Getting Started

Introduzione

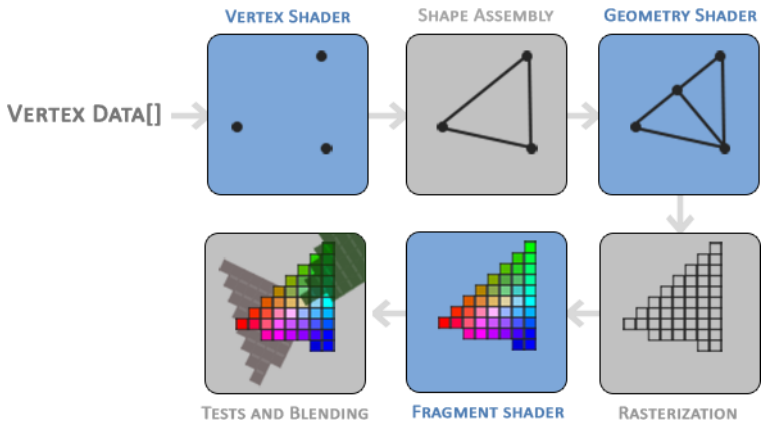
OpenGL è una libreria che fornisce un'interfaccia per la programmazione di applicazioni multipiattaforma per la renderizzazione di vettori grafica 3D (solo). OpenGL è una grande macchina a stati, composta da un insieme di variabili che definiscono il suo funzionamento. Ogni stato di OpenGL è detto comunemente **contesto OpenGL**. Questo stato si può semplicemente cambiare modificando alcune opzioni, buffer e, successivamente, eseguendo il rendering del contesto corrente si può visualizzare il risultato ottenuto. Quando si lavora in OpenGL, infatti, ci si imbatte in diverse funzioni di cambiamento di stato che permettono di cambiarlo, e altre funzioni che permettono di eseguire determinate operazioni basate sul contesto corrente.
Un oggetto in OpenGL non è altro che una raccolta di opzioni che rappresentano un sottoinsieme dello stato di OpenGL.

Concetti generali

In OpenGL ogni cosa è in uno spazio 3D, ma gli schermi e le finestre sono array di Pixel 2D. Dunque, gran parte del lavoro svolto da OpenGL è quello di trasformare coordinate 3D in pixel 2D visibili sullo schermo. Questo processo è svolto dalla **pipeline grafica** di OpenGL. Essa può essere divisa in 2 grandi parti: una prima trasforma le coordinate 3D in pixel (coordinate) 2D, ed una seconda che si occupa di trasformare i pixel (coordinate 2D) in pixel colorati.
Tra una coordinata 2D ed un pixel c'è una di differenza: una coordinata 2D è una rappresentazione molto precisa di dove il punto è in uno spazio 2D, mentre un pixel 2D è un'approssimazione del punto stesso nel limite dello schermo (finestra visibile).

La Pipeline

La pipeline grafica prende in input un insieme di coordinate 3D e le trasforma in pixel colorati 2D, mostrandoli sullo schermo. Per far ciò essa svolge determinati step, ognuno dei quali ha bisogno dell'output dello step precedente per funzionare. Tutti questi step sono altamente specializzati in determinate funzioni e possono essere facilmente eseguiti in parallelo.
Proprio grazie alla loro natura parallela, la maggior parte delle GPU odierne hanno migliaia di piccoli core di elaborazione per elaborare rapidamente i dati all'interno della pipeline eseguendo dei piccoli programmi per ogni step della pipeline stessa. Questi piccoli programmi sono detti **shaders**, i quali sono scritti in **OpenGL Shading Language**, (GLSL). Come input per la pipeline grafica passiamo una lista di tre coordinate 3D che dovrebbero formare un triangolo in un array chiamato Vertex Data. Questo è una raccolta di vertici. Un vertice, a sua volta, è una raccolta di dati per coordinate 3D. I dati di questo vertice sono rappresentati usando attributi dei vertici che per ogni vertice contengono la posizione 3D e il valore colore.



La prima parte della pipeline è il **vertex shader** che prende come input un singolo vertice. L'obiettivo principale di questo step è quello di trasformare coordinate 3D in nuove coordinate 3D applicando sui vertici in input e sui loro attributi diverse trasformazioni.
La fase **primaria di assemblaggio** prende in input tutti i vertici dal vertex shader e forma una primitiva assemblando tutti i punti nella forma primitiva data (un triangolo).
L'output della primitiva assemblata è passato in seguito al **geometry shader**, il quale prende in input una collezione di vertici che formano la primitiva ed ha la capacità di generare altre forme primitive, creando nuovi vertici.

L'output del geometry shader viene in seguito passato allo step seguente, la **rasterization**, nella quale le primitive vengono mappate ai pixel corrispondenti nella finestra visiva, creando dei fragment che saranno utilizzati in seguito dai fragment shaders. Prima che i fragment shaders vengano eseguiti, si ha una fase di clipping, in cui tutti i frammenti fuori dalla finestra vengono scartati, in modo da aumentare le performances.
Ogni fragment è l'insieme dei dati necessari ad OpenGL per renderizzare un singolo pixel.
Il principale obiettivo del **fragment shader** è quello di calcolare il colore finale di un pixel, fase in cui tutti gli effetti avanzati di OpenGL si verificano. Lo shader, come detto prima, contiene infatti tutti i dati per calcolare il pixel nale,quali luci, ombre, colore ecc...
Dopo che tutti i valori di colore corrispondente sono stati determinati, l'oggetto finale passerà attraverso un ulteriore stadio chiamato **alpha test e blending**. Questa fase controlla il corrispondente valore di profondità del frammento e li utilizza per verificare che il frammento risultate si trova davanti o dietro altri oggetti e deve essere scartato di conseguenza. Quindi, anche se un colore di output di un pixel viene calcolato nel fragment shader, è possibile che il colore finale assuma un colore diverso.

