

Computer Graphics

Appunti A.A. 21/22

Introduzione	3
Modello digitale	5
Una categorizzazione dei tipi di modelli digitali 3D	7
Point Clouds	8
Polygonal Representations	11
Da quad mesh a triangle mesh	11
Da point cloud a triangle mesh	12
Mesh processing	19
Texture	26
Parametric Surfaces	31
Curve parametriche	31
Superfici parametriche	33
Rappresentazioni Volumetriche	36
Mesh poliedrale	36
dataset voxellizzati	37
Modelli impliciti	39
Da nuvola di punti a mesh	41
Dataset superficiali regolari	43
height-field	43
Range scan	43
Geometry images	44
Subdivision Surfaces (cenni)	45
Immagini digitali in breve	47
Rendering: ray-tracing	49
Algoritmi	51
Rendering: Rasterization	52
Trasformazioni Spaziali	55
La sequenza di trasformazioni nel rendering:	56
visione di insieme	56

Trasformazioni nel rendering: Model	61
Trasformazioni nel rendering: View	63
Trasformazioni nel rendering: Projection	66
Graphic Hardware, Drivers, API e librerie (cenni)	69
Rasterizing primitives	71
Rimozione di superfici nascoste	75
basata sull'orientamento	75
Back-face Culling	75
Algoritmo Depth test e Depth Buffer	76
Partecipating media - fog	79
Illuminazione	82

PARTE I: 3D data structures

Introduzione

Computer Graphics = is a sub-field of computer science which studies methods for digitally synthesizing and manipulating visual content

La CG ha come obiettivo creare immagini usando software e hardware.

3D computer graphics = sometimes called CGI, 3DCGI or three-dimensional computer graphics (in contrast to 2D computer graphics), are graphics that use a three-dimensional representation of geometric data (often Cartesian) that is stored in the computer for the purposes of performing calculations and rendering 2D images. The resulting images may be stored for viewing later (possibly as an animation) or displayed in real time.

Metodi per la sintesi e la manipolazione di visual content.

La CG ha diverse applicazioni:

- Manifattura → (CAD = Computer-Aided Design is the use of computers to aid in the creation, modification, analysis, or optimization of a design). In particolare il processo di design si avvale della CG prima di passare alla fase di produzione. Inoltre tramite simulazioni in CG si può verificare la integrità dell'oggetto o della infrastruttura.
- 3D printing
- Visualizzazione scientifica → ha come obiettivo ottenere immagini per comprendere meglio alcuni campi
 - Medica
 - Dei dati
- Intrattenimento
 - Movies → Nel settore cinematografico si ha come obiettivo il foto-realismo durante la creazione degli effetti visuali (≠ effetti speciali), ovvero la simulazione di un oggetto 3D in una foto (o frame) in maniera foto-realistica. Si cerca di evitare di cadere nella uncanny valley
 - Video Games
- Architettura
- Militari
- Realtà virtuale
- Cultural Heritage → in maniera da facilitare la fruizione e la conservazione nel tempo delle opere d'arte.

La CG risponde alle domande:

- Come si rappresenta una entità 3D?

Le immagini 2D si codificano in diversi modi come ad esempio una griglia di pixel da comprimere o tramite una stringa ASCII. Le entità 3D sono composte da:

- Superfici
- Volumi
- Scene

Anche le entità 3D hanno diverse codifiche e le vedremo più avanti e vedremo diverse **strutture dati**.

rendering = is the process of generating a final digital product from a specific type of input. Quando si sintetizza l'immagine digitale di qualcosa.

3D rendering = is the 3D computer graphics process of converting 3D models into 2D images on a computer.

- Come si renderizzano queste entità? Ovvero come si passa da rappresentazione digitale 3D a visualizzazione **immagine digitale** a schermo?

Ci sono due modi di render che differiscono sul tempo di creazione delle immagini digitali:

- Real-time = in tempo reale. Si fa questo tipo di render nelle applicazioni interattive.
- Offline

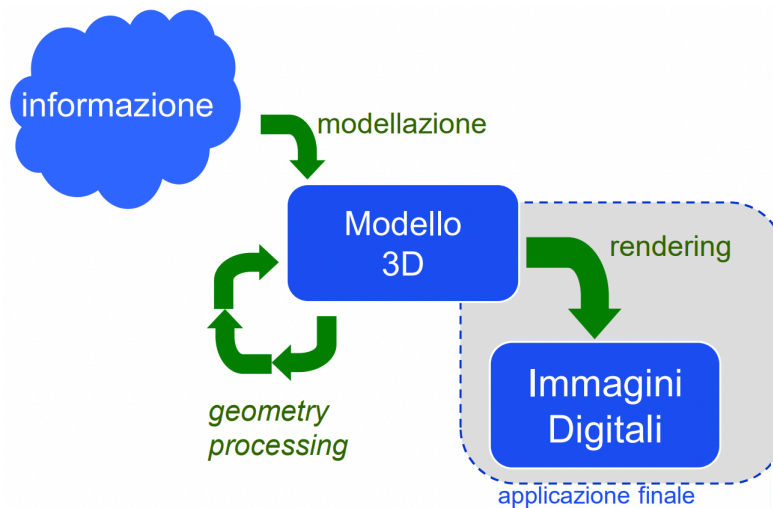
Vedremo diversi **algoritmi** che permettono questi due tipi di rendering.

//La CPU differisce dalla GPU poiché deve fare più calcoli in parallelo. Lo sviluppo nel campo della CG ha portato a un ampio progresso delle GPU che oggi vengono usate anche in campi diversi da quelli per cui sono stati originariamente creati (come bitcoin e machine learning, in questo caso si parla di GPGPU)

Seguendo uno schema di pipeline per l'acquisizione di un modello 3D a partire da un oggetto:

- Si parte da un oggetto reale
- Si fa una acquisizione di un modello 3D tramite una scansione dell'oggetto
- Si ha l'immagine 3D su cui si possono fare diverse operazioni (visualizzazione, analisi, catalogazione, studio,...)

