Copilota RCQF

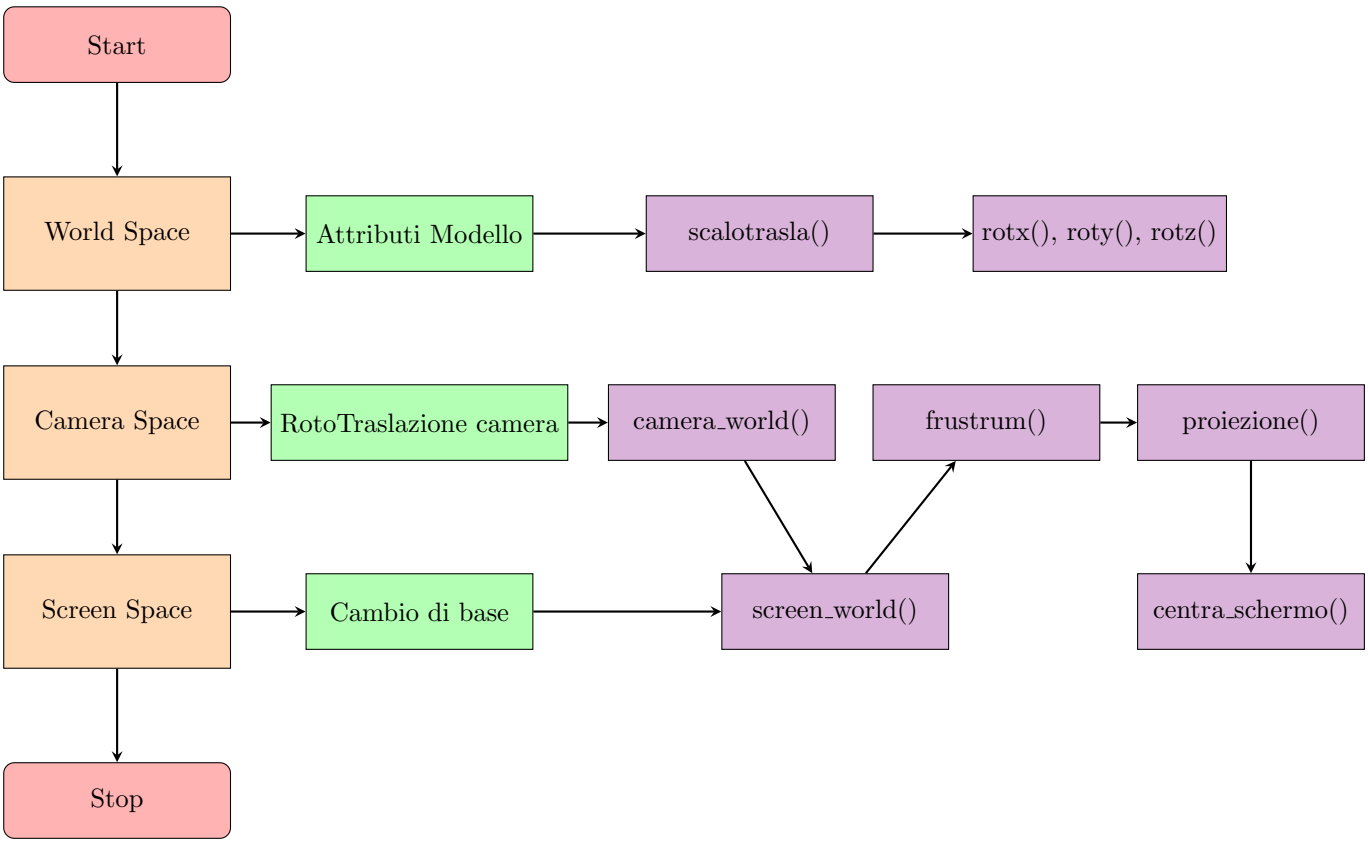
Alessio Cimma

4 luglio 2024

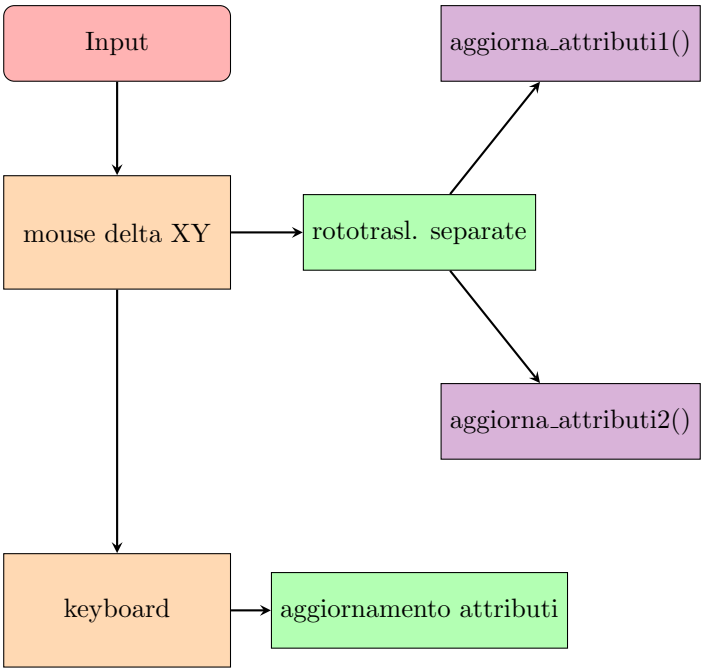
Indice

1 3D orientation

1.1 Pipeline di renderizzazione



1.2 Pipeline di aggiornamento



1.3 Teoria della pipeline usata

Modello: La geometria del modello solitamente è descritta e contenuta dentro ad un file `model.obj` c'entrata su un'origine a $XYZ = \{0, 0, 0\}$. Quando questo modello viene caricato, viene inserito in un'apposita classe che contiene anche attributi come posizione, scala e rotazione. Queste verranno applicate ad ogni refresh della pagina con il seguente ed immutabile ordine:

- Rotation X
- Rotation Y
- Rotation Z
- Scale and Traslation

Questo passaggio dev'essere il primo ad essere eseguito, in quanto posiziona tutti i modelli in uno spazio globale in relazione uno con l'altro.

Camera: Usando una camera e non un punto di vista fisso, dobbiamo spostarci dallo spazio globale a quello della camera, per farlo usiamo la relativa matrice che trasla e ruota la scena. (Immagino che in futuro aggiungerò la possibilità di avere più camere tra cui scegliere; durante la pipeline, basterà switchare la camera attiva). Una volta compiuto questo passaggio, nel caso ci trovassimo in modalità prospettiva, sarà necessario applicare `frustrum()` e `proiezione()`, che appunto generano l'effetto di proiezione dato un certo FOV.

Schermo: In questo programma noi usiamo un sistema di riferimento per cui:

- la destra locale è X - la destra sullo schermo è X
- l'alto locale è Z - l'alto sullo schermo è -Y
- il profondo locale è Y - il profondo sullo schermo è Z

Questo ci porta a dover cambiare la base per avere tutto orientato correttamente. Infine verrà applicata una scala per portarci da uno spazio normalizzato (-1, 1) ad uno spazio nella scala della dimensione dello schermo.