Vertex Input
Una volta che le coordinate del vertice sono state elaborate nel vertex shader, dovrebbero esserci coordinate del dispositivo normalizzate (normalized device coordinates), ovvero, un piccolo spazio in cui le coordinate x, y e z variano valori da -1.0 a +1.0. Tutte le coordinate che non rientrano in questo intervallo verranno scartate e non saranno visibili sullo schermo.
Dopo che i dati del vertice sono stati definiti, vorremmo inviarli come input al primo processo della pipeline grafica (vertex shader). Questo viene fatto creando memoria sulla GPU in cui archiviamo i dati dei vertici.
La memoria della GPU viene gestita tramite i **vertex buffer object** (VBO) che memorizzano un gran numero di vertici nella memoria della GPU. Il vantaggio nell'utilizzare i VBO è quello di poter inviare grandi quantità di dati contemporaneamente alla scheda grafica senza essere obbligati ad inviare un singolo vertice per volta. Far ciò è utilissimo in quanto il passaggio di dati verso la CPU è relativamente lento, quindi mandare il maggior numero di dati contemporaneamente è di vitale importanza per le prestazioni.

Vertex Shader
Il vertex shader è uno degli shader programmabili. OpenGL richiede la programmazione di un vertex shader e fragment shader per il redenring.
Nella programmazione grafica usiamo il concetto matematico di un vettore abbastanza spesso, poiché rappresenta ordinatamente posizioni/direzioni in qualsiasi spazio e ha proprietà matematiche utili. Un vettore in GLSL ha una dimensione massima di 4 e ciascuno dei suoi valori può essere recuperata tramite vec.x, vec.y, vec.z e vec.w rispettivamente dove ognuno di essi rappresenta una coordinata nello spazio. Si noti che il componente vec.w non viene utilizzato come posizione nello spazio a viene usato per qualcosa chiamato **prospective division**.

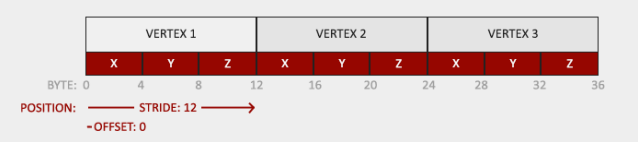
Fragment Shader
Il fragment shader è il secondo e ultimo shader programmabile per il rendering di un triangolo. Il fragment shader consiste nel calcolare il colore di output dei pixel. I colori in computer graphics sono rappresentati come una matrice di 4 valori: il componente rosso, verde, blu e alfa (opacità), comunemente abbreviato con RGBA. Quando si definisce un colore in OpenGL, si imposta l'intensità di ciascun componente su un valore compreso tra 0.0 e 1.0.
Il fragment shader richiede solo una variabile di output e questo è il vettore di dimensione 4 che definisce l'output del colore.

Shader Program
L'oggetto shader program è la versione finale collegata di più shader combinati. Per usare gli shader dobbiamo collegarli sullo shader program.

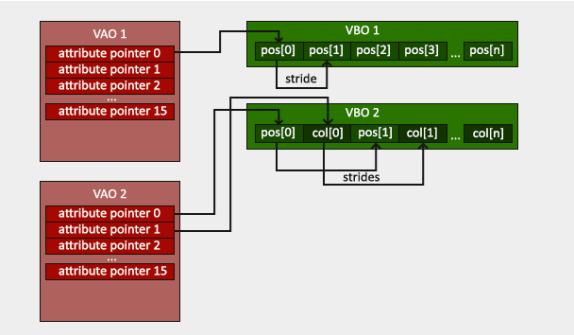
Linking Vertex Attributes
Il vertex shader ci consente di specificare qualsiasi input desideriamo sotto forma di attributi di vertici e mentre ciò consente una grande flessibilità, ciò significa che dobbiamo specificare manualmente quale parte dei nostri dati di input va a quale attributo di vertice nel vertex shader. Quindi, dobbiamo specificare come OpenGL dovrebbe interpretare i dati del vertice prima del rendering.

I buffer dei dati sono formati nel seguente modo:

- I dati di posizione vengono memorizzati come valori
  In virgola mobile a 32 bit (4 byte);
- Ogni posizione è composta da 3 di questi valori;
- Non c'è spazio tra ogni set di 3 valori.



Vertex Array Object



Un vertex array object (VAO) può essere associato esattamente come un vertex buffer object, in modo che qualsiasi chiamata successiva a quel vertice possa essere memorizzata all'interno del VAO. Questo ha il vantaggio che quando si configurano i puntatori degli attributi del vertice, è necessario effettuare tali chiamate una sola volta e ogni volta che si desidera disegnare l'oggetto, è posisbile associare il VAO corrispondente.

Element Buffer Objects

Un element buffer object (EBO) è un buffer che memorizza gli indici che OpenGL utilizza per decidere quali vertici disegnare.

Shaders

Gli shader sono piccoli programmi che lavorano sulla GPU. Questi programmi vengono eseguiti per ciascuno degli step della pipeline grafica. In un certo senso, gli shader non sono altro che programmi che trasformano gli input in output. Essi lavorano in modo isolato, in quanto non sono autorizzati a comunicare tra di loro. L'unica comunicazione degli shaders è attraverso input e output.

GLSL

Gli shader sono scritti nel linguaggio C-like GLSL, un linguaggio personalizzato per l'uso grafico contente utili funzioni specifiche mirate alla manipolazione di vettori e matrici.
Come qualsiasi altro linguaggio di programmazione, GLSL ha tipi di dati per specificare con quale tipo di variabile vogliamo lavorare. Oltre alla maggior arte dei tipi base, GLSL presenta due container *vectors* e *matrices*.
Un vettore in GLSL è un contenitore di 1,2,3 o 4 componenti per uno dei tipi base.
Ogni shader può specificare input e output utilizzando le parole chiave **in** e **out**. Fondamentale è che una variabile di output corrisponda a una variabile di input del successivo livello shader, cioè, la variabile "out" del primo livello di shader deve avere lo stesso nome e tipo di una variabile "in" del secondo livello di shader. Il vertex shader differisce per la tipologia del suo input, in quando, riceve l'input direttamente dal contenitore e dal buffer che contiene i dai del vertice. Per definire come sono organizzati i dati del vertice si specificano le variabili di input con metadati di posizione in modo da poter configurare gli attributi dei vertici sula CPU. Questo viene fatto da: *layout (location = 0)*.
Un'altra cosa importante è che il fragment shader richiede una variabile di output vec4 contenente il colore, poiché i fragmnet shader generano un colore di output finale, il quale, se non specificato sarà bianco o nero.