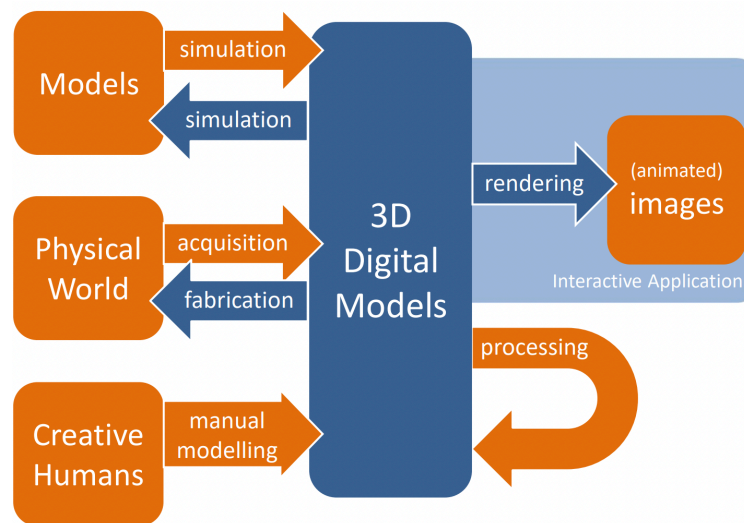
In generale tutte le applicazioni della CG citate in precedenza hanno in comune una informazione che ha va tradotta (o meglio modellata) in un modello 3D. In seguito, tramite rendering, si crea una immagine digitale.



Geometry Processing = or mesh processing, is an area of research that uses concepts from applied mathematics, computer science and engineering to design efficient algorithms for the acquisition, reconstruction, analysis, manipulation, simulation and transmission of complex 3D models.

Sono tutti i procedimenti che hanno come input e output un modello 3D.

Modello digitale



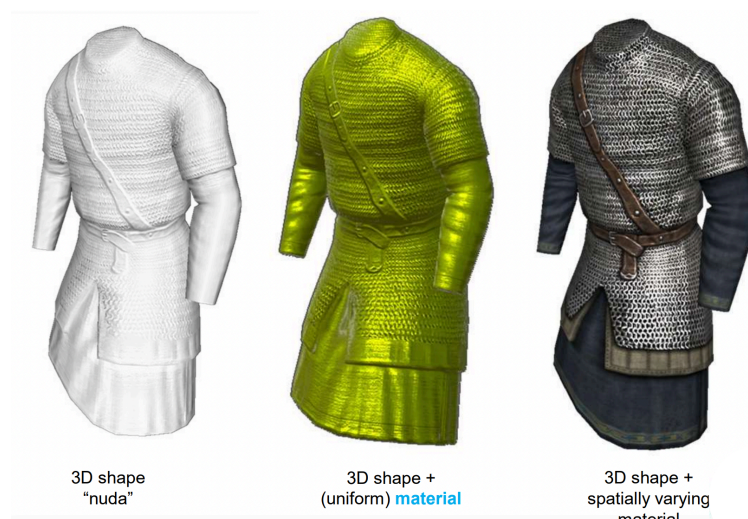
Un modello digitale è costruibile a partire da:

- **Simulazioni** (es. modello di maremoto) → A sua volta il modello 3D produce nuova informazione sul modello astratto che lo ha generato
- **Cattura del mondo reale** (acquisizione 3D come una scansione laser o una TAC) → Queste catture vengono restituite al mondo reale tramite la loro realizzazione fisica, come:
 - Fabrication = fabbricazione (come 3D printing o altre forme di rapid prototyping)
 - Manufacturing = industria manifatturiera (in grande scala)
- **Modellazione manuale** da parte di artisti digitali

I modelli sono classificabili come:

- **Volume-based** = Volumetrici, modellano l'oggetto come un volume. Rappresentato sia l'interno, che il bordo, che (a volte) anche l'esterno dell'oggetto (esempio: l'output di una TAC).
- **Boundary representations** = (3D surfaces) Di superficie, modellano solo il bordo dell'oggetto 3D, cioè una superficie. Questo modello è meno oneroso in termini di memoria ed è quello più comuni, infatti degli oggetti solidi (a meno che non siano trasparenti o traslucidi), si vede solo il bordo. Quindi il rendering richiede solo questo.

Materiale = in un modello 3D il materiale è la descrizione di come un oggetto reagisce alla luce. Contiene anche l'informazione del colore del modello e altre caratteristiche ottiche. Questo può variare spazialmente. È una informazione contenuta nel modello.



Risoluzione = (detta anche complessità) in un modello 3D la risoluzione viene determinata dalla sua quantità di poligoni o celle volumetriche. La risoluzione può essere:

- bassa → basso costo (in termini di RAM per storage, tempo di rendering...) usata spesso in applicazioni real-time.
- alta → alta qualità (in termini di quantità di dettaglio modellato, aderenza del modello all'originale...) usata per applicazioni offline.

La risoluzione di un modello 3D deve essere adeguata al contesto applicativo. Per ogni categoria di modello 3D, il concetto risoluzione si declina in maniera diversa.



Alcune strutture dati per modelli 3D ammettono risoluzione:

- **Adattiva** = risoluzione diversa in parte diverse dell'oggetto, contemporaneamente. (Per esempio in modo che dipende dalla loro importanza semantica, o dalla quantità di dettaglio geometrico locale).
- **Costante** = (fixed) come ad esempio in un'immagine raster, infatti non ha risoluzione adattiva: se sono 100 DPI (dot per inches) in alto a sinistra, sono 100 DPI in tutta l'immagine.

multi-risoluzione = livelli diversi di risoluzione per lo stesso modello. Lo stesso oggetto è modellato da un modello a risoluzione diversa in momenti diversi, da usare in modo dinamico in base all'occorrenza. Infatti consentire di utilizzare di volta in volta la risoluzione più adatta alle mutevoli situazioni.

