Vision and Cognitive Systems

Cristian Mercadante
UNIMORE | LAST UPDATE: 30/05/2020

Lecture #11

Wednesday 25th March 2020

Templates and shapes (1)

Iniziamo a pensare a qualche modello. Nel mondo in 3D abbiamo oggetti, ma quando parleremo di camera model, il 3D model è mappato sul piano 2D dell'immagine. Quindi avremo a che fare con forme. Le forme possono essere visual shape (mano, bocca, ecc.), ma anche ideal shape (basata su conoscenze a priori e sulla geometria: se vediamo una scatola, vediamo un rettangolo). Dopo parleremo di pattern e dell'utilizzo di feature visuali (come istogramma del colore o feature vector di che arriva dai layer di una NN).

Vogliamo capire come riconoscere una forma nell'immagine. Il **fitting** di un modello geometrico si riferisce al trovare i valori dei parametri di un modello per trovare il migliore allineamento corrispondente alle istanze del modello. RANSAC è un algoritmo molto generico per fare ciò. Vedremo Hough Transform.

Il **matching** è più forte del fitting. Non vogliamo solo trovare la miglior corrispondenza fra il modello e i dati, ma consideriamo un modello non parametrico. Il matching può essere fatto in 2D, in 3D, ma noi vedremo un caso semplice. È importante ad esempio nella robotica, quando un braccio meccanico deve prendere un oggetto, avendo un'idea di come questo sia fatto, oppure in medicina.

Il **template matching** è il modo più facile: ho un set di pixel che cerco nell'immagine, anche tenendo conto di modifiche della forma. Un template è una generica finestra di pixel che è agnostica della propria semantica (è diverso da un pattern). TM comporta anche la localizzazione del template in modo preciso nell'immagine. Abbiamo una forma e vogliamo trovare quella forma senza cambiamenti di orientamento nell'immagine o forma. Usiamo in genere global TM (considero tutto l'oggetto), local TM (considero solo una parte della forma) o block matching (considero il template solo come un blocco di pixel). Per cercare, posso usare il MSE, ma il metodo più semplice è la cross-correlation. Se usassimo il MSE, otterremmo una mappatura in cui il valore più alto è la posizione del template, però la potenza è onerosa computazionalmente. Quindi si è passati ad altre funzioni che approssimano il MSE, come il MAD e il SAD.

Ricordiamo che la correlazione è uguale alla convoluzione se il kernel è simmetrico. La funzione di match prende il template e un'immagine e va nello spazio R per trovare la misura. La funzione che usiamo è la **cross-correlation**.

$$F(m,n) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} g(m+i, n+j) \cdot t(i,j)$$

Perché usiamo al CC? Si parte dalla definizione di MSE, che è la miglior misura (ma costosa):

$$MSE = D_{i,j}(m,n) = \sqrt{\sum_i \sum_j \big(g(i+m, j+n) - t(i,j)\big)^2}$$

Se effettuiamo i calcoli nel temine interno e rimuoviamo la radice:

$$E2(m,n) = \sum_i \sum_j g^2(i+m, j+n) - 2g(i+m, j+n) \cdot t(i,j) + t^2(i,j)$$

Il terzo termine è la somma dell'intensità dei pixel nel template, quindi possiamo eliminarla. Supponiamo che la prima parte si possa rimuovere (anche se non è costante): quello che rimarrebbe assomiglierebbe molto alla CC. Il primo termine è il valore medio del livello di grigio: vorremmo poterlo eliminare, per avere un'approssimazione del MSE. Per eliminare il primo termine si passa per una normalizzazione. Si usa l'NCC: non si usano i valori originali dell'immagini o del template, ma si sottrae la media. Inoltre, ha il vantaggio di essere normalizzata fra 0 e 1.

$$\hat{t} = \frac{t - \bar{t}}{\sqrt{\sum (t - \bar{t})^2}} \qquad \hat{g} = \frac{g - \bar{g}}{\sqrt{\sum (g - \bar{g})^2}}$$

$$NCC(t,g) = C_{tg}(\hat{t}, \hat{g}) = \sum_{[i,j] \in \Re} \hat{t}(i,j) \cdot \hat{g}(i,j)$$

La **block matching** è utilizzata nella compressione video per trovare i motion vector dei blocchi.