Uniforms

Gli uniforms sono un altro modo per trasferire i dati dalla nostra applicazione sulla CPU agli shader della GPU. Gli uniforms sono leggermente diversi rispetto agli attributi del vertice. Prima di tutto, gli uniforms sono **globali** (condivisi fra tutti gli shader). In secondo luogo, qualunque sia l'impostazione del valore, gli uniforms manterranno i loro valori fino a quando non vengono rispristinati o aggiornati.

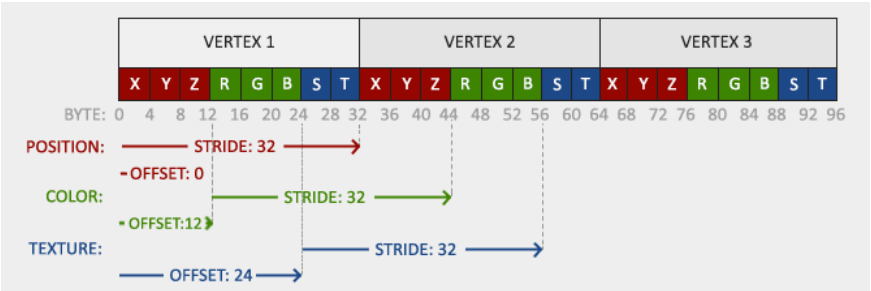
Textures

Una texture è un'immagine 2D usata per aggiungere dettaglio ad un oggetto. Poiché possiamo inserire molti dettagli in una singola immagine dobbiamo cercare di dare l'illusione che l'oggetto sia estremamente dettagliato senza dover specificare vertici aggiuntivi.
Per mappare una texture su un triangolo, abbiamo bisogno di associare ogni vertice di un oggetto con la parte di texture a cui dovrà corrispondere. Dunque, ogni vertice deve avere una **coordinata texture** associata che specifica la parte d'immagine da campionare e il fragment shader poi si occuperà della restante parte. Le coordinate della texture permettono il recupero della texture. Questo processo è detto **campionamento**. Le coordinate della texture vanno da (0,0), per l'angolo inferiore sinistro a (1,1), per l'angolo superiore destro dell'immagine della texture.
OpenGL si occupa di mappare ogni pixel ad un texel, tuttavia non c'è sempre una corrispondenza 1 a 1 specialmente quando la risoluzione della texture è molti differente da quella dei pixel a disposizione. Quindi, il compito di OpenGL è quello di decidere quale colore assegnare ad ogni pixel in base alla texture.

Ci sono due approcci possibili:

1) GL_NEAREST: assegnare ad un pixel un colore in base alla distanza più vicina fra il suo centro e la coordinata della texture (metodo di default);
2) GL_LINEAR: assegna al pixel un colore ottenuto interpolando il colore delle coordinate della texture più vicino ad esso approssimando il colore dei vari texel.

Dopo aver aggiunto le texture, il buffer dei dati sarà costruito nel seguente modo:



Mipmap

Supponiamo di visualizzare un grande ambiente con tanti oggetti di piccole dimensioni sui quali sono attaccati delle texture ad alta risoluzione. In questo contesto, è facile intuire che essi in realtà essendo di piccole dimensioni producono solamente pochi fragment di conseguenza OpenGL trova delle difficoltà nel trovare il giusto colore da assegnare ad ogni fragment a causa dell'alta risoluzione della texture e questo produrrà un effetto visibile su gli oggetti in questione, per non parlare, inoltre, dello spreco di memoria che comportano l'uso di texture ad alta risoluzione in questo caso. Per risolvere questa cosa OpenGL usa il concetto di mipmaps che è in sostanza una collezione di texture ognuna delle quali è due volte più piccola di quella precedente. A questo punto, esso utilizzerà la texture che meglio si adatta alla distanza dell'oggetto. Dato che l'oggetto è lontano, la risoluzione più piccola non sarà visibile all'utente e comunque le prestazioni risultano essere migliori.

Transformations

Si potrebbe rendere dinamici gli oggetti cambiando i loro vertici e riconfigurando i loro buffer ad ogni frame, ma questo sarebbe dispendioso a livello prestazionale. Esistono modi molto migliori per trasformare un oggetto, ovvero usando **oggetti matrici**.

Scaling

Scelto un punto C di riferimento, scalare un generico punto P significa riposizionare P sul piano cartesiano ricalcolando le sue distanze da C in base ad una nuova unità di misura. La scalatura ha quindi l'effetto di variare le dimensioni della primitova geometrica lungo gli assi principali in accordo ai fattori di scala s_1, s_2, s_3 selezionati.

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{pmatrix}$$

Translation

Traslare una primitiva geometrica nel piano cartesiano significa muovere ogni suo punto P di T_x unità lungo l'asse delle x, T_y unità lungo l'asse delle y e T_z lungo l'asse delle z fino a raggiungere la nuova posizione.

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

Rotation

Fissato un punto C di riferimento ed un verso di rotazione, ruotare un punto P rispetto a C significa muovere P attorno a C di un centro angolo nel verso assegnato.

Rotation around the X-axis:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

Rotation around the Y-axis:

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x + \sin\theta \cdot z \\ y \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

Rotation around the Z-axis:

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \\ 1 \end{pmatrix}$$

Combining matrices

Il vero potere dell'uso delle matrici per le trasformazioni è che possono combinare più trasformazioni in un'unica matrice grazie alla moltiplicazione tra matrici.

La moltiplicazione tra matrici non è commutativa, quindi, quando si moltiplicano le matrici per una trasformazione, la matrice più a destra viene applicata per prima.

GLM

GLM sta per OpenGLMathematics ed è una libreria di *sola intestazione*, il che significa che dobbiamo solo includere i file di intestazione corretti.

Per ottenere le matrici di trasformazioni per gli shader, utilizziamo mat4, in quando GLSL, oltre ai tipi vettori ha anche matrici. Adatteremo, quindi, il vertex shader per accettare una variabile uniforms mat4 per la trasformazione. La posizione finale sarà data, quindi, dalla moltiplicazione della matrice di trasformazione con il vettore posizione.

