Università degli Studi di Palermo

Corso di Laurea Magistrale in

Ingegneria Informatica

Visione Artificiale

Assignment 1

Tool for navigating 360° images

Autori

Castelli Giovanni
Costa Gabriele Nicolò
Sollazzo Ivan

Anno Accademico

2023/2024

Indice

1 Introduzione al problema

Ci viene richiesto di realizzare uno strumento di navigazione a partire da un video ripreso con telecamera 360. Il problema viene chiamato "Non-Equirectangular Projection", dal momento che il video che prendiamo è composto da frame che chiamiamo **immagini equirettangolari**, i cui pixel identificano latitudine e longitudine rispetto al centro della sfera.
Il nostro obbiettivo è quello di estrarre, a partire dai frame del video equirettangolari, una **immagine planare**, ovvero la "proiezione" senza distorsione del piano tangente la sfera nel punto individuato da latitudine e longitudine (specificato come input dall'utente).
In input l'utente difatti inserirà tre cose fondamentali: LAT (latitudine), LONG (longitudine) e FOV (field of view).

1.1 Immagine Sferica e Immagine Equirettangolare

Quando utilizziamo camere sferiche, l'immagine che otteniamo in realtà è una sfera. Dal momento che però le macchine fotografiche odierne utilizzano array di sensori, si rende necessaria una rappresentazione bidimensionale che riesca a coprire l'intera sfera (si pensi per esempio al famoso lecca-lecca Chupa Chups). Nell'immagine 1 possiamo difatti notare la corrispondente rappresentazione di una immagine equirettangolare a partire da una immagine sferica. Ogni punto tridimensionale nella sfera viene difatti mappato in un sistema

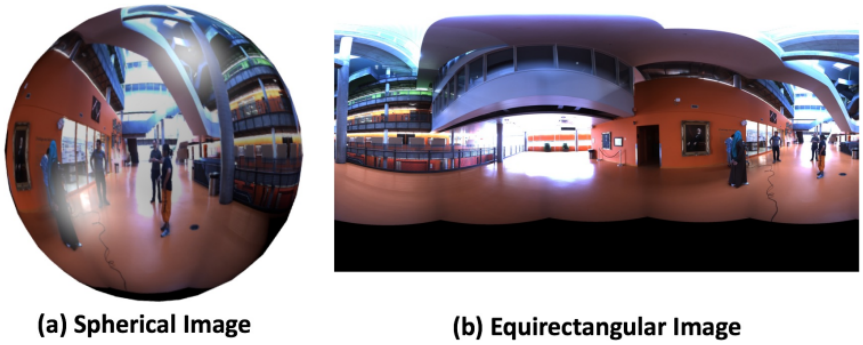


Figura 1: Immagine 1

di riferimento (θ, ϕ).
Come possiamo notare, è presente una deformazione molto evidente, quindi per estrarre una immagine planare dobbiamo ragionare in termini geometrici.

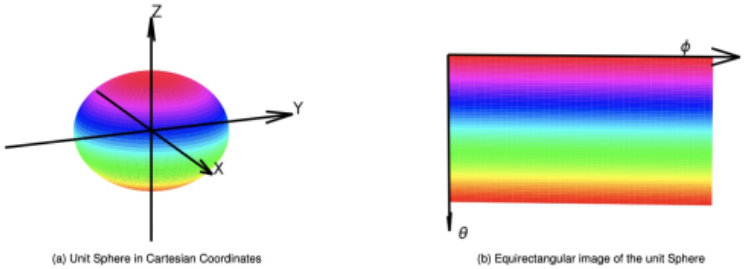


Figura 2: Sfera e rispettiva rappresentazione equirettangolare

1.2 Immagine Planare

Per estrarre l'immagine planare teniamo in considerazioni due cose:

- Possiamo pensare che ogni volta guardiamo solo una porzione dell'immagine sferica, quindi la proiezione su un piano tangente la sfera (cono visivo, quindi porzione della superficie sferica proiettata su un piano tangente la sfera nel suo centro di proiezione).
- Sappiamo come convertire un punto tridimensionale sul piano tangente in punti sulla sfera quindi in coordinate sferiche

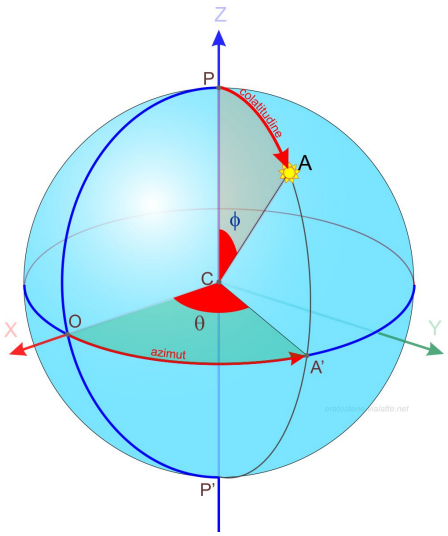


Figura 3: Coordinate sferiche

A questo punto quindi possiamo dire che:

- Ogni punto sull'**immagine equirettangolare** identifica un punto espresso tramite coordinate sferiche
- Ogni punto sull'**immagine sferica** è un punto tridimensionale equidistante dal centro della sfera, ma comunque un punto del tipo (x,y,z)
- Ogni punto sul **piano tangente** è un punto tridimensionale, trasformazione dell'**immagine planare** di uscita.