Ci sono due varianti della risoluzione:

- **discreta** = anche detti livelli di dettaglio discreti, si memorizza una piccola collezione di modelli 3D, ciascuno ad una risoluzione diversa, per es «alta» «media» «bassa»
- **continua** = quando è possibile estrarre dinamicamente da un'unica struttura dati un modello ad una risoluzione richiesta

Le capabilities (capacità) dei modelli 3D possono essere:

- **Editati** → Facili da editare/modificare o non consentono facilmente editing/modifiche
- **Illuminati** → static lighting only o relightable
- **Oggetto di simulazione** → Simulation grade ovvero adatti ad essere il soggetto di una simulazione o meno, a causa di vari difetti, come per esempio rumore, incompletezza, inconsistenze.
- **Animati** → Staticità o capacità di essere animato
- **Stampati** → Il modello deve avere una coerenza interna, consistenza che consente a un oggetto di usare il modello per una stampa (ad esempio, i modelli superficiali devono essere «chiusi», cioè devono separare un interno da un esterno)

Una categorizzazione dei tipi di modelli digitali 3D

		ELEMENTI DISCRETI			CONTINUI
		regolari «a griglia»	semi-regolari o irregolari		
			elementi simpliciali	elementi non simpliciali	
SUPERFICIALI	2-manifold «rappresenta una vera superficie»	Height Field Range Scan Geometry Images	Triangle Mesh	Polygonal Mesh Quad Mesh Quad dominant Mesh	Subdivision surfaces Parametric Surfaces (es. B- splines)
	non-manifold «non rappresenta una sup»	Set di Range Scan		Point Cloud	
	VOLUM ETRICI	(3-manifold)	Voxelized Volume Volumetric Textures	Tetra Mesh	Hexa Mesh

Point Clouds

È un insieme di campioni 3D di superficie. Non c'è alcuna struttura o relazione esplicita fra questi punti. Può essere visto come un array di vertici e per ogni campione è indicato:

- La sua **posizione** p (un punto)
- La sua normale **normale** n (un vettore unitario che rappresenta la local surface orientation)
- A volte, alcuni **attributi**, come il colore o altri dati che descrivono il materiale o cose dipendenti dal contesto applicativo, come temperatura, pressione, etc. in una simulazione.

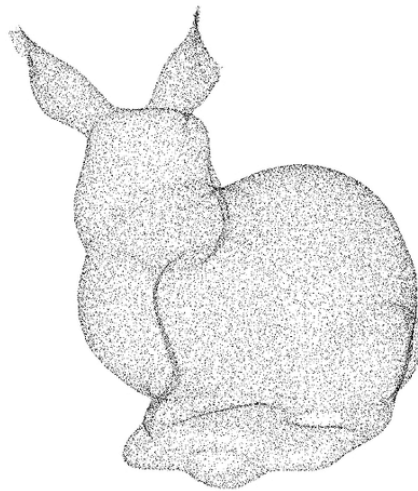
Tipicamente è un insieme molto numeroso che presenta difetti, come rumore, gaps, overlaps, e outliers (= punti errati, che non stanno affatto sulla superficie).

La nuvola di punti è un dato superficiale che descrive il boundary dell'oggetto 3D (la superficie) attraverso un semplice campionamento, ogni campione rappresenta un punto sulla superficie.

Le point cloud sono facili da **catturare**, da qui la loro popolarità.

fotogrammetria = creazione di point cloud a partire da fotografie.

Le point clouds sono facili da processare e da acquisire ma spesso di cattiva qualità.



Non sono però facili da utilizzare direttamente da un punto di vista:

- matematico → non definiscono davvero una superficie (ad es. una point cloud non separa un esterno da un interno, quindi, non è possibile determinare il volume di un modello)
- algoritmico → non esiste una relazione di vicinato fra punto (ad es. non è immediato trovare i campioni vicini ad un campione dato)

Altre caratteristiche della nuvola di punti:

- La sua **risoluzione** è definita dal suo **numero di punti** (o vertici)
- La risoluzione può essere **adattiva** (campionamento non necessariamente uniforme) e multi-risoluzione (basta prendere un sottoinsieme dei punti) e quindi continua
- può essere **re-illuminata** grazie alle normali
- Può essere **editata** con molta difficoltà (in pratica no)
- Non è printable

Il rendering di una point cloud avviene attraverso il **point splats** = uno **splat** è una regione di schermo che viene disegnata per rappresentare il punto.

La dimensione dello splat dipende dalla densità dei punti, la distanza, etc (si possono disegnare anche come pezzetti di superficie, questo è possibile grazie alle normali). La point cloud è semplice da gestire perché ogni punto è indipendente dagli altri. Ad esempio, una semplice multi-risoluzione si ottiene semplicemente prendendo un sottoinsieme. Strategia:

- preordinare la point cloud di N vertici in modo che i primi $M < N$ vertici siano ben distribuiti, per ogni M (meglio con random shuffle → buona approssimazione)
- in fase di rendering, disegnare solo i primi M vertici per avere una point cloud di risoluzione M

//Ripasso di algebra vettoriale

La struttura dati point cloud viene memorizzata su disco come un array di vertici contenenti la posizione e la normale, ovvero il suo orientamento.

$$normale = 1 = \sqrt{x^2 + y^2 + z^2} = \hat{n}$$

Il lighting di una nuvola di punti è calcolata a partire dalla normale. Senza le normali non si può cambiare la luminosità.

Tramite il **geometry processing** è possibile costruire le normali per ogni punto di una point cloud, se sono assenti.

Si lavora su ogni punto guardando il suo intorno, se condividono un piano allora si è trovata la normale. Il problema è: come si fa a trovare l'intorno dei punti (vicinato spaziale di ogni vertice)?

- Si mette un valore soglia ϵ e si includono tutti i punti a distanza ϵ dal punto preso in esame. Questo modalità è molto onerosa in termini di complessità e efficienza $\rightarrow O(n^2)$ dove n è il numero di punti. È anche difficile determinare quanto grande (o piccola) deve essere la soglia, infatti dipende dalla scala del modello (scale dependent).

$$||a - b|| \leq \epsilon^2 ||punto||$$

- Siccome è un modello adattivo è meno oneroso prendere un certo numero di vicini a un punto per calcolare la normale. Questo processo è chiamato **K-Nearest Neighbors** (= K-NN) traducibile come primi k-vicini. Preso questo insieme se ne ricava un ripiano e si estrae la normale. Complessità lineare $\rightarrow O(n)$

Un problema comune a entrambi i metodi è che non si conosce in realtà se la normale ricavata è nella direzione giusta o ribaltata, infatti un piano ha due normali di segno opposto. Si può risolvere ricontrollando gli scatti usati per creare il modello, infatti la normale è uscente verso l'obbiettivo.

