All'inizio c'è la parte dei simplessi e delle superfici.

***Processo acquisizione**: acquisisci misure geometrice tramite surface scanning -> allinea i dati da scan differenti, merge dei dati per costruire una sola superficie, poi finalizzazione (mesh)*

***Processo geometrico**: dai dati hai dei possibili errori, analizzi la qualità della superficie per rimuovere il noise. Semplifica per complexity reduction, ri-mesha per aumentare la qualità*

Le superfici sono regioni 2D in uno spazio 3D dove stai sostanzialmente eliminando una dimensione, la profondità. Alcune superfici le puoi rappresentare analiticamente tramite formule geometriche (vedi la sfera, il thorus, i cubi, etc.), altre (che poi sono quelle realmente presenti nel mondo reale) hanno delle forme più complesse. L'idea è quindi di vedere queste forme come una collezione di questi "building blocks" che noi abbiamo chiamato Simplessi.

K-simplesso: superfici convesse formate da k+1 punti indipendenti, chiamati vertici.
Un punto è uno 0-simplesso, una linea è un 1-simplesso, un triangolo è un 2-simplesso, etc.

Se mettiamo più k-simplessi insieme, otteniamo un **Complesso** (simpliciale). Questo è un insieme finito di simplessi che s'incontrano lungo una faccia comune, dove per "faccia" di un k-simplesso si intende un altro simplesso che è formato da un sottoinsieme dei vertici del k-simplesso.
Quindi tipo, in un triangolo, le facce saranno i lati o anche i vertici.

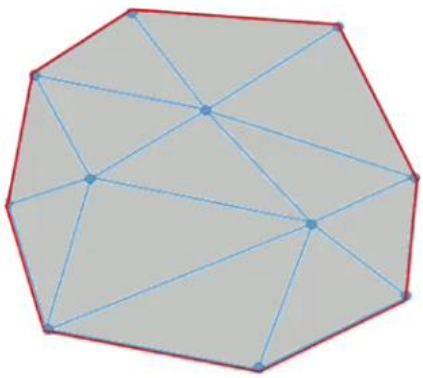
Un complesso si dice omogeneo se è formato solamente da k-simplessi.
Quindi se hai soltanto dei triangoli e non ulteriori punti/linee, quello è un complesso omogeneo.

Definito il complesso, si può dare anche la nozione di **Mesh** (le triangolari, che sono le più utilizzate). Essa è un complesso omogeneo 2-dimensionale (ovvero formato da triangoli).

Sulle mesh (e sui simplessi) abbiamo definito alcune quantità

Il **Contorno** di un simplesso normale, è l'insieme delle facce proprie di un simplesso
in un triangolo, il contorno sarà formato dai suoi lati.

Il **Contorno di una mesh triangolare** è un simplesso che si ottiene facendo la somma modulo 2 dei contorni di ognuno dei simplessi che lo formano.
In pratica, vai a prendere un lato sì ed un lato no di tutti i quanti i simplessi che formano la mesh.



La **Star** di un simplesso è l'insieme (unione) dei simplessi che contengono un certo simplesso.
La stella di un punto sono tutti quanti i triangoli che hanno quel punto in comune.

Il **link** è l'insieme di tutte quante le facce dei simplessi che stanno all'interno della stella ma che non intersecano il simplesso stesso.

Se consideri un punto sarebbe il contorno della stella, perché non conti gli archi che effettivamente toccano quel punto lì.

The *star* of a simplex Δ is the union of all the simplices containing Δ

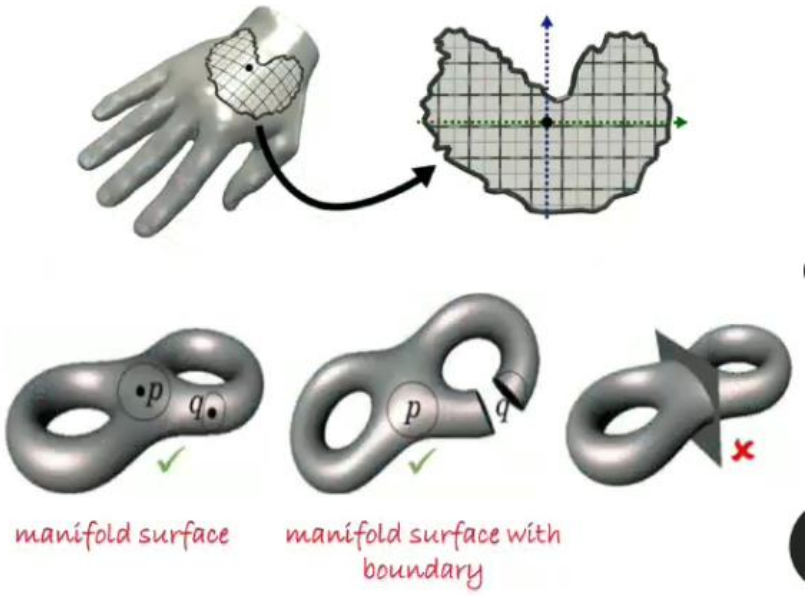
The *link* of Δ consists of all the faces of simplices in the star of Δ that do not intersect Δ

(a) A 0-simplex, (b) its star and (c) its link

Perché abbiamo definito queste quantità?

Perché entrano in gioco quando parliamo delle **Manifold** sulle superfici. La **Manifoldness** (in italiano, la "varietà") è uno spazio topologico in cui il vicinato di ogni punto è un piccolo pezzo di spazio Euclideo.

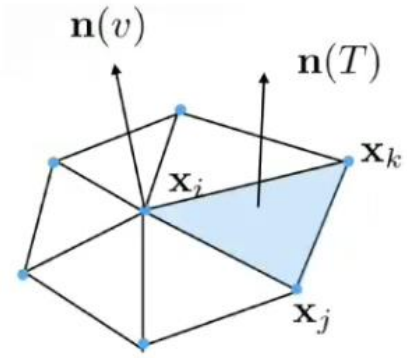
Ad esempio, una manifold per una retta è una curva, mentre una manifold per un piano è una superficie.



Le manifold di dim. 2 e 3 sono modelli matematici convenienti per la rappresentazione degli oggetti 3D.

In breve, li abbiamo introdotti perché per poter fare alcune operazioni sulle mesh (e.g. shading), abbiamo bisogno di una quantità fondamentale: la normale su un vertice. Come si calcola il vettore normale di una mesh? Facendo la somma pesata delle normali di ogni triangolo che forma la mesh. Ad esempio, in alcune delle tecniche di shading, si fanno le normali sui vertici (e.g. Gourad shading). Per calcolare una normale su un vertice, calcoli la somma pesata di tutti quanti i vettori normali della stella di quel punto.

$$\boldsymbol{n}(v) = \frac{\sum_{T \in N(v)} \alpha_T \boldsymbol{n}(T)}{\left\| \sum_{T \in N(v)} \alpha_T \boldsymbol{n}(T) \right\|}$$

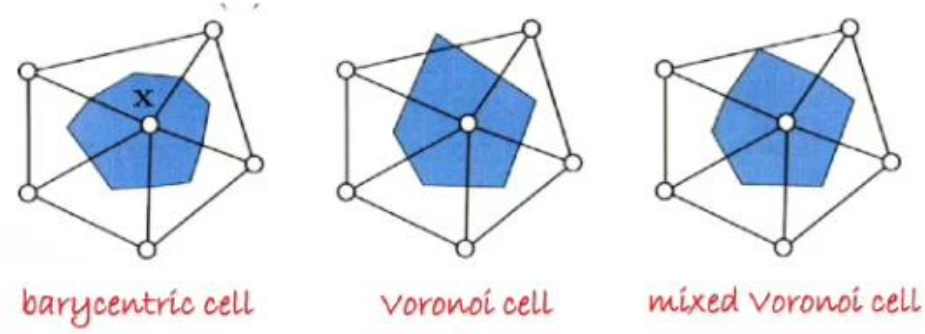


Hai la normale calcolata sul vettore sul punto V, tutto intorno ci sono triangoli, dovrai calcolare tutte quante le normali di tutti quei triangoli perché tutti fan parte della stella del vertice al centro.

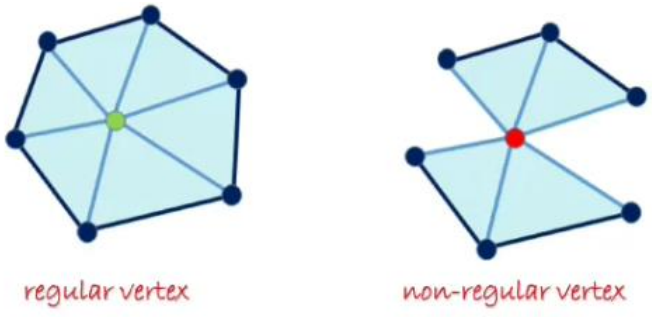
Il motivo della somma pesata è che con pesi costanti potresti ottenere risultati contro-intuitivi.

Ora, non è detto che questa somma pesata sia necessario farla su tutta l'area del triangolo: la puoi fare anche in diversi modi, tramite le **local averaging regions:**

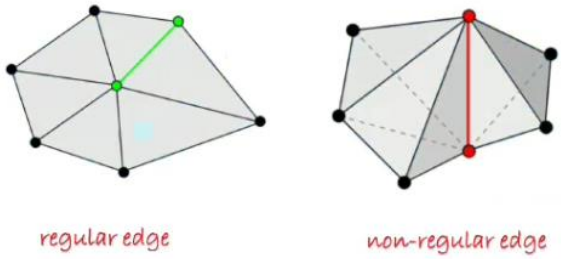
- **Baricentrico**: collega i baricentri dei triangoli con i punti medi dell'arco
- **Voronoi cells**: sostituisce i baricentri con i circocentri
- **Mixed Voronoi**: per i triangoli ottusi, usa il punto medio dell'arco opposto al vertice centrale



Per definire la manifoldness sulle mesh triangolari inoltre ci serve una nozione di "buon comportamento", simile al caso continuo. Un punto (0-simplesso) è **regolare** se è un vertice "interiore" o se sta "sul" boundary, altrimenti è non-regolare.



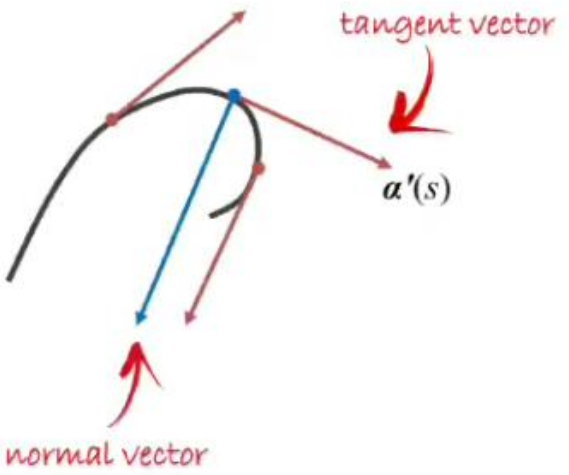
Seguendo la stessa logica per un segmento (1-simplesso):



Tornando a quel che dicevamo, abbiamo visto quindi che un esempio di "varietà" per una retta è una curva. Una **curva**, nel 2D, è come una mappa per un intervallo di numeri reali in uno spazio euclideo.

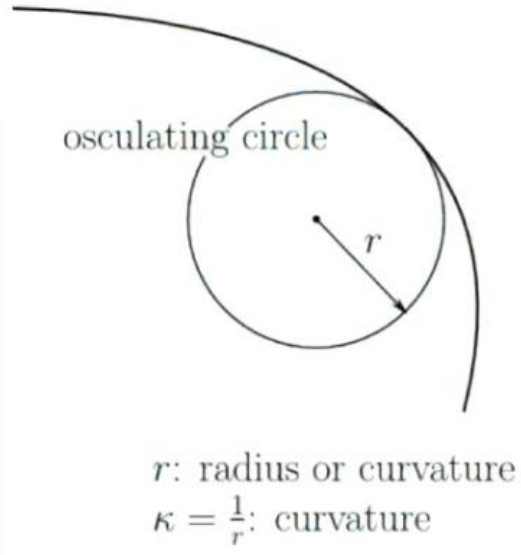
Il **vettore tangente** **alla curva** in un punto è la derivata di questa funzione di mapping in quel punto e ci dà la direzione e la velocità del movimento lungo la curva in quel punto.

Il **vettore normale** invece ci mostra quanto fortemente una curva devia dalla linea "dritta".



Se ruotiamo di 90° il vettore tangente, otteniamo il vettore normale.

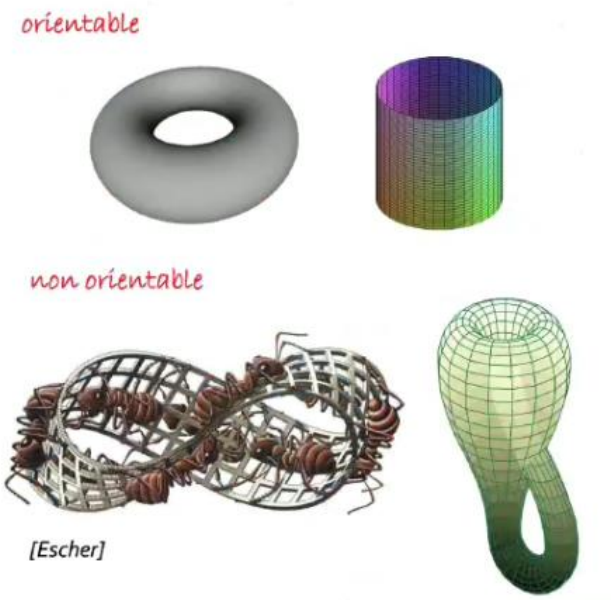
La **curvatura** di una curva è definita come l'inverso del raggio dell'oscurating circle (che è il cerchio tangente alla curva che la approssima al meglio).



Per arrivare ad una definizione di curva e curvatura sulla superficie (-> varietà) dobbiamo definire anche il **piano tangente** in un punto, che è il piano che viene attraversato dai vettori tangenti alle curve che passano da quel punto: così ottieni un piano.

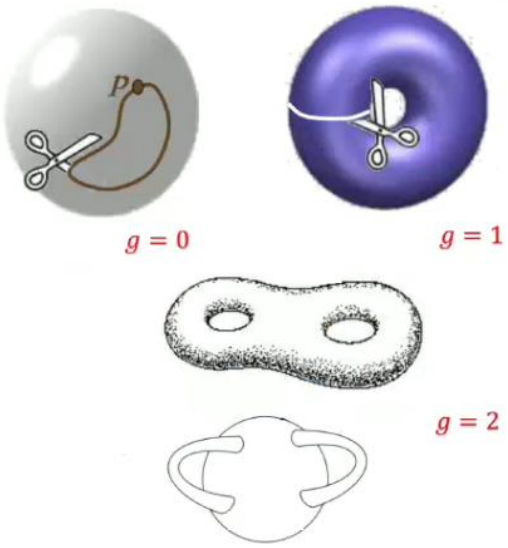
Il **vettore normale** è il vettore ortogonale al piano tangente.

Una superficie si dice **orientabile** se puoi orientare le normali di tutti i punti in modo consistente, altrimenti è **non-orientabile**.



La sfera ed il Thorus sono orientabili, perché nella sfera le normali le orienti appunto sempre verso l'esterno. Se consideri la bottiglia di Klein (?) oppure il nastro di Mobius non riesci a dargli un orientamento consistente, cioè non sai mai verso dove girare le normali.

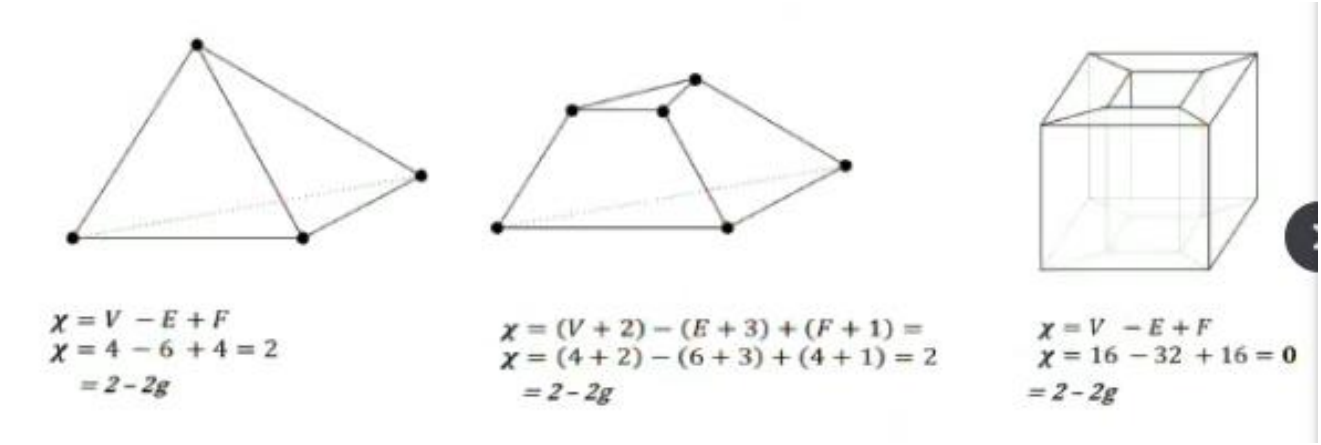
Altre quantità che abbiamo definito sulle superfici sono il **Genus** che ci dice il numero di tagli che possiamo fare prima di rendere disconnessa una superficie orientabile e connessa.



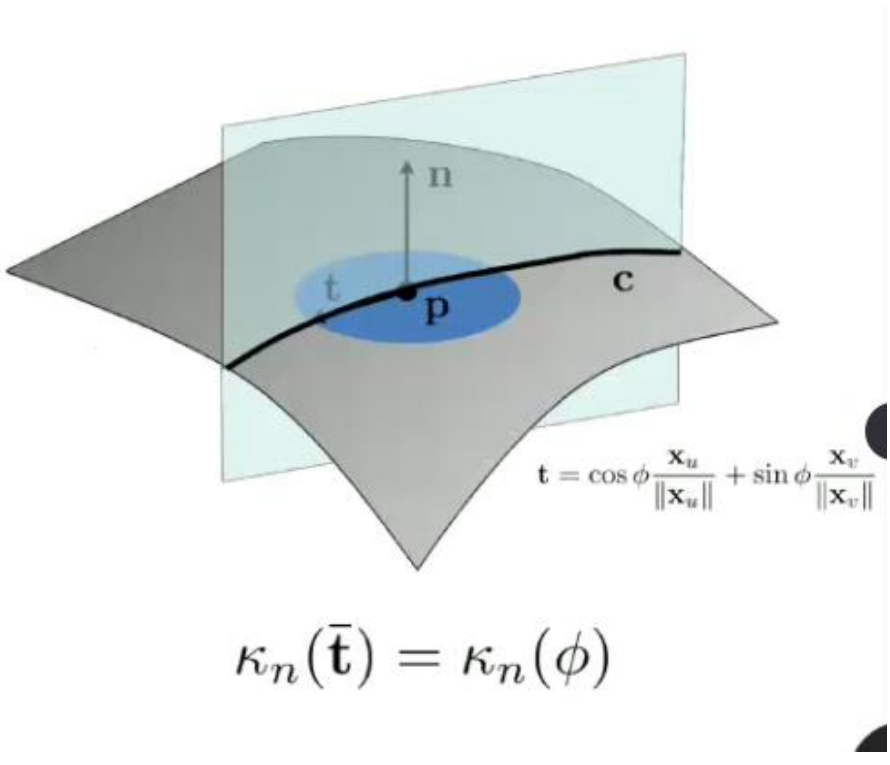
Per esempio, su una sfera è 0, su un Thorus è 1, etc.

Il Genus è topologicamente invariante (cerca GIF da Thorus a Tazza)

Poi abbiamo la **caratteristica di Eulero**, che è un numero che si calcola come **#vertici - #archi + #facce**, che è anche collegata al Genus perché è uguale anche a 2-2*Genus *[- #boundaries]*



Ora passiamo da curva e curvatura planari (2D) a curva e curvatura sulle superfici (3D), quindi estendendo il concetto alle superfici. Per farlo, parliamo di **curvatura normale** in un punto p, ed è la curvatura della curva planare che si ottiene intersecando la tua superficie con un piano che è attraversato dalla tangente e dalla normale del punto p. Essa ha un segno: è positiva se le curve vanno nella stessa direzione della superficie normale, negativa altrimenti.



