OpenGL: Disegnare in 3D

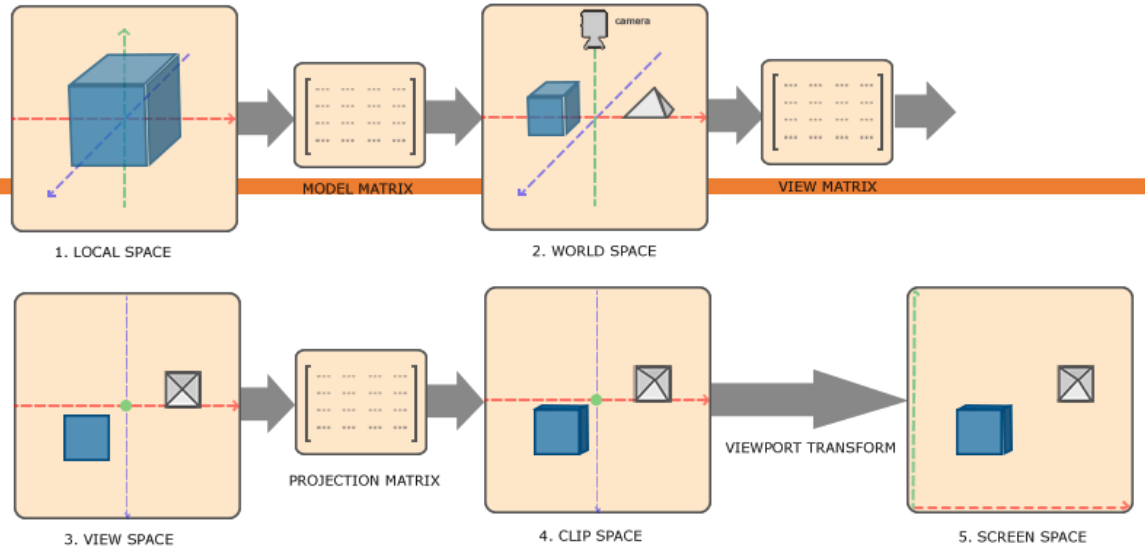
Riassumiamo le trasformazioni che subisce un vertice nel sottosistema geometrico prima di passare al sottosistema raster:

- OpenGL si aspetta che tutti i vertici, che vogliamo siano visibili, **dopo l'esecuzione del Vertex Shader**, siano **in coordinate del dispositivo normalizzate (NDC)** cioè, **le coordinate `x` , `y` `z` di ciascun vertice dovrebbero essere comprese tra `-1.0` e `1.0`; le coordinate al di fuori di questo intervallo non saranno visibili.**
- Quello che facciamo di solito è specificare le coordinate in un **sistema di riferimento dell'oggetto** e nel **vertex shader trasformiamo queste coordinate in coordinate del dispositivo normalizzate (NDC).**
- Queste NDC vengono quindi passate al **rasterizzatore per trasformarle in coordinate / pixel 2D sullo schermo**.
- La trasformazione delle coordinate in NDC viene di solito eseguita in modo graduale: i vertici di un oggetto vengono trasformati in diversi sistemi di coordinate prima di trasformarli infine in NDC.
- Il vantaggio di trasformarli in diversi sistemi di coordinate *intermedie* è che alcune operazioni / calcoli sono più facili in alcuni sistemi di coordinate.

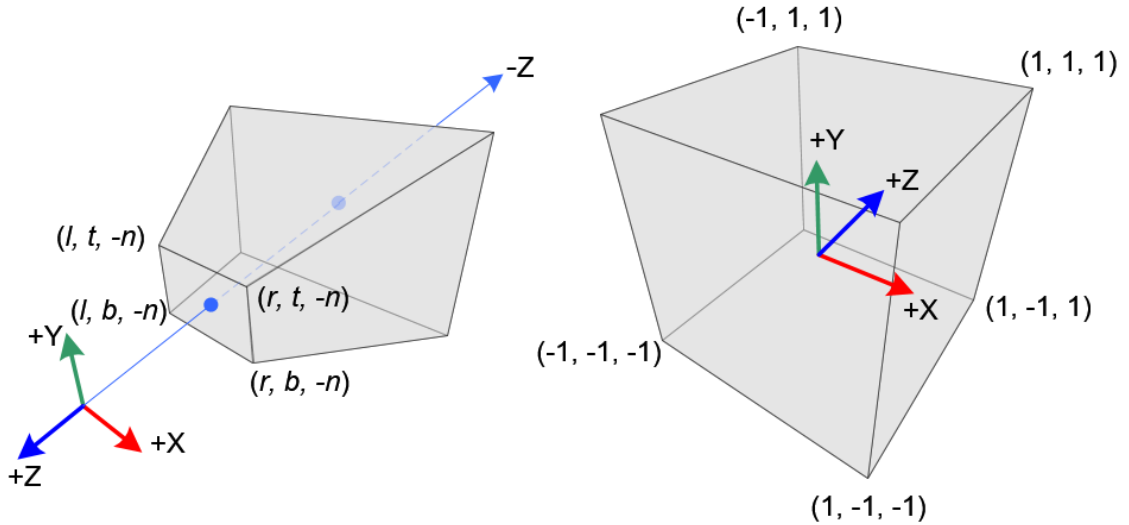
- 5 diversi sistemi di coordinate:

- Spazio locale (o spazio oggetti)
- World space - Spazio del Mondo
- View space (or Eye space) Spazio della vista
- Clip space - Spazio clip
- Screen space - Spazio sullo schermo
- Ogni vertice prima di diventare un frammento subisce queste 5 trasformazioni di coordinate.