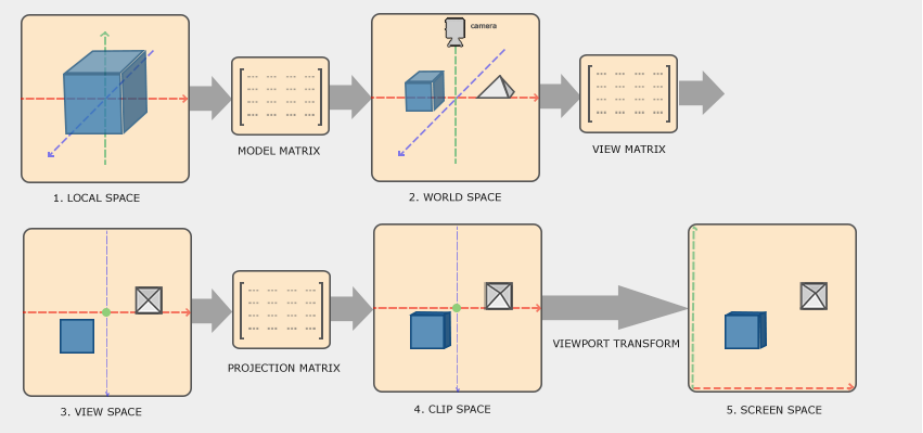
Coordinate Systems

La trasformazione delle coordinate in normalized device coordinates (NDC) viene di solito eseguita in più fasi, in cui gli oggetti vengono trasformati in diversi sistemi di coordinate, prima di arrivare alla forma finale. Il vantaggio di trasformarli in diversi sistemi di coordinate intermedie è che alcune operazioni sono più facili in certi sistemi rispetto che in altri. Ci sono un totale di cinque diversi sistemi importanti:

1. Local Space (or Object soace);
2. World Space;
3. View Space (or Eye space);
4. Clip Space;
5. Screen Space.

The global picture

Per trasformare le coordinate da uno spazio al successivo spazio delle coordinate useremo diverse matrici di trasformazione di cui le più importanti sono **model**, **view** e **projection**. Le coordinate del nostro vertice vengono inizialmente impostate come coordinate locali, successivamente elaborate in coordinate globali, coordinate di visualizzazione, coordinate clip e infine in coordinate dello schermo.



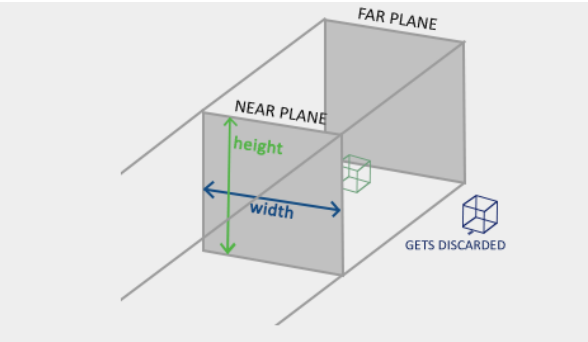
La trasformazione si basa su alcune fasi:

1. Le coordinate **local space** sono le coordinate del nostro oggetto relative alla sua origine locale;
2. Il prossimo passo è trasformare le coordinate locale in coordinate **world space**, cioè nello spazio globale. Queste coordinate sono relative ad un'origine globale, comune a tutti gli oggetti, in modo che sia possibile metterle in relazione con quelle degli altri oggetti presenti nel world space;
3. Successivamente trasformiamo le coordinate globali in coordinate **view space**, in modo tale che ciascuna coordinata sia trasformata dal punto di vista della telecamera o dello spettatore;
4. Dopo che le coordinate sono nello view space, vengono trasformate in coordinate **clip space**, nel quale vengono elaborate nell'intervallo -1.0 a 1.0 e si determinano i vertici che saranno visualizzati sullo schermo;
5. Infine, si trasformano le coordinate clip space in coordinate **screen space** attraverso un processo che trasforma le coordinate dall'intervallo -1.0 e 1-0, alla gamma di coordinate definite da *glViewport*. Le coordinate risultanti vengono quindi inviate alla rasterizzazione che li trasforma in frammenti.

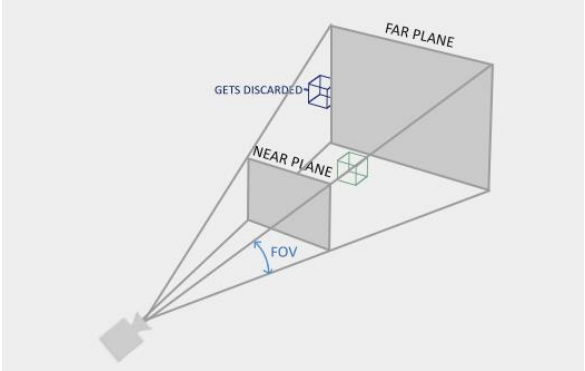
La funzione *glViewport* specifica il rettangolo effettivo della finestra per il rendering. La funzione richiedere 4 coordinate del rettangolo di visualizzazione. Le coordinate specificate indicano a OpenGL come deve mappare le sue coordinate normalizzate del dispositivo alle coordinate della finestra.

Per trasformare le coordinate del view space al clip space si definisce una **matrice di proiezione** che specifica un intervallo di coordinate, ad es. -1000 e 1000 in ogni dimensione.
Questa finestra di visualizzazione creata dalla matrice di proiezione viene detta **frustum**, ed ogni coordinata che finisce all'interno di questo riquadro si troverà nella schermata dell'utente.
La matrice di proiezione per trasformare le coordinate view space in coordinate clip space può assumere due forme diverse, in cui ogni forma definisce il suo unico frustum. Infatti, possiamo creare una matrice di proiezione ortografica o una matrice di proiezione prospettica.

Ortographic projection (distanza infinita tra centro e piano di proiezione)
Una matrice di proiezione ortografica definisce una scatola simile a un cubo che definisce lo spazio di ritaglio, in cui ogni vertice esterno a questo spazio è scartato. Quando si crea una matrice di proiezione ortografica, si specificano la larghezza, l'altezza e la lunghezza del frustum visibile. Tutte le coordinate che finiscono all'interno di questo frustum dopo la trasformazione in clip-space con la matrice di proiezione ortografica non verranno scartate.
Il frustum definisce le coordinate visibili ed è specificato da una larghezza, un'altezza e da due piani, uno vicino e uno lontano. Qualsiasi coordinata davanti al piano vicino viene ritagliata, lo stesso vale per le coordinate dietro il piano lontano. Il frustum ortogonale mappa direttamente tutte le coordinate all'interno del frustum attraverso coordinate del dispositivo normalizzate. Questo poiché la componente w di ciascun vettore non viene toccata; se la componente w è uguale a 1.0 la divisione prospettica non cambia le coordinate.