Our goal is to project part of the spherical surface onto a tangent plane to rectify the image and reduce deformation
The plane is tangent to the sphere at a 3D point
Only a portion of the defined sphere can be projected across a field of view expressed in degrees (or radiants)

Figura 4: Cono visivo

1.3 Algoritmo

Un possibile algoritmo per risolvere il problema del Non-Equirectangular Projection (quindi **Rectification Projection**) è quello di usare l'inverse mapping: **per ogni pixel dell'immagine planare di output cerchiamo il corrispondente punto nell'immagine equirettangolare**. Il processo quindi è riassunto nella figura seguente. Riassumendo dovremmo:

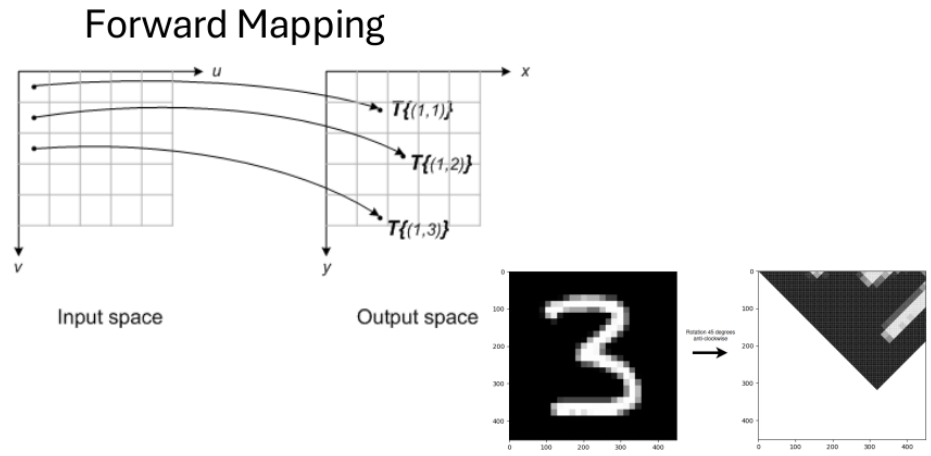


Figura 5: Forward Mapping

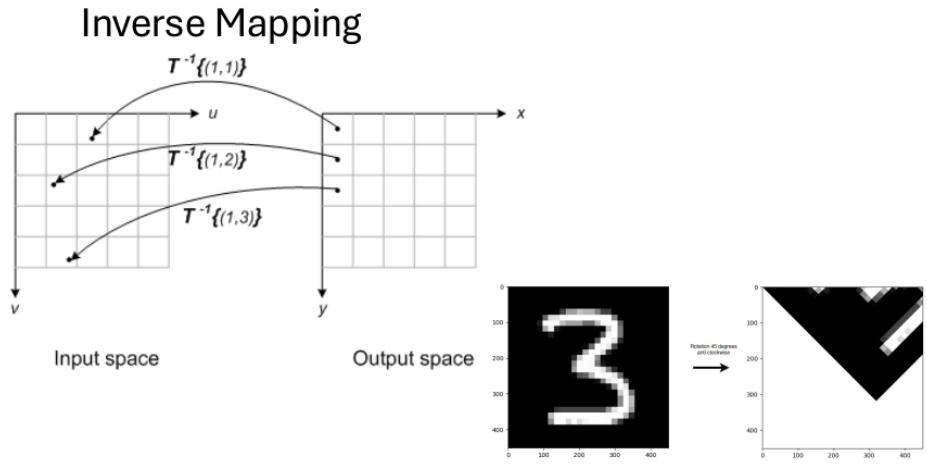


Figura 6: Inverse Mapping

- Per ogni pixel (u,v) della immagine planare
  - Troviamo il punto corrispondente nel piano tangente. Qui è conveniente pensare che il piano tangente canonico abbia centro nel punto $p = (1, 0, 0)$.
  - Per trovare il punto corrispondente a partire da (u,v) facciamo la prima trasformazione **chiamata T1** per far corrispondere l'origine del sistema di riferimento (u,v) con l'origine del sistema di riferimento individuato dal piano tangente.
  - Per questa parte far riferimento all'immagine PLANAR TO TANGENT
- Dopo aver trovato per ogni (u,v) il corrispondente nel piano tangente, dobbiamo ruotare questo piano tangente al fine di "coordinarci" con latitudine e longitudine fornite dall'utente. Vedere immagine

ROTATION. Per quanto concerne la rotazione, bisogna far riferimento al fatto che in base al nostro sistema di riferimento, l'asse z si trova sopra la nostra testa, mentre l'asse longitudinale e laterale di lato e di fronte a noi.