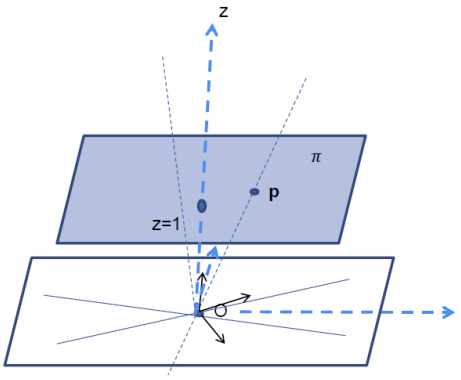
Se abbiamo un cambio di prospettiva, una trasformazione del pattern, dobbiamo capire come fare i cambiamenti opportuni. Serve la **geometria** perché:

- Il modello di camera si basa sulla geometria 3D
- Attorno a noi molti oggetti sono fatti in CAD, quindi hanno la geometria nel modello.

Consideriamo i punti su un piano cartesiano. Con questo piano non possiamo rappresentare il concetto di infinito. Possiamo usare un'altra geometria: le coordinate omogenee o **projective coordinates**. Invece che usare lo spazio cartesiano con N coordinate, possiamo lavorare in un nuovo spazio $P^n = \mathbb{R}^{n+1} - \vec{0}$, che è un piano in cui aggiungiamo una dimensione.

$$in\ 2D: \quad \boldsymbol{x} = (x, y) \in \mathbb{R}^2 \rightarrow \tilde{\boldsymbol{x}} = (\tilde{x}, \tilde{y}, \tilde{w}) \in P^2$$

Supponiamo di essere su un piano π in uno spazio 3D, che corrisponde a $z = 1$. Quindi immaginiamo di usare solo un piano di uno spazio 3D. Consideriamo l'origine e tutte le possibili linee che partono dall'origine e vanno ovunque (il fascio). Se consideriamo solo un punto $p \in \pi$, questo punto può essere rappresentato come $p = (x, y, 1) \in \pi$. Questo punto è in corrispondenza con $\{(wx, wy, w),\ w \in \mathbb{R}\} \in S$, perché questo punto può essere messo in corrispondenza con tutti i punti che sono sulla retta che passa per l'origine e per il punto stesso. Non consideriamo l'origine perché l'origine può essere rappresentata da ogni punto, perché ogni punto passa per l'origine (o meglio, per ogni punto p, il punto è in corrispondenza con tutti i punti sulla retta che passa per l'origine e per p, quindi anche con l'origine).



Possiamo rappresentare punti nel paino $z = 0$? Se usassimo (x, y) per il piano cartesiano, chiamiamo il corrispondente in coordinate omogenee, o **homogeneous vector**, $\tilde{\boldsymbol{x}} = (\tilde{x}, \tilde{y}, \tilde{w}) = \tilde{w}(x, y, 1) = \tilde{w}\bar{\boldsymbol{x}}$, dove $\tilde{\boldsymbol{x}}$ è detto **agumented vector** e il piano di proiezione è $P^2 = \mathbb{R}^3 - (0,0,0)$. È importante perché ogni oggetto in 3D può essere priettato nel projective plane. Possiamo quindi rappresentare tutti gli oggetti. Ma arriviamo alla risposta alla domanda di prima: se devo convertire coordinate disomogenee in coordinate omogenee, possiamo aggiungere una dimensione con $z = 1$; invece se volessimo tornare in dietro, se w non è 0, allora possiamo dividere per w, ma se $w = 0$, non posso dividere per 0. Significa che un punto con $w = 0$, rappresenta un punto all'infinito e quindi i punti sul piano $z = 0$ rappresentano i punti all'infinito. Notiamo infine che in coordinate omogenee i punti sono *scale invariant*, per il motivo della correlazione spiegato prima.