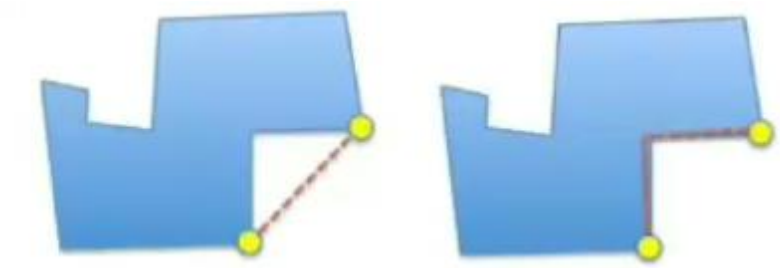
C'è una superficie ed il punto p. Calcoli il piano tangente al punto p, cioè il cerchio blu. Quel piano verrà definito da una tangente t e da una normale n (per due vettori passa un solo piano), e per quei due vettori fai passare questo piano che intersecherà la tua superficie. Intersecandola, viene fuori una certa curva, quella nera spessa dell'immagine sottostante. La sua curvatura è la curvatura normale nel punto p.

Le curvature normali sono fondamentali perché ci definiscono due valori estremi, detti **curvature principali**: la **curvatura massima** e la **curvatura minima**. Da queste due otteniamo due altre quantità altrettanto importanti che sono:

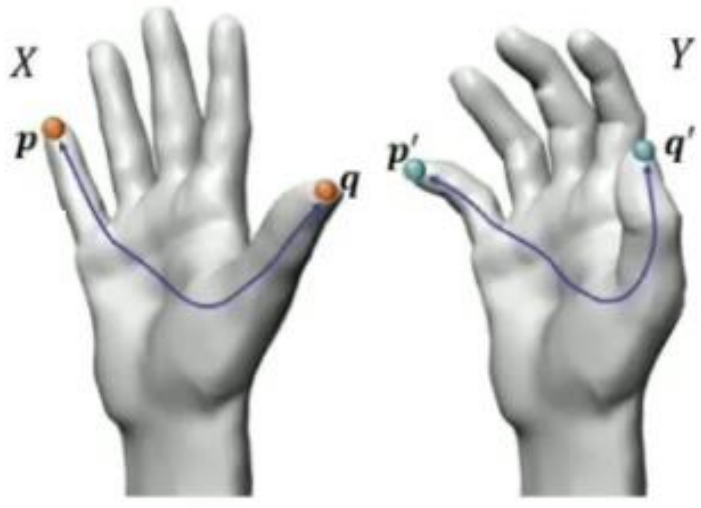
- La **curvatura media** (la media aritmetica dei due valori, è l'OR, (k1+k2)/2) ci serve a dire se una superficie è minimale. Se questa media è 0 ovunque, allora è una superficie minimale.
- La **curvatura gaussiana** sarebbe la moltiplicazione delle due curvature principali (l'AND, k1*k2) e ci aiuta a classificare 3 categorie di punti (vedi immagine sotto)

Se tutti i punti sono parabolici, possiamo dire che la superficie è "sviluppabile".
La curvatura gaussiana è utilizzata per fare "visual inspection", tipo per verificare la qualità delle superfici.

Poi, un'altra cosa importante è il prossimo concetto di distanza. Solitamente noi consideriamo la distanza nello spazio euclideo, cioè la lunghezza della linea che collega due punti. Se volessimo estendere questo concetto alle superfici, si parla di **distanza geodesica:** considera il percorso più breve che c'è tra due punti in una superficie.

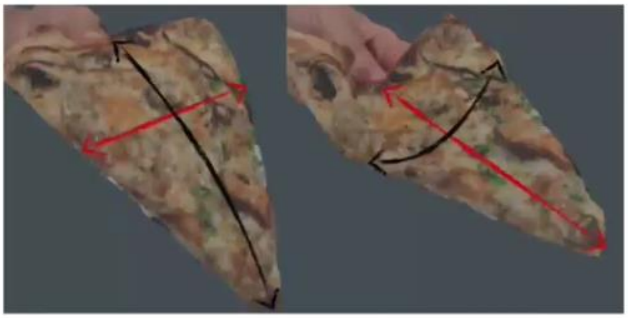


Questa è una quantità intrinseca, nel senso che non dipende dalla superficie o da come è fatta (se è curva, etc) e questo fatto che si preservi la distanza nonostante le trasformazioni delle superfici si dice isometria (o trasformazione isometrica).



La distanza tra p e q rimane invariata a prescindere dalla forma presa dalla mano.

Le trasformazioni isometriche sono anche alla base del **teorema Egregium**, che stabilisce che anche la curvatura gaussiana è una proprietà intrinseca, perché possiamo ottenerla calcolandola con 2 dimensioni su 3.



Questo perché è possibile? Se prendi ad esempio l'esempio del planisfero, noterai che ci sono delle proiezioni distorte. Questo perché quando vai a calcolare la curvatura gaussiana causerai una distorsione derivata dal fatto che stai cercando di preservare la distanza reale che c'è tra due punti e di trasportarla sul piano (da un geoide [la Terra] ad un piano): quindi stai facendo una trasformazione isometrica perché appunto devi mantenere invariata questa distanza. Poiché non c'è alcuna isometria tra la sfera ed il piano, allora avviene questo fenomeno di distorsione.

Poi ci sta la parte sulle strutture dati

Una volta che hai definito cos'è una mesh, bisogna vedere cosa si può fare con le strutture dati.
Quale mesh è più vicina ad un certo punto? Quali elementi sono dentro una regione? Quali sono intersecati da un raggio r? Queste due mesh s'intersecano? Se sì, dove? Etc.

Per rispondere efficientemente a queste domande si parla di "Spatial Search Data Structure".

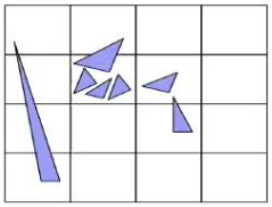
Ci sono due tipi principali di strutture dati: quelle **gerarchiche** e quelle **non-gerarchiche** (flat-based).

Quelle non gerarchiche sono:

- **Uniform Grid**: una griglia di celle dove ad ogni cella è assegnata una o più primitive. Questa è la più facile da utilizzare: se cerchi qual è la primitiva più vicina ad un punto, parti dalla cella che contiene il punto e poi ti espandi "a cerchi crescenti" andando verso l'esterno fino a toccare tutte le celle, cercando qual è la prima che incontri (crescendo ogni volta toccando una nuova riga/colonna). Se vuoi fare l'intersezione con un raggio, fai passare questo raggio per la griglia: per ogni cella che viene intersecata, *(n.b. intersezione raggio-cella più semplice di intersezione raggio-primitiva)* testi le primitizze intersecate e ci metti un flag su ognuna per evitare di testarle multiple volte. I pro sono che sono facili da costruire e da farci query sopra. Il contro è che occupi molta memoria **O(#celle + #primitive).**

Uniform Grid

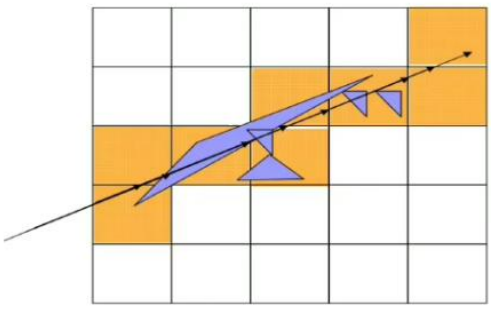
- **Description**: the space including the object is partitioned in **cubic cells**; each cell contains references to "primitives" (i.e. triangles)
- **Construction**.
  Primitives are assigned to:
  - The cell containing their feature point (e.g. barycenter or one of their vertices)
  - The eventually multiple cells spanned by each primitive



Lesson 4 – Spatial Indexing

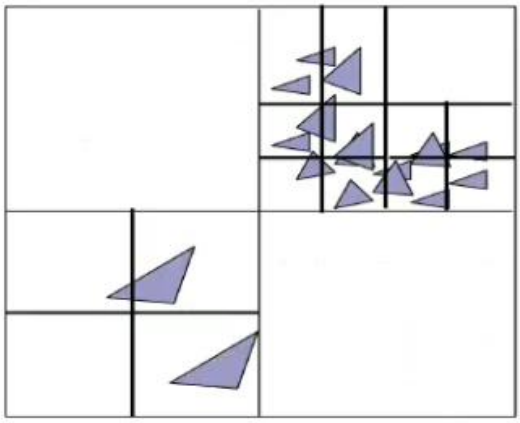
Uniform Grid

- **Intersection with a ray:**
  - Find all the cells intersected by the ray
  - For each intersected cell, test the intersection with the primitives referred in that cell
  - Avoid multiple testing by flagging primitives that have been tested (mailboxing)
- **Cost:**
  - Worst: $O(\#cells + n)$
  - Average: $O(\sqrt[d]{\#cells} + \sqrt[d]{n})$



- Per migliorare l'occupazione di memoria (che è il difetto del precedente), abbiamo considerato lo **spatial hashing**. Sarebbe la uniform grid, ma dove togli le celle vuote ed inoltre per ogni cella non-vuota calcoli l'hash di quella cella in modo tale da poterci accedere il più facilmente possibile: ovviamente ci possono essere collisioni. Come la precedente, anche questa (e pure di più), soffre per come sono predisposte le primitive nello spazio (averle sparpagliate, spatial hashing o uniform grid è praticamente la stessa cosa, in quanto non hai troppe celle vuote -> usalo solo se te ne aspetti molte). Quindi stessi pro, meno consumo di memoria, ma al caso pessimo O(#celle) per via delle collisioni.

Poi l'altro tipo di strutture, quelle gerarchiche, si tratta di dividere lo spazio in più sottoregioni.

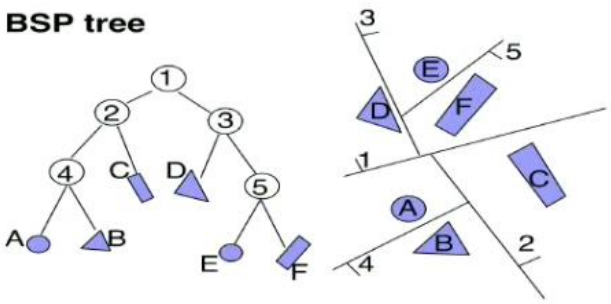


Classico Divide Et Impera: dividi lo spazio in sottooregioni ricorsivamente. Fare una query significa visitare un albero (e quindi complessita logaritmica e occupazione di memoria lineare)

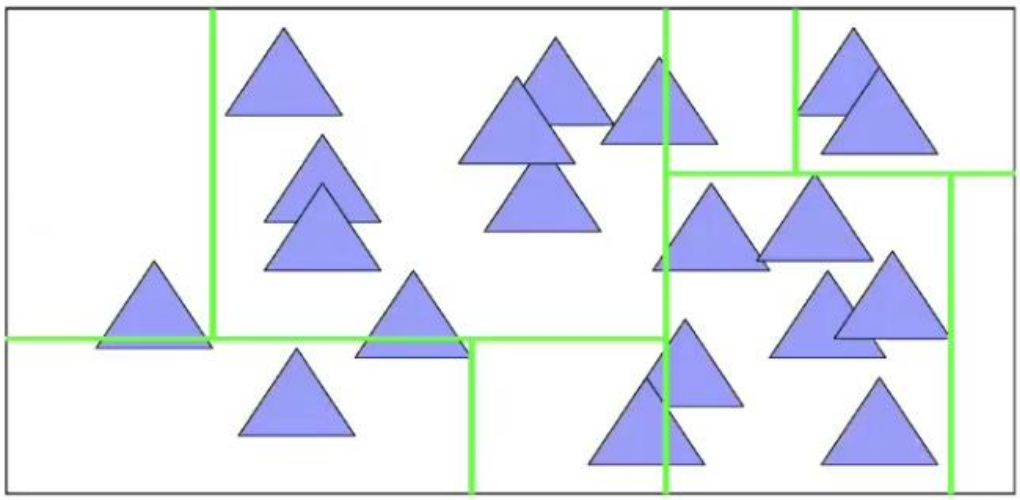
- Nel **Binary Space Partition-Tree** (BSP) dividi lo spazio sempre in due sottoregioni convesse tramite un iperpiano. La cosa più difficile di questa struttura è capire appunto come e dove far passare questo iperpiano. Una tecnica che si utilizza spesso è di far passare l'iperpiano proprio attraverso le primitive, in modo tale da dividere la primitiva stessa in parte sinistra e parte destra. Però non è l'unica strategia: di solito cambia rispetto all'applicazione che vogliamo fare (e.g. vuoi miglior balance? minimizzare numero splits?..?). Al caso pessimo, costa O(n), al medio O(log n).

Binary Space Partition-Tree (BSP)

- **Description**:
  - It's a binary tree obtained by recursively partitioning the space in **two** by a hyperplane
  - Therefore a node always corresponds to a **convex region**



- Un altro tipo è il **KD-Tree**, che è come un BSP speciale, dove i piani che separano tutta la superficie sono sempre paralleli ai due assi. Questo tipo di struttura dati è ottima per fare le "Range Query". Ovviamente l'algoritmo è semplice: mi calcolo l'intersezione tra il nodo e il box. Se il nodo sta completamente all'interno della box, aggiungo tutte le primitive. Se è completamente fuori, ritorni, se è parzialmente dentro, ricorsione al figlio. Costo O(n^(1-1/d) + k)

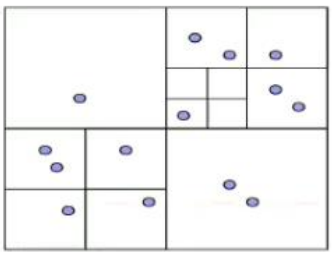


PROS: è estremamente veloce ad esplorare l'albero e consuma ancora meno memoria

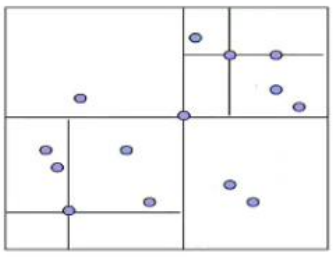
- Poi c'è il **Quad-Tree**, dove dividi il piano sempre in 4 sottoregioni tramite una coppia di piani ortogonali. Il centro di questa coppia può passare o dal centro della superficie che stai dividendo (Region Quad-Tree) oppure può passare attraverso un punto particolare (Point Quad-Tree), esempio uno dei punti della superficie o un punto delle primitive sulla superficie. Questa struttura dati si può estendere anche nel tridimensionale ed in questo caso si chiama Oct-Tree.

The plane is recursively subdivided in 4 subregions by couple of orthogonal planes

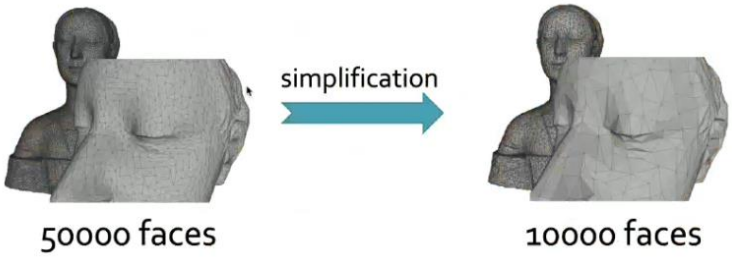
Region Quad-tree



Point Quad-tree



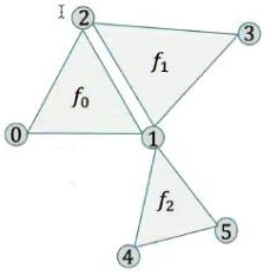
Ora vediamo per quanto riguarda l'effettivo SALVARE le mesh in memoria



Mesh semplification: *dando in input una forma complessa, vorremmo ottenere in output un complesso che rappresenta la stessa forma ma con meno celle.*

La più semplice è la **Indexed Data Structure** in cui ogni faccia immagazzina un puntatore a tutti i suoi vertici.

Indexed Data Structure

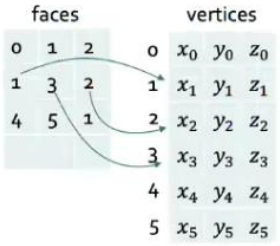



```
struct Face{
   int N;
   int vert[N];
};
struct Vertex{
   float pos[3];
};

vector<Face> faces;
vector<Vertex> vertices;
```

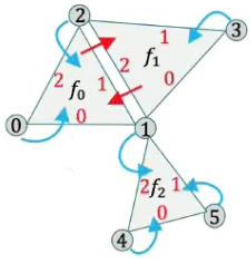
Operation	*Time*
Find subfaces	O(1)
Find co-faces	O(f*d)
VStar	O(f*d)
Vertex removal	O(f*d)
Face removal	O(1)
Size (trimesh)	$3f \approx 6v$

- Each face stores a pointer to each of its vertices
- It is the layout most commonly used to store the mesh on disk and/or in GPU memory for rendering



Poi, la **struttura dati indicizzata con le adiacenze:** come prima ogni faccia ha i puntatori ai suoi vertici, ma stavolta anche un puntatore a tutte le facce ad essa adiacenti.

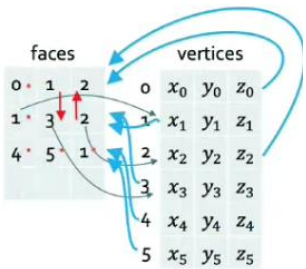
Indexed Data Structure with Adjacencies




```
struct Face{
   int N;
   int vert[N];
   int adjFaces[N];
};
struct Vertex{
   int oneAdjFace;
   float pos[3];
};
vector<Face> faces;
vector<Vertex> vertices;
```

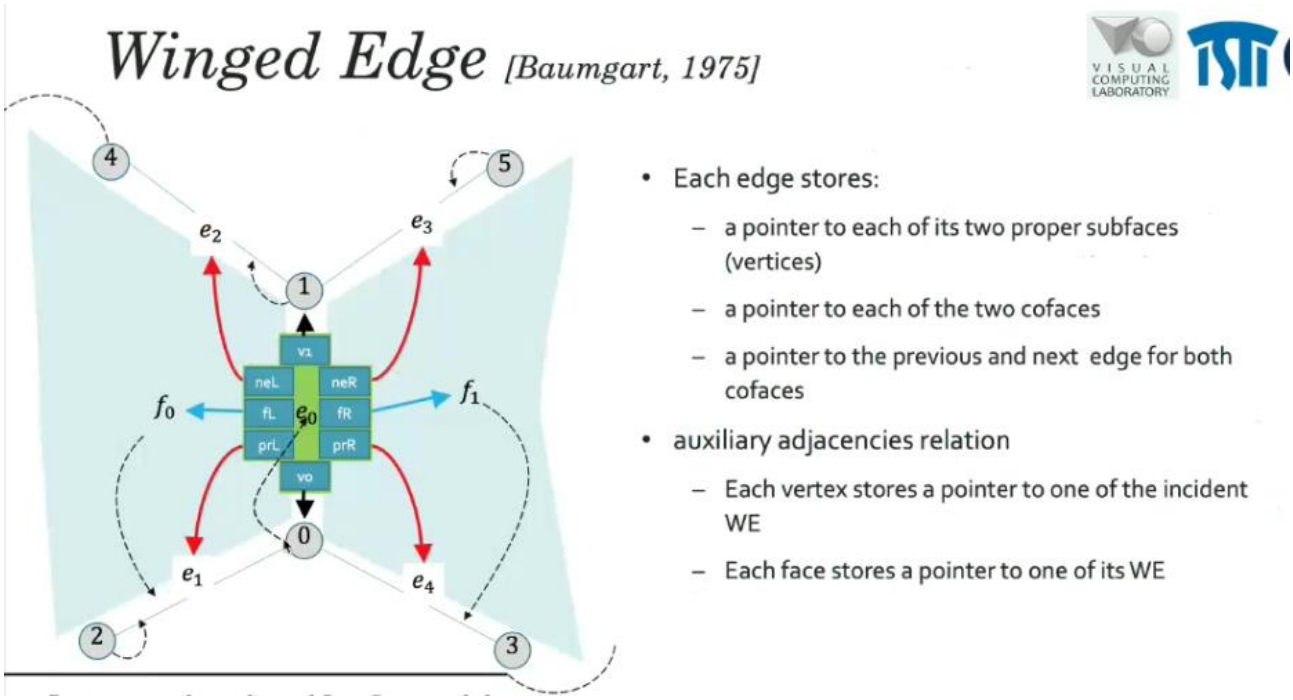
Operation	*Time*
Find subfaces	O(1)
Find co-faces	O(1)
FStar	O(d)
Vertex removal	O(d)
Face removal	O(1)
Size (tri meshes)	$(3+3)f + v \approx 13v$

- Each face stores:
  - a pointer to each of its vertices
  - a pointer to each of its adjacent faces
- Each vertex stores a pointer to one of its adjacent faces

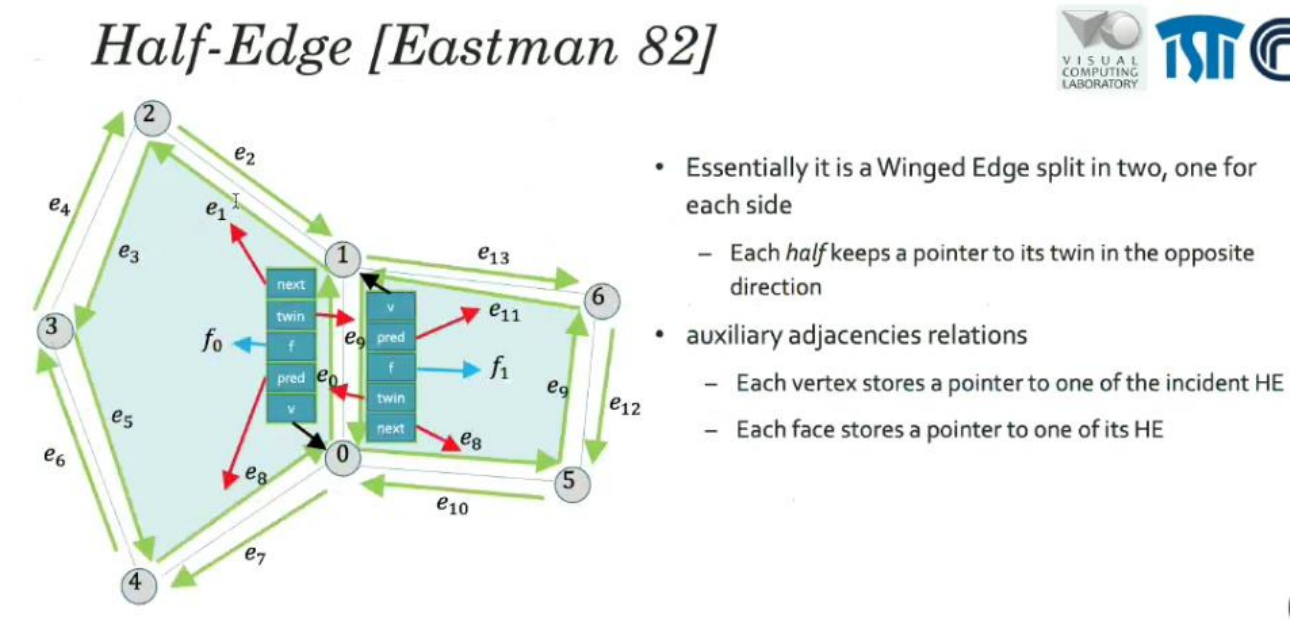


f0 ha il puntatore a f1 e viceversa. Inoltre, ogni vertice ha un puntatore ad una delle facce adiacenti (le frecce azzurre).