- Infine proiettare i punti del piano ruotato sulla sfera per mezzo della proiezione sferica (si faccia riferimento alla immagine TANGENT TO SFERIC)
- Quindi estrarre per ogni punto le coordinate (ρ, θ, ϕ) tramite formule opportune in funzione dei punti proiettati sulla immagine sferica
- Terminiamo facendo l'ultima trasformazione per portare (ρ, θ, ϕ) in pixel.

Di seguito quindi si prenderenno in considerazione le seguenti operazioni:

- Operazione 1: trasformare le coordinate planare in coordinate tridimensionali sul piano tangente (T1).
- Operazione 2: rotazione attorno al piano xy e attorno al piano zx, ovvero le rotazioni per raggiungere latitudine e longitudine
- Operazione 3: Proiezione dei punti tridimensionali ruotati sulla sfera
- Operazione 4: Conversione punti sulla sfera tridimensionali in punti in coordinate sferiche, assumendo per il momento raggio ρ costante pari ad 1 (nella realtà dei fatti ρ corrisponde allo zoom.
- Operazione 5: Dopo aver estratto quindi θ, ϕ dobbiamo effettuare l'ultima trasformazione per ottenere i punti in pixel dell'**immagine equirettangolare**

Alla fine del processo otteniamo una mappatura tale per cui ad ogni punto (u, v) corrisponde un pixel (x_f, y_f), dove quest'ultimi sono pixel del frame del video equirettangolare.

$$map[u] = x_f$$

$$map[v] = y_f$$

Utilizziamo quindi una funzione di opencv che ci permette di fare remap a partire da due array di mapping e un tipo di trasformazione (nel nostro caso, bilineare).

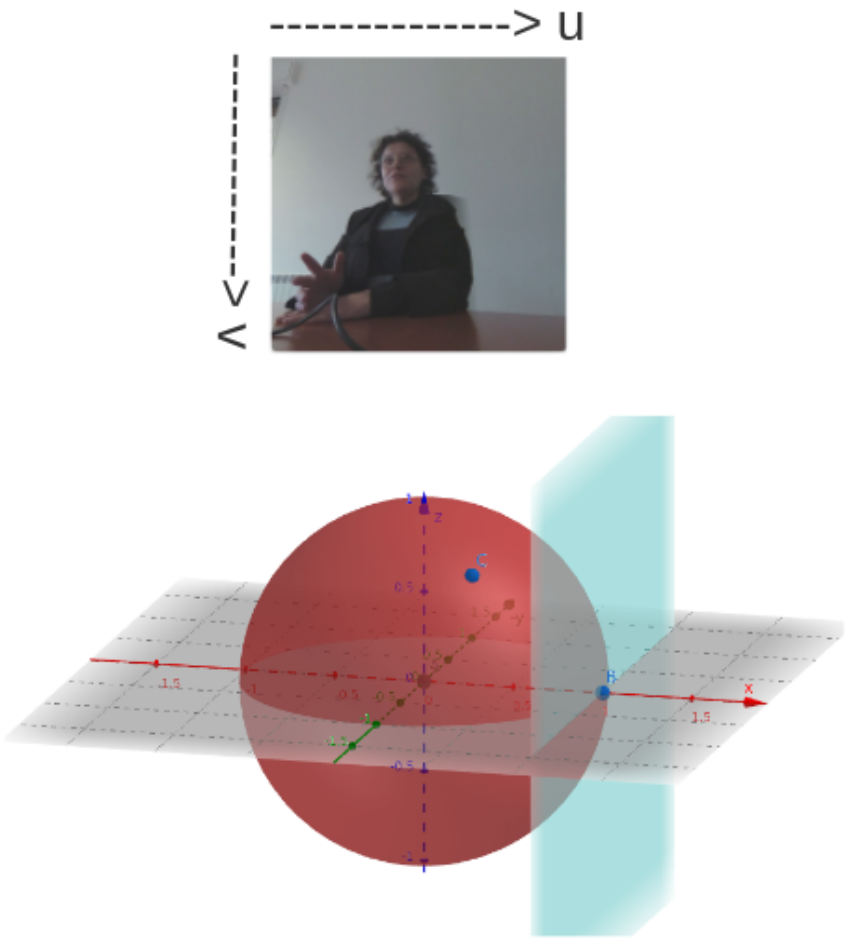


Figura 7: In alto l'immagine planare che nella realtà sarà una matrice numpy di n*m pixel, cui ogni punto va mappato nel piano tangente