Classe I: Euclidea (isometria)

Supponiamo di volere un algoritmo per traslare i punti. Applico una trasformazione lineare applicando la traslazione alle coordinate di un punto. Se però lavoro in uno spazio omogeneo, possiamo fare questa operazione con una moltiplicazione fra matrici:

$$\begin{bmatrix} X2 \\ Y2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X1 \\ Y1 \\ 1 \end{bmatrix}$$

Se vogliamo anche applicare una rotazione bisogna cambiare la matrice di identità inserendo seni e coseni di un angolo.

$$\begin{bmatrix} X2 \\ Y2 \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & dx \\ \sin\theta & \cos\theta & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X1 \\ Y1 \\ 1 \end{bmatrix}$$

In generale possiamo definire le isometrie in questo modo:

$$H = \begin{bmatrix} \varepsilon\cos\theta & -\sin\theta & t_x \\ \varepsilon\sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix} \quad con\ \varepsilon = \pm 1$$

Notiamo che abbiamo tre gradi di libertà: rotazione, traslazione su x, traslazione su y. Possiamo anche definirla come:

$$H = \begin{bmatrix} a & -b & t_x \\ b & a & t_y \\ 0 & 0 & 1 \end{bmatrix} \quad con\ a^2 + b^2 = 1\ e\ \begin{cases} a = \cos\theta \\ b = \sin\theta \end{cases}$$

Classe II: Similarità

È come il caso precedente, ma togliamo il vincolo $a^2 + b^2 = 1$, quindi otteniamo $\begin{cases} a = s \cdot \cos\theta \\ b = s \cdot \sin\theta \end{cases}$, s scalare. In questo modo possiamo anche scalare, quindi abbiamo un quarto grado di libertà. Questa trasformazione è *equiform*, perché preserva la forma.

Classe III: Affine

Permette anche di cambiare gli angoli. Le linee parallele però restano preservate. La matrice di trasformazione è:

$$H = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ 0 & 0 & 1 \end{bmatrix} \quad \forall a_{ij} \in \mathbb{R}$$

Classe IV: Omografia

È la trasformazione più completa, può fare tutto, preservando le linee dritte. La matrice di trasformazione è:

$$H = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \quad con\ h_{22} = 1\ oppure\ in\ generale\ h_{22} \neq 0$$

Algoritmi

Forward

Per ogni punto in input possiamo applicare la trasformazione diretta, calcolando la posizione di destinazione moltiplicando per la matrice di trasformazione. Questo metodo ha delle limitazioni, perché crea dei buchi, perché lo spazio di coordinate è intero, quindi il risultato della trasformazione potrebbe non essere intero, quindi il punto dovrebbe essere *splatted* fra i vicini e potrebbero esserci dei punti che non sono definiti.

Inverse

Si applica la trasformazione inversa, ovvero si prende ogni punto dell'immagine di arrivo e per ogni punto si prende il valore della sorgente sulla base della trasformazione inversa. Quindi per ogni punto di destinazione abbiamo il punto sorgente. In questo modo non ci sono buchi, è definita in tutti i punti e alla fine possiamo fare un'interpolazione finale.

Lecture #12

Thursday 26th March 2020 – Prof. Baraldi

Geometry

Le trasformazioni geometriche cambiano la posizione dei punti, e non cambiano (dal punto di vista concettuale) il contenuto dell'immagine. Si fanno con moltiplicazioni fra matrici: $f(x) = Tx.x$ è in coordinate omogenee e T è la matrice di trasformazione. Le due classi di trasformazioni che vedremo sono la affine (6 DoF) e la perspective (8 DoF). Ricordiamo che quando applichiamo la trasformazione alle immagini facciamo qualcosa che è discreto e quantizzato: se prendo tutti i pixel ed applico la trasformazione poi disegno la nuova immagine, non avrò dei buoni risultati, perché avrei più pixel trasformati nella stessa posizione, quindi dovrò gestire questa situazione; inoltre, potrei avere dei casi dell'immagine finale che non hanno un corrispondente nell'immagine iniziale. Quello che facciamo è applicare il *reverse mapping*: invece che iterare sui pixel nell'immagine iniziale, noi facciamo l'inverso, ovvero iteriamo sui pixel di output, applicando la trasformazione inversa, ottenendo il valore del pixel nell'immagine iniziale e applicandolo in output. Useremo $cv2.getAffineTransform(src, dst)$ e $cv2.getPerspectiveTransform(src, dst)$: OCV stima la trasformazione che porta dal caso iniziale all'output. Per applicare le trasformazioni usiamo $cv2.warpAffine$ e $cv2.warpPerspective$. Per stimare le trasformazioni in modo più robusto si usano altre funzioni che stimano la migliore trasformazione dato input e output. Per la perspective si usa RANSAC con $cv2.findHomography$. RANSAC è un algoritmo per stimare gli outlier: dato un set di punti corrispondenti, in cui qualche corrispondenza non è corretta, stima quali sono gli outlier e usa solo gli inlier per stimare la trasformazione. Prende dei sottoinsiemi di quattro punti, stima la trasformazione, la applica, calcola gli inlier e gli outlier e seleziona la migliore soluzione, ovvero quella che massimizza gli inlier.