  L'immagine seguente mostra il processo e mostra cosa fa ogni trasformazione:



- Le coordinate locali sono le coordinate dell'oggetto relative alla sua origine locale.
- Il prossimo passo è trasformare le coordinate locali in coordinate del MONDO
- **Successivamente trasformiamo le coordinate del mondo in coordinate dello spazio della Vista, in modo tale che ciascuna coordinata sia vista dalla telecamera o dal punto di vista dell'osservatore.**
- Dopo che le coordinate sono nello spazio della Vista (VIew-Space), **vogliamo proiettarle sulle coordinate di clip (Volume Canonico Normalizzato**: cubo con centro l'origine (0,0,0) e coordinate x,y,z che variano tra -1.0 e 1.0.
- E infine trasformiamo le **coordinate di clip in coordinate dello schermo in un processo che chiamiamo** viewport transform che trasforma **le coordinate da -1.0 e 1.0 nell'intervallo di coordinate definito da glViewport**. Le coordinate risultanti vengono quindi inviate al rasterizzatore dove vengono trasformate in frammenti.


-Z
(l, t, -n)
(l, b, -n)
(r, t, -n)
(r, b, -n)
+Y
+X
+Z
(-1, 1, 1)
(1, 1, 1)
+Y
+Z
+X
(1, -1, 1)
(-1, -1, -1)
(1, -1, -1)

Una volta che tutti i vertici vengono trasformati nello spazio di clip, viene effettuata un'operazione finale di divisione prospettica che consiste nel dividere le componenti x , y e z dei vettori di posizione per la componente w del vettore; la divisione in prospettiva è ciò che trasforma le coordinate 4D dello spazio di clip in coordinate del dispositivo 3D normalizzate. Questo passaggio viene eseguito automaticamente alla fine del vertex shader.

È dopo questa fase che le coordinate risultanti vengono mappate alle coordinate dello schermo (usando le impostazioni di glViewport) e trasformate in frammenti.

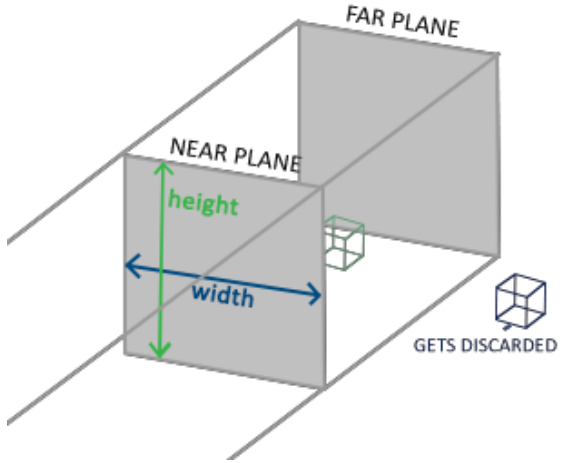
La matrice di proiezione per trasformare le coordinate di vista in coordinate di clip di solito assume due forme diverse, in cui ciascuna forma definisce il proprio frustum unico. Possiamo creare una matrice di proiezione ortografica o prospettica .

Proiezione ortogonale

Una matrice di proiezione ortografica definisce un volume di vista a forma di parallelepipedo che delimita lo spazio di clip, rispetto a cui viene tagliato ogni vertice ad esso esterno.

Quando creiamo una matrice di proiezione ortografica, specifichiamo la larghezza, l'altezza e la lunghezza del volume visibile.

Tutte le coordinate all'interno di questo volume verranno mappate all'interno del sistema NDC.



Il Volume di Vista definisce le coordinate visibili ed è specificato da una larghezza, un'altezza e un piano vicino ed un piano lontano.

Qualsiasi coordinata davanti al piano vicino viene ritagliata e lo stesso vale per le coordinate dietro il piano lontano.

Il frustum ortografico mappa direttamente tutte le coordinate all'interno del frustum su coordinate del dispositivo normalizzate senza effetti collaterali speciali poiché non toccherà la componente w del vettore trasformato;

se il componente w rimane uguale alla divisione in prospettiva 1.0 non cambierà le coordinate.

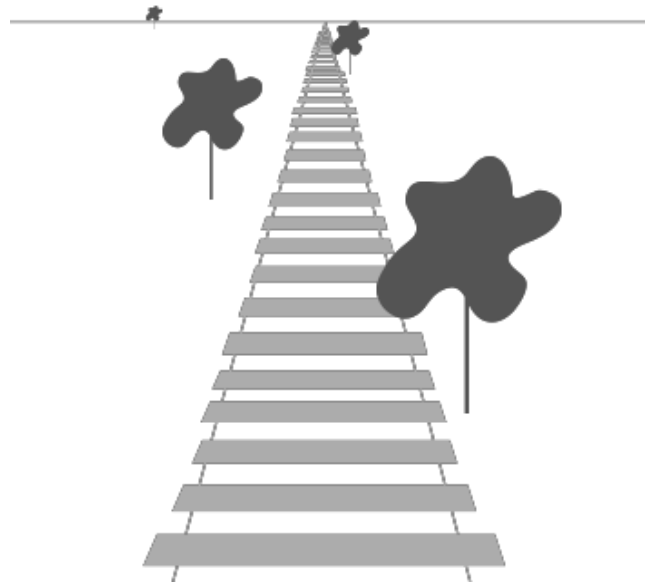
Per creare una matrice di proiezione ortografica utilizziamo la funzione di GLM ortho

ortho (0.0f, 800.0f, 0.0f, 600.0f, 0.1f, 100.0f);

I primi due parametri specificano la coordinata sinistra e destra del frustum
il terzo e il quarto parametro specificano la parte inferiore e superiore del frustum.
Con quei 4 punti abbiamo definito la dimensione dei piani vicino e lontano e con il 5 ° e 6 ° parametro definiamo le distanza tra il piano vicino e lontano.

Questa matrice di proiezione trasforma tutte le coordinate tra questi valori di intervallo x , y e z in coordinate del dispositivo normalizzate.

Proiezione prospettica



Nella vita reale gli oggetti più lontani appaiono molto più piccoli

A causa della prospettiva le linee sembrano coincidere a una distanza abbastanza lontana. Questo è esattamente l'effetto che la proiezione prospettica cerca di imitare e lo fa usando a matrice di proiezione prospettica .

La matrice di proiezione mappa una determinata gamma di frustum nello spazio di clip, ma manipola anche il valore w di ciascuna coordinata del vertice in modo tale che più lontano una coordinata del vertice si trova dallo spettatore, maggiore diventa questa componente w .