2 Come procedere

Riassumendo, abbiamo le seguenti operazioni:

- Operazione 1: trasformare le coordinate planare in coordinate tridimensionali sul piano tangente (T1).
- Operazione 2: rotazione attorno al piano xy e attorno al piano zx, ovvero le rotazioni per raggiungere latitudine e longitudine
- Operazione 3: Proiezione dei punti tridimensionali ruotati sulla sfera
- Operazione 4: Conversione punti sulla sfera tridimensionali in punti in coordinate sferiche, assumendo per il momento raggio ρ costante pari ad 1 (nella realtà dei fatti ρ corrisponde allo zoom.
- Operazione 5: Dopo aver estratto quindi θ, ϕ dobbiamo effettuare l'ultima trasformazione per ottenere i punti in pixel dell'**immagine equirettangolare**

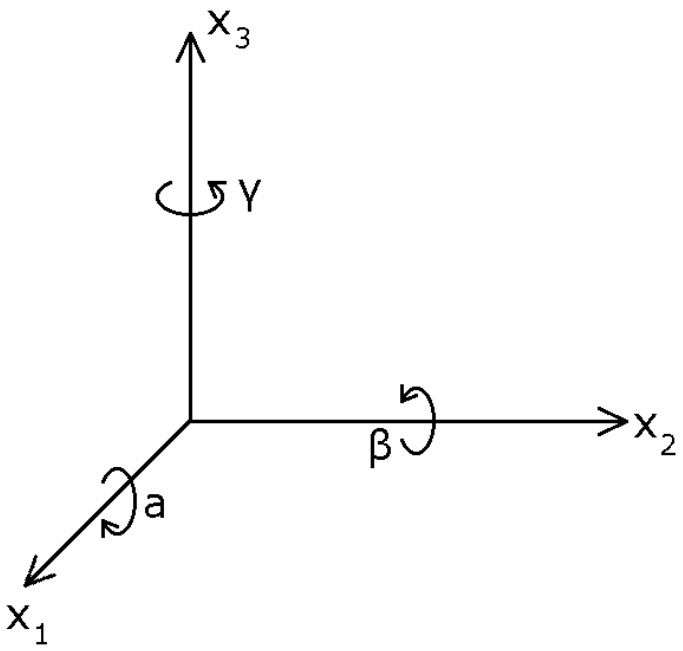
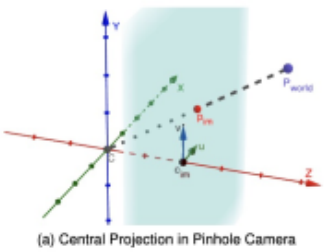


Figura 8: Sistema di riferimento per effettuare le rotazioni. Nel nostro caso inoltre x_3 corrisponde al nostro asse z, mentre x_1 corrisponde al nostro asse y, infine l'asse x_2 corrisponde al nostro asse di riferimento x

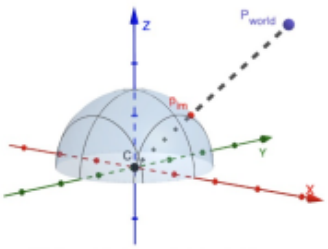
Una rotazione generale in 3 dimensioni può essere espressa come una composizione di 3 rotazioni intorno a tre assi indipendenti, come ad esempio gli assi x, y, z[1]. Quindi dati tre angoli α, β, γ, che indicano rispettivamente di quanto si deve ruotare intorno a ognuno degli assi, la matrice di rotazione risulta:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \cdot \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \cdot \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figura 9: Principi per applicare le matrici di rotazione in funzione del sistema di riferimento. Fonte: https://it.wikipedia.org/wiki/Rotazione_(matematica)



(a) Central Projection in Pinhole Camera



(b) Central Projection in Spherical Camera

To find projection on the plane we use

$$\bar{x} = \begin{bmatrix} f\frac{x}{z} \\ f\frac{y}{z} \\ 1 \end{bmatrix}$$

To find projection on the sphere we use

$$\bar{x} = \frac{x}{\|x\|}$$

Which is a 3D point

Figura 10: Proiezione punti tridimensionali sulla sfera

2.1 Operazione 1 - Dall'immagine planare al piano tangente

Per ogni (u,v) dell'immagine planare dobbiamo fare una trasformazione lineare che ci fa ottenere i punti sul piano tangente, ovvero $(1, u', v')$

$$u' = \frac{u - \frac{W_{planar}}{2}}{W_{planar}} \cdot 2\tan\Big(\frac{\alpha}{2}\Big)$$

$$v' = -\frac{v - \frac{H_{planar}}{2}}{H_{planar}} \cdot 2\tan\Big(\frac{\alpha}{2}\Big)$$