Un'altra struttura dati è la **Winged Edge**, in cui il soggetto che vai a salvare non è "la faccia con i puntatori" ma è un arco (o più precisamente gli archi). Gli archi li vedi come una serie di puntatori: due puntatori ai vertici che lo formano, due puntatori alle facce che hanno in comune l'arco stesso (ricordati che sei sempre su una mesh triangolare) 2 puntatori sia per il successivo che per il precedente dgli archi di entrambe le facce.



Se il Winged Edge lo spezzi a metà, quindi ti salvi tutte quante le informazioni del vertice di un lato dell'arco da una parte, e tutte le altre in un'altra, si sta parlando di **Half-Edge**. Entrambi hanno costo O(d) per le query che ci interessano (trova sottofacce, co-facce, rimuovi vertici, rimuovi facce, etc) ma la Winged occupa leggermente meno spazio.

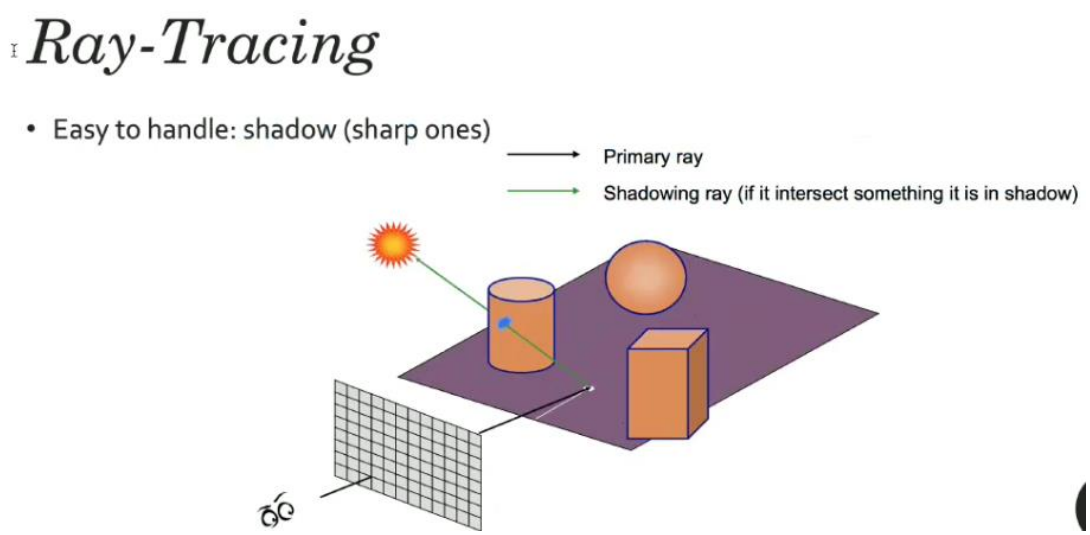


RENDERING PARADIGM

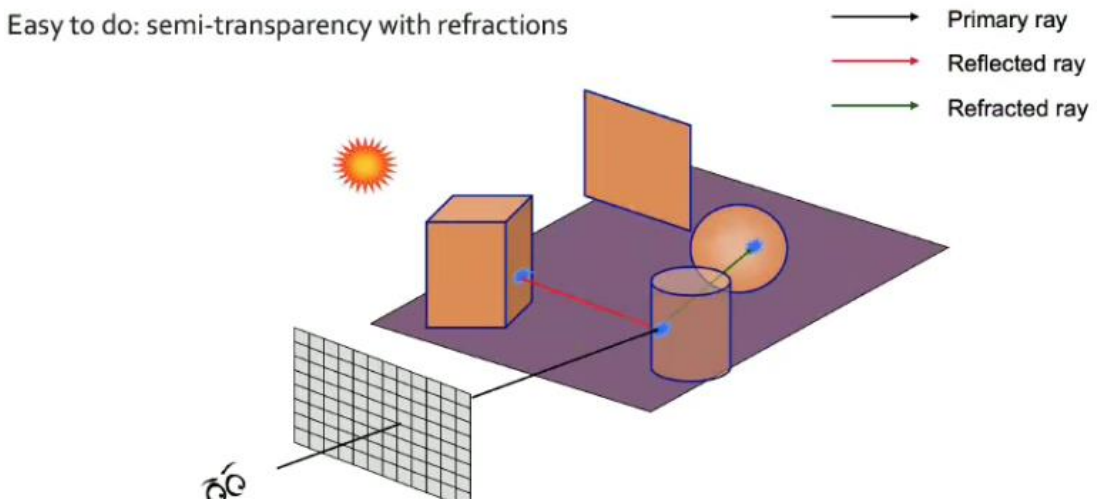
Il rendering è quel processo che fa passare dall'avere dei dati all'avere un'immagine. Ci sono diversi passaggi da fare, cioè la pipeline 3D (o la rendering pipeline), cioè tu vai dall'Object Space -> Word Space -> View Space -> Clip Space (se ne riparla dopo).

*Gli algoritmi di rendering sono di due famiglie: **ray tracing** e **rasterization-based**.*

Gli algoritmi di **RayTracing** si basano sull'avere questi raggi luminosi, quindi per ogni pixel "lanci" un raggio luminoso che dovrà "tornare indietro" alla sorgente di luce, in modo tale da capire quale primitiva interseca per prima.



Ci sono poi altri raggi per modellare altri effetti quali: riflessione (vedere come rimbalza questo raggio tra le varie primitive, per mostrare per esempio la luce riflessa), rifrazione (per modellare oggetti che possono essere [semi]trasparenti).



In generale comunque, questi algoritmi hanno come parte complicata il calcolare questa intersezione tra il raggio 3D e le primitive 3D, però abbiamo visto le strutture dati apposite per farlo in maniera efficiente.

In generale, l'algoritmo fa che "Per ogni pixel, genero un raggio e poi cerco le intersezioni con ogni primitiva, poi faccio il lighting".

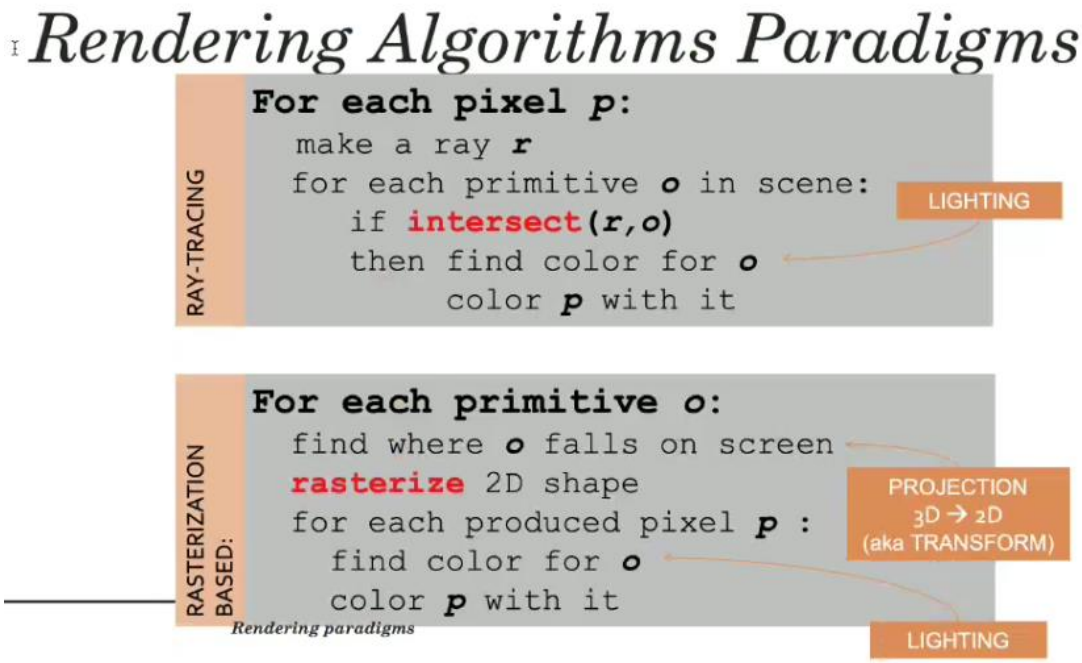
Invece, nel **Rasterization-Based**, si fa "il contrario". Si parte dalle primitive, si va a vedere dove queste primitive finiranno sullo schermo facendo rasterizzazione. Trasformo dal 3D al 2D in pratica, poi

questo triangolo lo divido in frammenti. Questi frammenti poi li processi appunto per ottenere i pixel finali (gli puoi applicare una texture, li puoi colorare, lighting, etc).

Nei rasterization based quindi "per ogni primitiva, la trasformi in 2D e poi per ogni pixel che hai generato fai lighting".

Quindi è meglio for each pixel (for each primitive) o for each primitive (for each pixel)?
Il primo (ray tracing) è concettualmente semplice, tiene conto degli effetti globali, fa un effetto di luce più realistico, c'è più libertà delle primitive (perché intersecare è più facile di proiettare+rasterizzare) e ha una alta scalabilità con la complessità della scena. *Usato nei film, immagini statiche, offline rendering.. è Software-based (CPU (o anche Specialized-HW per avere complessità sublineari -> velocità)*
Il secondo (rasterization) è più facilmente parallelizzabile, controlla meglio la complessità, è più facile con dati dinamici, etc. *Usato per i giochi, real-time rendering, … è Hardware-based (GPU). Anche questo può simulare il primo (come il contrario) usando algoritmi avanzati etc.*

"Rasterization is fast but needs cleverness to support complex visual effects. Ray tracing supportss complex visual effects but needs cleverness to be fast" – D. Luebke (NVIDIA)

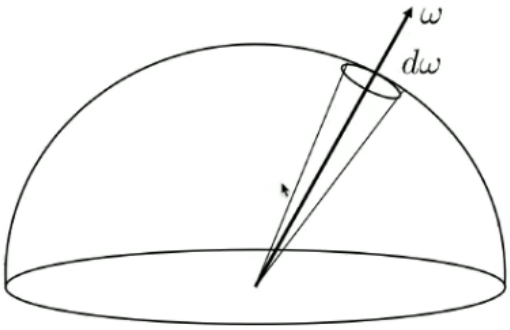


Come abbiamo detto sia per il raytracing che per il rasterization-based, l'ultima fase è il **lighting**, cioè scegliere il colore finale di ogni parte della scena. Ora, esistono diversi modelli di luce e noi abbiamo analizzato in particolare quello a raggi ottici: la luce viene modellata come dei raggi che seguono regole geometriche [legge di Fresnel e legge di Lambert, che ci permettono di modellare rispettivamente la riflessione e la rifrazione].

Per arrivare all'equazione di rendering completa (luce in un certo punto data da una certa direzione è = luce emessa in quel punto&direzione + luce riflessa in quel punto&direzione). Prima di ciò, serve di parlare di angolo solido (l'angolo planare, ma sulle superfici). Si definisce come "la dimensione angolare del conoide infinitesimale lungo una certa direzione", oppure come la direzione associata ad un punto di una sfera.

Solid angle

- It represents the angular dimension of an infinitesimal conoid along a given direction.
- It can be interpreted as a direction associated to an infinitesimal area on the unit sphere (unit measure: *steradians*).



Hai un punto sulla sfera, colleghi il punto al centro della sfera, quella direzione definita da questo segmento è l'angolo solido.

Poi, c'è da ricordarsi che l'intensità della luce emessa si riduce col quadrato della distanza (serve sempre per la rendering equation).

Radiometry

- Point light source emits the light uniformly in all directions producing:

$$E(x) = \frac{\Phi_s \cos\theta}{4\pi r^2}$$

$\Rightarrow$ The light intensity reduces with the square of the distance (!)

Quindi la sorgente di luce è un punto, questa sorgente emette luce uniformemente in tutte le direzioni: più ti allontani, più quest'intensità della luce diminuisce, con che rapporto? Il quadrato della distanza.

Ora, abbiamo detto come funziona l'angolo e come funziona la luce emessa.. Parliamo della luce riflessa: abbiamo visto la **BSSRDF (Bidirectional Scattering surface reflectance distribution function)** che sarebbe la funzione che definisce lo scattering, cioè il fenomeno per cui una luce che colpisce una superficie con una certa posizione, la lascerà in una posizione diversa da quella entrante. A noi interessa un'approssimazione di questa funzione, la **BRDF** che dice quanta luce riflessa c'è in un punto, data una certa direzione (che è proprio quel che ci serve nella rendering equation!).

$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega}_r) + L_r(x, \vec{\omega}_r)$$

Luce totale = luce emessa + luce riflessa.

La luce emessa si è detto che si può calcolare rapportata alla distanza dalla sorgente, mentre la luce riflessa la calcoliamo come integrale pesato di tutte quante le applicazioni di questa funzione BRDF ai diversi angoli di riflessione.

$$L_o(x, \vec{\omega}_r) = L_e(x, \vec{\omega_r}) + \int_{\Omega} f_r(x, \vec{\omega_i}, \vec{\omega_r}) L_i(x, \vec{\omega}_i)(\vec{\omega}_i \cdot \vec{n}) d\vec{\omega}_i$$

Però la rendering equation è troppo complicata per fare lighting per ogni primitiva. Quindi sono stati introdotti i **lighting model**, cioè modelli che mirano a semplificare/approssimare la rendering equation. Noi abbiamo visto il **Phong Lighting Model** (1975). Abbiamo sempre che la sorgente di luce è un punto, però abbiamo diverse semplificazioni quali:

- non è modellata l'inter-riflessione (quindi puoi interagire solo con materiali opachi)
- non si modella la rifrazione (non puoi interagire con materiali [semi]trasparenti)

L'approssimazione più forte però è che approssimi la rendering equation con due costanti: una per la **riflessione speculare** (Fresnel), ed una per la **riflessione diffusa** (Lambert). La legge di Fresnel, in questo contesto dove non abbiamo la rifrazione, ci dice che il raggio che incide su una superficie senza rifrazione (== no dispersione d'energia) lascerà la superficie con stessa energia e con un angolo uguale. Riguardo Lambert, dice in pratica la stessa cosa di Fresnel ma per delle superfici che hanno delle piccole microsfaccettature: per approssimazione, la luce in questo caso viene riflessa uniformemente in tutte le direzioni. Come si approssimano queste due quantità?

La riflessione diffusa si approssima con una costante che dipende dal materiale e dipende dall'angolo che c'è tra la normale nel punto in cui stiamo considerando e la direzione d'incidenza della luce.

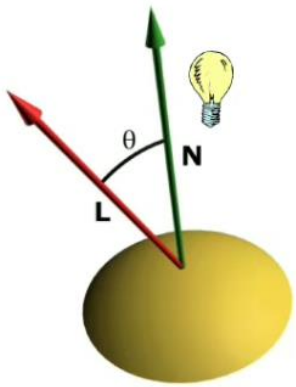
Diffuse Reflection

- The reflection function is approximated as a constant k_d which dependes on the material
- Lighting equation (diffuse):

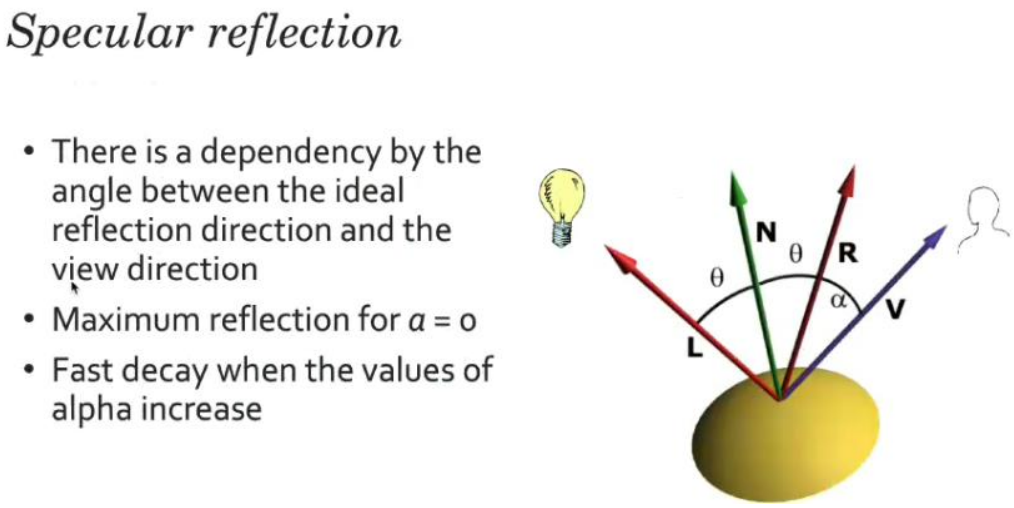
$$I = I_p k_d \cos\theta$$

or

$$I = I_p k_d (\vec{\mathbf{N}} \cdot \vec{\mathbf{L}})$$



Invece, la riflessione speculare è un'altra costante che però stavolta dipende anche dalla direzione dell'osservatore. Che significa? Quando si approssima la legge di Fresnel, si scopre che c'è questo effetto di Specular Highlight: la quantità di luce riflessa in un punto dipende da dove stai guardando. Si è scoperto che c'è questa dipendenza con l'angolo che c'è tra la riflessione e l'osservazione (vedi figura)



Il parametro n (specular reflection exponent) dipende del materiale
L'equazione approssimante è: $I = I_p k_s \cos^n \alpha$

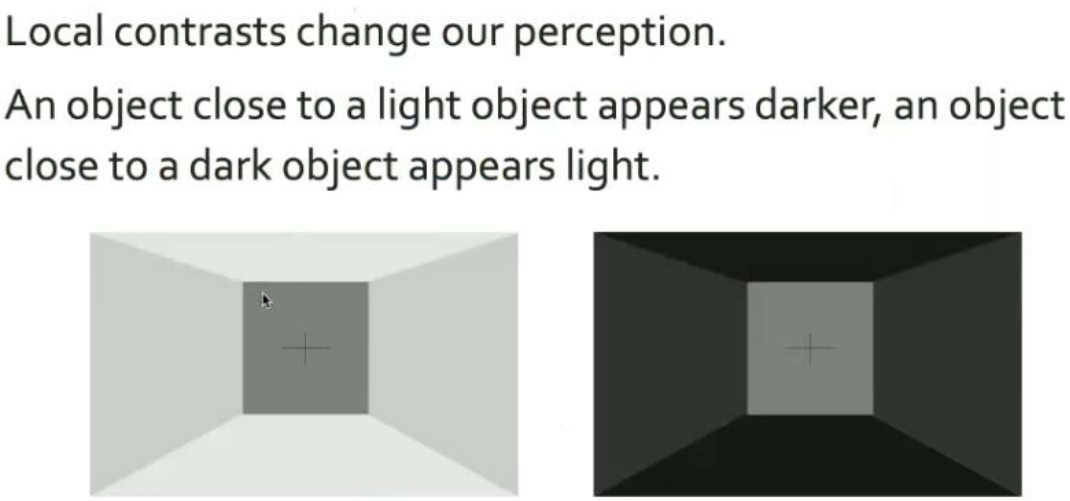
La lampadina che genera luce, ha direzione d'incidenza che sarebbe la L. Con lo stesso angolo Theta, si genera R, cioè la direzione di riflessione. L'angolo che si viene a creare tra R e V, sarebbe la direzione dell'osservatore, cioè Alpha, il coseno alla n di quell'angolo è proporzionale alla quantità di luce riflessa speculare.