Una volta che le coordinate vengono trasformate nello spazio di clip, si trovano nell'intervallo da - w a w (tutto ciò che è al di fuori di questo intervallo viene troncato.

OpenGL richiede che, come output del vertex shader, le coordinate visibili rientrino nell'intervallo `-1.0` e `1.0`, quindi una volta che le coordinate sono nello spazio di clip, la divisione della prospettiva viene applicata alle coordinate dello spazio clip

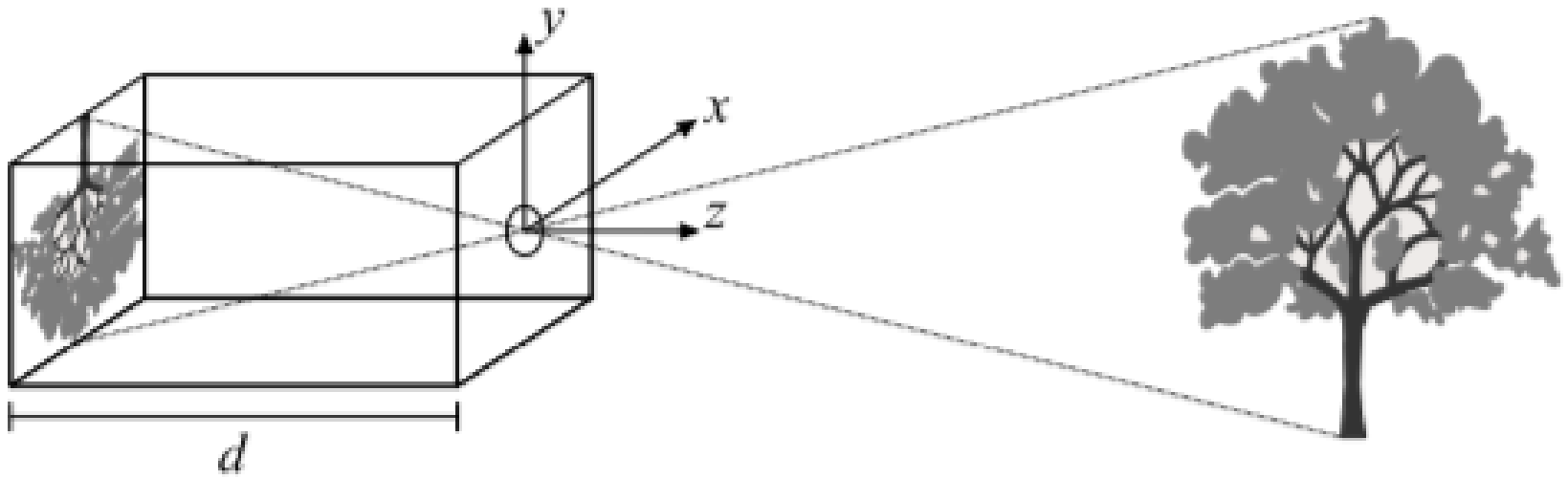
$$out = \begin{pmatrix} x/w \\ y/w \\ z/w) \end{pmatrix}$$

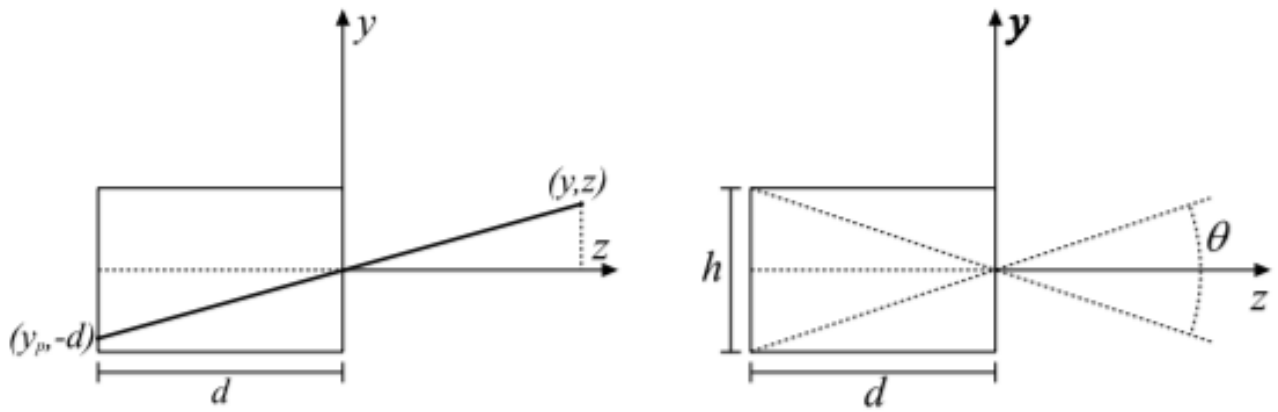
Ogni componente della coordinata del vertice viene divisa per la sua w dando coordinate del vertice più piccole quanto più un vertice è lontano dal visualizzatore.

Questo è un altro motivo per cui la componente w è importante, poiché ci aiuta con la proiezione prospettica.

Synthetic camera

La metafora più utilizzata per la descrizione delle relazioni scena-osservatore è certamente quella della macchina fotografica virtuale (**synthetic camera**).
Semplicemente possiamo immaginare di posizionare ed orientare liberamente una macchina fotografica all'interno della scena e scattare istantanee a nostro piacimento.
L'apparecchio fotografico oggetto della metafora è molto semplice ed è costituito da un **parallelepipedo in cui la faccia anteriore presenta un foro di diametro infinitesimo** (da qui il nome inglese pinhole camera dello strumento) e sulla faccia posteriore, coincidente con la pellicola fotografica, si formano le immagini.





E’ facile ricostruire la relazione geometrica che lega i punti della scena ai punti corrispondenti sull’immagine; il generico punto P = (x, y, z) della scena avrà coordinate $P_p = (x_p, y_p, -d)$ sul piano immagine, dove

$$x_p = -\frac{x}{z/d} \quad y_p = -\frac{y}{z/d},$$

e d indica la distanza focale.