Come si esegue un **offset di superficie**? ovvero come spostare i punti della point cloud di un certo k ?

$$\forall p \in P : p_i \leftarrow p_i + k \hat{n}_i$$

Una volta fatta la stima delle normali si procede con:

- **denoising** = rimuovere il rumore dalla point cloud, tentando di preservare il segnale. Si sposta ogni campione in modo da avvicinarlo alla superficie reale. Si suppone che le superfici tendano ad essere lisce: superfici molto frastagliate devono essere rese più lisce avvicinando ogni punto al piano passante per il suo vicinato.
- outlier removal

Spesso point clouds diverse fanno riferimento a parti diverse di uno stesso oggetto. Ad esempio, ho catturato due lati di una statua con due procedimenti, come laser scanning. Un task che vogliamo risolvere in questo caso è spostare (ruotare e traslare) una delle due in modo da portare le parti in comune a coincidere (quindi ci devono essere quindi sovrapposizioni). Questo problema viene detto **allineamento** (o registrazione) di point cloud. Risolvere l'allineamento significa applicare l'algoritmo **Iterative Closest Points** (= ICP):

1. Scegliere un insieme arbitrario di punti di una delle due nuvole (per esempio, qualche centinaio)
2. Per ognuno di loro, trovare il suo corrispondente sull'altra nuvola, come il punto più vicino ad esso
3. Scartare i punti che hanno il corrispondente troppo lontano, (oltre un valore soglia)
4. Trovare una rotazione + traslazione che avvicini il più possibile ogni punto sul suo corrispondente (sotto problema matematico)
5. Ripetere dal punto 1, fino a convergenza.

Ad ogni iterazione, le due nuvole si avvicinano e quindi si avranno corrispondenze migliori, e così via. Se nella situazione di partenza le due nuvole sono sufficientemente vicine alla soluzione

corretta, allora il sistema converge ad una soluzione **molto buona** (allineando le due point clouds in modo perfetto. Altrimenti, il sistema può convergere a soluzioni del tutto sbagliate, o non convergere del tutto. E' necessario partire da una soluzione che si avvicina a quella corretta. Distinguiamo quindi due problemi (due fasi), da risolvere in cascata:

- **Coarse registration** → (allineamento grossolano) trovare una prima soluzione approssimativa che metta in corrispondenza le due point-cloud in modo non necessariamente accurato (tramite machine learning o geometria computazionale)
- **Fine registration** → (allineamento fine) tramite ICP

Il task più comune è convertire la point-cloud in un'altra rappresentazione, più adatta a processing o a rendering. Bisogna ricordare che le point cloud sono diffuse soprattutto in virtù della loro facilità di acquisizione 3D (spesso da immagini). Tipico esempio: da point cloud a mesh poligonale.

//Le mesh poligonali possono essere viste come delle point clouds con i vari punti connessi da poligoni.

Polygonal Representations

Le **mesh poligonali** sono un insieme di elementi poligonali, dette **facce**, adiacenti lato a lato. Le facce sono composte da:

- **Vertici** = angoli della faccia
- **Edges** = (spigoli) lati della faccia, sono coppie di vertici

Le mesh poligonali possono essere viste come dei grafi dove i nodi sono i vertici e gli archi sono gli edges. Sono il modello 3D digitale più diffuso. E' GPU friendly, infatti le schede video sono pensate per renderizzare le mesh.

Connettività = (detta anche topologia) insieme delle **facce** che connettono/collegano i punti dell'insieme. Dice come sono connessi i vertici. Describes the incidence relations among mesh elements (e.g., adjacent vertices and edges of a face, etc).

Geometria = insieme dei **vertici**, ciascuno con pos (x,y,z), che compongono una mesh poligonale. È un campionamento della superficie. Specifies the position and specifies the position and other geometric characteristics of each vertex.

Elemento simpliciale = Ogni faccia della mesh è un poligono semplice (triangolare).

Triangle mesh = (o mesh simpliciale) è una mesh composta da poligoni di 3 vertici. Sono molto diffusi perché sono le mesh più facili da renderizzare, infatti l'HW del pc è circuitato al render di mesh triangolare. In generale il numero di facce è circa il **doppio** del numero dei vertici (Es.: una mesh triangolare di 10 facce ha circa 5 vertici). Questa mesh ha un vantaggio molto importante, le facce sono sempre **planari** perché tre punti nello spazio sono sempre co-planari.

Quad mesh = è una mesh composta da quadrilateri, poligoni di 4 vertici. In generale il numero di facce è circa **uguale** al numero dei vertici (Es.: una mesh triangolare di 12 facce ha circa 10 vertici). Questa tipologia di mesh è più vicina a come un oggetto viene progettato da un umano (CAD) ma viene poi scomposto in triangoli. Più difficili da generare automaticamente.

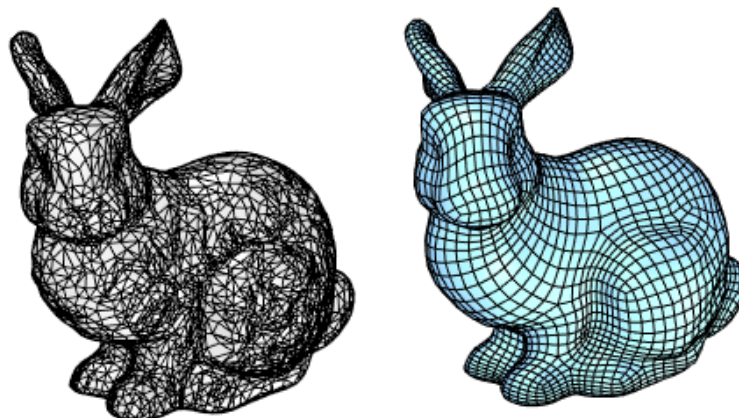
Mesh poligonali = (o ibride) è un termine generico che spesso indica un mix di mesh triangolari e quad (come la quad dominant).

Da quad mesh a triangle mesh

L'HW lo fa spesso in automatico, usando il **diagonal spilt**, ovvero dato un poligono con n lati lo si divide per le diagonali in modo da creare triangoli. Il numero di triangoli in un poligono sarà:

$$\text{\#triangoli} = n - 2$$

Più aumentano i vertici, più aumentano i modi di fare diagonal spilt.



Da point cloud a triangle mesh

Questo problema di ricostruzione della superficie è risolvibile tramite il **geometry processing**.

Front advancing = è una classe di algoritmi in cui si aggiunge un triangolo alla volta finché non si chiude il modello.

Ball pivoting = (rotolamento di sfera) è un algoritmo che si basa sul front advancing. Si fa ruotare una sfera sulla nuvola di punti finché non crea una mesh chiusa.