Un'altra costante che si può aggiungere è quella per modellare l'inter-riflessione (nel Phong model non c'è, però se si introduce questa costante la si può approssimare), semplicemente con $I = I_a k_a$. Anche questa costante dipende dal materiale ed è costante in tutti i punti della scena. Mettendo insieme queste parti (la componente ambientale che modella l'interriflessione, la componente diffusa e la componente speculare per ogni punto) si ottiene l'illuminazione della scena.

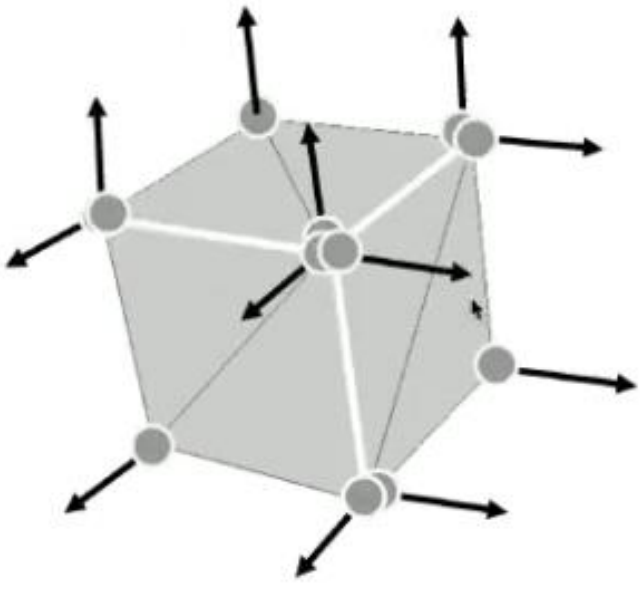
Se invece volessimo proprio togliere la rendering equation, per calcolare l'interazione tra la luce ed il materiale si parla di **Shading**. Esistono diverse tecniche per farlo:

- la più semplice è il **flat shading** in cui la superficie viene discretizzata. Cioè, la superficie sarà fatta di facce e andiamo a colorare ogni faccia di un colore.

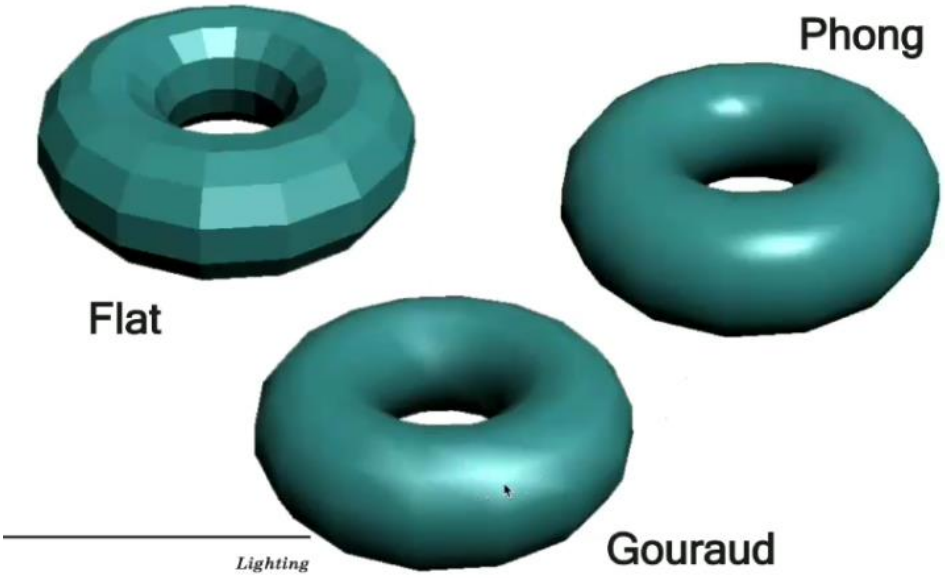
Oltre al problema della non continuità della superficie, si va anche incontro all'effetto di **Mach Banding:** ovvero che mettere vicini delle cose scure e delle cose chiare, il contrasto si noterà molto di più (il grigio centrale dell'immagine sottostante è lo stesso, ma sembra diverso!).



- Per ovviare, si parla di **Gourad Shading**. Qui non si calcola la luce su tutta la superficie, ma la si calcola solo in alcuni punti e poi fa interpolazione lineare durante la rasterizzazione. Come fa? Si prende la media delle normali sulle facce incidenti sui vertici (quella tecnica di shading che si diceva all'inizio per cui servono le normali). Quindi ti calcoli le normali sui vertici, poi fai interpolazione di queste normali e scopri la quantità di luce che avrai sulla faccia. Questo però introduce il problema del calcolare la luce sul vertice stesso: se considerassi la media pesata delle normali blabla, scopriresti che sui vertici, le normali è come se andassero verso l'esterno.. Quindi ti verrebbero illuminati i vertici, e tutto il resto no. Quindi cosa si fa? C'è la versione "**migliorata**" del Gourad Shading in cui poiché un vertice è condiviso tra più facce, tu ti devi calcolare lo shading per ogni faccia, per ogni faccia ti calcoli una normale diversa (vedi immagine). Anche questo ha un problema, ovvero che non si comporta bene con lo specular highlight (l'effetto secondo cui la luce riflessa cambia rispetto al punto d'osservazione). Questo perché facendo interpolazione è come se andassi ad espandere questo effetto su tutta la faccia, e quindi non è realistico.



- Per ovviare anche a questo, abbiamo visto il Phong shading, dove calcoli la normale in ogni punto e poi fai interpolazione delle normali però ovviamente è più precisa perché lo fai punto per punto (pixel per pixel).



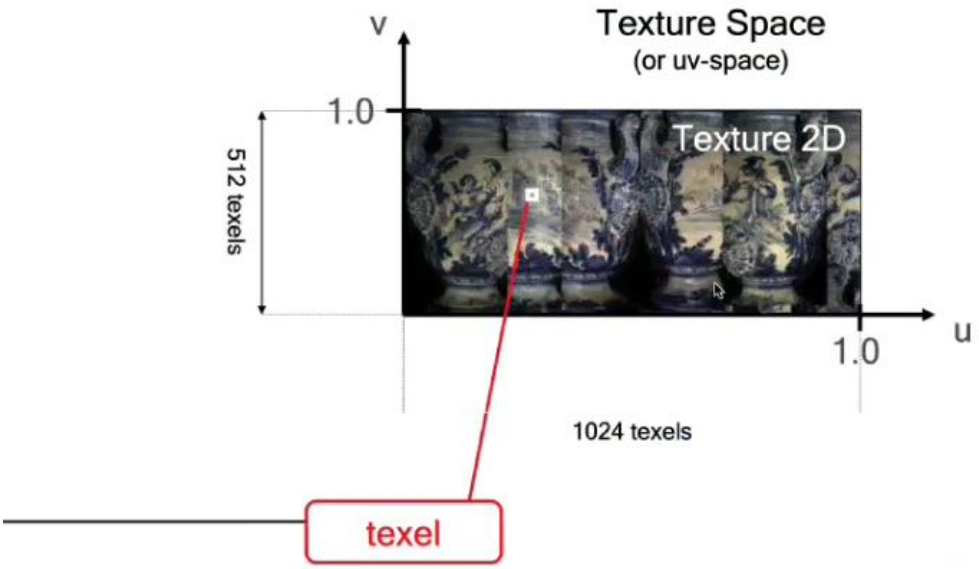
Gourad: solo alcuni punti, poi interpolazione. Phong punto per punto, quindi più realistico però più costoso.

TEXTURING

Quando dobbiamo modulare l'attributo di un vertice per ottenere un effetto visivo si parla di texturing. Se l'attributo è il colore, si parla di Texture Mapping.

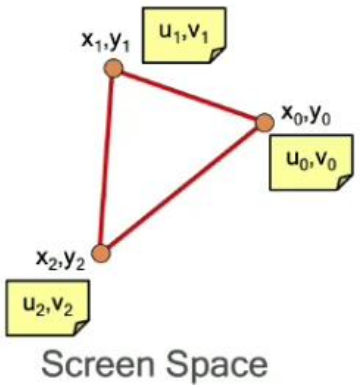
Partiamo da cos'è una Texture: un array di texels (1D, 2D o 3D) salvata nella texture RAM. Il **texel** è il sample di una texture (come il pixel è il sample di una picture). Il texel ha delle coordinate definite nel Texture Space (chiamato anche uv space): questo spazio è simil-piano cartesiano, ma è normalizzato a 1.

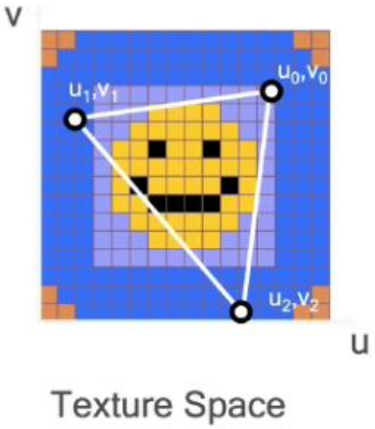
Texture Space (also called uv space)



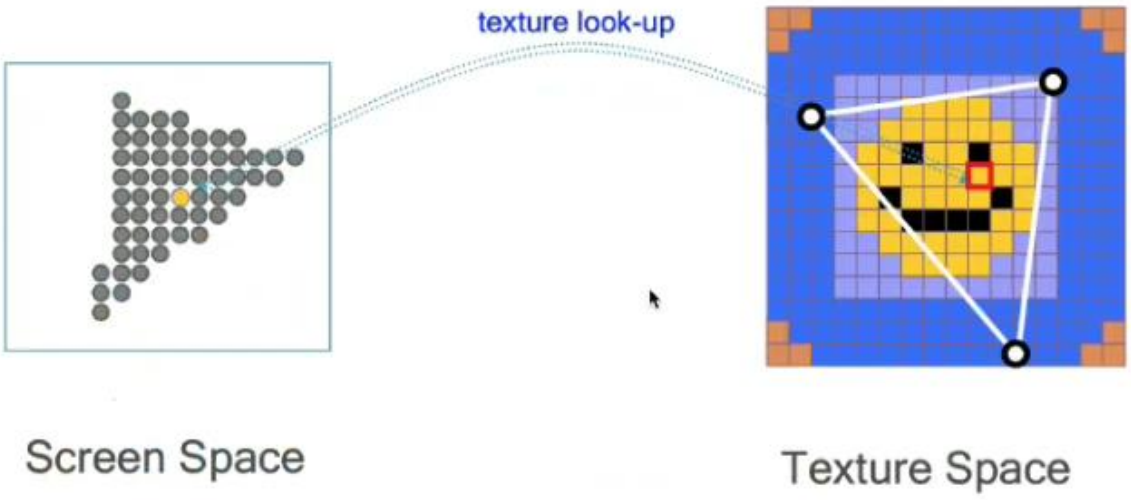
Come si fa il texture mapping? In due passaggi. Il primo è di rapportare le coordinate che hai nello Screen Space (lo spazio delle primitive) nel texture space (lo spazio uv), mentre il secondo è il Texture Mapping.

- The coordinates in uv space are assigned to each vertex.





Una volta definito ciò, si fa la frammentazione come facevamo nella rasterizzazione: si definiscono i pixel e poi si fa texture lookup (ovvero si colorerà ogni pixel in base alle sue coordinate all'interno della texture).



Quindi il primo passaggio è di passare dalle coordinate dello Screen Space alle coordinate del Texture Space: ci sono diversi modi di farlo.

Automatic creation of texture coordinates

- Planar projection (along the x axis)

$$(x, y, z) \mapsto (u, v) : \begin{cases} u = z \\ v = y \end{cases}$$

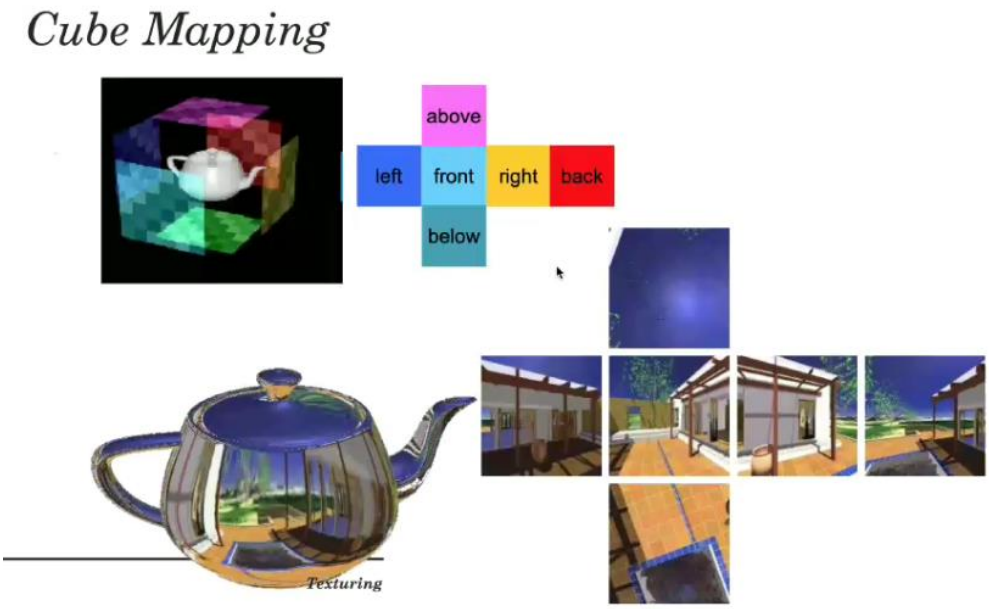
- Cylindric projection:

$$(x, y, z) \mapsto (u, v) : \begin{cases} u = \text{atan2}(z, x) \\ v = y \end{cases}$$

Note: the (u,v) coordinates have to be normalized in the [0, 1] interval

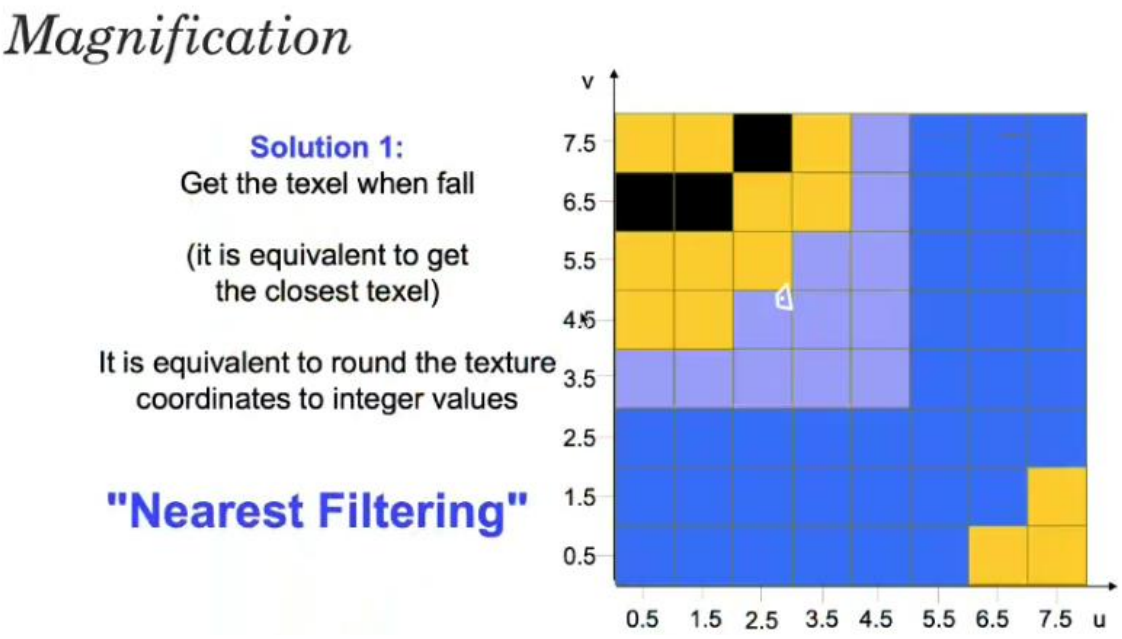
Puoi avere per esempio una proiezione planare, da cui passi dall'avere xyz all'avere uv (sostanzialmente elimini una dimensione). Oppure una proiezione cilindrica, l'arcotangente tra l'angolo tra Z e X.

Un modo carino che abbiamo visto è lo **Sphere Mapping**, dove si passa da coordinate cartesiane a coordinate sferiche, cioè è come usare questa Environment Map, cioè una texture che ti rappresenta il colore che assegneresti ad una scena in cui hai un ambiente riflesso (come nell'effetto del Fish Eye). Oppure il Cube Mapping, in cui la texture è formata da un cubo (6 facce) ed ogni faccia la applichi ad un lato della tua primitiva.



Un'ultima cosa da dire su questo spostare le texture da uno spazio all'altro è come comportarsi quando le coordinate escono fuori dalla texture. **Clamp mode** (allunghi gli ultimi texel di una texture all'infinito) oppure la **Repeat mode** (ripeti la texture all'infinito in tutte le direzioni che ti servono). Con quest'ultima sei ovviamente efficiente in termini di spazio (stessa texture per più primitive), ma potrebbe causare il Tiling (mettere la stessa texture affiancata, se non fatta bene, può creare un brutto effetto di discontinuità).

La seconda parte del Texture Mapping è il Texture Lookup, ovvero quando devi colorare ogni pixel. Quindi per ora hai creato i tuoi frammenti, e ora dovrai calcolarne i colori == applicare il colore della texture nella parte corrispondente. In questo step si possono verificare due fenomeni: **magnification** e **minification**. La **magnification** è quando un pixel è più piccolo di un texel:

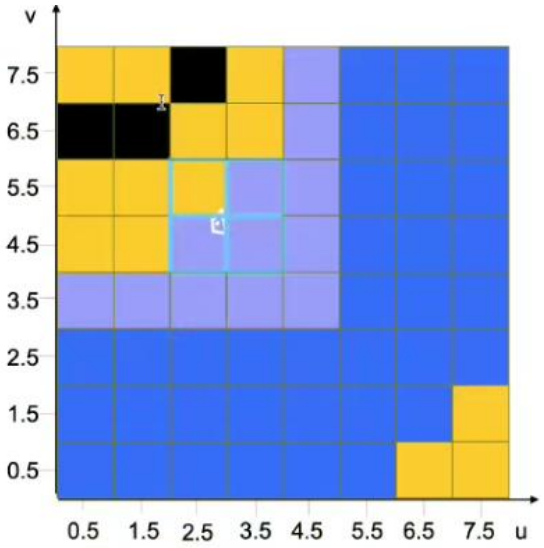


quindi hai due tecniche, il Nearest Filtering e la Bilinear Interpolation. La Nearest Filtering -> al pixel assegni il colore del texel in cui va a finire: questa però rende i visibili i texel, e quindi i pixel -> brutto. Nella bilinear interpolation, per rendere tutto più fluido, si fa interpolazione del colore facendo la media dei colori dei 4 texel più vicini al pixel.