L'immagine planare ha dimensioni $W_{planar} \cdot H_{planar}$, mentre α risulta essere il field of view (FOV) definito dall'utente. Questa trasformazione serve per fare in modo che i nuovi punti (u', v') vengano correttamente mappati nel piano tangente di dimensione $H = 2\tan\big(\frac{\alpha}{2}\big)$ a partire dal nuovo sistema di riferimento che corrisponde all'origine del centro del piano di coordinate (1,0,0).
Dunque fin qui:

$$(u, v) \Longrightarrow (1, u', v')$$

Va tuttavia specificata che questa tripla di punti corrisponde ad una tripla di punti nello spazio tridimensionale, ovvero sono i punti sopra il piano tangente.

2.2 Operazione 2 - Rotazione del piano tangente

Adesso che ho i punti sul piano tangente, devo comunque ruotarli. Adesso, se ragioni in termini di latitudine e longitudine, il mio punto di partenza risulta essere (1,0,0). Per ruotare il piano tangente dunque dobbiamo effettuare:

- Una rotazione di θ attorno al piano zx, ovvero attorno all'asse y, ovvero $R_y(\theta)$, nel nostro caso θ corrisponde alla **latitudine**
- Una rotazione di ϕ attorno al piano xy, ovvero rotazione attorno all'asse z, ovvero $R_z(\phi)$, nel nostro caso ϕ corrisponde alla **longitudine**
- Una rotazione di 0 attorno al piano yz, ovvero rotazione attorno all'asse x. (che deve quindi restare fisso)

I valori di rotazione vengono inseriti dall'utente, tramite appunto i valori di **latitudine** e **longitudine**.
Per fare questo utilizzo le matrici di rotazione:

$$R = R_y(\theta) \cdot R_z(\phi) \cdot R_x(0)$$

Nella determinazione dell'ordine nelle matrici di rotazione facciamo riferimento al fatto che **prima dobbiamo ruotare l'asse trasversale, ovvero y, poi ruotiamo l'asse longitudinale, ovvero z**. Che si semplifica considerando che $R_x(0) = I_{3x3}$ Quindi:

$$(x, y, z) = p_{tangente} = R \cdot (1, u', v')$$

Fin qui quindi, applichiamo la rotazione ai punti trovati al punto 1, ottenendo:

$$(1, u', v') \Longrightarrow (x, y, z)$$

2.3 Operazione 3 - Proiezione punti dal piano tangente alla sfera

Per proiettare i punti dal piano tangente alla sfera, devo semplicemente normalizzare e considerare che:

$$p_{sfera} = \frac{p_{tangente}}{\|p_{tangente}\|}$$

Quindi ottengo i punti proiettati sulla superficie sferica in coordinate tridimensionali:

$$(x', y', z') = \frac{(x, y, z)}{\|(x, y, z)\|}$$

Dunque abbiamo ottenuto:

$$(x, y, z) \Longrightarrow (x', y', z')$$

2.4 Operazione 4 - Cambio di coordinate, da cartesiane a polari

Per ottenere le coordinate polari a partire da quelle cartesiane utilizzo le seguenti formule inverse, che ci permettono di estrarre θ_{sfera} e ϕ_{sfera}):

$$p = costante$$

$$\phi_{sfera}) = \arctan\left(\frac{y'}{z'}\right)$$

$$\theta_{sfera} = \arccos\left(\frac{z'}{p}\right)$$

In prima approssimazione, possiamo supporre p=1, per semplicità di calcolo, però potrebbe essere pensato come lo "zoom" dell'utente. (altro parametro da specificare)
Se "droppiamo" la p, ovvero la consideriamo come costante ed unitaria, otteniamo le coordinate $(\theta_{sfera}, \phi_{sfera}))$ per ogni punto (u, v) di partenza!
Difatti, $(\theta_{sfera}, \phi_{sfera}))$ equivalgono ai "valori" che otteniamo guardando l'immagine equirettangolare: θ comanda le righe, ϕ) comanda le colonne.
Tuttavia manca ancora un passo: le coordinate che abbiamo trovato sono ancora in gradi, mentre l'immagine equirettangolare di uscita è in pixel!
Ad ogni modo, al termine di questa operazione ottengo il seguente risultato:

$$(x', y', z') \Longrightarrow (p, \theta, \phi)$$

2.5 Operazione 5 - Riscalatura e trasformazione finale

Per ottenere i pixel dell'immagine equirettangolare dobbiamo applicare una trasformazione lineare (l'ultima) ai punti trovati al punto di sopra per riportarci alle dimensioni del frame immagine equirettangolare.
Siano H_{frame}) e W_{frame}) le dimensioni dell'immagine equirettangolare che andiamo ad estrarre da un frame, sulla base di $(\theta, \phi))$ applichiamo la seguente trasformazione:

$$x_{frame} = \phi \cdot \frac{W_{frame}}{2\pi}$$

$$y_{frame} = \frac{\theta - \frac{\pi}{2}}{\pi} \cdot H_{frame}$$

I punti x_{frame} e y_{frame} risultano essere i pixel del frame immagine equirettangolare. Alla fine quindi otteniamo:

$$(p, \theta, \phi) \Longrightarrow (x_{frame}, y_{frame})$$

2.6 Conclusioni finali

Se si segue l'intero processo di **inverse mapping**, riusciamo a determinare una matrice di corrispondenze e quindi fare interpolazione bilinare:

$$map_x[u, v] = x_{frame}$$

Nota bene: deve essere della stessa dimensione dell'immagine di destinazione, ovvero planare.

$$map_y[u, v] = y_{frame}$$

Per fare ciò utilizziamo la funzione di OpenCV remap() che, presa una immagine sorgente (il nostro frame immagine equirettangolare, map_x e map_y) per mappare i punti dall'immagine di destinazione nell'immagine sorgente usando interpolazione bilineare.

3 Funzionamento del software

3.1 Funzione di inverse mapping

La suddetta spiegazione dell'inverse mapping è implementata per mezzo di un'opportuna funzione scritta in codice Python, la quale accetta come parametri:

- `input_image`: è l'immagine equirettangolare originale. Rappresenta nel caso specifico una matrice Numpy;
- `w_output` e `h_output`: la larghezza e l'altezza desiderate per l'immagine planare, la quale è l'output;
- `latitude` e `longitude`: le coordinate che definiscono il punto centrale della vista nell'immagine di output, espresse in gradi;
- `fov`: il campo visivo FOV (field of view) per l'immagine di output, anch'esso espresso in gradi.

In uscita restituisce `output_image`, la quale è una matrice Numpy che rappresenta l'immagine planare. In particolare, la funzione di trasformazione è la seguente:

```
def inverse_map(input_image, w_output, h_output, latitude, longitude, fov):

    # Calcolo dimensioni immagine equirettangolare
    h_input, w_input = input_image.shape[0], input_image.shape[1]

    # Conversione da gradi a radianti
    latitude_rad = np.radians(latitude)
    longitude_rad = np.radians(longitude)
    fov_rad = np.radians(fov)

    #u = np.linspace(-np.tan(fov_rad / 2), np.tan(fov_rad / 2), w_output)
    #v = np.linspace(-np.tan(fov_rad / 2), np.tan(fov_rad / 2), h_output)
    # Creazione di un piano tangente scalato
    u = np.linspace(-1, 1, w_output) * np.tan(fov_rad / 2)
    v = np.linspace(-1, 1, h_output) * np.tan(fov_rad / 2)
    U, V = np.meshgrid(u, v)

    # Costruzione dei vettori nel piano tangente, assumendo x = 1
    X = np.ones_like(U) * zoom
    Y = U.copy()
    Z = -V.copy()

    # Definizione delle matrici di rotazione
    Ry = np.array([[np.cos(latitude_rad), 0, np.sin(latitude_rad)], [0, 1, 0], [-np
        .sin(latitude_rad), 0, np.cos(latitude_rad)]])
    Rz = np.array([[np.cos(longitude_rad), -np.sin(longitude_rad), 0], [np.sin(
        longitude_rad), np.cos(longitude_rad), 0], [0, 0, 1]])

    # Costruzione dei vettori intersecanti il piano
    V_initial = np.stack((X, Y, Z), axis=-1)

    # Rotazione del piano
    V_rotated = V_initial @ Ry @ Rz

    # Normalizzazione
    V_normalized = V_rotated / np.linalg.norm(V_rotated, axis=-1, keepdims=True)

    # Calcolo delle coordinate sferiche
    theta = np.arccos(V_normalized[..., 2])
    phi = np.arctan2(V_normalized[..., 1 ], V_normalized[...,0])
```

```
    # Mappatura alle coordinate dell'immagine equirettangolare
    map_x = (((phi / np.pi) + 1) * (w_input / 2)).astype(np.float32)
    map_y  = ((theta / (np.pi)) * h_input).astype(np.float32)

    # Creazione di un'immagine planare
    output_image = np.zeros((h_output, w_output, 3), dtype=np.uint8)

    # Utilizzo di remap per effettuare l'interpolazione
    output_image = cv2.remap(input_image, map_x, map_y, interpolation=cv2.
        INTER_LINEAR)

    return output_image
```

Le operazioni di trasformazione si concretizzano nelle seguenti porzioni di codice:

3.1.1 Creazione delle matrici di coordinate

```
    u = np.linspace(-1, 1, w_output) * np.tan(fov_rad / 2)
    v = np.linspace(-1, 1, h_output) * np.tan(fov_rad / 2)
    U, V = np.meshgrid(u, v)

    X = np.ones_like(U) * zoom
    Y = U.copy()
    Z = -V.copy()
```

Qusta parte di codice costituisce il primo step della conversione e permette di generare il piano tangente alla sfera. Nel dettaglio:

- La funzione `np.linspace(...)` genera due array, uno per l'asse orizzontale (u) e uno per l'asse verticale (v), entrambi con valori compresi nell'intervallo $[-1, 1]$ e scalati in base al campo visivo. Il numero di valori è pari a `w_output` (nel caso dell'asse orizzontale) e `h_output` nel caso dell'immagine verticale;
- La funzione `np.meshgrid(u,v)` combina i vettori u e v per formare una griglia di coordinate, dove U e V contengono le coordinate (U, V) su questo piano;
- `X = np.ones_like(U) * zoom` crea una matrice delle stesse dimensioni di U, dove vi sono tanti 1 quanti sono gli elementi di U. Inoltre, ciascun 1 viene moltiplicato per il valore di `zoom`;
- `Y=U.copy()` e `Z=-V.copy()` copiano le coordinate di U e $-V$ nelle matrici Y e Z rispettivamente. L'uso di $-V$ per Z inverte l'asse verticale.

3.1.2 Rotazione del piano per l'allineamento al punto specificato

```
    # Definizione delle matrici di rotazione
    Ry = np.array([[np.cos(latitude_rad), 0, np.sin(latitude_rad)], [0, 1, 0], [-np
        .sin(latitude_rad), 0, np.cos(latitude_rad)]])
    Rz = np.array([[np.cos(longitude_rad), -np.sin(longitude_rad), 0], [np.sin(
        longitude_rad), np.cos(longitude_rad), 0], [0, 0, 1]])

    # Costruzione dei vettori intersecanti il piano
    V_initial = np.stack((X, Y, Z), axis=-1)

    # Rotazione del piano
    V_rotated = V_initial @ Ry @ Rz
```

La suddetta porzione di codice si occupa di definire le matrici di rotazione, di costruire i vettori intersecanti il piano e di ruotare il medesimo. Nel dettaglio:

- Vengono definite le matrici di rotazione attorno all'asse Y e attorno all'asse Z con riferimento ai valori di `latitude_rad` e `longitude_rad`;
- I vettori iniziali vengono costruiti impilando le matrici X, Y e Z lungo l'asse delle z. Questo permette quindi di creare un array tridimensionale che rappresenta i punti nello spazio;
- I vettori vengono poi ruotati applicando le matrici di rotazione precedentemente definite. L'operazione di prodotto matriciale `@` applica la rotazione a tutti i vettori in `V_initial`. In particolare:
  - La rotazione avviene prima attorno all'asse Y (per la latitudine) e poi attorno all'asse Z (per la longitudine);
  - Il risultato `V_rotated` è un insieme di vettori che sono stati ruotati per allinearsi con la direzione specificata dalle coordinate di latitudine e longitudine fornite.

3.1.3 Normalizzazione

```
# Normalizzazione
V_normalized = V_rotated / np.linalg.norm(V_rotated, axis=-1, keepdims=True)
```

Questa porzione di codice permette semplicemente di effettuare la normalizzazione dei vettori.

3.1.4 Passaggio alle coordinate sferiche

```
# Calcolo delle coordinate sferiche
theta = np.arccos(V_normalized[..., 2])
phi = np.arctan2(V_normalized[..., 1 ], V_normalized[...,0])
```

Questa porzione di codice permette di ottenere le coordinate sferiche, ovvero i vettori `theta` e `phi`. In particolare:

- `V_normalized[...,2]` sono i valori della coordinata Z, quindi `theta` è calcolato con riferimento all'arcocoseno di tali valori;
- `V_normalized[...,1]` e `V_normalized[...,0]` sono rispettivamente i valori della coordinata Y e della coordinata X, quindi `phi` è calcolato con riferimento all'arcotangente2 di Y/X. Il vantaggio di eseguire l'arcotangente2 sta nel fatto che l'angolo considerato rientra nell'intervallo $[-\pi, \pi]$ e non $[-\pi/2, \pi/2]$

3.1.5 Mappatura alle coordinate dell'immagine equirettangolare

```
map_x = (((phi / np.pi) + 1) * (w_input / 2)).astype(np.float32)
map_y = ((theta / (np.pi)) * h_input).astype(np.float32)
```

Questa porzione di codice permette invece di calcolare le coordinate mappate `map_x` e `map_y`, le quali saranno utilizzate per proiettare l'immagine equirettangolare originale su un'immagine planare con riferimento alle coordinate sferiche.