Nonostante l'estrema semplicità, la pinhole camera presenta caratteristiche ottime:
immagini sempre nitide ed a fuoco in quanto per ogni punto del soggetto soltanto un raggio luminoso (proiettore) riesce ad attraversare il foro (centro di proiezione) e colpire il piano della pellicola.

La profondità di campo è illimitata;

l'angolo θ di vista del dispositivo può essere agevolmente modificato variando il rapporto d tra la lunghezza della scatola (distanza focale) e le dimensioni del piano immagine.

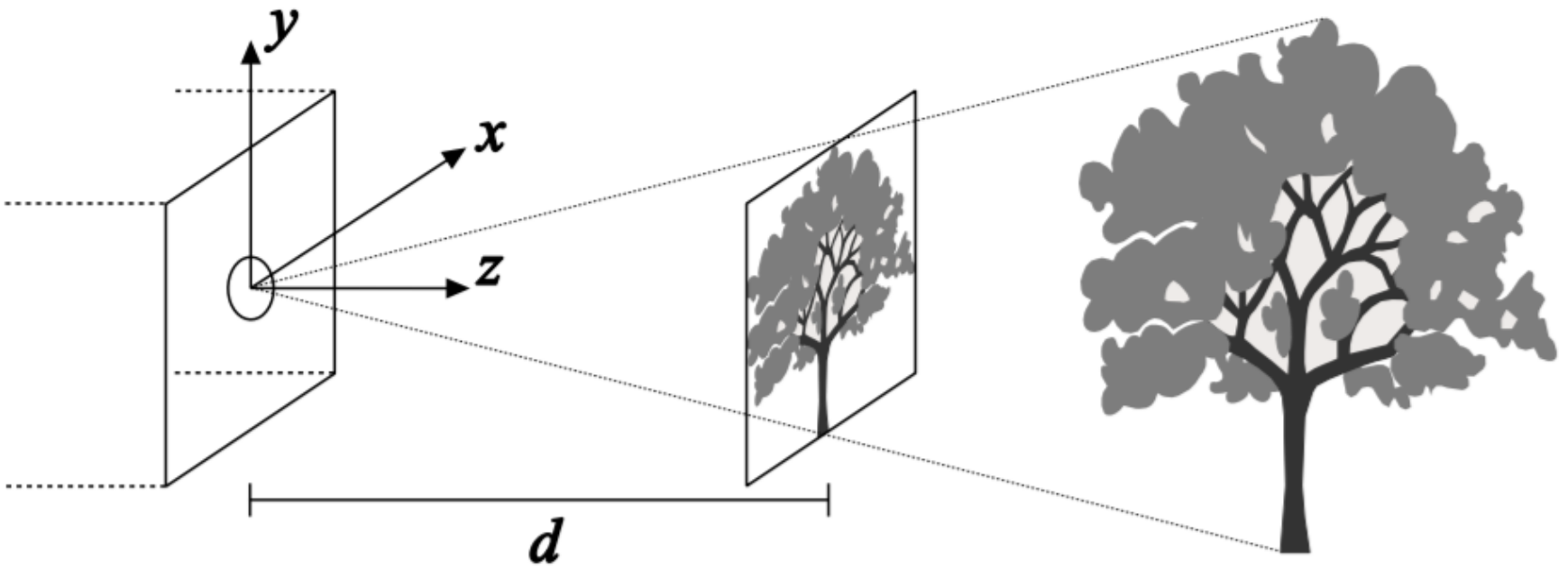
Se con h indichiamo la misura della diagonale del piano di proiezione avremo infatti che

$$\theta = 2 \arctan(h/2d).$$

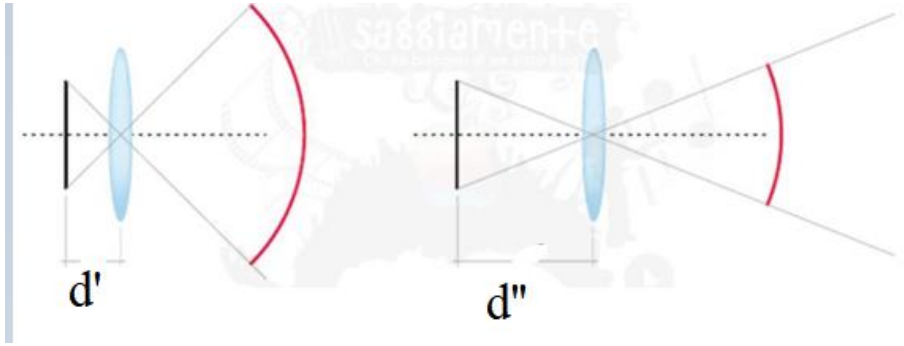
Le fotografie prodotte dalla pinhole camera sono capovolte. Ciò accade perché, il piano di proiezione si trova dietro il centro di proiezione, tuttavia questo può essere evitato se il piano di proiezione si trova sullo stesso lato della scena. Nel mondo reale , il piano dell'immagine non può ovviamente essere posizionato davanti all'apertura, ma nel mondo virtuale dei computer, costruire la nostra fotocamera in questo modo non è affatto un problema.

Concettualmente, per costruzione, questo ha portato a vedere il foro della fotocamera (che è anche il centro di proiezione) come la posizione effettiva dell'occhio e il piano dell'immagine, l'immagine che l'occhio sta guardando.

Definire la nostra macchina fotografica virtuale in questo modo, ci mostra più chiaramente come costruire un'immagine seguendo i raggi di luce da qualsiasi punto della scena da cui vengono emessi, fino a raggiungereall'occhio, risulti essere un semplice problema geometrico a cui è stato dato il nome di proiezione prospettica. La proiezione prospettica è un metodo per costruire un'immagine attraverso questo apparato che è una sorta di piramide il cui vertice è allineato con l'occhio, e la cui base definisce la superficie di un piano su cui viene "proiettata" l'immagine della scena 3D.



Definiamo distanza focale d, la distanza tra l’occhio della telecamera ed il piano di vista.