Solution 2:
Compute a weighted average of the four closest texels

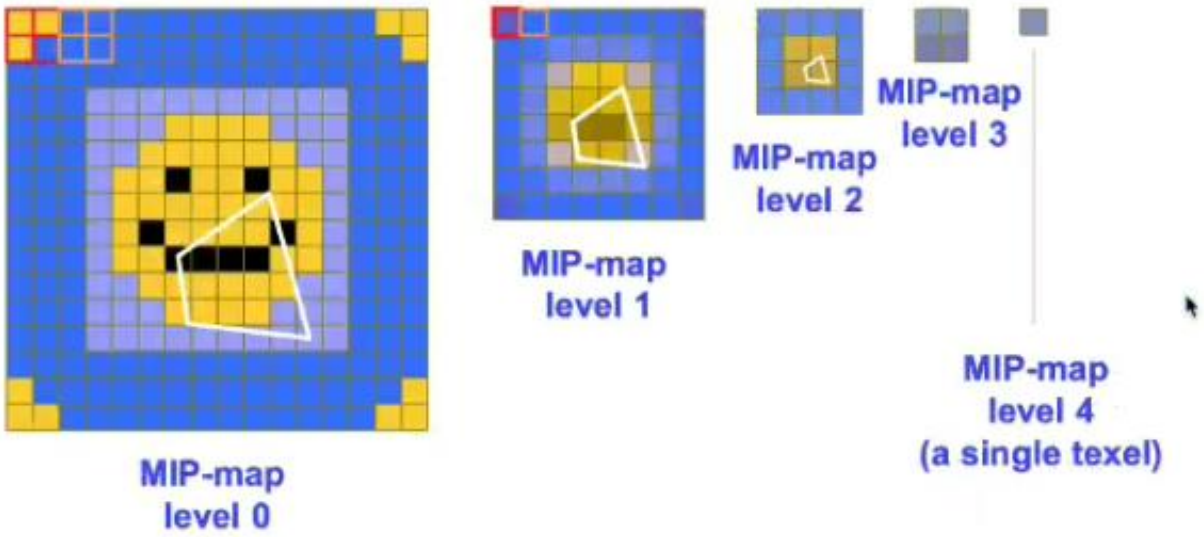
Bilinear Interpolation

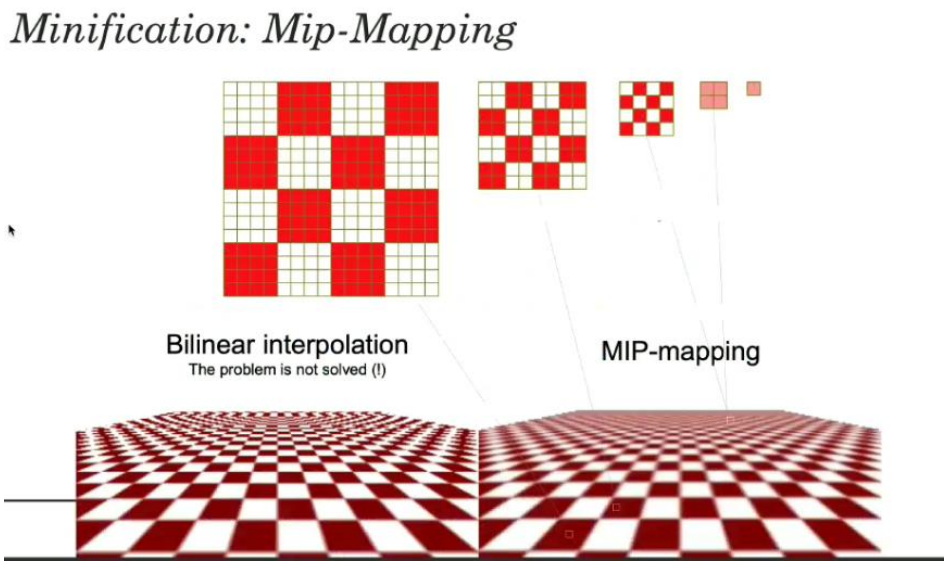
$$
\begin{aligned}
\alpha &= x - \lfloor x \rfloor \\
\beta &= y - \lfloor y \rfloor \\
p(x,y) &= (1-\alpha)(1-\beta)p_{11} + \\
&+ \alpha(1-\beta)p_{12} + \\
&+ (1-\alpha)\beta p_{21} + \alpha\beta p_{22}
\end{aligned}
$$



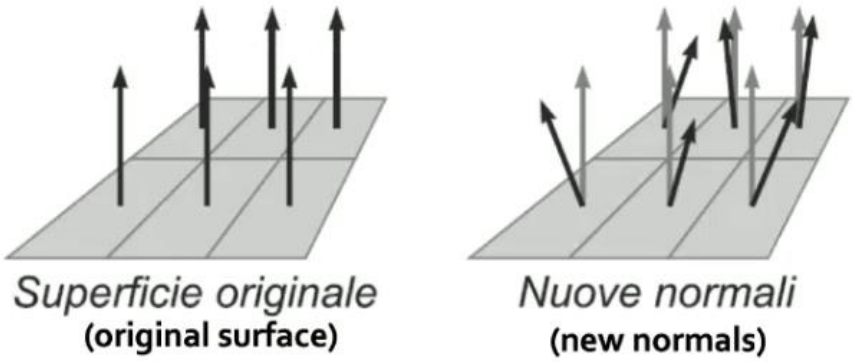
In questo modo rendi tutto più omogeneo.

L'altra cosa è la **minification**, quando il pixel è più grande del texel, cioè al pixel sono associati più colori perché più texel può significare più colori. In questo caso, si definisce un valore di scala, Rho, che sarà il rapporto tra la grandezza dei texels e quella dei pixels. Si utilizza il mip mapping, dove hai la stessa texture salvata più volte in versioni con risoluzioni diverse e, in base al tuo mipmap level (log2rho) assegnerai la texture.

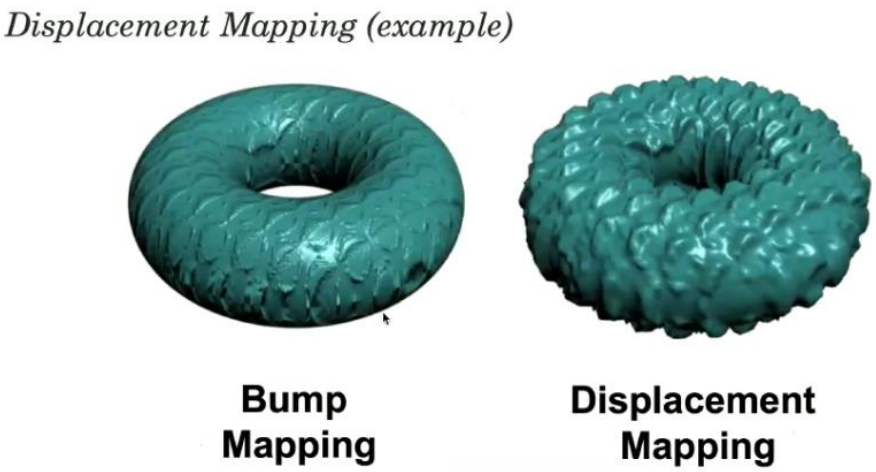




Abbiamo visto anche altri tipi di mapping, che quindi cambiano altri attributi, il Bump mapping per esempio serve a modificare come un oggetto appare senza però cambiare la sua geometria.



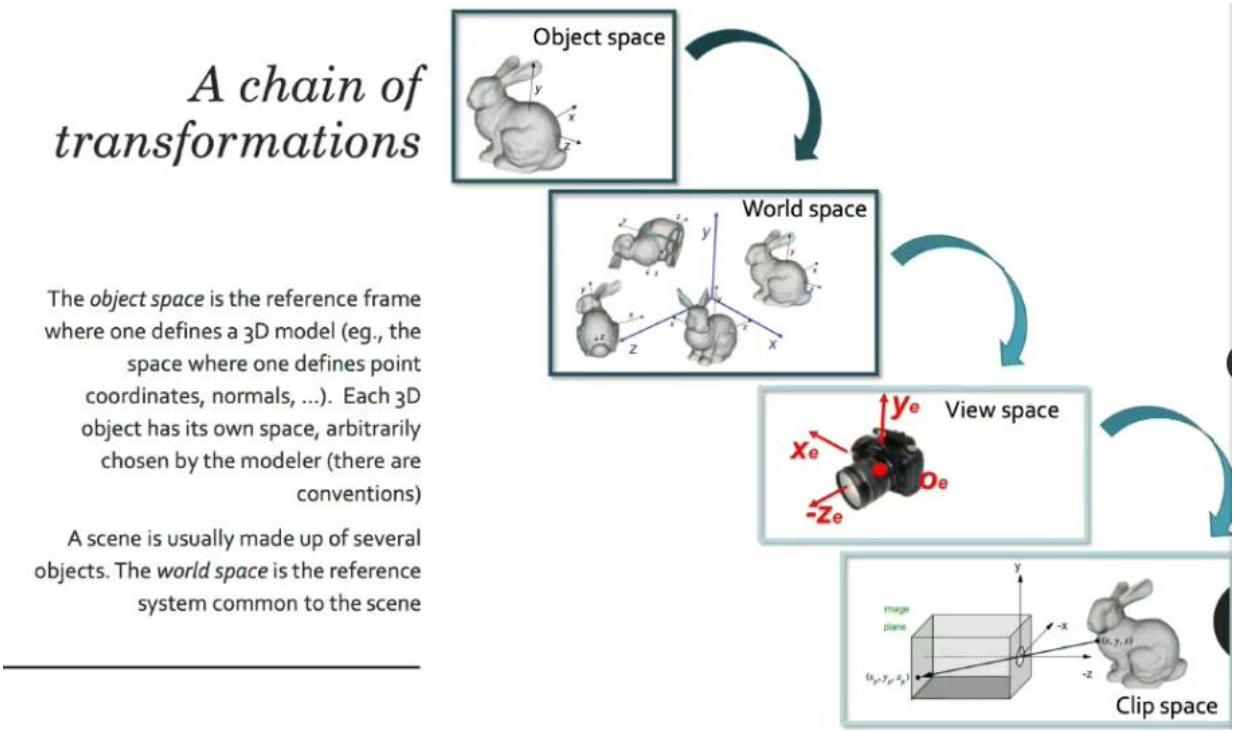
Oppure il displacement mapping, dove si prendono le normali delle superfici e gli si aggiunge un certo noise in modo tale da ottenere quest'effetto (si esegue a rendering time):



Quindi mentre nel primo hai comunque mantenuto il fatto che sia un Thorus al quale hai applicato un effetto sopra, nel secondo invece è proprio un'altra cosa perché modifica la geometria dell'oggetto in quanto non va a modificare le normali ma va a spostare i punti sulla superficie.

L'ultima cosa sono le trasformazioni geometriche

Per poter arrivare a mostrare una scena su uno schermo, dovrai intanto partire dallo spazio degli oggetti, in cui ogni oggetto che vorrai mettere nella scena ha il suo spazio di coordinate. Questo oggetto lo dovrai posizionare all'interno del World Space, ovvero dovrai costruire la tua scena. Dal World Space passerai al View Space, che è uno spazio di coordinate che è centrato nella camera (lo strumento che utilizzi per catturare la scena). Infine, dal View Space passerai al Clip Space. Se le prime 3 sono trasformazioni affini, l'ultima è una proiezione.



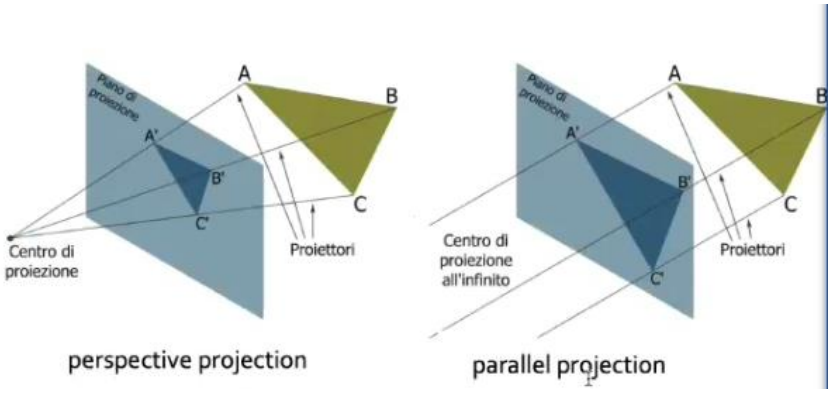
<u>Cosa sono le trasformazioni affini?</u> Nel 2D, sono la traslazione, lo scaling (rispetto ad un punto fisso) e la rotazione (r.a.u.p.f.). Nel caso del 2D questo punto è l'origine degli assi. Queste trasformazioni le userai componendole: ma questo non si può fare con le trasformazioni affini normali, perché? Perché la traslazione è una somma di un vertice in movimento (prendi coordinate e sommi di quanto si devono spostare sugli assi x e y), mentre scaling e rotazione sono una moltiplicazione. Nello scaling è un fattore di scala (*2, *3, …), nella rotazione invece per il coseno e seno dell'angolo. Per poterlo fare allora si considerano le **coordinate omogenee**: se un punto prima era definito dalle coordinate xy, ora è definito dalle coordinate "x su w", "y su w", 1. w è un numero diverso da 0. Che vantaggio otteniamo? Puoi trattare i punti e i vettori (quindi locazioni e direzioni) allo stesso modo. Quindi se prima scaling e rotazione li facevi rispetto ad un punto e la traslazione era visto come un vettore di movimento, ora entrambi sono viste allo stesso modo. Quindi hai rimosso l'ambiguità che c'era tra punti e vettori e poiché ora puoi esprimere tutto come prodotto, puoi anche comporre queste trasformazioni.

*In particolare, riflettere un'immagine è un particolare caso dello scaling (rispetto alla X, Y * -1 e viceversa, se entrambi sia X che Y * -1). Lo shear (distorsione) è una composizione di scaling e rotazione in modo tale da distorgere l'immagine (rispetto ad uno o entrambi gli assi).*

Tutto quel che abbiamo detto finora è anche applicabile al 3D, però la rotazione è diversa perché devi prima scegliere l'asse rispetto al quale ruoterai. A seconda di quale scegli, cambia la matrice che andrai a moltiplicare.

L’ultima cosa da fare per mostrare l’immagine, è proiettarla sul piano di proiezione. Anche questa è una moltiplicazione matriciale e fai una proiezione che può essere di due tipi. Proiezione **perspettiva** e proiezione **parallela**. Si differenziano rispetto a dove si trova il centro di proiezione: se è un punto, tipo la pinhole camera, è perspettiva e devi collegare il centro di proiezione coi vertici della tua primitiva tramite questi “proiettori”. Facendo così intersecherai il piano di proiezione e quindi andrai a delineare il contorno della tua primitiva. Se invece il centro di proiezione è all’infinito, i proiettori saranno tutti paralleli tra di loro e quindi è come copincollare esattamente una faccia della tua primitiva sul piano di proiezione.

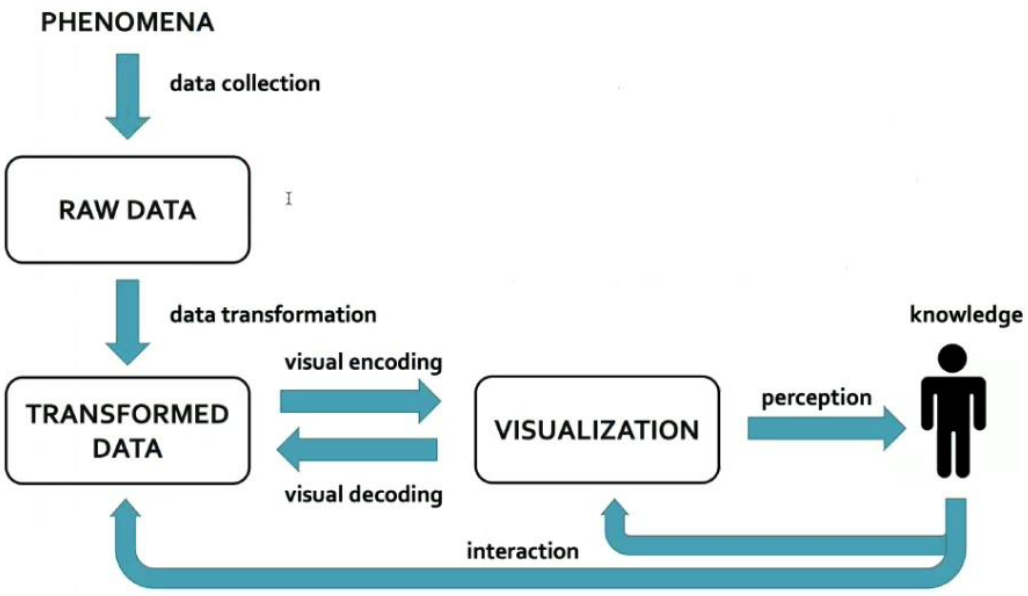
Perspective and parallel projections



Seconda parte

Information visualization

Quando vuoi visualizzare dei dati astratti con grafici in modo tale da facilitarne la compresione e cognizione.



*Prima di parlare di ciò, vanno ovviamente collezionati questi dati (**Raw Data**), che trasformerai in base a ciò che devi visualizzare/mostrare. A questi dati trasformati potrai poi fare **visual encoding**, ovvero il passare dai dati alla loro rappresentazione. In cosa consiste? Nello scegliere il grafico giusto. Come si sceglie? Tramite gli elementi del grafico stesso.*

Permette il processing parallelo d'informazioni, tramite explaination, exploration e confirmation

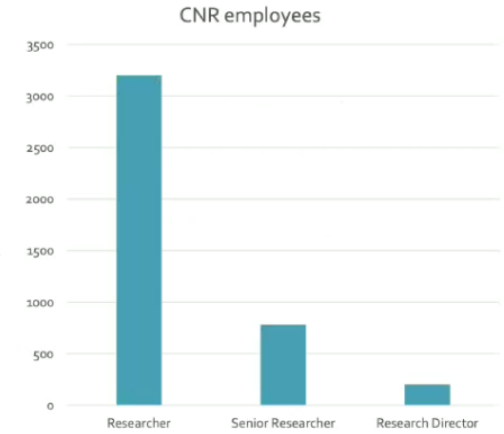
Quali sono gli elementi?

- **Mark**: tutti gli elementi visivi del grafico. Punti, barre, linee, aree, …
- **Channel**: tutto ciò che riesce a mostrare uno o più attributi dei dati. Posizione, colore, angolo, texture, forme, dimensioni, …
- **Componenti contestuali**: tutte le componenti che arricchiscono il grafico. Legenda, annotazioni, griglie, …

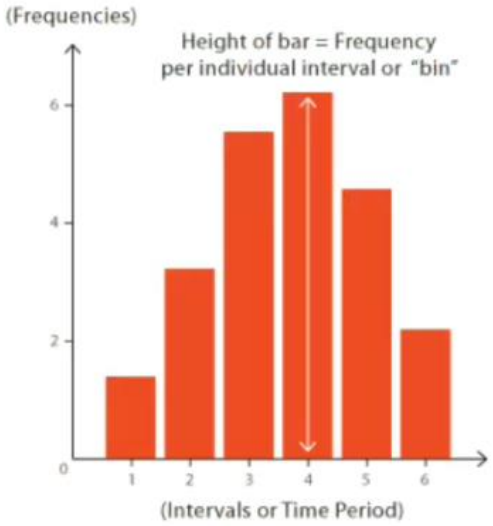
Nei channel abbiamo parlato di **attributi** e **dati**. Un dato si definisce come un'informazione che può essere utilizzata per effettuare delle statistiche o delle misure. I dati possono essere strutturati o non-strutturati: nel primo caso, parliamo di dati con attributi (ovvero proprietà del dato, e.g. dato: Persona, attributi: Nome, Cognome, …). Gli attributi li abbiamo poi divisi in categorici (mostrano categorie == valori discreti. Se possono essere ordinati sono detti Ordinali, altrimenti Nominali) e quantitativi (mostrano quantità misurate == valori continui).

Abbiamo visto diversi tipi di grafici (tutti per dati bivariati == che puoi mettere su 2 [o 3?] dimensioni):

- **Bar Chart:** tipo di grafico che mostra la correlazione tra un attributo categorico (asse X, solitamente indipendente) con un attributo quantitativo (asse Y, solitamente dipendente).

