezza, e questo procedimento prende il nome di normalizing.

Fra i vettori è possibile effettuare la somma, sottrazione, moltiplicazione e divisione.

Nella moltiplicazione fra 2 vettori bisogna far distinzione fra gli operatori {“.”, “x”} il primo ritorna uno scalare



a differenza del prodotto con l’operatore “x”



Ovviamente possiamo estendere questi concetti anche alle matrici dove è possibile fare anche la moltiplicazione fra matrice e matrice, matrice e vettore, matrice e scalare.

Sistemi di coordinate

OpenGL si aspetta che le coordinate x, y e z di ogni vertice siano nel formato normalized device

coordinates (NDC) per essere visualizzate dopo essere state elaborate dal ogni vertex shader.

Le coordinate (x,y,z) devono essere comprese nell'intervallo [-1,1] e all'esterno di questo non

verranno visualizzate. Le coordinate verranno definite da noi in un intervallo preferito e poi

il vertex shader le trasformerà in NDC, queste verranno poi trasferite al rasterizzatore che le trasformerà in coordinate 2D/pixles dello schermo che verranno poi visualizzate.

Ci sono 5 sistemi di coordinate molto importanti:

1. Local space
2. World space
3. View space
4. Clip space
5. Screen space

Local space è lo spazio di coordiante locale all’oggetto in questione, dove il suo centro di massa è sempre (0, 0, 0).

World space tutti gli oggetti presenti nella scena adesso avranno le stesse coordinate, quindi c’è il ritroveremo tutte in (0, 0, 0). Per poter posizionarli come vogliamo dovremmo prima di tutto applicare delle trasformazioni su di essi, che vengono memorizzate nella model matrix.

La model matrix è una matrice di trasformazioni che scala ruota e trasla l’oggetto.

View space è il risultato della trasformazione dalle word-space coordinates a come la camera vede la scena. Questo procedimento è realizzato effettuando delle trasformazioni memorizzate all’interno di una view matrix.

Clip space le view-space coordinates vengono proiettate nel piano di clippong, all’intervallo [-1, 1] e determinano i vertici che verranno visualizzati nello schermo.

Dopo che viene eseguito ogni vertex shader, OpenGL si aspetta che le coordinate siano all'interno di uno specifico intervallo, tutte le coordinate che non rientrano in questo intervallo vengono "clipped" e quindi scartate. Le coordinate rimanenti diventeranno fragments che verranno visualizzate nello schermo.

Per trasformare le coordinate dal view-space al clip-space bisogna definire la matrice di proiezione (projection matrix) la quale stabilisce un intervallo di coordinate, per esempio [-1000,1000], per ogni dimensione. La matrice di proiezione trasforma dunque le coordinate all'interno di questo specifico intervallo in NDC nell'intervallo [-1,1]. Viene poi applicato il clipping. Se una parte della primitiva si trova all'esterno del volume di clipping (clipping volume), OpenGL ricostruirà il triangolo e lo dividerà in triangoli più piccoli in modo da rappresentare quelli che rientrano nel clipping range.

Questa viewing box, che la projection matrix crea, è chiamata frustum e determina le coordinate che saranno visualizzate nello schermo e saranno dunque visibili, in base al fatto se si trovano al suo interno o meno. Il processo di convertire le coordinate all'interno del range dell'NDC in modo da poter essere mappato alle coordinate 2D del view space si chiama projection, siccome la matrice di proiezione proietta le coordinate 3D a quelle 2D dell'NDC.

Una volta che tutte tutti i vertici sono stati trasformati nel clip space viene eseguita un'ultima operazione chiamata perspective division: vengono divise le coordinate x,y,z per le componenti omogenee w del vettore; la perspective division trasforma le coordinate 4D del clip space in coordinate 3D del NDC. Questa operazione viene eseguita automaticamente alla fine di ogni vertex shader.

Le proiezioni ortografiche una orthographic projection matrix definisce un frustum box simile a un cubo che definisce lo spazio di clipping, tutto ciò che si trova all'esterno viene scartato. Il frustum definisce le coordinate visibili ed è definito da una larghezza, altezza e due piani: near e far plane. Tutte le coordinate davanti al near plane e dietro il far plane vengono scartate.