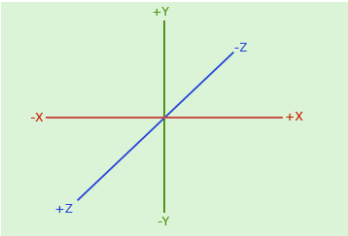
Prospective projection (distanza finita tra centro e piano di proiezione)
Nella proiezione prospettica si aggiunge il concetto di distanza, ovvero si cerca di simulare la realtà, in cui un oggetto, a seconda della distanza a cui ci troviamo, sembra cambiare dimensione.
La matrice di proiezione mappa un dato intervallo di frustum per ritagliare lo spazio, ma manipola anche il valore w di ogni coordinata in modo tale che quanto più lontana è una coordinata dal visualizzatore (utente), tanto più alto diventa w.
Una volta che le coordinate vengono trasformate nel clip space, sono comprese nell'intervallo da -w a w (qualsiasi cosa al di fuori di questo intervallo viene tagliata). OpenGL richiede che le coordinate visibili rientrino nell'intervallo -1.0 e 1.0 come output finale del vertex shader, quindi una volta che le coordinate si trovano nel clip space, viene applicata la divisione prospettica. Il valore **fov**, che sta per campo visivo, imposta quanto grande è lo spazio della vista. Per una vista realistica, di solito è impostato su 45 gradi.



Putting it all together
Si crea dunque una matrice di trasformazione per ciascuna delle fasi sopra indicate: model, view e projection. Una coordinata di viene quindi trasformata in coordinate clip space con i seguenti passaggi:

$$V_{clip} = M_{projection} * M_{view} * M_{model} * V_{local}$$

Right-handed system



Per convenzione, OpenGL è un sistema destrorso. Ciò che sostanzialmente dice questo è l'asse x positivo è alla nostra destra, l'asse y positivo è in alto e l'asse z positivo è all'indietro.

Z-Buffer

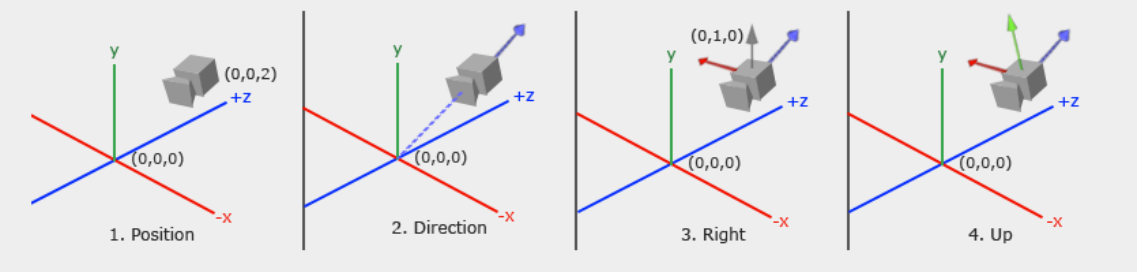
OpenGL memorizza tutte le informazioni di profondità in un buffer z, noto anche come buffer di profondità. La profondità è memorizzata all'interno di ogni frammento e ogni volta che il frammento vuole produrre il suo colore, OpenGL confronta i suoi valori di profondità con il buffer z e, se il frammento corrente è dietro l'altro frammento, viene scartato, altrimenti sovrascritto. Questo processo è chiamato test di profondità e viene eseguito automaticamente da OpenGL.

Camera

OpenGL di sui non ha una vera e propria camera, ma possiamo simularla muovendo tutti gli oggetti nella scena nella direzione opposta dando l'illusione del movimento.

Camera/View space

Per definire una camera abbiamo bisogno della sua posizione nel world space, della direzione in cui sta guardando, di un vettore che punta a destra e di un vettore che punta verso l'altro della camera. Si tratta effettivamente di un sistema di coordinate con 3 assi unitari perpendicolari, con la posizione della telecamera come origine.



1. Camera position: altro non è che un vettore posizionato nl world space;
2. 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;
3. Right axis: rappresenta l'asse positivo della x della camera. Esso viene calcolato eseguento l'intersezione fra vettore up (0.0f, 1.0f, 0.0f) e la camera direction;
4. Up axis: rappresenta l'asse positivo della y ed è calcolato come l'intersezione della camera direction e camera right.

Look At

La Look At è una matrice risultato del prodotto delle seguenti matrici:

$$LookAt = \begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

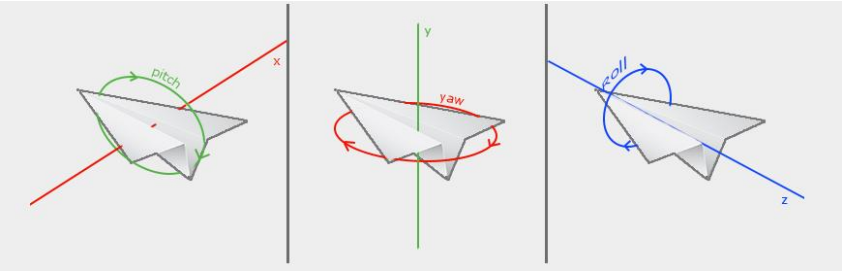
dove R è right vector, U è up vector, D è direction vector e P è camera position.
L'uso di questa matrice trasforma le coordinate da world space a view space.

Movement speed

Quando muoviamo 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 **delta time**, che è calcolato come la differenza di tempo per renderizzare il frame precedente.

Euler angles

Oltre al movimento della camera c'è anche la rotazione, data dall'unione delle seguenti operazioni: pitch, yaw e rool. Ovviamente, possono essere eseguite singolarmente o insieme, ma non simultaneamente.



Gli angoli di Eulero sono 3 valori che possono essere rappresentare qualsiasi rotazione in 3D. Ci sono 3 angoli di Eulero: pitch, yaw e roll.
Il **pitch** è l'angolo che mostra quanto stiamo guardando in alto o in basso come visto nella prima immagine.
L'angolo **yaw** rappresenta la grandezza che stiamo guardando a sinistra o a destra.
L'angolo **roll** rappresenta la quantità di *rotoli* utilizzata principalmente nelle telecamere spaziali.