Aggiornamento degli attributi: Nel caso dei modelli, basterà aggiornare la nuova pos / rot / scala. Nel caso della camera il discorso si fa più complesso. Dobbiamo suddividere il discorso in due parti:

- Orientamento: aggiornato seguendo la sequenza di rotazioni di Eulero (XYZ)
- Posizione: la rotazione attorno al primo asse (Y) è facilmente applicabile, il problema compare quando si cerca di applicare la seconda rotazione (X). Infatti ora la dovremo applicare lungo la X locale, che è diversa dalla X globale. Non possiamo quindi applicare `rotx()`, ma `rot_ax()` attorno all'asse locale X

1.4 Codice delle varie funzioni

Listing 1: Matrici di Rotazione (3 con assi globali, 1 con asse custom)

```python
@staticmethod
def rotx(ang: float) -> np.ndarray[np.ndarray[float]]:
    return np.array([
        [1, 0, 0, 0],
        [0, np.cos(ang), np.sin(ang), 0],
        [0, -np.sin(ang), np.cos(ang), 0],
        [0, 0, 0, 1]
    ])

@staticmethod
def roty(ang: float) -> np.ndarray[np.ndarray[float]]:
    return np.array([
        [np.cos(ang), 0, np.sin(ang), 0],
        [0, 1, 0, 0],
        [-np.sin(ang), 0, np.cos(ang), 0],
        [0, 0, 0, 1]
    ])

@staticmethod
def rotz(ang: float) -> np.ndarray[np.ndarray[float]]:
    return np.array([
        [np.cos(ang), np.sin(ang), 0, 0],
        [-np.sin(ang), np.cos(ang), 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]
    ])

@staticmethod
def rot_ax(axis: np.ndarray[float], ang: float) -> np.ndarray[np.ndarray[float]]:
    K = np.array([
        [0, -axis[2], axis[1], 0],
        [axis[2], 0, -axis[0], 0],
        [-axis[1], axis[0], 0, 0],
        [0, 0, 0, 1]
    ])

    return np.eye(4) + np.sin(ang) * K + (1 - np.cos(ang)) * np.dot(K, K)
```