Il raggio della sfera, che spesso dipende dalla **densità** della point cloud, è importante, se è:

- Grande → si avrà come risultato una mesh formata da triangoli grandi, perdendo molti dettagli
- Piccolo → triangoli piccoli, molti dettagli.

Questa tecnica è però problematica in quanto:

- rischia di includere rumore a causa degli outlier
- i punti esterni dominano quelli interni

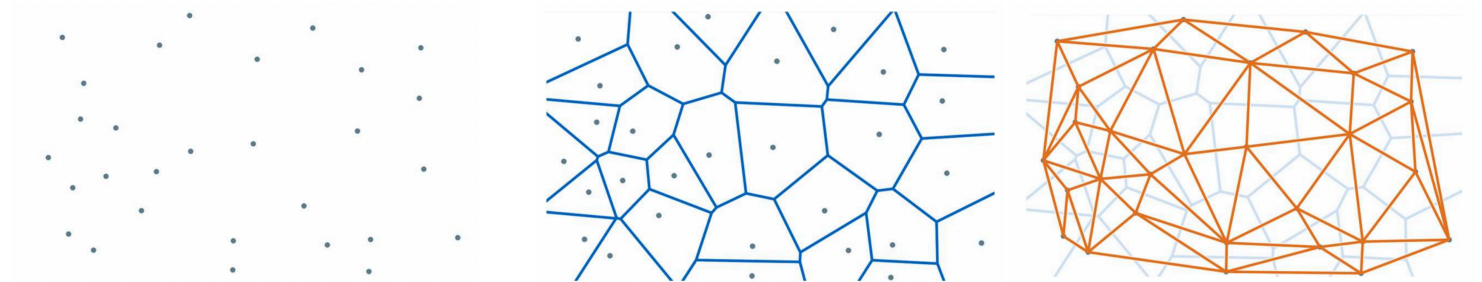
Per togliere gli errori si può fare una media su diverse misurazioni (si calcolano vari modelli e se ne fa una media).

Speso prima di passare direttamente da point cloud a mesh esiste un passaggio intermedio, dove la point cloud diventa un dato **volumetrico** (rappresentazione volumetrica), in particolare **DFS** (signed distance field).

Da nuvola di punti a triangle mesh in 2D il problema ha un'ottima soluzione: la triangolazione di Delaunay, usando il diagramma di Voronoi (è la soluzione di un problema di geometria computazionale).

Diagramma di Voronoi = Una scomposizione in regioni di piano indotta da n punti 2D (seed). Ogni seed produce una regione. Una regione di un seed s_j è formata da l'insieme di punti p del piano che sono più vicini a s_j che ad ogni altro seed.

Triangolazione di Delaunay = Il duale di un diagramma di Voronoi, è la connettività ottenuta connettendo con un edge ogni coppia di seed di regioni confinanti. Si ottiene una triangolazione con ottime proprietà, come ad esempio: il **circocentro** che inscrive ogni triangolo non include nessun altro seed (per ogni 3 punti si fa passare un cerchio in modo che non includa nessun altro vertice. Si cerca di evitare di inserire triangoli troppo stretti).



Alcuni metodi aggiungono solo la **connettività**:

- la nuvola di punti costituisce già la geometria della mesh
- Front Advancing, come Ball Pivoting
- In 2D: Delaunay triangulation (da Voronoi diagram)

Altri metodi **ricampionano** anche i vertici come l'uso di rappresentazioni volumetriche intermedie. È utile quando la una nuvola di punti è affetta da errori e outlier, infatti questi metodi consentono di ottenere denoising e outlier removal come effetto collaterale.

Geometria:

- Procedurale → dice come gestire dati 3D in modelli 3D
- Computazionale → risolve problemi di geometria

Risoluzione → il numero di facce (o di vertici) che compongono la mesh. Può essere:

- Hi-res → più accuratezza. Più facce ci sono migliore è livello di approssimazione e si ha come risultato una risoluzione maggiore, diventa quindi più oneroso da renderizzare.
- Low-res → più efficienza, detta anche low-poly mesh

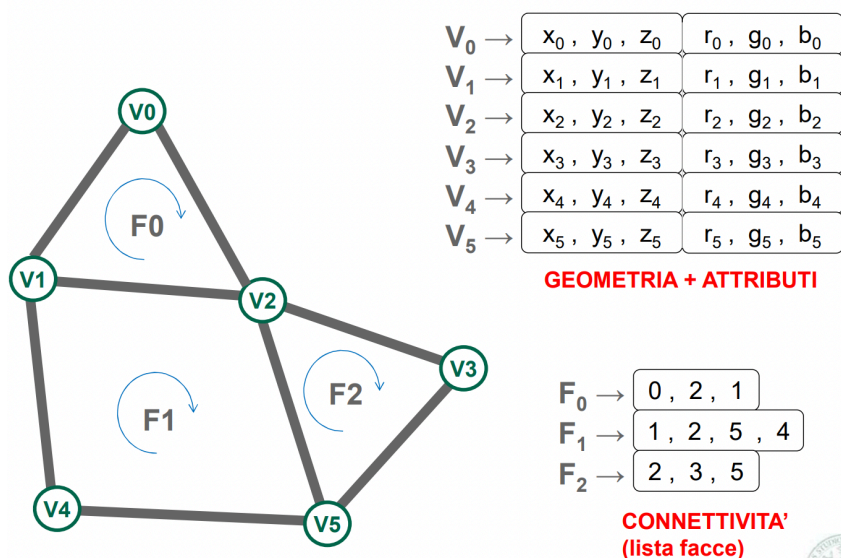
Le mesh poligonali hanno una **risoluzione adattiva**, infatti dove la superficie è piana ci sono meno triangoli, mentre c'è un tassellamento più fine (campionamento più fitto) dove necessario.

In media un modello poligonale quante facce ha? Dipende, infatti più la tecnologia avanza più è possibile usare sempre più poligoni (la risoluzione aumenta). Dipende anche dal modello che si vuole realizzare.

Quanto deve essere grande un triangolo su un display per sembrare seamless? 4,5 pixel

Come rappresento internamente una mesh? Ovvero, come si memorizzano le mesh in una **struttura dati**?