Hough transform

È un algoritmo per trovare linee dritte oppure cerchi. Si chiama la funzione e si cercano gli iperparametri migliori. Dobbiamo innanzitutto applicare dell'edge detection, per definire meglio le linee dritte e i cerchi. Poi si chiamano le funzioni $HoughLines$ e $HoughCircles$ che ritornano una lista di linee e una lista di cerchi trovate.

Template matching

Vogliamo cercare un template all'interno dell'immagine. Funziona come la convoluzione: abbiamo un input e qualcosa che assomiglia al kernel (il template). Si prende il template, lo si mette in una sliding windows e si calcola una misura di similarità, che può essere MSE, SAD, MAD, ecc.

PyTorch (1)

Quando scriviamo una rete, non la otteniamo, ma ne definiamo solo il grafo. Non si usa PT in ricerca, perché ci sono cose che non funzionano e che non possiamo testare perché in esecuzione si compila in C++. Il tensore di PT è lo stesso di numpy (come concetto). Si differenzia perché può vivere su diversi device: quello di numpy rimane sulla CPU, mentre su PT può essere anche su GPU o altro acceleratore (come VSLA). Autograd è un componente che calcola i gradienti. Un modulo invece è un layer di una NN, che quindi può conservare uno stato e dei pesi imparabili. Tutto è un tensore.

Le size e le stride sono utilizzate per ottenere i dati dalla memoria fisica. Per capire di quante posizioni muoversi nella memoria fisica, dato $stride = S = [s_0, s_1]$ e un indice del fato del tensore 2x2 (che abbiamo preso come esempio) $tensor[i_0, i_1]$, la posizione in memoria si trova facendo $sum(S \cdot tensor[i_0, i_1]) = i_0 \cdot s_0 + i_1 \cdot s_1 = position$. Come viene calcolato lo stride? Se abbiamo un tensore con shape (n, c, h, w), lo stride sarà $[chw, hw, w, 1]$.

Dato un tensore 2x2, se mi sposto di una colonna, mi sposto nella memoria fisica di una posizione, mentre se le righe hanno due elementi, se mi sposto di una riga, mi sposto di due posizioni nella memoria fisica.

PT cambia lo stride per effettuare delle operazioni senza riallocare i dati, ad esempio quando facciamo slicing. PT crea un'astrazione di alto livello dove vediamo cose diverse, senza cambiare ciò che sta in memoria, ci dà solo una view del tensore originale. Quindi crea un nuovo tensore che punta agli stessi dati del tensore originale, ma ne cambia i metadati.

Se facciamo $tensor[1, :]$, creiamo un nuovo oggetto tensore con size 2, stride 1, e poi aggiunge un dato offset, uguale a 2, che vuol dire che inizieremo a legge dall'indice 2 in memoria. Se facciamo $tensor[:, 0]$, avremo size 2, stride 2, nessun offset perché partiamo dall'inizio.

Lecture #13

Friday 27th March 2020

Templates and shapes (2)

Che vantaggio abbiamo nel lavorare in coordinate omogenee, anziché in coordinate cartesiane? Ogni trasformazione diventa una moltiplicazione fra matrici, che è più semplice.

Se vogliamo avere un TM generale, dobbiamo creare un gran numero di trasformazioni (non parliamo di data augmentation). Per ogni possibile trasformazione del template e per ogni punto dell'immagine, calcolare il miglior NCC. A volte possiamo supporre che non ci sia rotazione. Si parla quindi di *multiscale NCC*.