Le proiezioni perspettive mappa un dato intervallo di frustum al clip space, ma manipola

anche il valore w di ogni coordinata in modo tale che più è lontano dall'osservatore, più grande diventa la componente w. Una volta che le coordinate vengono trasformate e adattate al clip space si trovano nell'intervallo [-w,w] (tutto quello che si trova all'esterno viene scartato). Ogni componente del vettore è diviso dalla sua componente w; questo dà come output un vettore 2 con coordinate più piccole man mano che ci si allontana dall'osservatore.

Screen space le coordinate nell'intervallo [-1,1] vengono trasformate in coordinate relative alla dimensione dello schermo, questo processo si chiama viewport transformation che trasforma le coordinate dello step precedente nell'intervallo definito da glViewport. Le coordinate risultanti sono inviate al rasterizzatore che li trasforma in fragments.

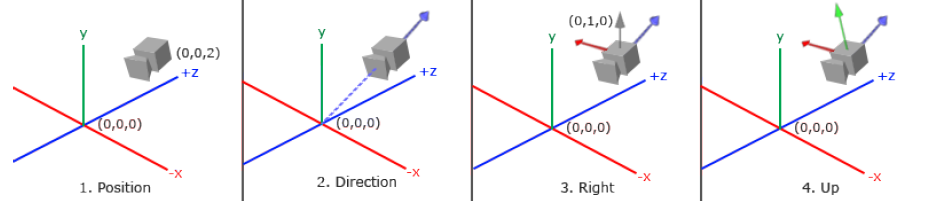
Le matrici di trasformazione sono usate per muovere i vettori da uno spazio di coordinate ad un altro, attraverso i prodotti matriciali. In questo modo è possibile ruotare, scalare, traslare, ecc, un oggetto all’interno del mondo.

OpenGL memorizza tutte le informazioni di profondità in un z-buffer, conosciuto anche come depth buffer. GLFW lo crea automaticamente al posto nostro, al suo interno vi sono memorizzati la profondità ogni fragment e ogni volta che il fragmento vuole far uscire il suo colore, OpenGL compara il suo valore di profondità con lo z-buffer e se il fragment è dietro qualche altro fragment verrà scartato, altrimeti verrà sovrascritto. Questo processo prende il nome di depth testing e viene eseguito automaticamente da OpenGL.

Camera

OpenGL di suo non ha una vera e proprio camera ma possiamo simularla muovendo tutti gli oggetti nella scena nella direzione opposta dandoci l’illusione che ci stiamo muovendo.

Per poter creare una camera avremmo bisogno di una posizione nel world space, un vettore direzione in cui sta guardando, un vettore direzione per la destra uno puntato verso l’alto come si evince dalla foto.



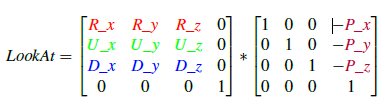
La camera position altro non è che un vettore posizionato nel world space.

La camera direction è un vettore che è la sottrazione della posizione della camera con l’oggetto a cui sta guardando (in genere l’origine). Se dovessimo invertire la sottrazione punteremo verso l’asse negativo della z.

Right axis che rappresenta l’asse positivo della x della camera. Esso viene calcolato eseguendo l’intersezione fra un vettore up (0.0f, 1.0f, 0.0f) e la cameraDirection.

Up axis che rappresenta l’asse positivo della y calcolato come l’intersezione della cameraDirection e la cameraRight.

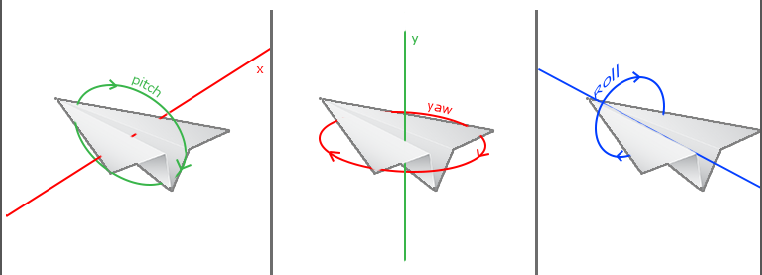
La LookAt è una matrice risultato del prodotto delle seguenti matrice:



Dove R right vector, U up vector, D direction vector e P camera’s position.

Quando moviamo la camera utilizziamo un valore per calcolarci la velocità con cui farlo, essa infatti è data dal prodotto di uno scalare (scelto arbitrariamente) per un valore chiamato deltaTime, che è calcolato come la differenza di tempo per renderizzare il frame precedente.