- **lungo vettore di poligoni** = (anche detto: zuppa di triangoli) per ogni poligono (di n lati) c'è un vettore di n vertici, dove per ogni vertice viene memorizzata la posizione (x,y,z) ed eventuali attributi (colore r,g,b, e/o altro). Questa struttura ha però un grosso difetto: la replicazione dei vertici, con conseguente spreco di spazio e complessità per updates (bisogna mantenere le copie uguali durante il processing).
- **indexed mesh** = dove:
 - **Geometria** → array di **vertici** ($V[0] \dots V[N]$). in ogni vertice, per ogni vertice viene memorizzata la posizione e attributi.
 - **Connettività** → Array di **poligoni** (lista **facce**). Ogni poligono viene rappresentato da una sequenza, ordinata e ciclica, di indici di vertice, uno per «angolo di faccia» (o «corner», o «wedge»).



In C++

```
class Vertex {
    vec3 pos;
};
class Face{
    vector<int> vertexIndex;
    rgb color; /* attribute 1 */
    vec3 normal; /* attribute 2 */
};
class Mesh{
    vector<Vertex> vert; /* geom + attr */
    vector<Face> face; /* connettivita' */
};
```

OBJ e OFF (Object File Format) sono i formati più diffusi. Qua sotto si può vedere un esempio di OFF, dove le prime due righe di sinistra sono l'header, il resto della colonna di sinistra esprime la geometria mentre la colonna di destra la connettività.

facce

edge

vertici

OFF

12

10

0

0.0 0.0 0.0

3.0 0.0 0.0

3.0 1.0 0.0

1.0 1.0 0.0

1.0 5.0 0.0

0.0 5.0 0.0

0.0 0.0 1.0

3.0 0.0 1.0

3.0 1.0 1.0

1.0 1.0 1.0

indice 0

indice 1

indice 2

indice 3

x,y,z

2ndo vert

LetteraL.off

1 5 1

0 5 1

4 3 2 1 0

4 5 4 3 0

4 6 7 8 9

4 6 9 10 11

4 0 1 7 6

4 1 2 8 7

4 2 3 9 8

4 3 4 10 9

4 4 5 11 10

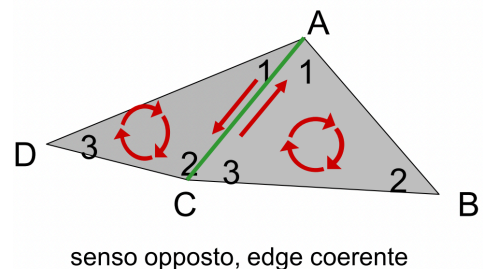
4 5 0 6 11

prima faccia:
4 vertici:
con indici
3, 2, 1 e 0

The **orientation** of a face is a cyclic order of the incident vertices.

The orientation of two adjacent faces is **compatible**, if the two vertices of the common edge are in opposite the common edge are in opposite order.

Una mesh si dice **ben orientata** se, nelle sua struttura dati, tutti gli edge sono percorsi nello stesso senso (orario o antiorario).



A mesh is a **manifold** if (1) each edge closed fan is incident to only one or two faces and (2) the faces incident to a vertex form a closed or an open fan.

A manifold mesh is **orientable** if any two adjacent faces have compatible orientation.

Two-manifold = (una "varietà due") rappresenta in effetti una superficie. molti algoritmi di geometry processing necessitano che questo sia il caso. Non tutte le mesh lo sono. Le facce di una mesh rappresentano sempre (pezzi di) superficie, ma su edge e vertici le cose possono andare storte. Ci sono delle condizioni che si devono verificare:

- un edge interno è two-manifold se è condiviso da **al massimo due facce**
- un vertice è two manifold se ha **un solo fan** (ventaglio = è un insieme di facce connesse edge a edge che condividono in unico vertice)

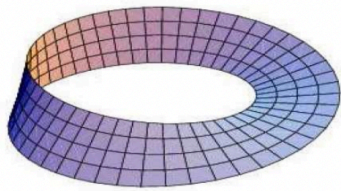
Una mesh two manifold può essere ben orientata oppure no.

Non manifold = se non rispetta le condizioni di una mesh two-manifold.

Aperta = se una mesh ha edge di bordo (un edge condiviso da solo 1 faccia è di **bordo**, mentre un edge condiviso da 2 facce è **interno**). Quando una mesh è aperta si può calcolare il volume del modello 3D.

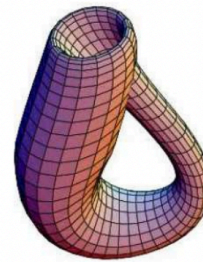
Chiusa = se una mesh non ha edge di bordo

Non tutte le mesh sono orientabili (invertire il senso di percorrenza dei vertici di una faccia non serve, perché la mesh rimane non orientabile) come il Nastro di Moebius (non orientabile, aperta) e la Bottiglia di Klein.



Nastro di Moebius

(non orientabile, aperta)



Bottiglia di Klein

(non orientabile, chiusa)

Una mesh two-manifold, ben-orientata e chiusa separa uno spazio interno da quello esterno e quindi rappresenta un oggetto **solido**. Possiamo calcolarne il volume (oltre che l'area) e il suo baricentro, possiamo convertirla in un modello 3D volumetrico, dato un punto, possiamo calcolare se è esterno o interno, possiamo stamparla in 3D.

Attributi = dati addizionali che si possono memorizzare per ogni:

- **Faccia** → in questo caso si indica un valore che rimane costante al suo interno (quindi discontinuità $C0$, non uniformità, fra le facce in corrispondenza, cioè, degli edge)
- **Vertice** → gli attributi dei vertici di una faccia vanno interpolati/mischiati dentro la faccia stessa

Gli attributi dipendono dalla applicazione e modellano quantità che variano sulla superficie:

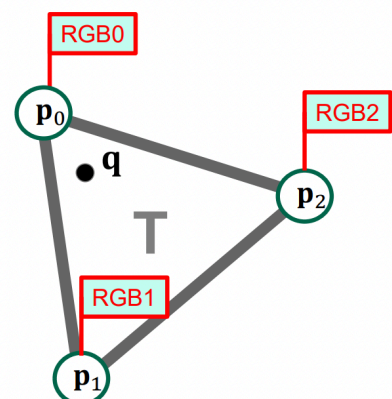
- Colore → diffusivo, tipicamente espresso come RGB, diventa parte della descrizione del materiale
- Vettore normale → quale orientamento ha la superficie in quel punto?
- Temperatura e pressione alla superficie (in una simulazione fisica)
- "Coefficiente di vulnerabilità" (in un gioco)
- Qualità della ricostruzione, in un modello di un oggetto reale (quanto è accurata la superficie in un punto)

Gli attributi possono essere di qualsiasi tipo: **scalari** (es: temperatura, qualità...) o **vettori** (es: colore, normale...).