3.1.6 Interpolazione col metodo `cv2.remap()`

```
output_image = cv2.remap(input_image, map_x, map_y, interpolation=cv2.
    INTER_LINEAR)
```

Infine, la funzione `cv2.remap()` consente di trasformare l'immagine di input basandosi su un set di mappe di coordinate specificate, risultando in un'immagine di output modificata secondo le mappe fornite. Nel dettaglio, vengono riallocati i pixel dell'immagine di input a nuove posizioni nell'immagine di output in base alle mappe `map_x` e `map_y`.

3.2 Funzione di gestione della navigazione da tastiera

La seguente funzione permette di navigare fluentemente all'interno dell'immagine:

```
def tastiera():
    global lat, long, FOV, zoom, w, h, paused
    # Aspetta l'input da tastiera
    key = cv2.waitKey(1) & 0xFF
    #print(key) utile per capire il codice del tasto che clicco
    # Se la freccia su e' premuta
    if key == ord('w'):
        lat += 1
    # Se la freccia giu e' premuta
    elif key == ord('s'):
        lat -= 1
    # Se la freccia destra e' premuta
    elif key == ord('a'):
        long += 1
    # Se la freccia sinistra e' premuta
    elif key == ord('d'):
        long -= 1
    # Se il tasto c e' premuto zooma
    elif key == ord('c'):
        zoom +=0.1
        if zoom > 10:
            zoom = 10
    # Se il tasto v e' premuto dezooma
    elif key == ord('v'):
        zoom -= 0.1
        if zoom < 1:
            zoom = 1
    # Se il tasto x e' premuto aumenta il FOV
    elif key == ord('x'):
        FOV += 1
    # Se il tasto z e' premuto diminuisci il FOV
    elif key == ord('z'):
        FOV -= 1
    # Se il tasto 'q' e' premuto, esce dal ciclo
    elif key == ord('q'):
        return False
    # Se il tasto 't' e', metti in pausa il video
    elif key == ord('t'):
        if not paused:
            paused = True
        else:
            paused = False
    return True
```

Qui sotto un elenco di comandi che si possono usare per navigare nell'immagine:

- **w** ti permette di modificare la latitudine, aumentandola.
- **s** ti permette di modificare la latitudine, diminuendola.
- **a** ti permette di modificare la longitudine, aumentandola. (spsota verso sinistra)
- **d** ti permette di modificare la longitudine, diminuendola. (spsota verso destra)
- **x** ti permette di modificare il fov, aumentandolo.
- **z** ti permette di modificare il fov, diminuendolo.

- **c** ti permette di modificare il zoom, aumentandolo.
- **v** ti permette di modificare il zoom, diminuendolo.
- **t** ti permette di catturare il frame, fermando il video.

3.3 Funzione per la cattura dei frame di un video

L'idea è quella di ottenere il numero totale di frame del video, prima catturandoli e poi salvandoli in una lista vuota creata per memorizzare i fotogrammi:

```
def capture_video_frames(path):
    video = cv2.VideoCapture(path)
    nFrames = int(video.get(cv2.CAP_PROP_FRAME_COUNT))
    frames = []

    for i in range(1, nFrames):
        frame = capture_frame(path, i)
        print(f"Saving frame {i} of {int(nFrames)}...")
        frames.append(frame)

    return frames
```

3.4 Funzione per la riproduzione del video

```
def play_video(video_array):

    # Metodo di processamento del frame
    def process_frame(frame):
        plane_image = inverse_map(frame, w, h, lat, long, FOV)
        return plane_image

    # Loop di esecuzione
    i = 0
    running = True

    while running:
        if i < len(video_array):
            frame = video_array[i]
            plane_image_final = process_frame(frame)

            if not show_frame(plane_image_final):
                break
            if paused:
                while True:
                    plane_image_final = process_frame(video_array[i])
                    if not show_frame(plane_image_final):
                        running = False
                        break
                    if not paused:
                        break
                if not running:
                    break
            i += 1
        else:
            cv2.destroyAllWindows()
            running = False
```

In tale codice, il frame viene processato dalla funzione *process_frame*, nel loop:

1. il frame viene processato
2. il frame processato viene mostrato tramite la funzione *show_frame*. Se tale funzione restituisce false, il ciclo si interrompe.
3. Se il video è in pausa (premendo t come scritto sopra), il frame corrente conmtinua ad essere processato e mostrato fino a quando non è più in pausa.
4. Se *show_frame* restituisce false durante la pausa, il ciclo si interrompe

3.5 Riproduzione dei frame del video:

```
paused = False
video_array = capture_video_frames("video_1.MP4")
play_video(video_array)
```