Oltre al movimento della camera c’è anche la rotazione, Euler angles essa è data dall’unione delle seguenti operazioni: pitch, yaw, rool, che ovviamente posso essere eseguite singole o insieme, ma non simultaneamente.



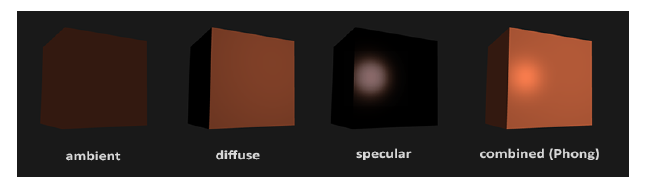
Colors

I colori che noi vediamo provenire dagli oggetti altro non sono che i colori riflessi da essa, ovvero quelli che non sono assorbiti. In OpenGL noi diamo un valore ad una light source che moltiplicheremo col colore di un oggetto in modo da ottenere la luce riflessa da quell’oggetto. Possiamo quindi definire il colore di un oggetto come la quantità di ciascun componente di colore che esso riflette da una sorgente luminosa.

Basic Lighting

Il sistema di illuminazione in OpenGL è basata su un’approssimazione della luce reale, che dipende da troppi fattori per essere rappresentata virtualmente, queste modelli di luce sono basati sulla fisica delle luci. Uno di essi è Phong lighting model. Esso consiste nella composizione di 3 modelli:

1. Ambient lighting: esiste sempre una qualche luce, anche quando è quasi tutto buio, infatti non c’è mai completamente buio in un rappresentazione di OpenGL.
2. Diffuse lighting: simula l’impatto direzionale di un luce che ha sugli oggetti, ed è la più significativa delle luci.
3. Specular lighting: simula un punto luminoso di una luce che appare sugli oggetti luminosi



Ambient lighting viene utilizzata per dare un minimo di luce alla scena in quanto a differenza degli algoritmi di illuminazione globali che sono complessi ed onerosi, questo è davvero semplice da inserire.

Diffuse lighting a differenza della luce ambientale che non fornisce una grande quantità di luce, la luce direzionale illumina maggiormente gli oggetti colpiti da essa, poiché possiamo immaginarla come un fascio di luce proveniente da una direzione ben precisa, ma non si conosce la direzione di riflessione. Colpisce ogni punto con intensita' che dipende dall'angolo formato dalla direzione della luce incidente con la normale alla superficie in quel punto. Se la luce è perpendicolare all’oggetto il suo impatti sarà massimo. Per misurare l’angolo fra il raggio luminoso e il fragment useremo un normal vector che è un vettore perpendicolare alla superfice del fragment.  
L'osservatore percepisce la stessa intensita' di colore indipendentemente dalla sua posizione.

Specular lighting proprio come la luce diffusa, la luce speculare è proprietà direzionale, ma interagisce molto più intensamente con le superfici. A causa della sua natura altamente direzionale, è anche possibile che a seconda della posizione dello spettatore, la luce speculare può non essere visibile. Un faretto e il sole sono buoni esempi di fonti che producono forti luci speculari.

Per valutare il suo contributo alla definizione del colore in un punto, si deve calcolare prima il vettore che va dallo spettatore alla scena e il vettore inverso del vettore luce. Il prodotto scalare fra questi due vettori determina il potere riflettente (più è alto, minore sarà la luce speculare prodotta).

L’equazione del modello di Phong:

Dato un materiale con:

* una proprietà “ambiente” ka
* una proprietà “diffusione” kd
* una proprietà “speculare” ks
* un fattore di lucentezza

Data una luce con:

* una proprietà “ambiente” ia
* una proprietà “diffusione” id
* una proprietà “speculare” is

La formula completa del modello di Phong è:

lp = kaia + kd (L \* N)id + ks(R \* V)is

dove:

lp è l’intensità della luce al punto p

N è il normale della superficie

L è il versore dal punto in ombra al punto luce

R è il vettore di riflessione del negativo del vettore luce L nel piano definito da N

V è il vettore che va dallo spettatore alla scena

Materiali

Nella realtà ogni oggetto riflette in modo differente la luce, per poter simulare un comportamento simile possiamo creare per ogni oggetto un materiale. Ogni materiale è formato dalle 3 luci ed in aggiunta un componente chiamato lucentezza. Per poterlo creare basterà creare una struct in fragment shader e poi memorizzarlo in un uniform.