Listing 2: Matrice di ScaloTraslazione

```python
@staticmethod
def scalotrasla(obj):
    return np.array([
        [obj.sx, 0, 0, 0],
        [0, obj.sy, 0, 0],
        [0, 0, obj.sz, 0],
        [obj.x, obj.y, obj.z, 1]
    ])
```

Listing 3: Matrice di cambio sistema di riferimento (camera)

```
@staticmethod
def camera_world(camera: np.ndarray) -> np.ndarray[np.ndarray[float]]:
    return np.array(
        [[1, 0, 0, 0],
            [0, 1, 0, 0],
            [0, 0, 1, 0],
            [-camera.pos[0], -camera.pos[1], -camera.pos[2], 1]]
    ) @ np.array(
        [[camera.rig[0], camera.dir[0], camera.ups[0], 0],
            [camera.rig[1], camera.dir[1], camera.ups[1], 0],
            [camera.rig[2], camera.dir[2], camera.ups[2], 0],
            [0, 0, 0, 1]]
    )
```

Listing 4: Matrici di applicazione prospettiva

```
@staticmethod
def frustrum(W: int, H: int, h_fov: float = np.pi / 6) -> np.ndarray[np.ndarray[
    float]]:
    v_fov = h_fov * H / W
    ori = np.tan(h_fov / 2)
    ver = np.tan(v_fov / 2)
    return np.array([
        [1 / ori, 0, 0, 0],
        [0, 1 / ver, 0, 0],
        [0, 0, 1, 1],
        [0, 0, 0, 0]
    ])

@staticmethod
def proiezione(vertici: np.ndarray[np.ndarray[float]]) -> np.ndarray[np.ndarray[
    float]]:
    distanze = vertici[:, -1]
    indici_dist_neg = np.where(distanze < 0)

    ris = vertici / vertici[:, -1].reshape(-1, 1)
    ris[(ris < -12) | (ris > 12)] = 2
    ris[indici_dist_neg] = np.array([2, 2, 2, 1])

    return ris
```

Listing 5: Matrice di cambio sistema di riferimento (schermo)

```
    @staticmethod
    def screen_world() -> np.ndarray[np.ndarray[float]]:
    return np.array([
        [1,0,0,0],
        [0,0,1,0],
        [0,-1,0,0],
        [0,0,0,1]
        ])
```

Listing 6: Matrice di adattamento spazio normalizzato a dimensione dello schermo in px

```
@staticmethod
def centra_schermo(W, H):
return np.array([
    [W/2, 0, 0, 0],
    [0, H/2, 0, 0],
```

```
    [0, 0, 1, 0],
    [W/2, H/2, 0, 1]
    ])
```

Listing 7: prima e seconda parte dell'aggiornamento degli attributi della camera

```
def rotazione_camera(self) -> None:
    '''
    Applico le rotazioni in ordine Eulero XYZ ai vari vettori di orientamento della
        camera
    '''
    self.rig = self.rig_o @ Mate.rotx(self.becche)
    self.ups = self.ups_o @ Mate.rotx(self.becche)
    self.dir = self.dir_o @ Mate.rotx(self.becche)

    self.rig = self.rig @ Mate.roty(self.rollio)
    self.dir = self.dir @ Mate.roty(self.rollio)
    self.ups = self.ups @ Mate.roty(self.rollio)

    self.rig = self.rig @ Mate.rotz(self.imbard)
    self.ups = self.ups @ Mate.rotz(self.imbard)
    self.dir = self.dir @ Mate.rotz(self.imbard)

    '------------------------------------------------------------------'

    self.pos -= self.focus
    self.pos = self.pos @ Mate.rotz(- self.delta_imbard)
    self.pos = self.pos @ Mate.rot_ax(self.rig, self.delta_becche)
    self.pos += self.focus
```

2 Camera sync

Dal momento che usiamo le matrici per generare la prospettiva nel caso del wireframe e generiamo dei raggi per ricreare lo stesso effetto nel raytracer, è necessario trovare un sistema che unifica questi metodi per riottenere la stessa immagine. Ci baseremo sul fornire ad entrambi i sistemi una variabile `fov` che indicherà l'apertura focale in radianti.