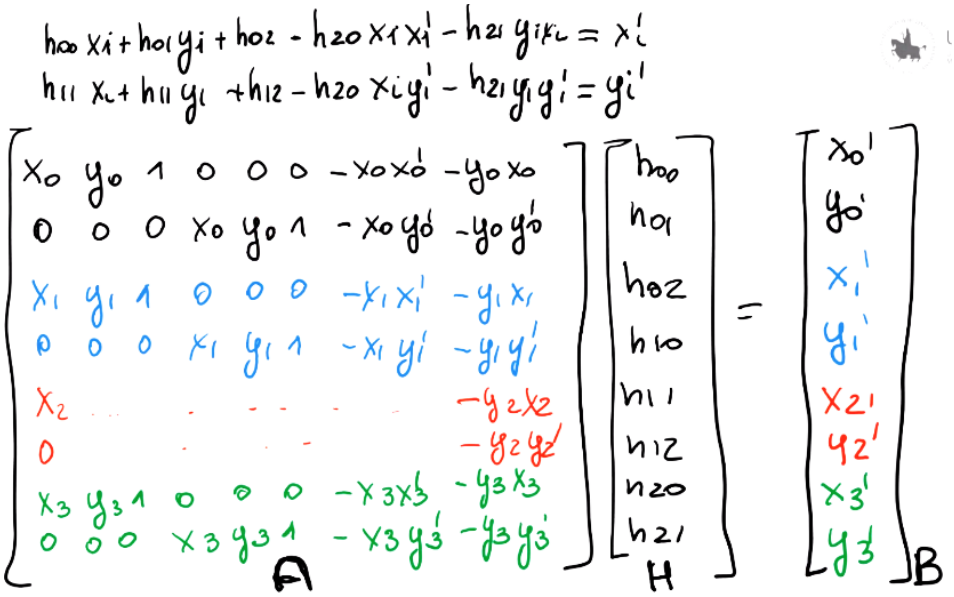
Tutto quello che abbiamo fatto in uno spazio 2D possiamo farlo anche in uno spazio 3D aggiungendo una dimensione (quindi la matrice di trasformazione avrà una dimensione in più).

In 3D possiamo applicare la traslazione in modo analogo al 2D, ma la rotazione è diversa, perché potremmo ruotare attorno all'asse x, y, o z. Lo scaling può essere su tutti gli assi. L'affine è una matrice 3x4. La perspective è una matrice 4x4, dove l'ultimo elemento deve essere diverso da 0.

Possiamo fare trasformazioni composite moltiplicando le matrici di trasformazione fra di loro. Inoltre, avendo un oggetto in 3D possiamo proiettarlo in 2D applicando queste trasformazioni.

So volessimo capire quanto è alta una persona, potremmo cercare dei punti su cui fare una trasformazione e ottenere una *bird view*. Si prendono quattro punti che non sono allineati e che si trovano sul pavimento e se sappiamo la misura della mattonella, allora possiamo ricostruire l'altezza di una persona sdraiata per terra. Il piano del pavimento risulterà non distorto, ma tutti gli altri piani saranno distorti.

Come si può trovare la matrice di trasformazione date due sorgenti (immagini)? Si fissano almeno quattro coppie di punti corrispondenti nelle due immagini e si risolve un sistema per risalire agli elementi della matrice di trasformazione. Il risultato finale è:



Hough transform

L'**Hough transform** è molto utilizzata. Per ora siamo ancora agnostici della semantica (non sappiamo cosa c'è nell'immagine), ma sappiamo che cerchiamo qualcosa di dritto, rotondo, quadrato, ecc. Come possiamo trovare un modello geometrico? Una possibilità è utilizzare **RANSAC (RANdom SAmple Consensus)**. Cerca di trovare un buon fitting dei dati: se abbiamo dei dati, cerchiamo fra questi dei dati che rappresentino bene la figura geometrica cercata. RANSAC è però molto complesso computazionalmente, quindi nelle immagini è difficile da utilizzare, troppa complessità computazionale. Quindi si usa HT che è meno complesso (quadratico e non cubico). Solitamente prima si fa edge detection e poi si applica la HT. Solitamente in CV si definisce HT una qualsiasi trasformazione che va dallo spazio delle coordinate a uno spazio parametrico dove la detection è più facile.