L'interpolazione degli attributi definiti sui vertici è definita dentro a facce triangolari (mesh simpliciali). Quindi, per prima cosa, gli altri poligoni devono essere suddivisi in triangoli.

Ad ogni punto q in un triangolo T viene implicitamente attribuita un'interpolazione di dei tre attributi definiti sui vertici di T , usando, come peso dell'interpolazione, le **coordinate baricentriche** di q in T . Gli attributi, salvati sui vertici, sono così estesi implicitamente su tutta la superficie.

La GPU è specializzata nel computo delle coordinate baricentriche di punti dentro ai triangoli, e la conseguente l'interpolazione degli attributi definiti sui vertici. Durante il rendering, questo procedimento viene fatto per ogni pixel che compone



l'immagine un triangolo (nota: i pixel corrispondono a punti dentro le facce di una mesh, non a vertici di una mesh).

Un triangolo T ha tre attributi colore assegnati ai suoi vertici p_0 , p_1 e p_2 . Prendiamo un punto q su T di coordinate baricentriche k_0 , k_1 e k_2 in T , cioè con

$$q = k_0 p_0 + k_1 p_1 + k_2 p_2$$

Allora a q assegno il colore:

$$q = k_0[RGB0] + k_1[RGB1] + k_2[RGB2]$$

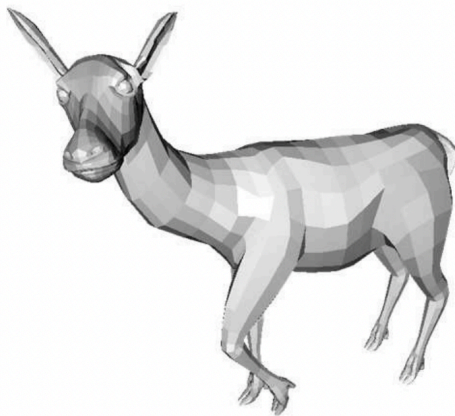
Se le normali di una superficie sono

- costanti $\rightarrow$ superficie piatta
- variano con continuità $\rightarrow$ superficie curva
- variano con discontinuità $\rightarrow$ superficie con un crease (il crease è costituito dai punti dove la normale è discontinua)

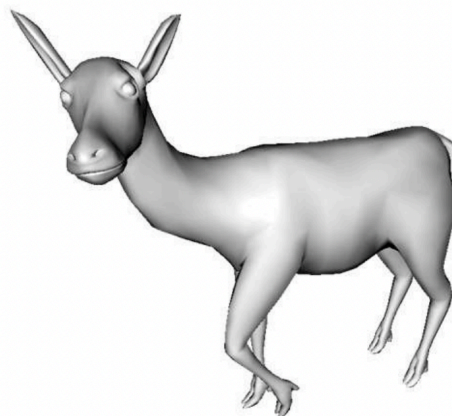
Nel caso in cui l'attributo è la **normale** per:

- faccia $\rightarrow$ si ha come risultato una normale costante per faccia, si hanno facce (dall'aspetto) piatto. **Discontinuità C0** sugli edge, gli edge sono "spigoli taglienti", detti edge di crease, o hard edges. Coerentemente col modello digitale, ma a differenza (spesso) dell'oggetto 3D che si intendeva rappresentare.
- vertice $\rightarrow$ si ha come risultato una normale che varia sulla faccia (vengono interpolate dentro le facce, come qualsiasi altro attributo), si hanno facce (dall'aspetto) curvo. **Continuità C0**, aspetto smooth (liscio) (ma non **continuità C1**)

Normali per faccia



Normali per vertice



Le normali risultano visibili all'osservatore attraverso l'**illuminazione** del modello digitale. Il calcolo dell'illuminazione è un task del **rendering**.

Come ottengo una faccia che appaia piatta? Uso una stessa normale come attributo dei tre vertici di un triangolo

Come abbiamo visto, interpolando vettori unitari si ottengono vettori non unitari, che vanno quindi **ri-normalizzati**. Questo significa che gli attributi di tipo vettore unitario, come le normali, devono essere rinormalizzati dopo ogni l'interpolazione.

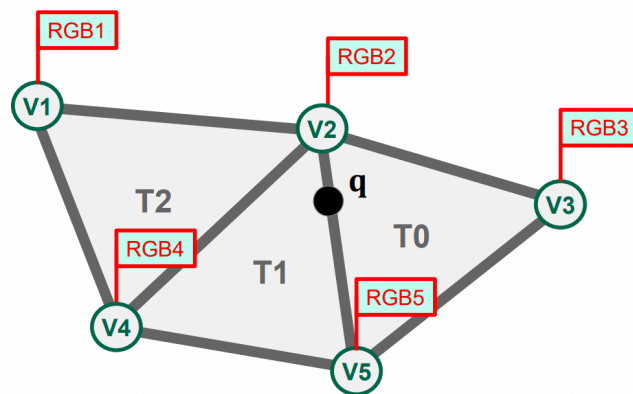
Esercizio: determinare con una stima se questo comporta un onere di calcolo eccessivo, per una GPU, durante un rendering in real-time.

Stimare quante operazioni in virgola mobile (Floating Point Operation) sono necessarie per normalizzare un vettore. Soluzione: circa 10.