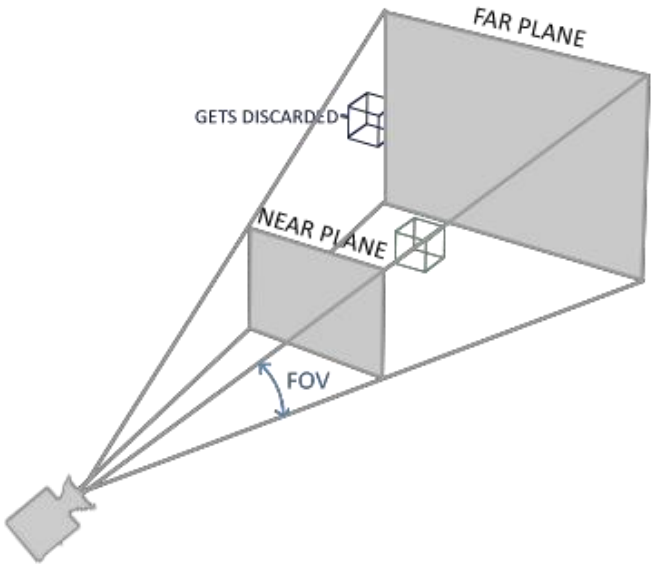


Ad una distanza focale d’ minore, corrisponde un campo visivo maggiore, mentre valori più grandi di distanza focale d’’ permettono di restringere il campo, ottenendo l'effetto di avvicinare i soggetti.

Una matrice di proiezione prospettica può essere creata in GLM con la funzione perspective come segue:

```
mat4 proj =perspective(radians(45.0f),(float)width/(float)height,0.1f, 100.0f);
```

`perspective` crea un Volume di Vista che è un tronco di piramide.



Il suo primo parametro definisce il valore **fov**, che sta per campo visivo e imposta la dimensione dello spazio di visualizzazione. Per una visione realistica di solito è impostato a 45 gradi.

Restringere il campo visivo, equivale adavvicinare la scena (quindi a farne uno zoom), allargare il campo visivo, equivale ad allontanare la scena (a vederla nel suo insieme)

Il secondo parametro imposta **le proporzioni** che vengono calcolate dividendo la larghezza della finestra per la sua altezza.

Il **terzo e il quarto parametro impostano il piano *vicino* e *lontano* del frustum**.

I piani di clipping near (vicino) e far (lontano) sono piani virtuali situati davanti alla telecamera e paralleli al piano dell'immagine (il piano in cui è contenuta l'immagine).
La posizione di ogni piano di ritaglio viene misurata lungo la linea di vista della telecamera (l'asse z locale della telecamera).
Sono utilizzati nella maggior parte dei modelli di fotocamere virtuali e non hanno equivalenti nel mondo reale.
Gli oggetti più vicini del piano di ritaglio vicino o più lontani del piano di ritaglio lontano sono invisibili alla telecamera.

Normalmente impostiamo la distanza vicina a `0.1` e la distanza lontana a `100.0`. Verranno visualizzati tutti i vertici tra il piano vicino e lontano all'interno del frustum.

Esempio: Modifcare il FOV interagendo con la rotellina centrale del mouse

Nel main: `glutMouseWheelFunc(mouseWheelCallback);`

```
void mouseWheelCallback(int wheel, int direction, int x, int y) {
if (direction > 0) {
SetupProspettiva.fov -= 2.0f; //  Restringere il campo visivo, equivale ad avvicinare la scena (quindi a farne uno zoom),
}
else {
SetupProspettiva.fov += 2.0f; //  Aumentare il campo visivo, equivale ad allontanare la scena (vederla nel suo insieme),
}

// Limita il FOV per evitare valori non desiderati
if (SetupProspettiva.fov < 1.0f) {
SetupProspettiva.fov = 1.0f;
}
if (SetupProspettiva.fov > 180.0f) {
SetupProspettiva.fov = 180.0f;
}
// Richiede il ridisegno della scena
glutPostRedisplay();
}
```

Creiamo una matrice di trasformazione per ciascuna delle fasi sopra menzionate: modello, matrice di visualizzazione e proiezione

$$V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local}$$

Una coordinata di vertice viene quindi trasformata in coordinate di clip.
L'ordine di moltiplicazione della matrice è invertito (ricorda che dobbiamo leggere la moltiplicazione della matrice da destra a sinistra). Il vertice risultante dovrebbe quindi essere assegnato a `gl_Position` nello shader di vertici e OpenGL eseguirà automaticamente la divisione prospettica ed il clipping.

L'output del vertex shader richiede che le coordinate siano nello spazio clip, che è quello che abbiamo appena fatto con le matrici di trasformazione.
OpenGL quindi esegue la *divisione prospettica* sulle *coordinate dello spazio clip* per trasformarle in *coordinate del dispositivo normalizzato* .
OpenGL utilizza quindi i parametri da glViewport per mappare le coordinate del dispositivo normalizzato alle coordinate *dello schermo in* cui ciascuna coordinata corrisponde a un punto sullo schermo (ad esempio una finestra sullo schermo 800x600). Questo processo è chiamato *trasformazione* del *viewport* .

Per iniziare a disegnare in 3D, creeremo prima una matrice di tipo Model che trasforma le coordinate dei vertici, espresse nel sistema di riferimento dell'oggetto, nel sistema di riferimento del mondo . La matrice Model è composta da taslazioni, scalature e / o rotazioni.

Moltiplicando le coordinate del vertice con questa matrice del modello stiamo trasformando le coordinate del vertice in coordinate del mondo.

Se non si specifica alcuna matrice di Vista, e quindi la matrice di vista è l'identità, il sistema di riferimento della camera coincide con il sistema di riferimento del mondo. La scena viene quindi rappresentata dal suo interno.

Vogliamo spostarci leggermente indietro nella scena in modo che l'oggetto diventi visibile.

Si può o spostare la telecamera ... o spostare la scena. Sebbene non sia pratico nella vita reale, questo è davvero semplice e utile in Computer Graphics.
Spostare una telecamera all'indietro equivale a spostare l'intera scena in avanti.