HT per le straight lines: se abbiamo un modello rappresentabile geometricamente come $f(\boldsymbol{x}, p) = 0$, dove $\boldsymbol{x}$ è la coordinata dei punti e p è il set di parametri, allora possiamo usare la HT. Una linea retta può essere rappresentata come $y = mx + c$. Con dei punti, si cerca una trasformazione in cui i punti, trasformati in un nuovo spazio, sono delle rette che passano per lo stesso punto. I punti iniziali sono nella forma $\boldsymbol{x} = (x, y)$, mentre nello spazio parametrico di arrivo sono nella forma $p = (m, c)$, quindi sono delle rette. Se i punti del piano cartesiano sono allineati, allora nello spazio parametrico votano per lo stesso punto, ovvero sono delle rette che si intersecano nello stesso punto. In generale, nello spazio di arrivo ogni retta vota per i punti per cui passa, e se più rette passano per un punto (cioè si intersecano in un punto), allora votano per quel punto.

Il problema della rappresentazione $y = mx + c$ è l'infinito, quindi si usa $x \cdot \cos(\theta) + y \cdot \sin(\theta) = r$, la classica rappresentazione parametrica di una linea, dove r è una distanza trigonometrica, quindi può essere anche negativa. Perché può esserlo? Se abbiamo due linee parallele allora hanno lo stesso angolo. Con questa nuova trasformazione, i punti iniziali nello spazio di proiezione non sono più delle rette, ma sono delle sinusoidi, che passano per lo stesso punto nel pianto di proiezione (che è (r, θ), con r sia positivo che negativo e $\theta \in [0{,}180]$).

Algoritmo:

- Quantizzare lo spazio dell'HT, ovvero identificare il minimo valore di r e θ. Se un'immagine è $N \times N$, dobbiamo trovare lo spazio di arrivo, che è una matrice di accumulazione con dimensioni che dipendono dal valore massimo e minimo di r e θ. θ va da 0 a 180, ma r? È il doppio della diagonale dell'immagine ($2N\sqrt{2}$), perché r è la distanza dall'origine del piano (r, θ), e la massima distanza che possiamo avere dall'origine è quella della diagonale, raddoppiata perché può essere anche negativa. Infine, a seconda della precisione che vogliamo, dividiamo le dimensioni in bin più o meno grandi.
- Creiamo una matrice/array di accumulazione inizializzata a 0.
- Per ogni punto dell'immagine, se il punto è d'interesse (edge), quel punto voterà per tutti i punti della sinusoide nel piano di proiezione. Votare vuol dire incrementare di 1 il bin di arrivo nello spazio di proiezione (nella matrice di accumulazione).

HT non fornisce la localizzazione, ci dice solo i parametri della linea. Il vantaggio è che si trovano le linee anche se sono occluse. Si può fare anche per i cerchi, usando l'equazione di un cerchio, oppure una proiezione. Si volta per un cono di punti. Si può usare anche per parabole ed ellissi.

Si può creare una **Hough transform generalizzata** che possa trovare una qualsiasi forma nell'immagine? È un approccio basato sul ML, però noi non abbiamo molti esempi: come facciamo con una sola immagine? Per ogni punto dobbiamo capire l'angolo fra la normale e la distanza dal centro. Dopo creiamo una hash table: per ogni angolo abbiamo più di un punto con una certa distanza.

- Per ogni punto computiamo la direzione del gradiente
- Accumuliamo nell'accumulatore il possibile raggio e il possibile centro a seconda dell'immagine.
- Si vota quindi per il centro.

Lecture #14

Tuesday 31st March 2020

Camera Model (1)

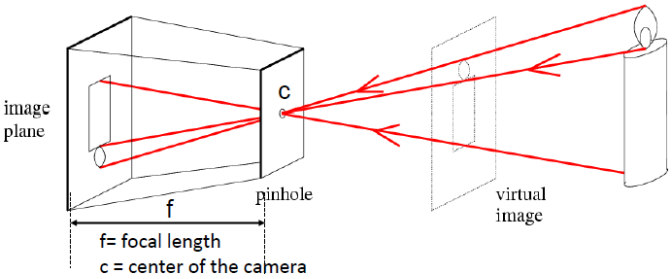
Vogliamo partire da qualcosa nel mondo 3D e poi ottenerne una rappresentazione 2D. La camera usa CCD o CMOS, riceve la luminanza e crea il piano dell'immagine. Come possiamo partire dal piano dell'immagine e misurare qualcosa nel mondo 3D?