Lighting

Colors

I colori sono rappresentati digitalmente usando un componente rosso, verde e blu. Usando diverse combinazioni di questi 3 valori possiamo rappresentare tutti i colori conosciuti. I colori che vediamo nella vita reale non sono i colori effettivi degli oggetti, ma solo i colori riflessi dall'oggetto a contatto con la luce; i colori che non assorbiti dagli oggetti sono i colori che noi percepiamo. Queste regole di riflessione del colore si applicano direttamente nella land-graphics.
Quando definiamo una sorgente luminosa in OpenGL si da a questa fonte di luce un colore. In seguito, se si moltiplica il colore della sorgente luminosa con il valore del colore di un oggetto, il colore risultante è il colore riflesso dell'oggetto.

Basic Lighting

L'illuminazione in OpenGL si basa su approssimazioni della realtà usando modelli semplificati che sono molto più facili da elaborare e sembrano relativamente simili. Uno di questi modelli è il modello di illuminazione di **Phong**. I principali elementi costitutivi del modello di Phong sono:

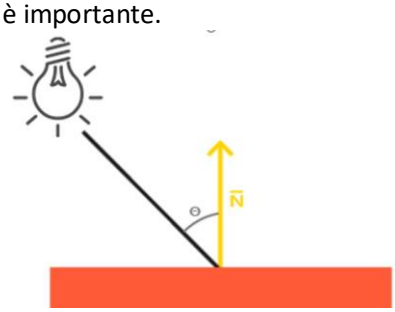
- **Ambient lighting**: anche quando è buio, di solito c'è sempre una fonte di luce, anche se lieve, in qualche parte del mondo. Di conseguenza gli oggetti non sono quasi mai completamente al buio. Per simulare ciò, si utilizza una costante di illuminazione ambientale che conferisce all'oggetto sempre un po' di colore.
- **Diffuse lighting**: simula l'impatto direzionale di una fonte di luce su un oggetto. Questo è il componente visivo più significativo del modello di illuminazione. Più una parte di un oggetto è rivolto verso la sorgente luminosa, più diventa luminosa.
- **Specular lighting**: simula il punto luminoso di una luce che appare sugli oggetti lucidi. I punti illuminati della luce speculare sono spesso più inclini al colore della luce stessa, rispetto al colore dell'oggetto.

Ambient lighting

Di solito non c'è un'unica fonte di luce, ma molte sparse intorno a noi, anche quando non sono visibili. Una delle proprietà della luce è che può disperdersi e rimbalzare in molte direzioni raggiungendo punti che non si trovano nelle sue immediate vicinanze; la luce può quindi riflettersi su altre super ci e avere un impatto indiretto sull'illuminazione di un oggetto. Gli algoritmi che prendono in considerazione ciò sono chiamati algoritmi di illuminazione globale, ma questi sono dispendiosi o complessi. Dato che non siamo grandi fan di algoritmi complicati e costosi, inizieremo utilizzando un modello molto semplicistico di illuminazione globale, ovvero l'illuminazione ambientale. Per aggiungere illuminazione ambientale alla scena si prende il colore della luce, lo si moltiplica con un piccolo fattore ambientale costante, lo si moltiplica con il colore dell'oggetto e lo usiamo come colore del frammento.

Diffuse lighting

L'illuminazione ambientale di per sé non produce i risultati più interessanti, ma l'illuminazione di usa inizia a dare un impatto visivo significativo sull'oggetto. L'illuminazione di usa conferisce all'oggetto più luminosità quanto più i suoi frammenti sono allineati ai raggi di luce provenienti da una qualsiasi sorgente luminosa. Conoscere l'angolo col quale il raggio di luce tocca il frammento è importante.



Se il raggio di luce è perpendicolare alla superficie, la luce ha l'impatto maggiore. Per misurare l'angolo tra il raggio di luce e il frammento si usa un vettore detto **normale** che è un vettore perpendicolare alla superficie del frammento. Più il Θ è grande (larghezza angolo), minore è l'impatto che la luce dovrebbe avere sul colore del frammento. Il raggio di luce diretto (**lighDir**) è un vettore di direzione risultante dalla differenza tra la posizione della luce e la posizione del frammento. Per calcolare questo raggio di luce abbiamo bisogno del vettore di posizione della

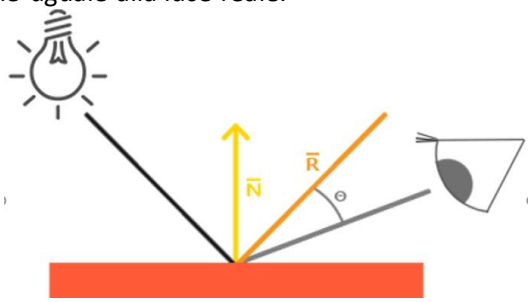
luce e del vettore di posizione del frammento.

Calculating the diffuse color

La prima cosa che dobbiamo calcolare è il vettore di direzione tra la sorgente di luce e la posizione del frammento. Come detto prima il vettore di direzione della luce è il vettore di differenza tra il vettore di posizione della luce e il vettore di posizione del frammento.
Successivamente, vogliamo calcolare l'effettivo impatto di uso che la luce ha sul frammento corrente, prendendo il **dot product** (tipologia di moltiplicazione tra vettori) tra il vettore normale e il vettore lightDir. Il valore risultante viene quindi moltiplicato con il colore della luce per ottenere la componente di diffusa. Maggiore è l'angolo tra entrambi i vettori e più scuro sarà il colore.

Specular Lighting

Proprio come l'illuminazione diffusa, l'illuminazione speculare si basa sul vettore di direzione della luce e sui vettori normali dell'oggetto, ma si basa anche sulla direzione della visuale, ad esempio la direzione con la quale il giocatore guarda il frammento. L'illuminazione speculare si basa sulle proprietà della luce di riflettersi sugli oggetti. Se pensiamo alla superficie dell'oggetto come a uno specchio, l'illuminazione speculare è la più forte poiché la luce riflessa sulla superficie sarà pressoché uguale alla luce reale.