Traslare l'intera scena verso l'asse negativo, equivale a spostare la camera lungo l'asse positivo, ad allontanare la telecamera dalla scena, per permetterci di vederla dall'esterno.

```
#version 430 core
layout (location = 0) in vec3 aPos;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
gl_Position = projection * view * model * vec4(aPos, 1.0);   }
```

Esercizio: Disegnare un cubo 3D.
Fare ruotare intorno all'asse y il cubo di un angolo , aggiornato mediante una funzione update chiamata da una glutTimerFunc :

```
model = rotate (model, radians (angolo), vec3(0.0f, 1.0f, 0.0f));
```

segue il cubo usando glDrawArrays di 36 vertici.

```
glDrawArrays (GL_TRIANGLES, 0, 36)
```

Per effettuare il disegno corretto, in termini di visibilità di ognuno dei triangoli disegnati per renderizzare ogni faccia del cubi, bisogna configurare OpenGL per eseguire il test di profondità.

Z-buffer

OpenGL memorizza tutte le informazioni di profondità in un buffer z, noto anche come buffer di profondità

FreeGlut crea automaticamente tale buffer (proprio come un buffer di colore che memorizza i colori dell'immagine di output).

La profondità è memorizzata all'interno di ogni frammento (come valore `z` del frammento) e ogni volta che il frammento vuole produrre il suo colore, OpenGL confronta i suoi valori di profondità con il buffer z.

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.

Tuttavia, se vogliamo assicurarci che OpenGL esegua effettivamente i test di profondità, è necessario abilitare i test di profondità; è disabilitato di default. Possiamo abilitare i test di profondità utilizzando glEnable . Tale funzionalità viene quindi abilitata / disabilitata fino a quando non viene effettuata un'altra chiamata per disabilitarla / abilitarla

- Per abilitare il test di profondità bisogna abilitare GL_DEPTH_TEST :

```
glEnable (GL_DEPTH_TEST);
```

- Dato che stiamo usando un buffer di profondità, vogliamo anche cancellare il buffer di profondità prima di ogni iterazione di rendering (altrimenti le informazioni sulla profondità del frame precedente rimangono nel buffer).

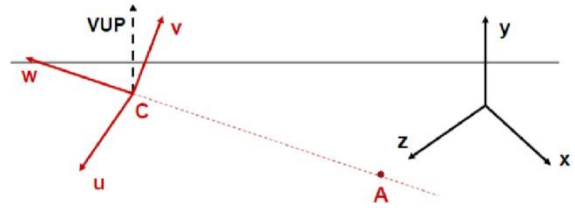
- Proprio come cancellare il buffer di colore, possiamo cancellare il buffer di profondità specificando DEPTH_BUFFER_BIT nella funzione glClear :

```
glClear (GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT)
```

Impostare una telecamera in OpenGL

Camera / View-Space

La matrice di vista trasforma tutte le coordinate del mondo in coordinate di vista, relative alla posizione e direzione della telecamera. Per definire una telecamera abbiamo bisogno di:



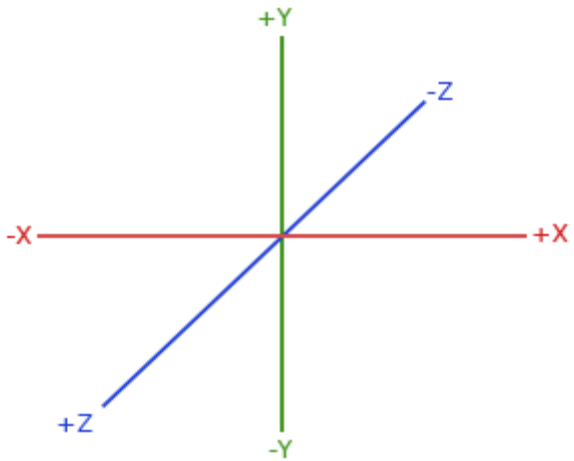
- la sua posizione in coordinate del Mondo
- la direzione in cui sta guardando,
- un vettore che punta a destra e un vettore che punta verso l'alto dalla fotocamera. Stiamo creando un sistema di coordinate con 3 assi unitari perpendicolari , aventi come origine la posizione della telecamera.

1. Posizione della telecamera

La posizione della telecamera è un vettore nello spazio del mondo che punta alla posizione della telecamera.

```
vec3 C = vec3(Cx, Cy,Cz);
```

Se vogliamo che la telecamera si sposti all'indietro, ci muoviamo lungo l'asse z positivo.


+Y
-Z
-X
+X
+Z
-Y

2. Direzione della telecamera

Direzione della telecamera: in quale direzione sta puntando?

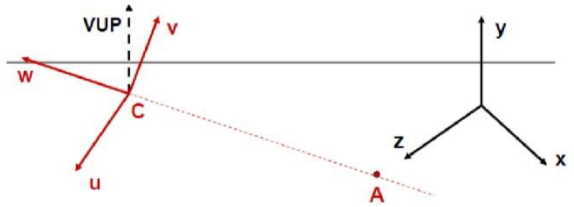
Supponiamo che punti verso il punto $A \equiv (Ax, Ay, Az)$

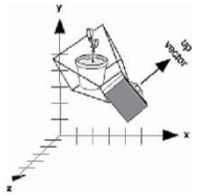
Nello spazio affine, la differenza tra 2 punti ci dà un vettore.

Sottraendo le coordinate della posizione della telecamera dalle coordinate del punto in cui guarda la telecamera si ottiene quindi il vettore di direzione desiderato.

Nel sistema di riferimento della telecamera , l'asse z è positivo verso l'interno, e perché per convenzione (in OpenGL) la telecamera punta verso l'asse z negativo, è necessario cambiare di segno il vettore di direzione. Basta cambiare l'ordine di sottrazione tra i due punti, e poi normalizzare il vettore ottenuto (per costruire il versore)

```
vec3 A = vec3(Ax, Ay, Az,1);
vec3 w = normalize(C- A);
```





Specifichiamo un vettore VUP che punta verso l' alto (in coordinate del mondo).