2.1 Approccio matriciale

Listing 8: Matrici di applicazione prospettiva

```
@staticmethod
def frustrum(W: int, H: int, h_fov: float=np.pi/6) -> np.ndarray[float]:
    v_fov = h_fov * H / W
    ori = np.tan(h_fov / 2)
    ver = np.tan(v_fov / 2)
    return np.array([
        [1 / ori, 0, 0, 0],
        [0, 1 / ver, 0, 0],
        [0, 0, 1, 1],
        [0, 0, 0, 0]
    ])
```

Spigazione: Con questo sistema noi possiamo calcolare la divergenza della geometria sfruttando il funzionamento della prospettiva stessa, ovvero che i punti tendono a seguire uno spostamento di cot(fov), il quale successivamente verrà diviso per la propria distanza.

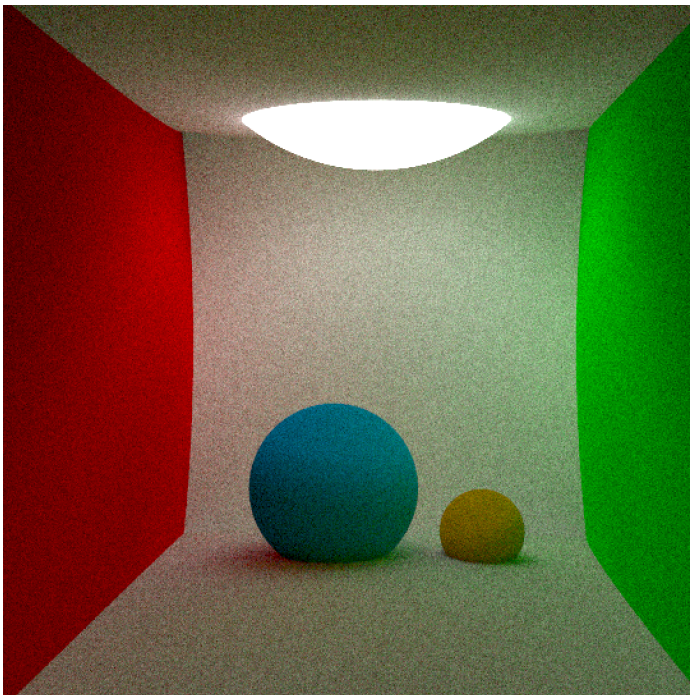
2.2 Approccio generazione raggi

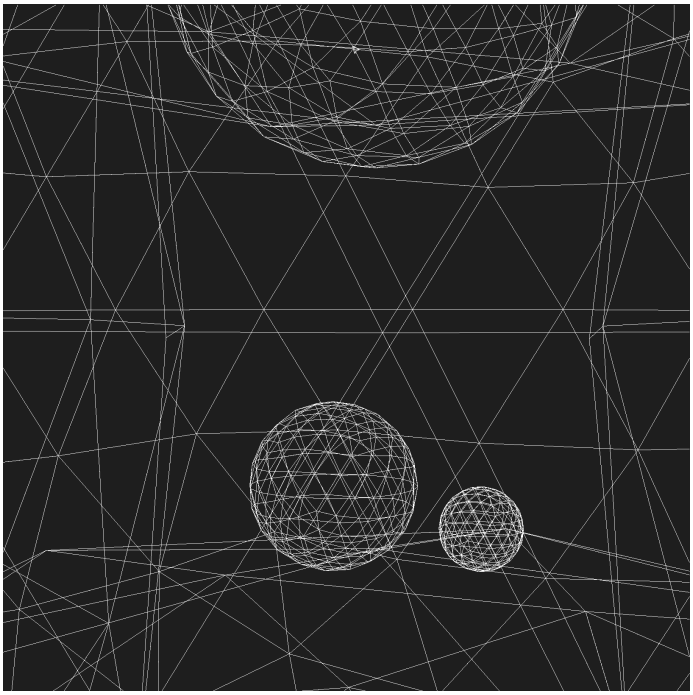
Listing 9: Calcolo della direzione del raggio generato in base al fov

```
def direzione(i, j) -> float:
    return camera.dir[:3] * (1 / np.tan(camera.fov / 2)) + (2 * (i / wo) - 1) *
        camera.rig[:3] + (2 * (j / ho) - 1) * (- camera.ups[:3]) / (wo/ho)
```