Lighting maps

Definire un materiale non ci permette di rappresentare veramente la realtà, poiché in generale un oggetto ha diversi materiali, per questo si è pensato di introdurre una mappa per le luci diffuse e speculari.

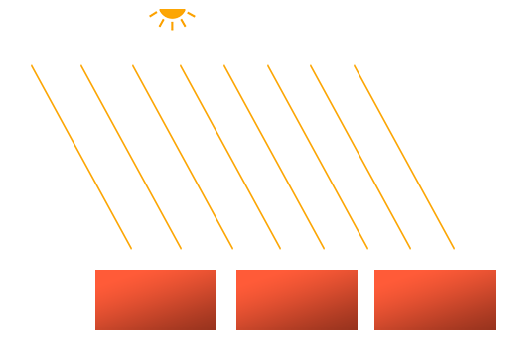
Diffuse maps è una tecnica che prevede di utilizzare una texture per ogni fragment dell’oggetto, infatti adesso metteremo nella nosrta struct del materiale la texture che prenderà il posto della luce diffusa. Inoltre rimuoveremo la luce ambientale in quanto è in genere uguale alla luce diffusa.

Specular maps lo stesso concetto della diffuse maps, utilizzare una texture al posto del componente diffuso.

Light casters

Generalmente in uno scenario reale non abbiamo un solo punto luce come visto finora ma diversi . Una sorgente luminosa che getta luce su altri oggetti viene chiamata light caster.

Direction Light è una luce che emana raggi luminosi paralleli e vicini fra di loro, un esempio pratico potrebbe essere il sole



Visto che tutti i raggi luminosi sono paralleli non importa come ogni oggetto è rivolto verso la luce poiché la posizione della luce rimane la stemma per ogni oggetto della scena.

Point light è una sorgente luminosa con una data posizione da qualche parte nel mondo che illumina in tutte le direzioni dove i raggi luminosi affievolendosi ad una certa distanza.

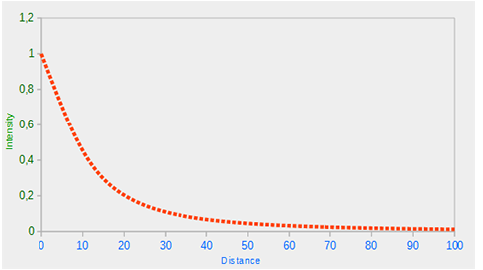
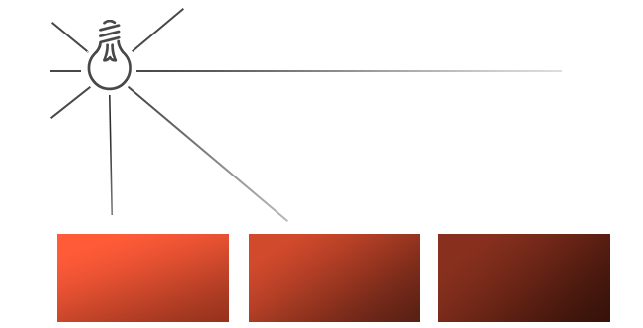
Per poter ridurre l’intensità luminosa dopo una certa distanza si una tecnica chiamata attenuation. Il vero problema è che la luminosità di una sorgente diminuisce rapidamente all’inizio mentre quella rimanente diminuisce più lentamente dopo una certa distanza.



Dove:

* I è l’intensità luminosa
* d rappresenta la distanza dalla luce
* K\_c è un costante generalmente vale 1.0 e viene utilizzata per assicurarci che il risultato determinato non sia inferiore ad uno.
* K\_l è un termine lineare che viene moltiplicato con la distanza
* K\_q è un termine quadrato moltiplicato il quadrato della distanza

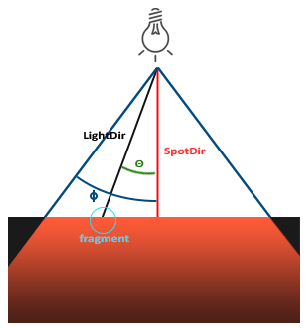
I termini costanti, lineari e quadrati hanno dei valori prestabiliti visibile in una tabella.



Spotlight è una sorgente luminosa che è posizionata da qualche parte nell’ambiente e che anziché emettere raggi luminosi verso tutte le direzione lo fa verso una direzione ben precisa.