3. Asse u

Per ottenere **l'asse u** che punta a destra della telecamera, facciamo un **prodotto vettoriale tra il vettore VUP e il vettore di direzione, w**. Poiché il risultato di un prodotto vettoriale è un vettore perpendicolare a entrambi i vettori, otterremo un vettore che punta nella direzione dell'asse **u**

positivo (se si cambia l'ordine del prodotto vettoriale si ottiene un vettore che punta verso l'asse u negativo):

```
vec3 up = vec3(0.0f, 1.0f, 0.0f);
vec3 u = normalize(cross (VUP, w));
```

4. Asse v

Ora che abbiamo sia il vettore dell'asse u che il vettore dell'asse w, possiamo ottenere il terzo asse v positivo del sistema di riferimento della fotocamera facendo il prodotto vettoriale del vettore del vettore, **w** e del vettore che **u** punta verso destra,

```
vec3 v = cross (w, u);
```

Utilizzando questi vettori possiamo creare una matrice del sistema di riferimento della camera.

Abbiamo visto a lezione che la matrice

$$M = \begin{bmatrix} u_x & v_x & w_x & C_x \\ u_y & v_y & w_y & C_y \\ u_z & v_z & w_z & C_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

fornisce la matrice cha trasforma le coordinate espresse nel sistema di riferimento della camera, il VCS, nel sistema di riferimento del mondo , il WCS. A noi interessa la matrice inversa di questa matrice.

La matrice M si può scrivere come:

$$M = \begin{bmatrix} 1 & 0 & 0 & C_x \\ 0 & 1 & 0 & C_y \\ 0 & 0 & 1 & C_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & w_x 0 \\ u_y & v_y & w_y 0 \\ u_z & v_z & w_z 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Per il teorema di Binet, l'inversa del prodotto di due matrici è uguale al prodotto delle loro inverse prese in ordine

inverso e poiché $\begin{bmatrix} u_x & v_x & w_x 0 \\ u_y & v_y & w_y 0 \\ u_z & v_z & w_z 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ è ortogonale e la sua inversa coincide con la sua trasposta, e l'inversa di

$\begin{bmatrix} 1 & 0 & 0 & C_x \\ 0 & 1 & 0 & C_y \\ 0 & 0 & 1 & C_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$ è uguale a $\begin{bmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Segue che la matrice di vista che applicata ai vertici in coordinate del mondo li trasforma in coordinate del sistemadi riferimento della telecamera è data da:

$$\boldsymbol{M^{-1}} = \begin{bmatrix} \boldsymbol{u_x} & \boldsymbol{u_y} & \boldsymbol{u_z 0} \\ \boldsymbol{v_x} & \boldsymbol{v_y} & \boldsymbol{v_z 0} \\ \boldsymbol{w_x} & \boldsymbol{w_y} & \boldsymbol{w_z 0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{-C_x} \\ \boldsymbol{0} & \boldsymbol{1} & \boldsymbol{0} & \boldsymbol{-C_y} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} & \boldsymbol{-C_z} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} \end{bmatrix}$$

La funzione `lookAt` della libreria `GLM` crea questa matrice:

```
mat4 view;
view  =lookAt  (vec3(0.0f, 0.0f, 3.0f),vec3(0.0f,0.0f,0.0f),
vec3(0.0f, 1.0f, 0.0f));
```

La funzione lookAt richiede rispettivamente una **posizione, un target e un vettore che punta verso l'alto della telecamera.**

```
typedef struct {
vec4 position;
vec4 target;
vec4 upVector;
vec4 direction;
} ViewSetup;

ViewSetup SetupTelecamera;
```

```
SetupTelecamera.position = glm::vec4(0.0, 0.5, 20.0, 1.0);
SetupTelecamera.target = glm::vec4(0.0, 0.0, 0.0, 1.0);
SetupTelecamera.direction = SetupTelecamera.target - SetupTelecamera.position;
SetupTelecamera.upVector = glm::vec4(0.0, 1.0, 0.0, 0.0);
```

Esercizio: Ruotare la videocamera attorno alla scena.

Mantenere il target in cui guarda la telecamera a `(0,0,0)`.

Mantenere fissa la quota y della posizione della telecamera.

Creare una coordinata `x` e `z` che per ogni frame dell'animazione rappresenta un punto su un cerchio di raggio radius: saranno la `x` e la `z` della posizione della telecamera.

```
const float radius = 50.0f;

SetupTelecamera.position.x = sin(glutGet(GLUT_ELAPSED_TIME)) * radius;

SetupTelecamera.position.z = cos(glutGet(GLUT_ELAPSED_TIME)) * radius;
```

Ricalcolando le coordinate `x e z` durante l'animazione, la telecamera ruota attorno alla scena.

Aggiornare la matrice Viewper ogni fotogramma:

```
View = lookAt(vec3(SetupTelecamera.position), vec3(SetupTelecamera.target),
vec3(SetupTelecamera.upVector));
```

Telecamera guidata dalla tastiera

Ogni volta che premiamo uno dei tasti `WASD` , la posizione della telecamera viene aggiornata di conseguenza.

W: per spostarsi avanti

S: per spostarsi indietro

A: per spostarsi a sinistra

D: per spostarsi a destra

float cameraSpeed=2.5 //velocità a cui si muove la telecamera

```
void moveCameraForward()
{

vec4 direction = SetupTelecamera.target - SetupTelecamera.position;    //Direzione lungo cui
si sposta la telecamera in coordinate del mondo
SetupTelecamera.position += direction * cameraSpeed;
}

void moveCameraBack()
{

vec4 direction = SetupTelecamera.target - SetupTelecamera.position; //Direzione lungo cui si
sposta la telecamera in coordinate del mondo
SetupTelecamera.position -= direction * cameraSpeed;
}
```

```
void moveCameraLeft()
{
vec3 direction = SetupTelecamera.target - SetupTelecamera.position; //Direzione lungo cui si
sposta la telecamera in coordinate del mondo
vec3 slide_vector = cross(direction, vec3(SetupTelecamera.upVector)) * cameraSpeed;
SetupTelecamera.position -= vec4(slide_vector, .0);
SetupTelecamera.target -= vec4(slide_vector, .0);
}

void moveCameraRight()
{

vec3 direction = SetupTelecamera.target - SetupTelecamera.position;  //Direzione lungo cui si
sposta la telecamera in coordinate del mondo
vec3 slide_vector = cross(direction, vec3(SetupTelecamera.upVector)) * cameraSpeed;
SetupTelecamera.position += vec4(slide_vector, .0);
SetupTelecamera.target += vec4(slide_vector, .0);
}
```

Velocità di movimento

Abbiamo usato un valore costante per la velocità di movimento della telecamera, `cameraSpeed`.