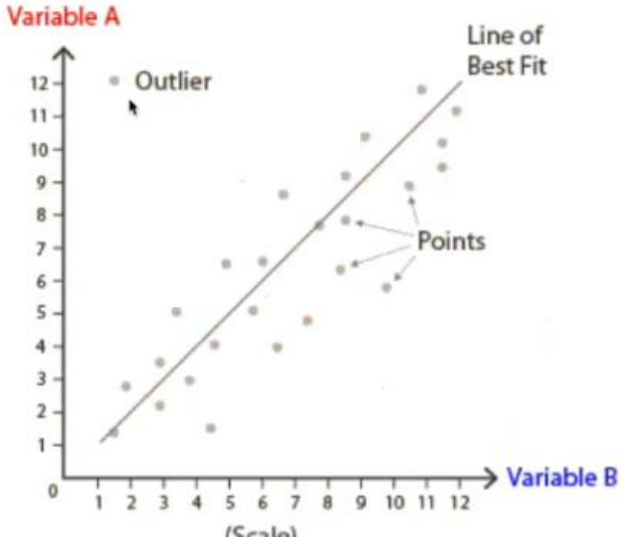


- **Istogrammi:** particolare tipo di Bar Chart in cui sull'asse delle X si mette UN attributo indipendente che però viene discretizzato in intervalli (bins). Sull'asse Y si mette la frequenza, quindi contare quanti valori di quell'attributo ricadono in quel bin. *(Nota: prima avevi 3 attributi indipendenti)*

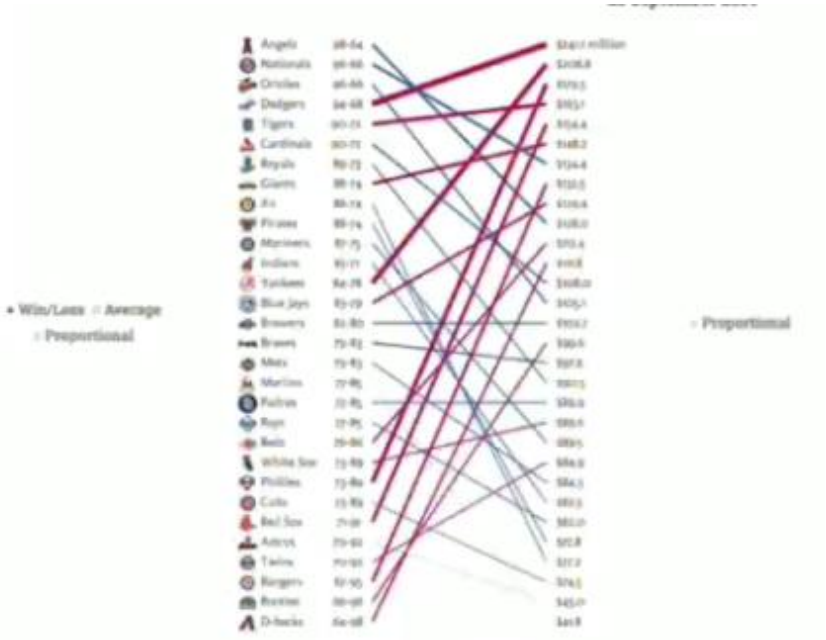


- **Pie Chart:** mostrano le proporzioni e percentuali tra categorie.
- **Donut Chart:** dei Pie Chart bucati al centro in modo tale da concentrarsi sulla lunghezza dell'arco piuttosto che sull'area dello spicchio.

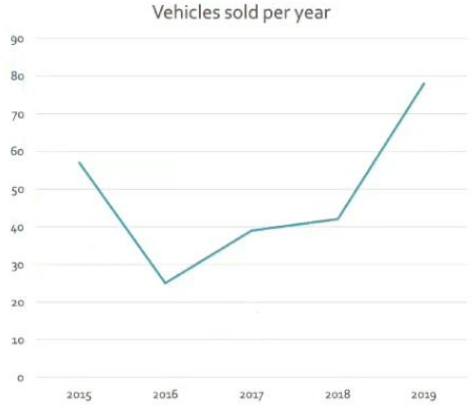
- **Scatter Plot:** servono a mettere in relazione attributi quantitativi indipendenti. Servono per cercare correlazione tra due di questi attributi e rendono facile l'individuazione degli outliers.



- **Slope Charts:** sono Scatter Plots particolari in cui i punti sono stavolta definiti da una linea che collega i due valori che creerebbero il punto nello Scatter Plot normale.

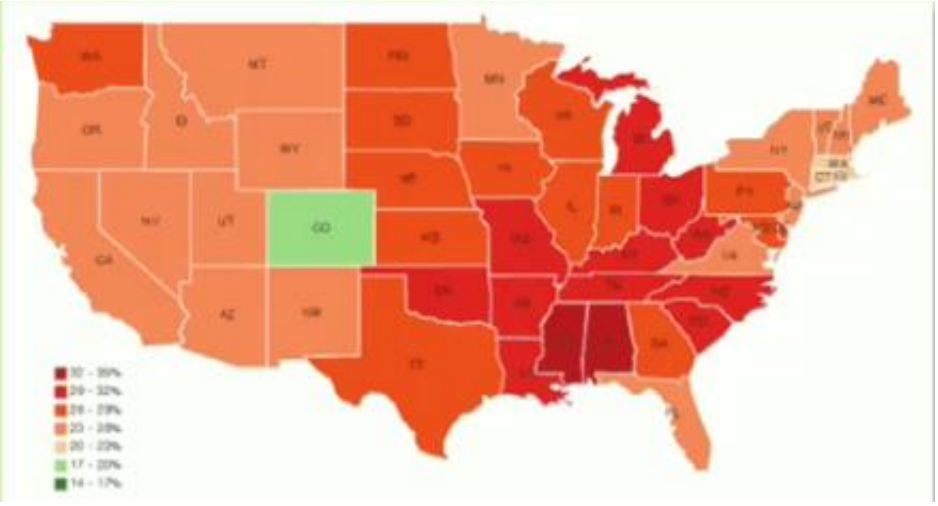


- **Line Chart:** fanno vedere l'andamento di un attributo (solitamente nel tempo).

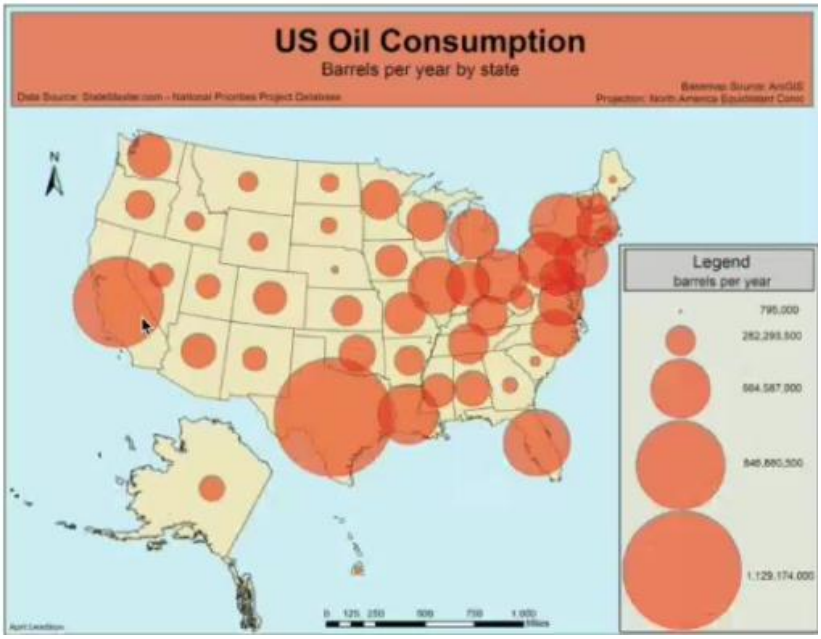


- **Area Chart:** stessa cosa del Line Chart, ma l'area sotto la linea è colorata così da far notare maggiormente i cambiamenti

- **Cloropleth Map:** per le regioni geografiche, sono quelle in cui ogni regione viene colorata di un colore secondo una scala/palette di colori in modo tale da mostrare un confronto lieve tra le varie regioni. Non è adatto per mostrare un confronto accurato, solo buona panoramica!

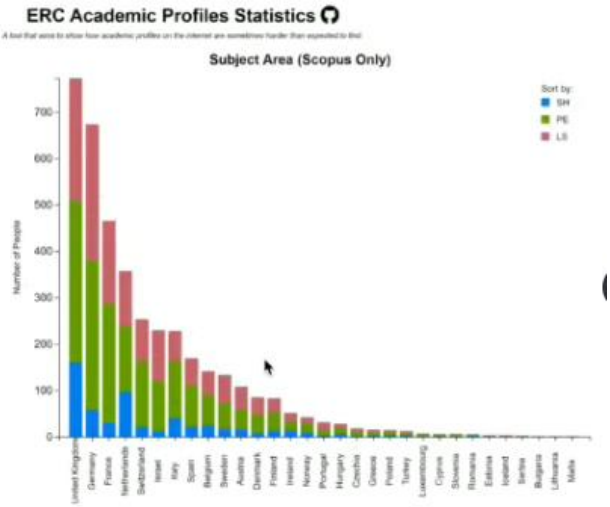


- **Symbol Map:** una versione che risolve il problema precedente. Su ogni regione, usi un simbolo di cui modificherai un attributo (colore, dimensione, …) per differenziare le varie regioni. Un esempio comune è usare un cerchio e la sua grandezza cambia proporzionalmente all'entità che vogliamo mostrare.

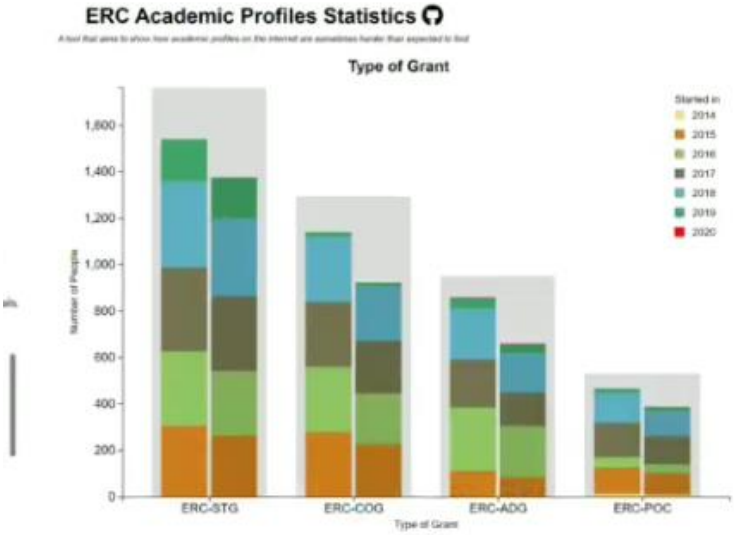


[Per attributi multipli]

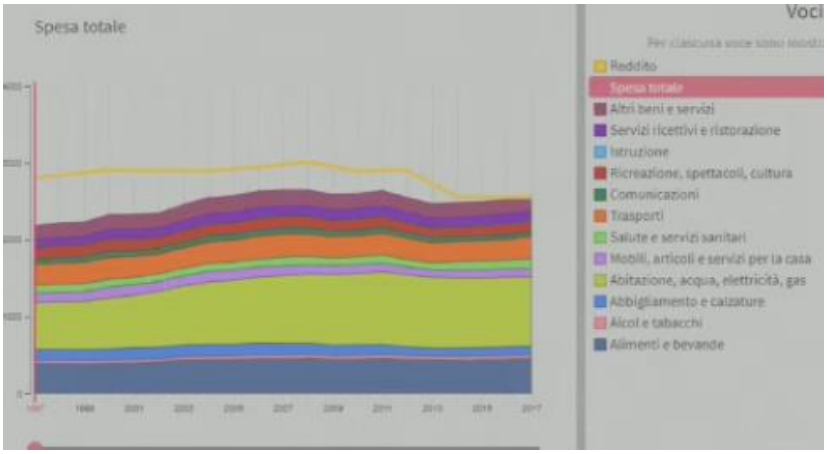
- **Stacked Bar Chart:** mettono in relazione diversi attributi. Esiste sia nella versione per valori assoluti, per dare enfasi alle differenze totali, sia in percentuale, per mostrare meglio la proporzione tra i diversi gruppi.



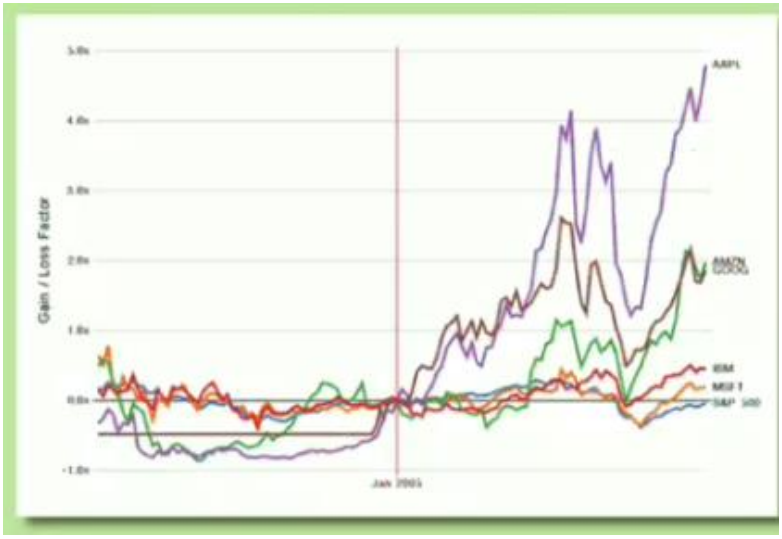
- **Grouped Bar Charts:** Bar Charts messi uno affianco all'altro, li ripeti tante volte quanti sono gli attributi che vuoi confrontare. Utile per comparare valori individuali.



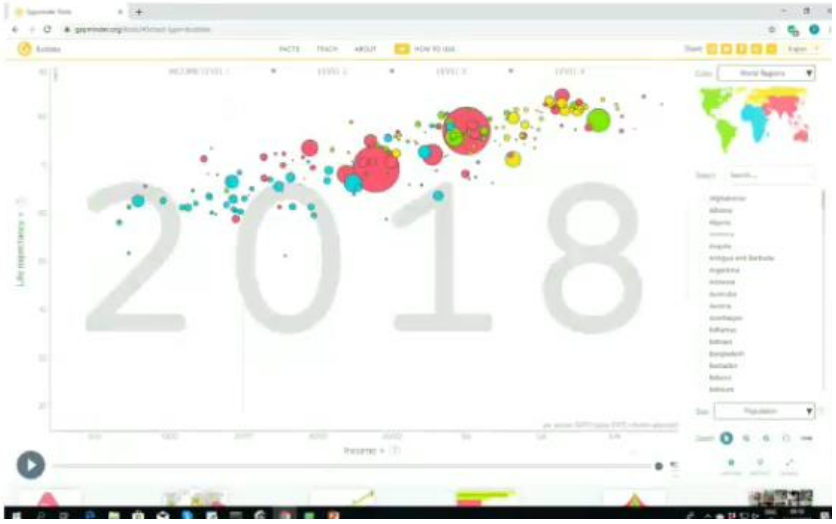
- **Stacked Line Chart:** Line Charts dove metti più linee. Usati quando sai che queste linee non andranno ad incrociarsi ma soprattutto per dati che non devono essere sommati.



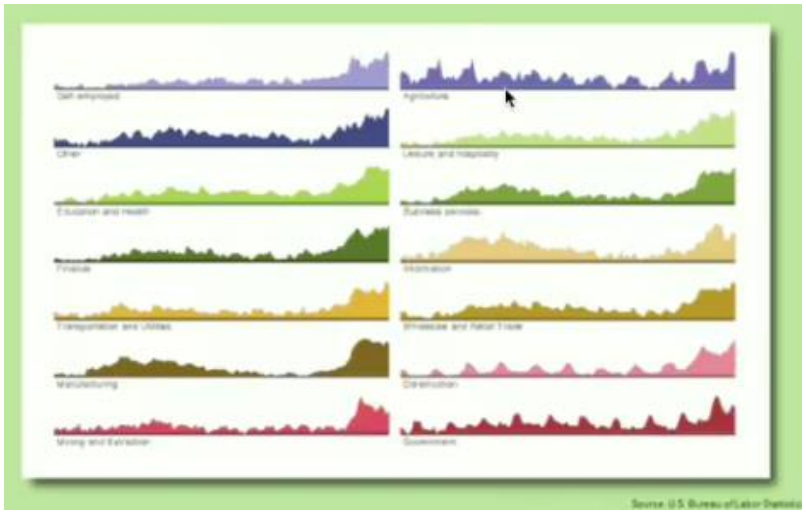
- **Line Chart Series:** come prima, ma confronti l'andamento di più valori che sai che si incroceranno. Sono spesso mal-utilizzati perché vengono messe troppe linee che rendono il grafico incomprensibile.



- **Bubble Chart:** sono Scatter Plots in cui al posto dei punti, vengono usate altre componenti che possono mostrare altre proprietà (es. un cerchio e la sua dimensione, colore ,..).



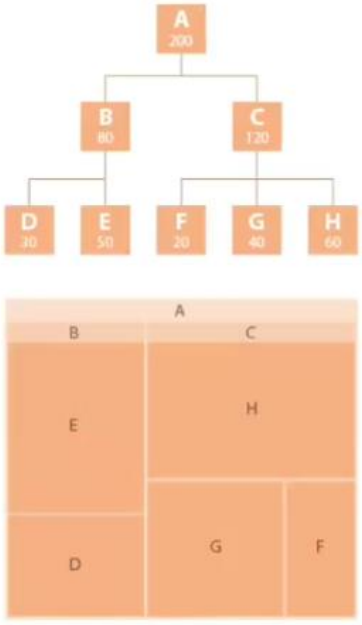
- **Small Multiples:** lo stesso tipo di grafico messo uno a fianco all'altro che però mostrano l'andamento di attributi diversi.



[Per mostrare gerarchie]

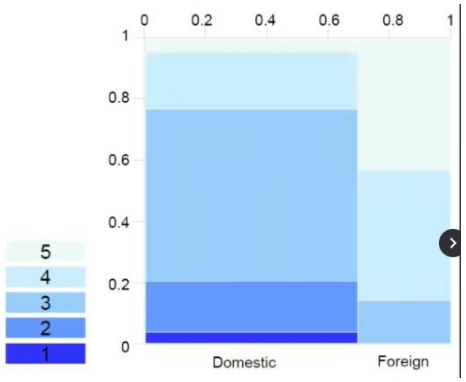
- **Sunburst Charts:** praticamente dei Pie Chart concentrici dove ogni anello rappresenta un livello di gerarchia. Partendo dal centro (la radice), "uscire" è come visitare l'albero.

- **Tree Map:** ancora più come un albero. Il rettangolo esterno è la radice, all'interno di ogni rettangolo puoi avere dei sottorettangoli che indicano i figli.

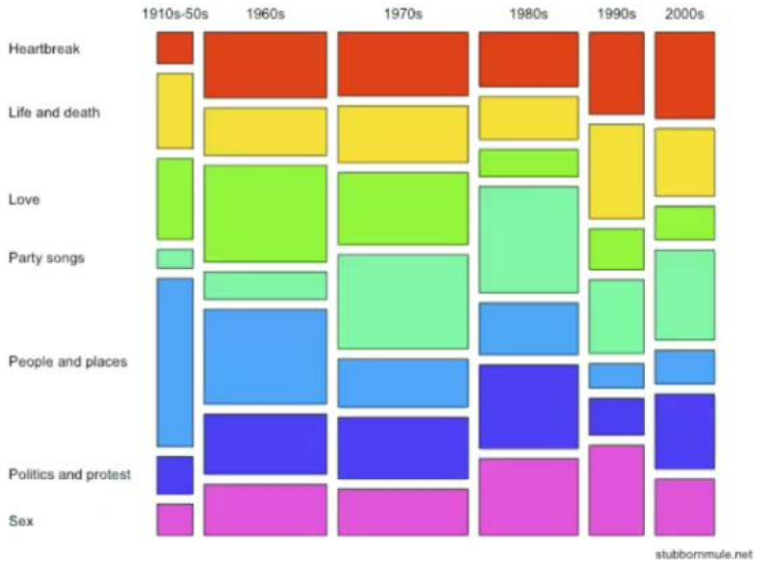


[Dati multidimensionali]

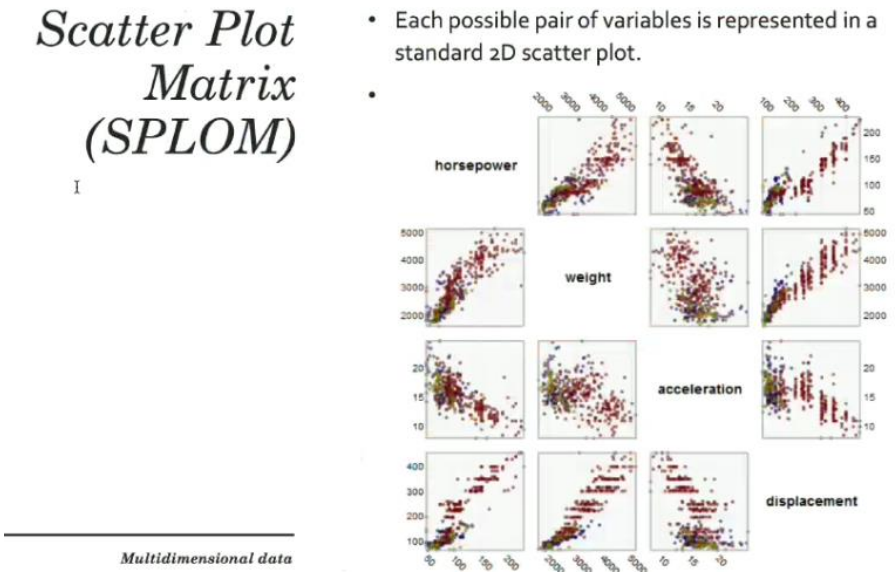
- **Spine Plot:** sono particolari Stacked Bar Chart in cui mostriamo entrambe le possibilità (sia proporzioni che percentuali tra variabili).