Si calcola un vettore di riflessione riflettendo la direzione della luce attorno al vettore normale. In seguito, si calcola la distanza angolare tra questo vettore di riflessione e la direzione della visuale. Più piccolo è l'angolo tra di loro, maggiore è l'impatto della luce speculare. L'effetto risultante è che vediamo un po' di luce quando guardiamo la direzione della luce riflessa attraverso l'oggetto.

Gli sviluppatori erano soliti implementare il modello di illuminazione di Phong nel vertex shader. Il vantaggio è che è molto più efficiente poiché in genere ci sono molti meno vertici rispetto ai frammenti, quindi i calcoli (costosi) dell'illuminazione vengono eseguiti meno frequentemente. Tuttavia, il valore di colore risultante nel vertex shader è il colore di illuminazione risultante di quel solo vertice e i valori di colore dei frammenti circostanti sono quindi il risultato di colori di illuminazione interpolati. Il risultato fu che l'illuminazione non era molto realistica a meno che non venissero utilizzati grandi quantità di vertici. Quando il modello di illuminazione di Phong è implementato nel vertex shader, viene chiamato **Gouraud shading** invece di **Phong shading**.

Materials

Nel mondo reale, ogni oggetto ha una diversa reazione alla luce ed inoltre, ogni oggetto risponde in modo diverso alle luci speculari. Alcuni oggetti riflettono la luce senza troppe dispersioni con conseguenti piccole luci speculari e altri si disperdono molto dando alla luce un raggio più ampio. Per simulare questi effetti in OpenGL bisogna specificare i tipi di **materiali**.
Quando descriviamo gli oggetti, possiamo definire un colore per ciascuno dei 3 componenti di illuminazione: ambientale, diffusa e speculare. Specificando un colore per ciascuno dei componenti, abbiamo un controllo a **fine-grained** sull'output del colore.
Nel fragment shader si crea una struttura per memorizzare le proprietà del materiale. Si definisce un vettore che descrive un colore per ciascuno dei componenti dell'illuminazione di Phong. Il vettore materiale *ambientale* definisce il colore che la superficie riflette durante l'illuminazione ambientale, che di solito è lo stesso del colore della superficie. Il vettore materiale *diffuso* definisce il colore della superficie in condizioni di illuminazione diffusa, che di solito è impostato sul colore desiderato per l'oggetto. Il vettore materiale *speculare* determina l'impatto cromatico che una luce speculare ha sull'oggetto. Infine, la *lucentezza* influisce sullo scattering/raggio dell'evidenziazione speculare.

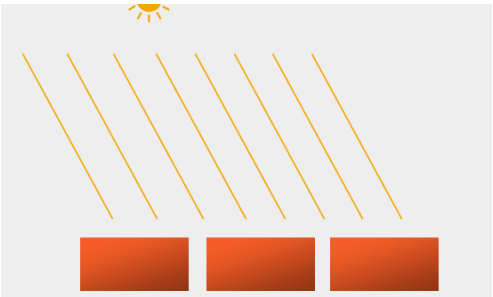
Lighting maps
Ogni oggetto ha la possibilità di avere un proprio materiale che reagisce in modo diverso alla luce. Questo è fondamentale per dare ad ogni oggetto un aspetto proprio in una scena illuminata, ma non offre ancora troppa flessibilità sull'output visivo dell'oggetto. Gli oggetti nel mondo reale di solito non sono costituiti da un singolo materiale, ma sono costituiti da diversi materiali. Per dare un aspetto più realistico al nostro mondo ci sono le *mappe diffuse e speculari*, le quali permettono di influenzare la diffusione e la componente speculare di un oggetto con molta più precisione.

Diffuse maps
In base alla posizione del frammento sull'oggetto è possibile recuperare un vettore di colore. Questo procedimento è molto simile a quello usato per le textures. La differenza è solo il nome che viene utilizzato per lo stesso principio di base: utilizzare un'immagine per wrappare un oggetto, il quale viene indicizzato per vettori di colori univoci per ogni frammento. Nelle scene illuminate questo è chiamato **diffuse maps** poiché un'immagine texture rappresenta tutti i colori diffusi dell'oggetto.

Specular maps
Per controllare il modo in cui ogni parte dell'oggetto riflette la luce speculare, ognuna con intensità diversa, si usano le **specular maps**. Ci troviamo di fronte a un discorso simile per le diffuse maps, si può usare anche una mappattura texture solo per le luci speculari. Il tutto si riduce nel generare una trama in bianco e nero che definisce l'intensità con la quale ogni parte d'oggetto si rapporta alla luce speculare.

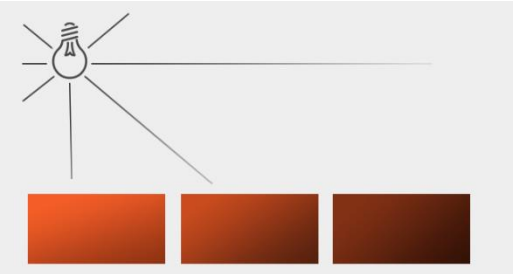
Light caster
Nella realtà noi non abbiamo una sola sorgente di luce che è un singolo punto nello spazio, bensì abbiamo diversi tipi di luce che agiscono tutte in modo differente. Una sorgente di luce *proietta* luce sugli oggetti è chiamata **light caster**.

Directional Light
Quando una sorgente di luce è lontana, i raggi di luce provenienti da essa sono vicini e paralleli fra di loro. Essi sembrano provenire tutti dalla stessa direzione, indipendentemente da dove l'oggetto e/o l'osservatore si trovi. Quando una sorgente luminosa è modellata per essere *infinitamente* lontana, è chiamata **directional light**, poichè tutti i suoi raggi hanno la stessa direzione ed è indipendente dalla posizione della sorgente di luce.



Point light

Le luci direzionali sono ottime per simulare le luci globali che illuminano l'intera scena ma al di là di essa noi vorremmo simulare anche diversi punti di luce sparpagliati all'interno della scena. Un punto di luce è una sorgente di luce con una determinata posizione all'interno di un mondo che illumina in tutte le direzioni finchè i raggi di luce non svaniscono a causa della distanza.