In pratica i computer hanno potenza di calcolo diversa, e il risultato è che alcuni utenti sono in grado di eseguire il rendering di molti più frame al secondo di altri. Alcuni utenti si muovono molto velocemente e altri molto lentamente a seconda della configurazione del loro hardware. Quando si esegue un'applicazione, sarebbe desiderabile che funzionasse allo stesso modo sui diversi tipi di hardware.

Le applicazioni grafiche e i giochi di solito tengono traccia di una variabile **deltaTime** che memorizza il tempo impiegato per il rendering dell'ultimo fotogramma.

Quindi l'idea è di moltiplicare la velocità per `deltaTime` . Il risultato è che **quando abbiamo un `deltaTime` elevato in un frame**, cioè l'ultimo frame ha richiesto più tempo della media, anche la velocità per quel frame sarà un po' più alta per bilanciare tutto. Quando si utilizza questo approccio, non importa se si dispone di un PC molto veloce o lento, la velocità della fotocamera verrà bilanciata di conseguenza in modo che ogni utente abbia la stessa esperienza.

Per calcolare il valore `deltaTime` teniamo traccia di 2 variabili globali:

```
float deltaTime = 0.0f; // Tempo tra il frame corrente ed il precedente
float lastFrame = 0.0f; // Tempo dell'ultimo frame
```

All'interno di ciascun frame calcoliamo quindi il nuovo valore `deltaTime` per un uso successivo:

```
float currentFrame = glutElapsedTime ();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;
```

Ora che abbiamo `deltaTime` possiamo tenerne conto nel calcolo delle velocità:

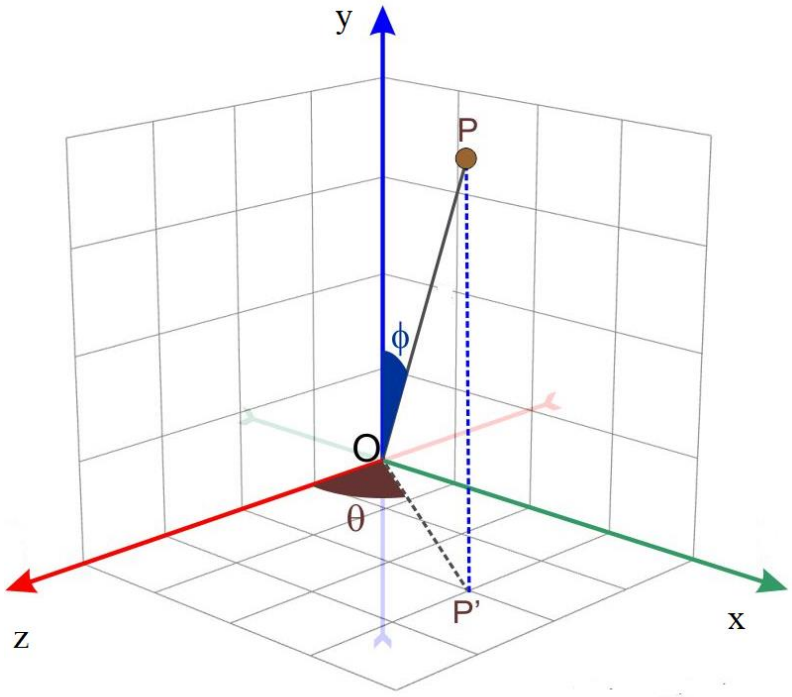
```
float cameraSpeed = 2.5f * deltaTime; ...
```

Dal momento che stiamo usando `deltaTime`, la telecamera ora si sposterà a una velocità costante di `2.5` unità al secondo

Trovare le componenti di una direzione della telecamera specificata con la posizione del mouse

ϕ : **yaw** rotazione attorno all'asse y ,

θ : **pitch** rotazione attorno all'asse x.



Le coordinate sferiche (r,ϕ,θ) di un punto P di coordinate cartesiane (x,y,z) si possono anche definire nel modo seguente.

Sia $r = x^2 + y^2 + z^2$ la distanza di P dall'origine O,

P' la proiezione di P sul piano xz

ϕ l'angolo formato tra semiretta OP uscente dall'origine e passante per P e l' asse y (co-latitudine)

θ l'angolo formato dal semiasse positivo delle z e la semiretta nel piano xz uscente dall'origine e passante per la proiezione P' di P su tale piano (longitudine)

$x = OP'\cos(\theta = r \sin(\phi)\cos(\theta)$

$y = PP' = r \cos(\phi)$

$z = OP' \sin(\theta) = r \sin(\phi)\sin(\theta)$,

$\theta \in [0, 2\pi]$

$\phi \in [-\pi/2, \pi/2]$

Siano xoffset e yoffset le variazioni nelle coordinate del cursore del mouse (movimento passivo del mouse sulla finestra) .

```
float xoffset = xpos - lastX;
float yoffset = ypos - lastY;
```

Si aggiornano gli angoli theta e Phi sommandogli queste variazione

```
Theta += xoffset;
Phi += yoffset;
```

```
Si ricavano le coordinate x,y,z del punto che sulla sfera è individuato
dagli angoli Theta e Phi
```

```
vec3 front;
front.x=cos(radians(Theta)) * cos(radians(Phi));
front.y = sin(radians(Phi));
front.z = sin(radians(Theta)) * cos(radians(Phi));
direction = vec4(normalize(front), 0.0);
```

```
SetupTelecamera.direction = vec4(normalize(front), 0.0); //Aggiorno la direzione della
telecamera
SetupTelecamera.target = SetupTelecamera.position + SetupTelecamera.direction; //aggiorno il
punto in cui guarda la telecamera
```