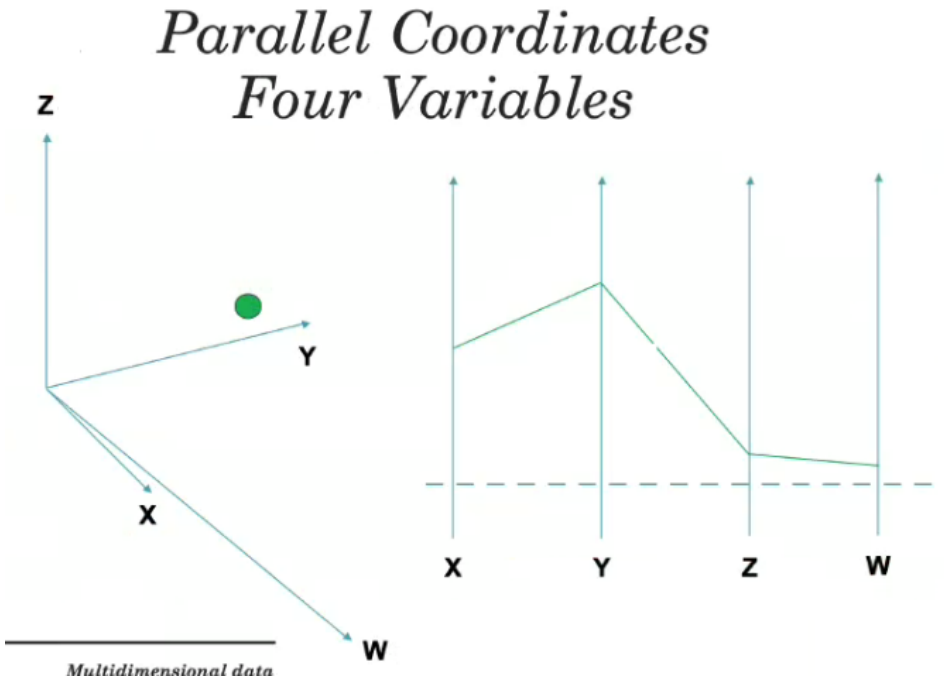
- **Mosaic Plot:** versione generica degli Spine Plot. Solitamente non sono utilizzati perché confusionari all'incrementare del numero di attributi.



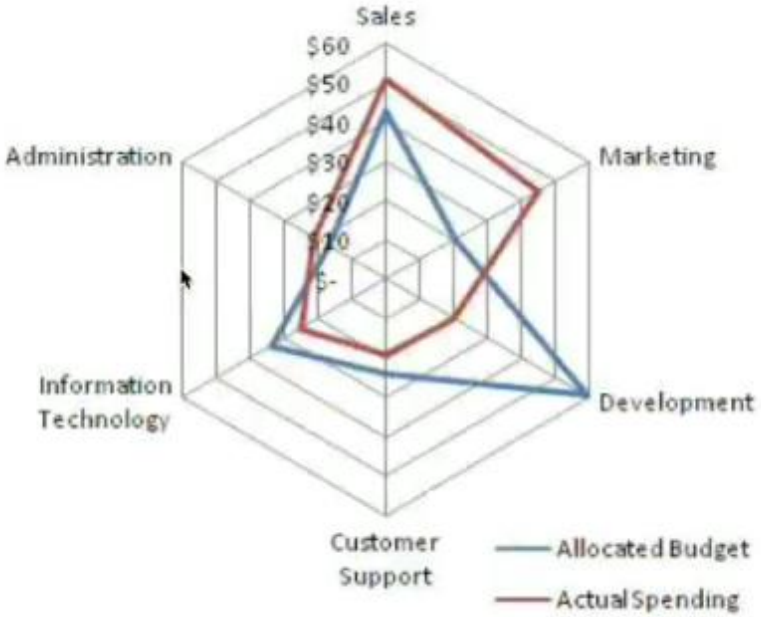
- **SPLOM (Scatter Plot Matrix):** una matrice di Scatter Plots in cui a 2 a 2 vai a correlare attributi quantitativi.



- **Parallel Coordinates:** sfruttando il fatto che avere dati multidimensionali significa avere più assi, ordini questi assi (secondo qualche logica) e per indicare un punto multidimensionale si collegano i valori che quel punto assume su ogni asse. La parte difficile è l'ordinamento degli assi, che è fondamentale, e ci sono *n!* combinazioni possibili.

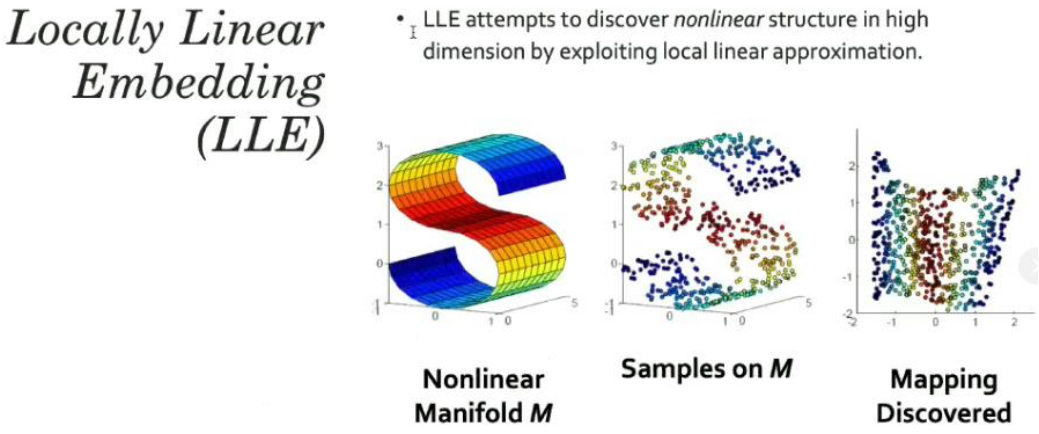


- **Star Plot:** detti anche Spider Chart. Invece di usare le coordinate cartesiane, utilizzi le coordinate polari: dividi un angolo giro (360°) in tanti angoli uguali quante sono le tue dimensioni. Non importa il punto d'inizio: poi unisci i valori similmente a prima per andare a creare un poligono.

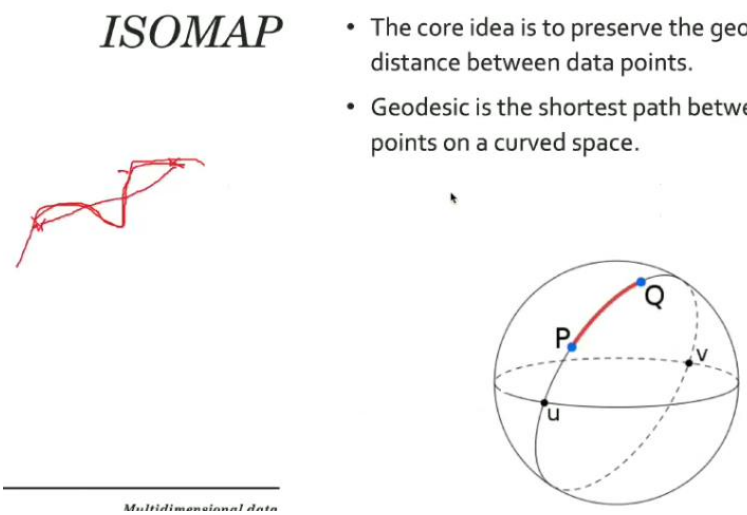


Buono per comparare proprietà. Non è facile capire il trade-off tra variabili differenti e quindi non è adatto per essere usato con molte variabili

- **Dimensionality Reduction:** un approccio diverso per esprimere la multidimensionalità. Quindi, passare da dati a N dimensioni a dati in 2/3 dimensioni in modo da utilizzare tutti i grafici visti finora. Quindi formalmente è un mapping di punti (NON una trasformazione geometrica) da uno spazio ad alte dimensioni ad uno a basse dimensioni. Abbiamo visto diverse tecniche, tipo:
    - **Principal Component Analysis (PCA):** tecnica lineare non parametrica dove cambi la base dei tuoi dati: fai in modo che la varianza dei dati sia massimizzata e che la covarianza sia minimizzata. Come fai? Applicando una combinazione lineare alla tua vecchia base, ottenendo una nuova base. Cosa significa? Moltiplicarla per un certo vettore. Per calcolare questo vettore, parti dalla matrice di covarianza: è simmetrica e quadrata, in cui sulla diagonale si trova la varianza (quanto varia il valore di un certo dato) di una certa variabile, mentre nel resto della matrice si trova la covarianza delle coppie di variabili (covarianza coppia AB, al variare di A come varia B e viceversa -> simmetrica). Il nostro scopo è massimizzare la varianza e minimizzare la covarianza (ridondanza), in quanto avere una covarianza alta significa che al cambiare di una certa variabile cambia anche un'altra (es. anno di nascita, età -> una delle due variabili è inutile, puoi ridurre la dimensione dei dati). Per ottenere ciò, basta diagonalizzare (massimizzare i valori sulla diagonale e minimizzare gli altri) la matrice di covarianza.
      Alla fine, la PCA trova k componenti (solitamente 2 o 3) e proietta i dati solo lungo queste 2-3 dimensioni. Cons: Fallisce per distribuzioni non-gaussiane.
    - **Local Linear Embedding:** un'altra tecnica che, come ISOMAP, parte costruendosi il grafo del vicinato. Quindi ogni punto viene scritto come combinazione lineare dei suoi vicini. Poi utilizzi un'operazione di ottimizzazione basata sugli autovettori della matrice risultante da queste combinazioni lineari affinché i punti siano trasformati in basse dimensioni. Però c'è una particolarità: quando ci si sposta in uno spazio a basse dimensioni, la combinazione lineare dei nuovi vicini del punto X deve continuare a rappresentare il punto X.



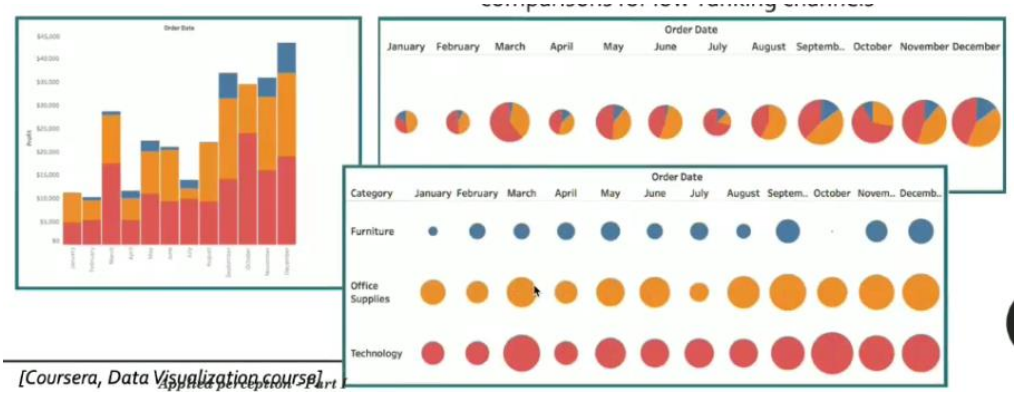
    - **ISOMAP:** qui invece non vuoi perdere la distanza che hai tra i vicini (come la distanza geodesica). Questo perché ISOMAP si basa su questa proprietà intrinseca. Parti dal Neighborhood Graph, calcoli lo Shortest Path (tramite algoritmo di Floyd) tra un punto ed i suoi vicini e crei la matrice delle pairwise distances. Ora si utilizza la tecnica di Multi Dimensional Scaling (MDS): prende una matrice di pairwise distances e la trasforma in punti rappresentabili sul piano cartesiano (quindi da una matrice di distanze -> punti piano cartesiano). Sempre tramite una trasformazione fatta (chissà come), si va in uno spazio a meno dimensioni mantenendo intatte le distanze che ci sono tra i punti.



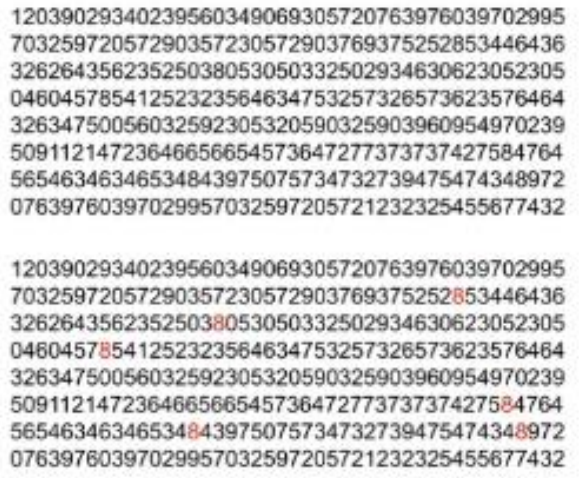
- **Stochastic Neighbor Embedding (SNE):** per passare da alte a basse dimensioni si utilizza la distribuzione di probabilità. Sono due distribuzioni: una calcolata nello spazio ad alte dimensioni (assegna una probabilità a coppie di punti che è tanto più alta quanto sono simili), l'altra viene calcolata nello spazio a basse dimensioni. Ora si cerca di minimizzare la differenza che c'è tra queste due, chiamata Divergenza di Kullback-Leibler [entropia relativa] ("quanto sono diverse due distribuzioni di probabilità", fatto tramite il Gradient Descent).
  In questa versione, lo SNE soffre del Crowding Problem, ovvero che non si riesce sempre a preservare la distanza con i vicini. È dimostrato che la probabilità che viene data ad una coppia AB non è per forza detto che sia la stessa che assegneresti a BA: se invece lo fai, stai usando la versione simmetrica dello SNE, chiamato **T-SNE.** Non elimina completamente questo problema, ma comunque in parte lo risolve.

Tornando indietro, noi abbiamo detto che dopo che collezioni e trasformi i dati puoi fare encoding. L'operazione inversa si chiama **visual decoding**, quindi passare dalla visualizzazione ai tuoi dati ("Decostruisci la rappresentazione grafica nelle sue parti"). Per sapere quando è fatto bene, si considerano due proprietà:

- **Espressività:** quando visualizzi qualcosa, vedi solo le relazioni che ci sono tra i dati senza "inventarne" altre. Esempio: grafico dove colori in modo diverso i punti, ma quel colore non rappresenta nulla.
- **Efficacia:** serve a capire quali sono le parti più rilevanti del grafico/rappresentazione. Abbiamo visto che l'efficacia dei canali visivi (che sono una parte delle componenti di un grafico) può essere misurata tramite alcuni parametri, quali:
  - **Accuratezza:** quanto accuratamente un canale rappresenta un attributo. Quindi ovviamente è sempre importante scegliere i canali con più alta accuratezza.

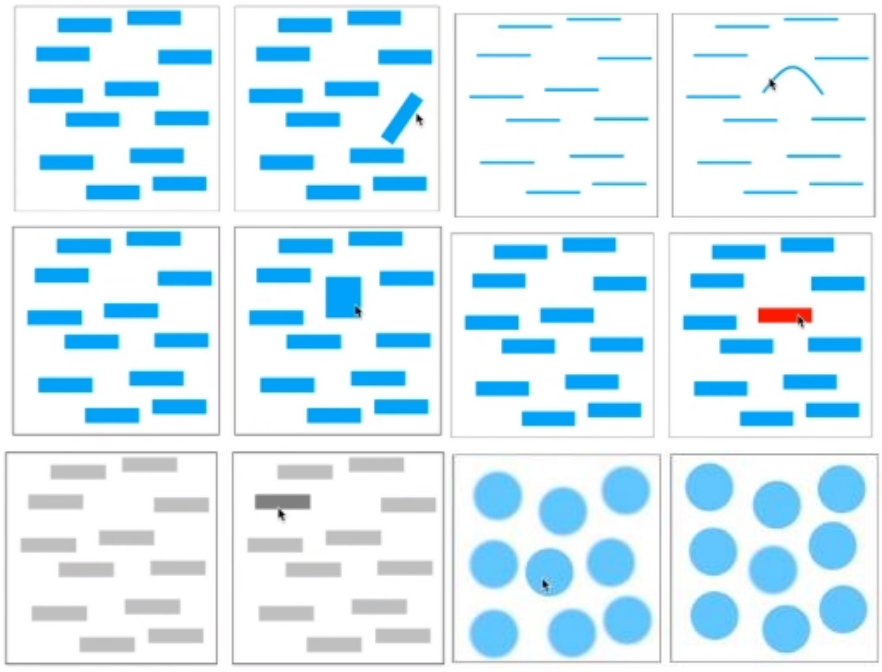


- **Discriminabilità:** quanti valori riesci a percepire con un singolo canale. Dipende ovviamente dal canale, tipo la posizione, i colori, etc.
- **Rilevanza:** quanto qualcosa riesce a risaltare rispetto al resto. Orientare, curvare, colorare, sfumare, etc. sono esempi che possono provocare rilevanza.
  - ➔ Processi Pre-Attentive: tutte quelle feature che risaltano all'occhio velocemente.

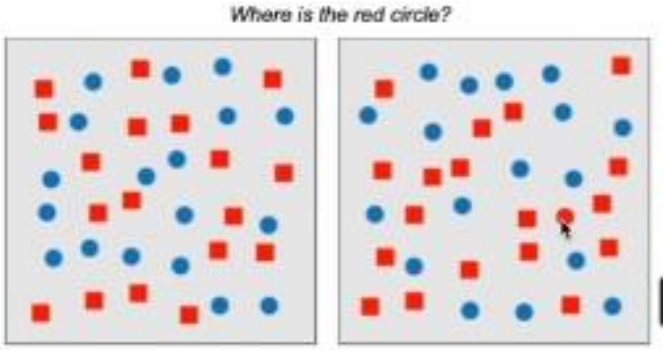


Tipo l'8 nell'immagine.

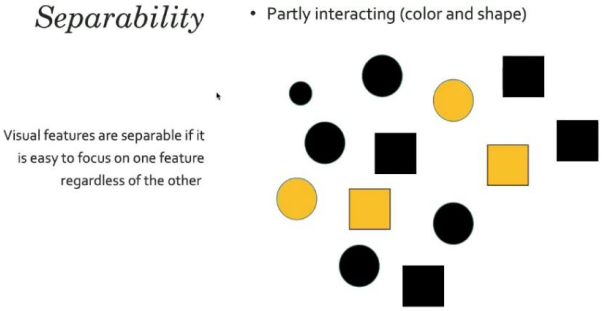
Tutti i processi pre-attentivi portano a maggiore rilevanza, tipo appunto orientamento, curva, forma, grandezza, colore, …



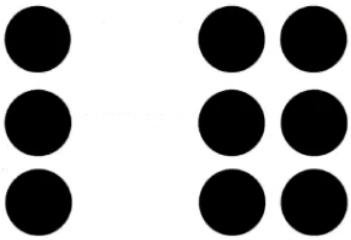
<u>Nota bene</u>: non è detto che sommando delle feature pre-attentive si ottenga un risultato migliore, per esempio



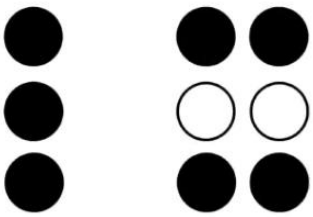
- **Separabilità:** quanto riesci a focalizzare l'attenzione su un canale rispetto che su un altro.