Spigazione: Dal momento che della camera noi conosciamo la direzione, la sua destra e il suo alto, possiamo calcolare di quanto dovrebbe estendersi il versore direzione affinchè sommato con il versore destra e alto, ci restituisca la direzione corretta. Nel codice manca la normalizzazione del risultato (eseguita più avanti). La necessità di dividere per 2 l'angolo del `fov` è dovuta alla scala usata. Infatti nel calcolo matriciale la base del triangolo isoscele che si forma considerando il vertice coincidente con la posizione della camera, avrà dimensione 1. Nel Case invece dei raggi generati, i versori saranno $\pm$ 1, dandoci un Δ tot = 2, e che quindi corrisponde a scala doppia.

2.3 Risultati





3 Ray - Sphere intersection

Il problema principale risiede nel calcolare il risultato dell'equazione per ottenere l'intersezione (minima e massima). Per risolverla dobbiamo risolvere l'equazione che mette in relazione: origine del raggio, direzione del raggio, posizione della sfera e raggio della sfera.

$$|ray.ori + ray.dir * t|^2 = |sphere.pos + sphere.radius|^2$$

Sappiamo che risulta tutto più comodo se la sfera è centrata sull'origine, possiamo quindi fare una trasformazione e chiamare oc la differenza tra $\vec{ray.ori} - \vec{sphere.ori}$.

$$|oc + ray.dir * t|^2 = |sphere.radius|^2$$

$$oc^2 + (ray.dir * t)^2 + 2 * oc * ray.dir * t = sphere.radius^2$$

$$ray.dir^2 * t^2 + 2 * oc * ray.dir * t + oc^2 - sphere.radius^2 = 0$$

Se volessimo risolvere per t, otterremmo un'equazione di secondo grado di forma: $ax^2 + bx + c = 0$, dove $a = ray.dir^2$, il quale essendo un versore sarà uguale a 1; $b = 2 * oc * ray.dir$; $c = oc^2 - sphere.radius^2$.
Da qui risolviamo il discriminante e otteniamo i possibili risultati.

Discussione risultati: Dobbiamo sempre considerare il risultato minimo positivo per eseguire la corretta intersezione. Attenzione che nel caso del vetro bisognerà considerare quello più lontano in certi casi, qualora si volesserò simulare trasmissioni ecc.

Spiegazioni del codice: Eseguire il prodotto scalare di un vettore per se stesso, ci darà il valore del suo modulo. La presenza del test > 0.001 invece, è dovuto per evitare errori di appossimazioni in cui l'origine del raggio successivo è di poco al di sotto della superficie, risultando quindi in una collisione con se stesso.

Listing 10: Codice intersezione raggio - sfera

```
def collisione_sphere(self, ray, record: Record):
    oc = ray.pos - self.pos;

    a = 1.0;
    b = np.dot(oc, ray.dir) * 2.0;
    c = np.dot(oc, oc) - (self.radius) ** 2;

    discriminante = b ** 2 - 4.0 * a * c;

    if discriminante >= 0.0:
        sqrt_discr = discriminante ** 0.5

        delta_min = (- b - sqrt_discr) / (2.0*a);
        delta_max = (- b + sqrt_discr) / (2.0*a);

        if delta_max > 0.001:
            t = delta_max
            if delta_min > 0.001 and delta_min < delta_max:
                t = delta_min

            if t < record.distanza:
                record.colpito = True;
                record.distanza = t;
                record.indice_oggetti_prox = self.indice;
                record.punto_colpito = self.punto_colpito(ray, record.distanza);
                record.norma_colpito = self.normale_colpita(record.punto_colpito);
                record.materiale = self.materiale;

    return record
```

4 Ray - Triangle intersection

Questo sarà l'algoritmo su cui si baserà la maggior parte dei test. Le sfere verranno usate per la luce, i triangoli invece per le mesh