Il risultato è che ogni oggetto che incontra il suo raggio in un certo raggio verrà illuminato.

Una spotlight è rappresentata da una posizione una direzione ed da un angolo chiamato cutoff, che specifica il raggio della luce.



Dove:

* LightDir è il vettore direzione dal fragment alla luce
* SpotDir la direzione della della luce
* Φ l’angolo che rappresenta il raggio della luce
* Θ l’angolo fra la LightDir e la SpotDir, questo valore

Dovrebbe essere più piccolo di Φ.

Flashlight è una spotlight particolare, perché essere è posizionata esattamente dove c’è l’osservatore. Una peculiarità di questa luce e che essa è in continuo aggiornameto poiché viene attacca alla camera, ma un difetto è che i fragment al di fuori del raggio sono scuri quindi vi è uno stacco netto fra i fragment illuminati e quelli non.

Per sviare questo problema utilizziamo delle tecniche di smussatura della luce, cominciamo con simulare una spotlight avente un cono interno ed uno esterno. Settiamo il cono interno come la spotlight normale mentre quello esterno che gradualmente si attenua. Per definire il cono esterno basta semplicemente calcolare un altro coseno che rappresenta l’angolo fra la direzione della spotlight e il vettore del cono esterno. Se il fragmento è il cono esterno e quello interno dovremmo calcolare la sua intensità compresa fra {0.0, 1.0}.



Dove epsilon è il la differenza fra il coseno del cono interno con il coseno del cono esterno.

Stencil testing

Una volta che il fragment shader ha processato il fragment, viene eseguito un stencil test che come il depth test, può scartare i fragments. Questo test è basato sul contenuto di un altro buffer chiamato stencil buffer che ci permetto di aggiornarlo durante la renderizzazione.

A stencil buffer contiene solitamente 8 bits per stencil value che sommati fanno un totale di 256 differenti stencil value per pixel/fragment.

Lo stencil buffer viene prima riempito con tutti 0 e poi si mette 1 nelle posizioni dove vogliamo che il test venga superato e 0 altrimenti, ora i fragments della scena vengono processati e renderizzati ogni volta che il stencil value del fragment è 1.

Object outlining è esattamente, come suggerisce il nome, una bordo attorno ad ogni oggetto colora, questo potrebbe risultare untile nei videogame quando si seleziona un oggetto.

Cubemaps

Fino ad ora abbiamo usato solo le texture 2d, ma esisto altri tipi di texture, uno fra questa è la cubemap, ovvero una combinazione multipla di texture.

Un cubemap è praticamente una texture che contiene 6 texture 2d e che ogni forma una lato di un cubo. Uno dei sui vantaggi è quello che le 6 unità possono essere indicizzati come un vettore direzione. L’unione delle 6 texture è un vettore chiamato textures\_faces.

Skybox è un enorme cubemap che circonda l’intera scena.

Environment mapping usando un cubemap con un ambiante possiamo dare agli oggetti della scena delle proprietà, la reflection e la refraction.

Advanced lighting

Il modello di phong è abbastanza buona per le luci ma soffre quando parliamo di sfumature, e quindi introduciamo una sue estensione, Blinn-Phong.

Il modell di Blinn-Phong è una estensione di Phong, ma differisce sull’approccio del componente speculare, perché anziché utilizzare riflettere la luce utilizzando il vettore di riflessione utilizzato da Phong esso utilizza un nuovo vettore chiamata halfway vector che è un unit vector esattamente a metà strada fra la direzione dell’osservatore e il vettore direzione



Gamma correction

I vecchi monitor a tubo a raggi catodici avevano la proprietà fisica che se in input entrava il doppio della tensione non significa che uscirà il doppio della quantità luminosa. Il raddoppio della tensione di ingresso comporta una luminosità uguale ad un rapporto di 1/2,2 conosciuta col nome di gamma. L’idea del gamma correction è quella di applicare l’inversa gamma del monitor all’output finale dei colori prima di mostrarli a video.