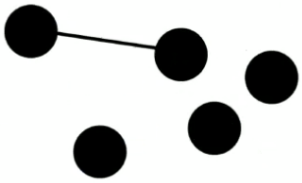
- **Raggruppamento:** come dei patterns possono emergere da un singolo elemento.
- **Gestalt Law:** riguarda il grouping, ovvero quando riesci a vedere più oggetti con un singolo elemento. Queste leggi cercano di spiegare quali sono questi pattern perception, come vengono creati, etc. Prossimità, struttura, similitudine, continuità (collegati da un segmento), simmetria, chiusure, …
  - <u>Prossimità</u>:



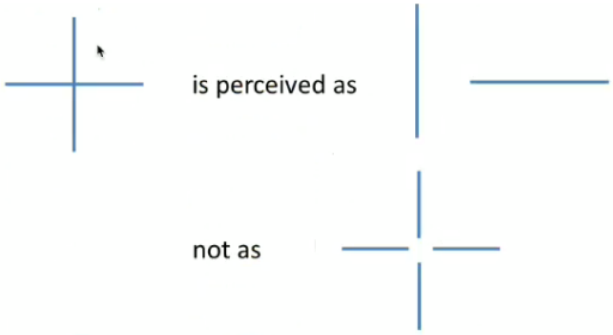
  - <u>Similarità</u>



  - <u>Continuità</u>



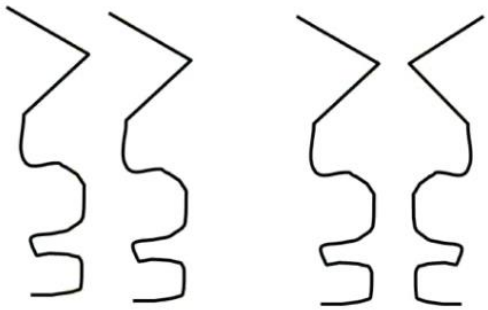
We expect that a line or an edge continue to follow its direction and do not deviate from it



- Simmetria

Objects arranged simmetrically are perceived as forming a visual whole instead of being preceived as separated entities

Simmetry is best perceived for horizontal and vertical axes



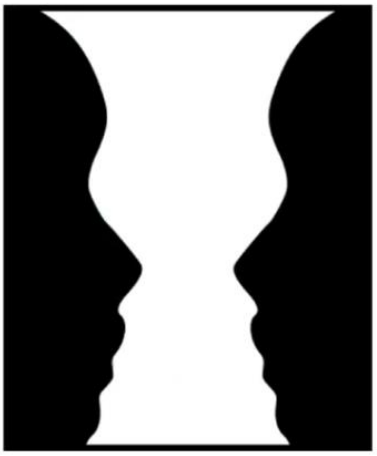
- Chiusura

We tend to perceive the complete appearance of an object: our brain fills the gap in case of missing parts.

- Common Fate

We tend to perceive as a group objects that move in the same direction

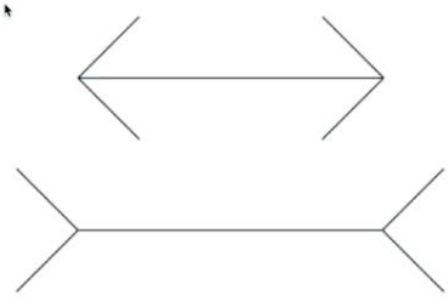
- Figure Ground



L'effetto di formare una figura dal background (due omini o un vaso)

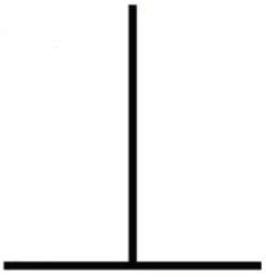
Müller-Lyer Illusion

- These two lines have equal length, but we perceive that they have different length.
- Two explanations:
  - Perspective explanation
  - Centroid explanation

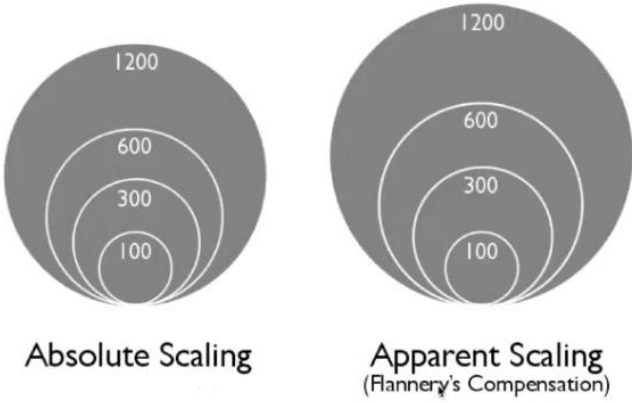


Horizontal–Vertical Illusion

- Another simple illusion discovered by Wundt.
- The vertical line is perceived 30% more length than the horizontal line.
- Cross-cultural (small) differences have been noticed.
- This is true also for intersecting lines.



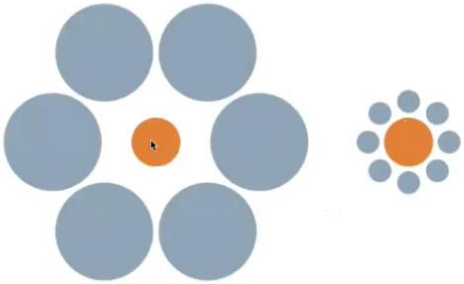
Flannery's Perceptual Scaling



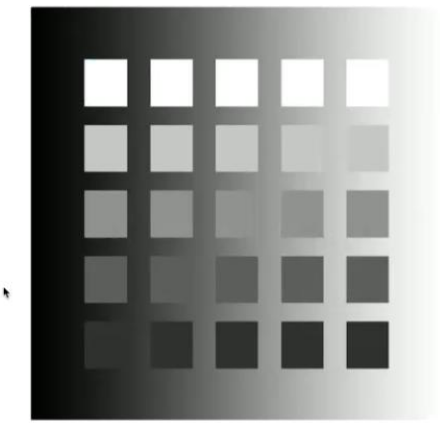
Confrontare aree è difficile: le proporzione vengono sottostimate

Comparing Area

- Perceptual scaling may be insufficient. Things are more complex from a perceptual point of view → *Heidenberg illusion*.



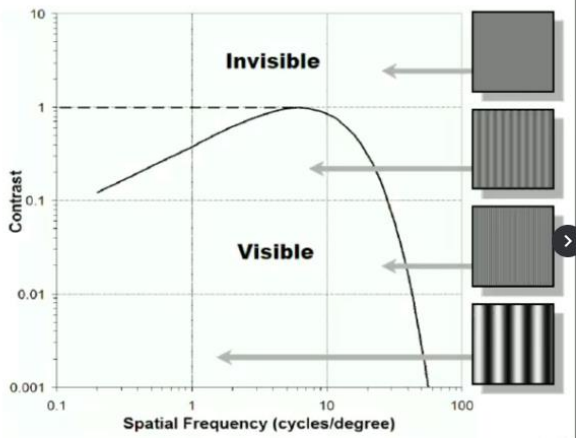
Weber's Law

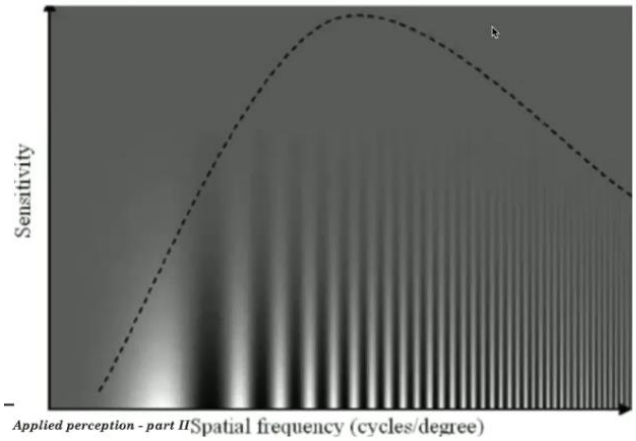


Weber Law: il contrasto delle zone circostanti gioca un ruolo fondamentale. Come percepiamo la luminosità.

Contrast Sensitivity Function (CSF)

Our perception is sensitive to pattern contrast, frequency and orientation.

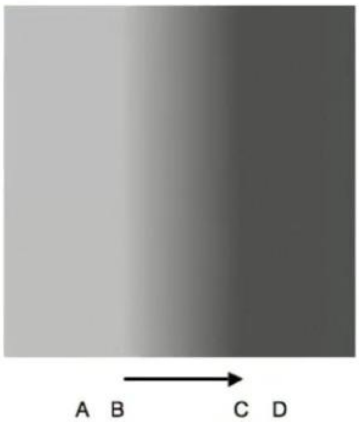


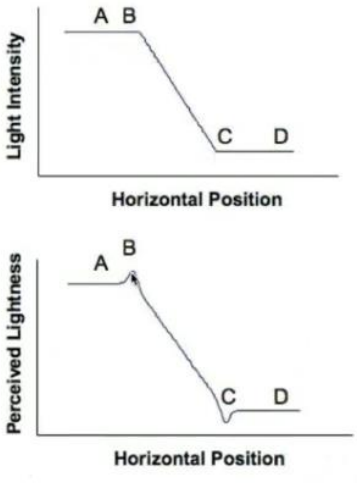


Il campo recettivo di una cella è la zona sulla quale la cella risponde alla luce.

Ricordi il vaso mach banding?

Mach Banding

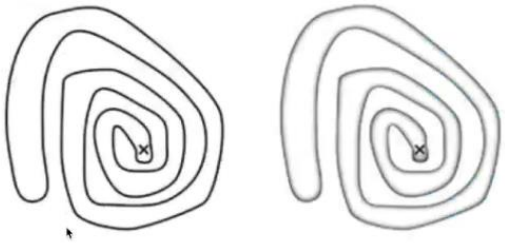




Mappa di grigi da usare solo per pochi valori! *Usala per bounded regions, important items (riduci contrasto della luminosità di elementi non importanti), aggiusta la luminosità di background per ottenere maggiore leggibilità.*

Cornsweet Effect

The Cornsweet effect can be used to highlight bounded regions.



Colori

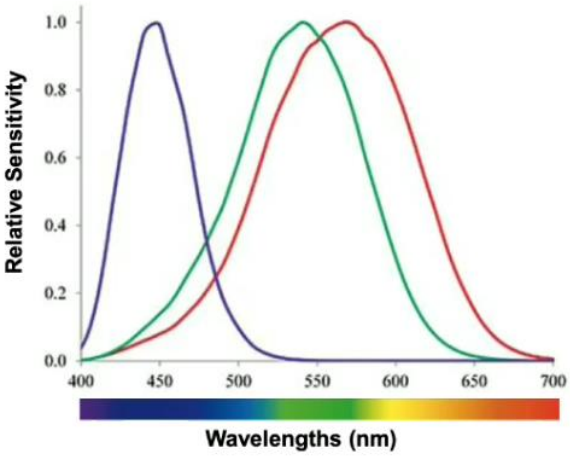
Per quanto riguarda la percezione del colore (estremamente importanti per mostrare pattern, fare labeling e/o highlighting), ci basiamo su due teorie: la **teoria della tricromia** e la **teoria dei colori opposti**. Entrambe si basano sul fatto che nella nostra retina abbiamo Rods e Coni. I primi servono quando c'è poca luce, quindi ci concentriamo sui coni che catturano la luce nello spettro di 3 colori principali: blue *(poco)*, verde *(lunghezza d'onda media)* e rosso *(lunghezza d'onda alta)*.

Trichromacy theory

Cones are sensitive to different wavelengths (short, medium, long)

Hence, they absorb light around the spectrum of blue, green, and red

The theory says that we perceive color as a three-channel system. All color spaces, even if designed to different purposes, are three-dimensional

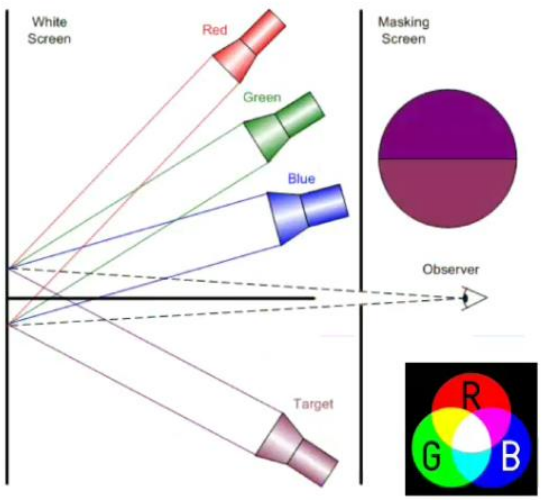


La tricromia si basa su questi tre colori principali, vedi l'RGB.

Colour measurement and specification

Since only three different receptors are involved in color vision, it is possible to match a patch of color light using a mixture of three color lights, called *primaries*

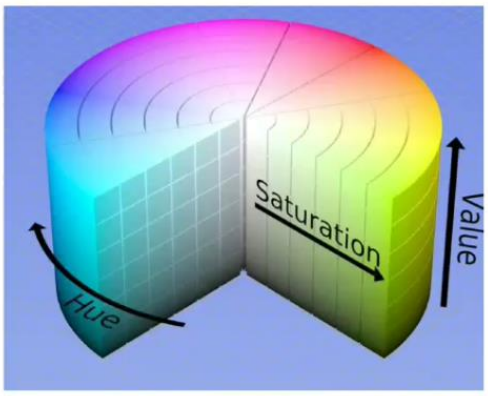
Given a *standard* set of primaries, one can use a transformation to create the same color on different output devices



Abbiamo visto che ci sono altri spazi di colore come l'HSV (Hue Saturation and Value).

HSV/HSL color space

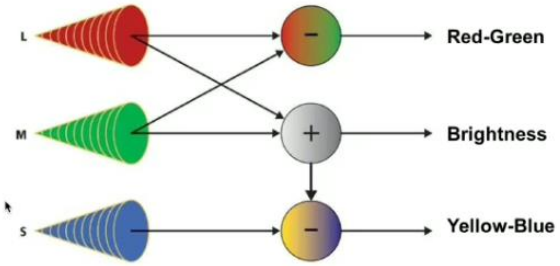
A cylindrical color space



L'altra teoria, quella dei processi opposti, per la quale i coni combinano i loro stimoli per formare tre coppie di colori. Una (bianco e nero) che definisce la luminosità, poi (rosso e verde) e (giallo e blu) che definiscono il colore vero e proprio.

Opponent process theory

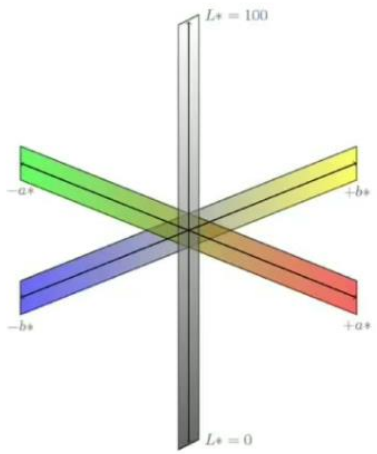
In the late 19th century, German psychologist Hering proposed the theory (later supported by experimental evidence) that cones combines their stimulus forming three pairs of colours that compete together to form the final one. These pairs, called *opponent pairs, are*: black-white, yellow-green, and yellow-blue



Seguendo questa teoria, è stato definito il Lab Color Space (L = asse Y su cui definisci bianco e nero, A e B due costanti che definiscono la quantità di verde&rosso e la quantità di blu&giallo). Con questo spazio di colore riesci a mostrare il tutto effettivamente come lo percepiresti nel mondo reale.

CIE Lab color space

Perceptually uniform color spaces (remember opponent process theory?)

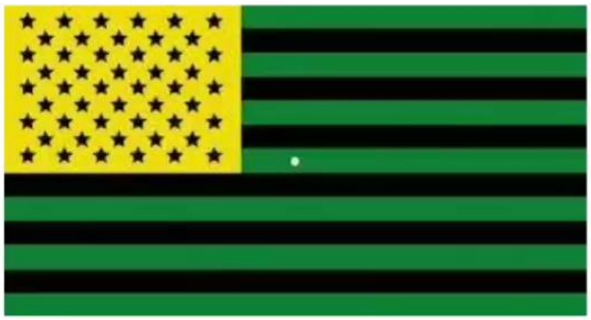


Nota: è stato dimostrato come i termini dei colori primari sono consistenti fra culture diverse!

Opponent process theory

Look at this picture for at least 30 (or 60) seconds, and look/focus at the little white dot that is in the middle.
Then, switch to a white slide: what you see on the white background is the flag of the United States.

That is because when you are staring at these colors, you are exciting the same source of these colors, and when you switch to the white background, since these sensors have been excited for too long, they inhibit those colors and the only colors.



Color differences

Though, uniform color spaces only provide a rough first approximation of how color differences will be perceived

An important influencing factor is size (we are much more sensitive to differences between large patches)

Tip: Use saturated colors when coding small symbols o thin lines, and less saturated colors for large areas

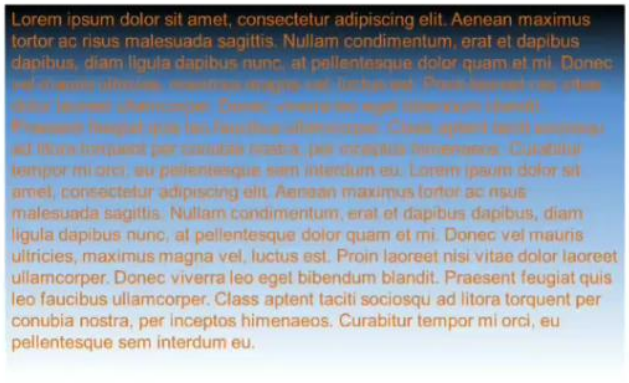
[Redrawn from C. Ware, Information for Visualization

Luminance and visualization

The red-green and yellow-blue chromatic channels are each capable of carrying only about 1/3 of the amount of detail carried by the black and white channel (Mullen, 1985)

Purely chromatic differences are not enough to display fine details

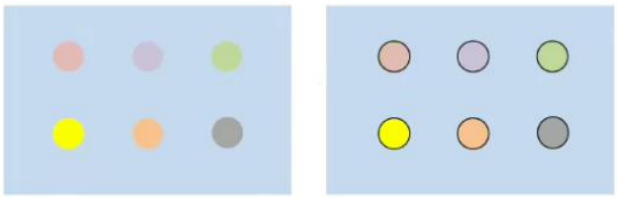
Ensure adequate luminance contrast with the background (also if colors with different chromaticity are used)



[Brown text on a blue gradient. From C. Ware, Information for Visualization]

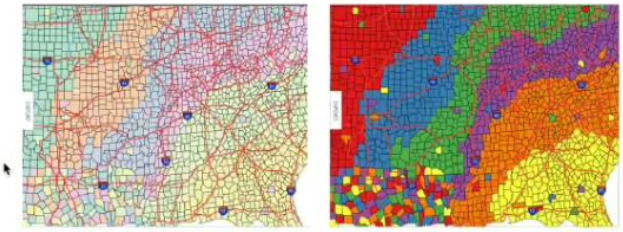
Luminance and visualization

A contrast boundary can improve the readability of colored symbols



Saturation and visualization

Use saturated colors for coding small symbols/fine details, and less saturated colors for coding large



Color for labeling

12 colors recommended by Colin Ware in its book: red, green, yellow, blue, black, white, pink, cyan, gray, orange, brown, purple

Widely agreed-upon category names, and reasonably far apart in color space

Solo una piccola parte di ciò che entra tramite i nostri occhi è effettivamente salvata in memoria!

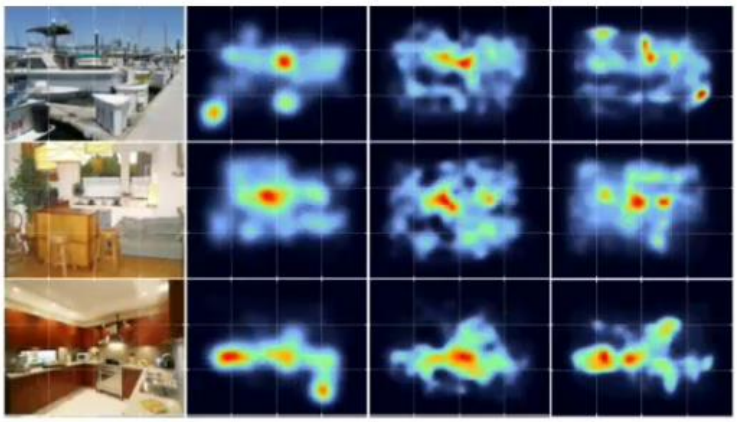
Saccading eye movements

Saccades take 20-180 ms. Both eyes move in the same direction. The movement may not be a simple linear trajectory

A fixation is composed of slower and fine movements (microsaccades, tremor and drift) that help the eye align with the target. A fixation takes 50-600 ms

Visual attention

We only focus our attention on a small part of the entire field of view



Domande orale Jacopo:

1) [Argomento a piacere]
2) Light Model -> Phong
3) Tutta la parte iniziale, simplessi, complessi, mesh, contorno, star, link, ..
4) Perché abbiamo definito star e link? perché quando andiamo a parlare delle manifold -> calcolare il vettore normale di una mesh (dato dalla somma pesata dei vettori sui triangoli che formano la mesh [quindi i triangoli all'interno della stella del vertice centrale])
5) Aree dove si può fare (baricentro, Voronoi, mixed)
6) Texturing O Trasformazioni affini
7) Cosa si utilizza per rappresentare dati multidimensionali
8) Visual decoding (espressività & efficacia)

Domande orale mio:

1. Curva -> Curvatura planare -> Curvatura massima e minima su Thorus (interno e esterno) / Sfera
2. Texture -> MIP-Mapping
3. Trasformazioni Geometriche (sia 2D che 3D) -> proiezione perspettiva e parallela
4. Principal Component Analysis
5. Visual Decoding -> Espressività ed Efficacia (accuratezza, discriminabilità, rilevanza, gestalt law, ...)
6. Colori -> Teoria tricromia e teoria colori opposti, in dettaglio