Quando abbiamo una camera, dobbiamo tenere conto:

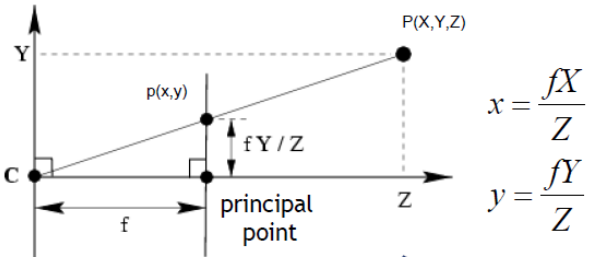
- Della geometria dell'immagine: dov'è il piano dell'immagine.
- Della fisica della luce: come possiamo ricevere queste informazioni.

Supponiamo che la luminanza arrivi solo da un punto, e che questo punto illumini un oggetto riflettente: una parte della radiazione è assorbita, una parte è radiata con uno specifico angolo e parte dell'irradianza arriva al sensore.

Quando abbiamo un oggetto che irradia, e la radiazione arriva in un film della camera, in genere il film dovrebbe diventare tutto bianco, sovraesposto. Per non avere questo risultato, si aggiunge una barriera con un piccolo buco (**pinhole model**). Se il buco è così piccolo per far passare solo un raggio di luce, allora sul film arriva solo un raggio e si forma l'immagine inversa e non siamo sovraesposti. Questo modello è alla base dalla CV. Se abbiamo una candela nel piano 3D, questa se è illuminata irradia da tutte le parti. I raggi passano per il pinhole che è il **centro della camera**, e risultano ribaltati nel piano dell'immagine. La distanza fra il piano dell'immagine e il pinhole si chiama distanza focale. Noi non considereremo l'image plane, ma il **virtual image**, che è esattamente uguale, ma ribaltato (ovvero dritto, non capovolto).



Come possiamo partire da un punto (x, y) nel piano dell'immagine e tornare indietro nel piano 3D (X, Y, Z)? Supponiamo di avere un triangolo nello spazio 3D, che viene proiettato nell'image plane. Il **centro della camera** C viene chiamato anche O, ovvero **optical flow**. Chiamiamo principal point p il centro della virtual image, quindi il punto della virtual image che si trova dall'intersezione con l'optical axis. f è la distanza focale e il piano dell'immagine è π. In genere, supponiamo di avere un virtual plane con distanza $-f$ dal centro e che non sia ribaltato.



Se abbiamo un punto P nello spazio 3D, questo punto è proiettato nel piano virtuale nel punto p. Notiamo che P rappresenta tutti i punti sulla retta che passa per O e per P (stiamo appunto parlando del piano di proiezione). Il rapporto fra P e p possiamo farlo con la similarità: $y: f = Y: Z$.

Se vogliamo calcolare y, usiamo $\frac{y}{f} = \frac{Y}{Z}$. Il problema è che non conosciamo né f, né Z. Infatti, se vedo un pixel nell'immagine noi non possiamo capire la distanza che abbiamo da quel pixel.

Spesso si parla di **normalized camera model**, quando si suppone che $f = 1$. Però $f \neq 1$, perché è deciso dalla casa produttrice della camera. Il camera model è il modello classico con distanza focale f, e qualche volta si usa il modello normalizzato con $f = 1$, ma spesso si usa la **orthographic perspective**: è simile alla projective perspective, ma con una differenza, ovvero se un oggetto non è molto distante dalla camera e ΔZ (variazione di Z fra il punto dell'oggetto più vicino e il punto più lontano rispetto alla camera) è trascurabile rispetto alla distanza dalla camera $\left(\frac{\Delta Z}{Z_0} \ll 1\right)$, allora possiamo considerare che Z sia costante per tutti i punti dell'oggetto.

Il modello generale è il modello perspective: abbiamo una trasformazione da $[x, y, z, 1]$ a $[x', y', 1]$. Perché la matrice di trasformazione è 3x4? Perché passiamo dal 3D al 2D. Il problema è che i valori della matrice sono sconosciuti.