Molte texture sono in sRGB space quindi vi è già applicata una gamma correction ma questo rendere l’immagina troppo luminosa e quindi vi sono 2 soluzioni:

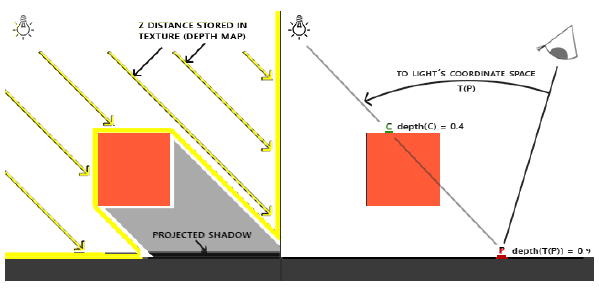
1. Dire agli artisti di lavorare in un linear space, ma molto probabilmente non sanno nemmeno cosa sia
2. Effettuare una re-correct, quindi trasformare le textures da un sRGB space ad uno lineare prima di poter calcolare il colore.

Le texture utilizzate per colorare gli oggetti, come texture diffuse sono quasi sempre sRGB, metre quelle utilizzate per il recupero dei parametri delle luci sono lineari.

Un altro aspetto importante è quello della attenuazione, ovvero l’illuminazione si attenua strettamente in basa al quadrato della distanza, cioè che la forza della luce è ridotta lungo la distanza della sorgente al quadrato. Ma in questo modo l’attenuazione della luce è troppo forte, senza aver eseguito una correzione gamma, e quindi di calcola la luce rapportata alla distanza invece che al suo quadrato, nel caso in cui eseguiamo prima una correzione gamma utilizzeremo il quadrato.

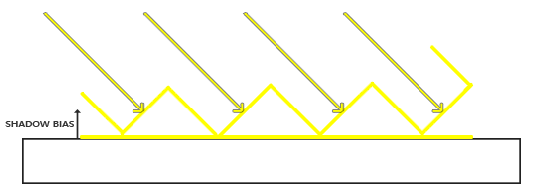
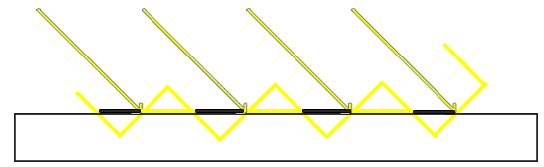
Shadow Mapping

Le ombre sono i il risultato dell’assenza di luce dovuta all’occlusione di essa. Danno un ottimo senza di profondità alla scena. Non esiste ancora un algoritmo davvero ottimo per le ombre ma solo delle buone tecniche di approssimazione. Un tecnica usata da mosti videogame è quella del shadow mapping. L’idea dietro questa tecnica è abbasta semplice, renderizzamo la scene a partire da ogni sorgente di luce verso l’osservatore e tutto ciò che vediamo è illuminato e tutto ciò che non vediamo è in ombra. Memorizziamo questi valori in una texture chiamata depth map o shadow map.



Prima di tutto ci creiamo la depth map, come detto precedemente, emettiamo dei raggi luminosi paralleli e tutto ciò che non vediamo è in ombra, e dobbiamo memorizzare questi dati in framebuffers. Attraverso il fragment shaders possiamo renderizzare la scena usando Blinn-Phong e in questo modo dedurre l’ombra se il valore è 1.0 altrimenti 0.0 quando è illuminato, il risultato viene moltiplicato con la luce diffusa e quella speculare.

Questa tecnica ha un piccolo difetto, ovvero i raggi luminosi provenienti da una sorgente Questa tecnica genera un artefatto della luce al quanto sgradevole, poiché la shadow map ha una risoluzione limitata i fragments più lontani dalla sorgete luminosa potranno avere lo stesso valore nell depth map, questto effetto prende il nome di shadow acne. Questo diventa problematico quando i raggi luminosi colpiscono gli angoli, così facendo otterremo delle discrepanze nelle ombre, e quindi alcuni fragments risulteranno in ombre pur non essendoli e si genera un patter a righe.



Questo è dovuto dal fatto che la shadow map ha una risoluzione limitata, quindi più frammenti nel depth map risultano con lo stesso valore. Per risolvere questo problema si utilizza una piccolo un shadow bias col quale possiamo compensare la profondità della superfice di una piccola quantità in modo tale che i fragments non risulteranno erroneamente al di sotto della superfice.

Questo risolve il problema dello shadow acne ma ne genera una nuovo, il problema di peter panning dove praticamente gli oggetti sembrano librarsi leggerme sopra la superfice. Per risolverlo utilizziamo un face-culling così facendo non andremmo a considedare il punto più in alto del fragment ma quello più basso.