Attenuation

Il fenomeno che consente di ridurre l'intensità della luce, lungo la distanza sul quale lavora un raggio di luce, è generalmente chiamato **attenuazione**. Un modo per ridurre l'intensità in base alla distanza è quello di utilizzare semplicemente una equazione lineare la quale si occuperà di questo e farà risultare gli oggetti più distanti meno illuminati.

Tuttavia, qualche equazione lineare tende un pò a sbagliare. Nel mondo reale, le luci sono generalmente abbastanza luminose nelle vicinanze ma la brillantezza della sorgente di luce diminuisce velocemente all'inizio mentre successivamente l'intensità rimanente diminuisce molto più lentamente lungo la distanza. Perciò abbiamo bisogno di una formula differente per effettuare la riduzione. La formula

$$Fatt = \frac{1.0 * I}{K_c + K_l * d + K_q * d^2}$$

calcola un valore di attenuazione, basato sulla distanza del fragment dalla sorgente di luce, il quale sarà poi moltiplicato per il vettore di intensità della luce. Nella formula *d* rappresenta la distanza dal fragment alla sorgente di luce. Quindi per calcolare l'attenuazione vengono definiti 3 termini (configurabili): un termine costante K_c, un termine lineare K_l e un termine quadratico K_q.

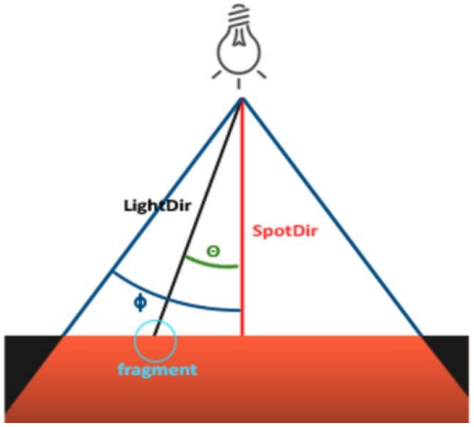
Il termine costante è solitamente 1.0 che permette di mantenere il risultante denominatore sempre più grande di 1 poiché altrimenti incrementerebbe l'intensità arrivati ad una determinata distanza e questo non è l'effetto che stiamo cercando.

- Il termine lineare è moltiplicato per il valore della distanza che riduce l'intensità in modo lineare;
- Il termine quadratico è moltiplicato con il quadrato della distanza e porta ad un decremento quadratico di intensità per la sorgente di luce;
- Il termine quadratico sarà meno significante comparato al termine lineare quando la distanza è piccola ma ha un'importanza molto più significativa del termine lineare quando la distanza aumenta.

Spotlight

Spotlight è una fonte di luce presente all'interno dell'ambiente che invece di spargere i raggi di luce in tutte le direzioni, li dirige in una sola direzione specifica. Il risultato è che solo gli oggetti all'interno del raggio formato dalla direzione vengono illuminati, mentre gli altri rimangono oscurati. Per capire perfettamente il funzionamento di una spotlight basta pensare ad un lampione o una torcia elettrica.

Uno spotlight in OpenGL è rappresentato da una posizione nel world space, una direzione e un angolo di taglio che



specifica il raggio dello spotlight. Per ogni frammento si calcola se il frammento si trova tra la direzione di taglio dello spotlight e, in tal caso, si illumina il frammento di conseguenza.

- **LightDir**: il vettore che punta dal frammento alla fonte di luce.
- **SpotDir**: la direzione verso cui lo spotlight punta.
- **Phi Φ**: l'angolo di taglio che specifica il raggio dello spotlight. Tutto al di fuori di quest'angolo non è illuminato dallo spotlight.
- **Theta Θ**: l'angolo tra il vettore LightDir e il vettore SpotDir. Il valore di Θ deve essere inferiore al valore di Φ per essere all'interno dello spotlight.

Quindi, ciò che fondamentalmente dobbiamo fare, è calcolare il dot product (il coseno dell'angolo tra due vettori unitari) tra il vettore LightDir e il vettore SpotDir e confrontarlo con l'angolo di **cutoff**.

Flashlight

Una fashlight è una spotlight situato nella posizione dello spettatore, di solito puntato dritto dalla prospettiva del giocatore. Fondamentalmente, una fashlight è uno spotlight normale, ma con la sua posizione e direzione continuamente aggiornate in base alla posizione e all'orientamento del giocatore.

Smooth/Soft edges

Per creare l'effetto di uno spotlight senza spigoli si simula uno spotlight con un cono interno e uno esterno. Si imposta il cono interno come il cono definito nella sezione precedente, ma si imposta anche un valore che offusca gradualmente la luce dall'interno ai bordi del cono esterno.
Per creare il cono esterno, si definisce semplicemente un altro valore del coseno che rappresenta l'angolo tra il vettore di direzione dello spotlight e il vettore del cono esterno (uguale al suo raggio). Ad esempio, se un frammento si trovasse tra il cono interno e il cono esterno, dovrebbe calcolare un valore di intensità compreso tra 0,0 e 1,0. Se il frammento si trova all'interno del cono interno, la sua intensità è uguale a 1,0 e 0,0 se il frammento si trova all'esterno del cono esterno. Possiamo calcolare tale valore usando la seguente formula:

$$I = \frac{\Theta + \gamma}{s}$$

dove *s* è il coseno della differenza tra il cono interno (Φ) e il cono esterno (Γ), dunque E = Φ − γ. Il risultato I è l'intensità dello spotlight sul frammento.

Multiple Lights

Per utilizzare più di una fonte di luce nella scena si incapsulano i calcoli di illuminazione in funzioni GLSL. Il motivo è che il codice diventa rapidamente fastidioso quando vogliamo eseguire calcoli di illuminazione con più luci, con ogni tipo di luce che richiede calcoli diversi. Quando si usano più luci in una scena, l'approccio è solitamente il seguente: abbiamo un singolo vettore di colore che rappresenta il colore di uscita del frammento. Per ogni luce, il colore di contributo risultante viene aggiunto al vettore del colore di uscita del frammento. Quindi ogni luce nella scena calcolerà il suo impatto individuale sul frammento e contribuirà al colore finale nell'output. Se per esempio due sorgenti di luce sono vicine al frammento, il loro contributo combinato darebbe luogo a un frammento più illuminato del frammento che verrebbe illuminato da un'unica fonte di luce.