Copilota RCQF

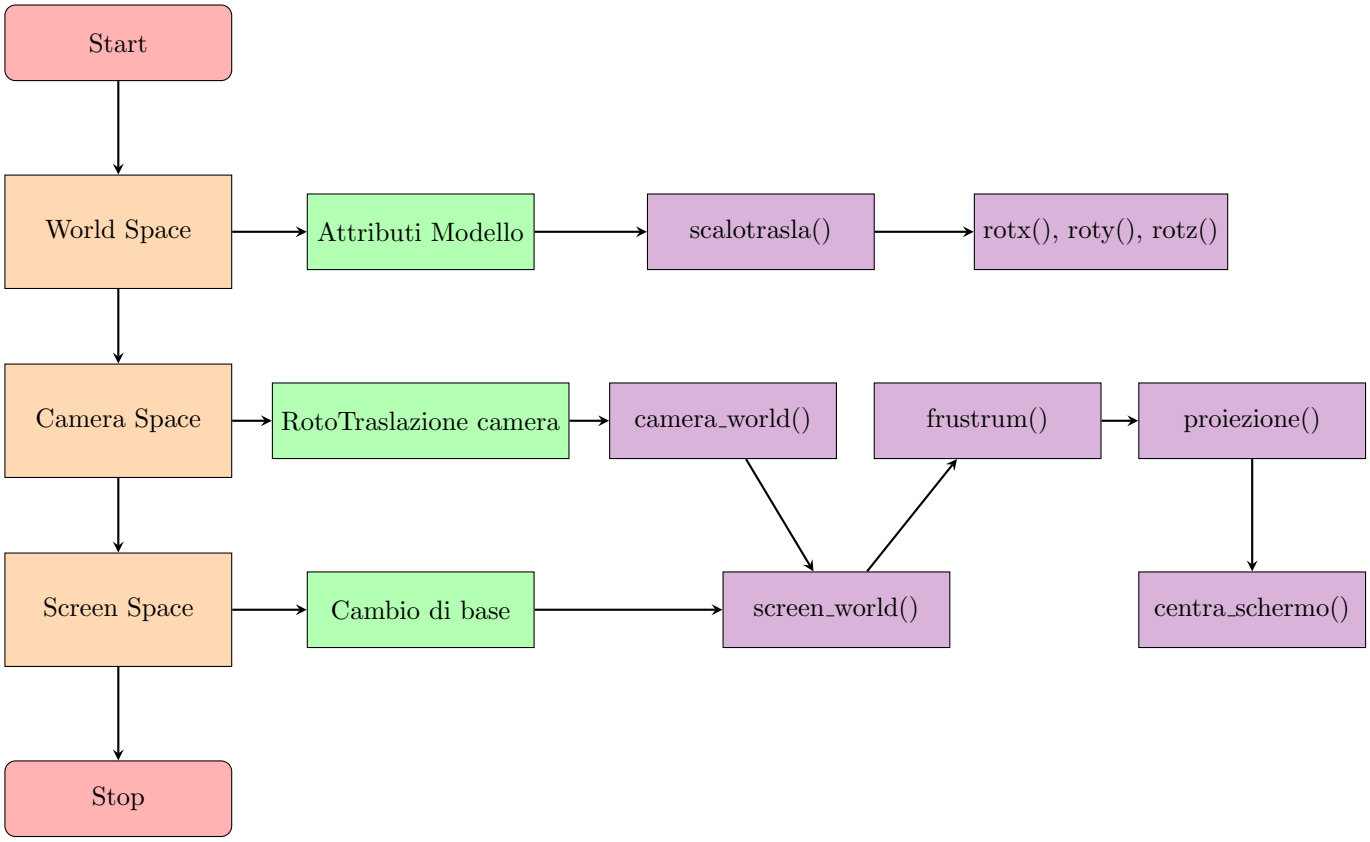
Alessio Cimma

19 giugno 2024

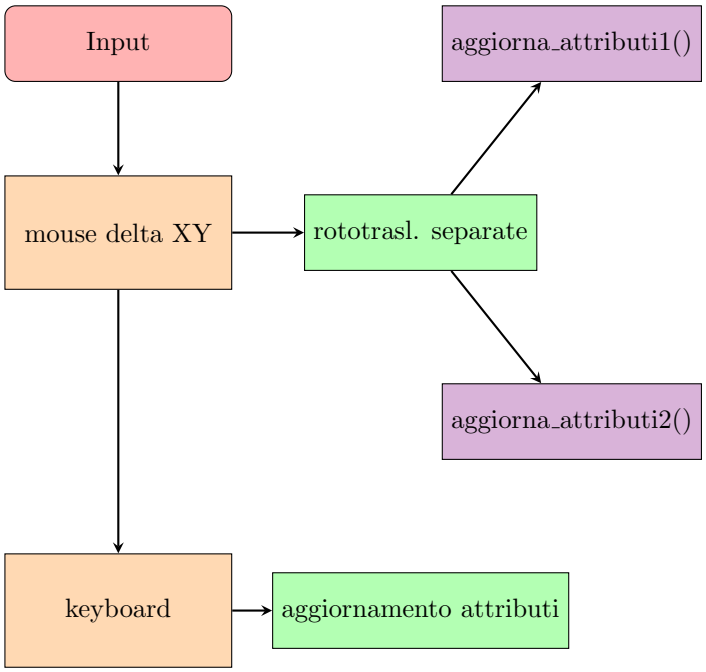
Indice

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)

```
@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

```
@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]]:
    # qua c'e' un meno per sistemare l'orientamento della camera, altrimenti
       ottieni un'immagine specchiata in prospettiva
    v_fov = h_fov * H / W
    left = np.tan(h_fov / 2)
    right = -left
    top = np.tan(v_fov / 2)
    bottom = -top
    far = 1000
    near = 0.01
    return np.array([
        [-2 / (right - left), 0, 0, 0],
        [0, 2 / (top - bottom), 0, 0],
        [0, 0, (far + near) / (far - near), 1],
        [0, 0, -2 * near * far / (far - near), 0]
    ])

@staticmethod
def proiezione(vertici: np.ndarray[np.ndarray[float]]) -> np.ndarray[np.ndarray[
    float]]:
    ris = vertici / vertici[:, -1].reshape(-1, 1)
    ris[(ris < -2) | (ris > 2)] = 0
    return ris
```

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

```python
@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

```python
@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

```python
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
```