Stimare quante volte sarà necessario ri-normalizzare un attributo «normale» in un rendering. Soluzione:

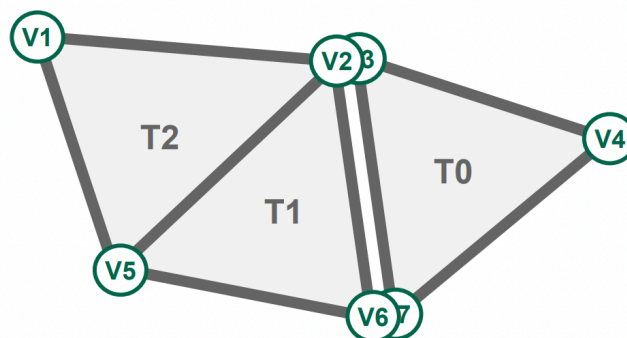
- stimiamo, a braccio, 1 volta in ogni pixel, quindi circa $1000 \times 1000 = 10^6$
- frame per secondo, in un rendering in tempo reale: stimiamo 60
- totale Floating Point Operation al Secondo (FLOPS) necessari a questo passaggio: $10 \times 10^6 \times 60 = 6 \times 10^8 = 0.6$ Giga-FLOPS
- Una GPU in commercio è capace di erogare 2/3 Tera-FLOPS = migliaia di Giga-FLOPS

Gli attributi definiti per vertice (interpolati dentro le facce) sono per costruzione continui (C_∞) dentro alle facce. Anche fra coppie di facce adiacenti, abbiamo una continuità di attributo (C_0).



Infatti: si prenda un punto q a cavallo fra T_0 e T_1 . L'attributo associato a q , se considerato parte di T_0 è lo stesso attributo è associato a q , se lo si considera parte di T_1 .

Se è necessario, è possibile modellare delle discontinuità (C_0) di attributo lungo un edge, ma solo duplicando i vertici ai suoi estremi. La mesh avrà due vertici geometricamente coincidenti (stessa posizione) ma con attributi associati differenti. Due edge tecnicamente aperti saranno quindi perfettamente sovrapposti, causando una superficie continua (senza «spiragli»). Facce adiacenti useranno una oppure l'altra di questa coppia di vertice.



Questa configurazione si chiama **seam** o vertex seam.

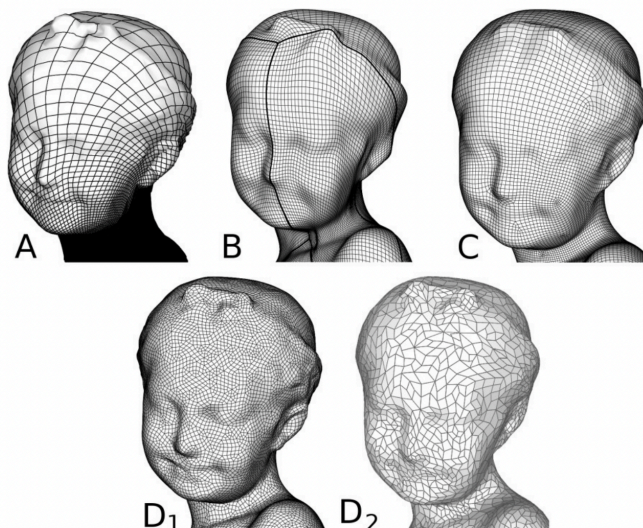
La regolarità di una mesh è determinata dalla sua **valenza**.

Valenza di un vertice = è il numero di facce o di edge che condividono (che sono adiacenti) ad quel vertice.

Vertice regolare = (per vertici interni) quando una quad mesh ha valenza 4 per ogni vertice (Quad mesh regolare = è un grigliato) e una tri mesh ha valenza 6 per ogni vertice (3 per hexagonal mesh).

Le tri-mesh e le quad-mesh (pure o dominant) hanno un grado maggiore o minore di regolarità, che dipende da quanti dei loro vertici sono regolari:

- **Tutti** = mesh (perfettamente) regolare o structured
- **Quasi tutti** = mesh semi-regolare.
- **Pochi** = (per es 2/3 o la metà) mesh **irregolare**



Le mesh regolari e semi regolari hanno alcuni vantaggi come:

- **flessibilità** → (rispetto alle mesh pure-quad: facile ottenere la forma voluta)
- **editabilità** → grigliati regolari: sono intuitivi e utili
- sono **comprimibili** con algoritmi di compressione
- **minore errore geometrico** a parità di risoluzione (di solito)

Una mesh regolare ha delle direzioni di edge ben definite, e seguono il senso della **curvatura** (la curvatura può essere vista come una misura della variazione della normale). Gli edge in una mesh regolare si trovano nel punto di incontro di massima e minima curvatura (che sono ortogonali tra di loro).

Le mesh **irregolari** sono a volte preferite per una questione di adattività della risoluzione. Infatti c'è la libertà di far variare la distribuzione del numero di vertici localmente. Mentre in una mesh regolare bisognerebbe aumentare la risoluzione ovunque.

Alcuni metodi che producono mesh generano mesh di triangoli irregolari (è un problema di geometria procedurale molto complesso quello di trovare una mesh regolare). Questi metodi sono:

- algoritmi (es.: front advancing methods)
- la maggior parte dei metodi di acquisizione 3D (scanning)
- Mesh sculpting (metodo più recente)
- Direct low poly editing

Altri metodi per generare mesh producono tipicamente mesh regolari, spesso di quad, come:

- Tramite la creazione di superficie di rivoluzione e di superfici di spiazzamento (a parte i poli del modello che sono irregolari)
- CAD modelling

Con il diagonal split di una mesh di quad (semi-)regolare si ottiene una tri-mesh (semi-)regolare. A condizione che la diagonale sia sempre nello stesso verso, per tutta la superficie.

Le mesh prodotte con un programma sono dette **mesh procedurali**.

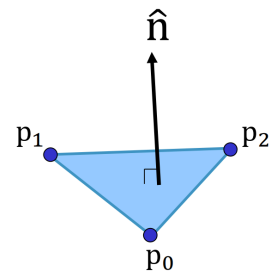
Mesh processing

Mesh processing = (geometry processing) is an area of research that uses concepts from applied mathematics, computer science and engineering to design efficient algorithms for the acquisition, reconstruction, analysis, manipulation, simulation and transmission of complex **3D models**. As the name implies, many of the concepts, data structures, and algorithms are directly analogous to signal processing and image processing.

Calcolo delle normali per **faccia** di una mesh

Il mesh processing è usato, ad esempio, per trovare la normale di un triangolo (di una faccia). In questo caso si usa il prodotto cross tra due **edge-vector** (il prodotto cross da come risultato un vettore chiamato **area vector**, un vettore ortogonale al triangolo lungo quanto la doppia area del triangolo).

$$\vec{n} = (p_1 - p_0) \times (p_2 - p_0)$$
$$\hat{n} = \frac{\vec{n}}{||\vec{n}||}$$



La normale delle facce triangolari è data dal semplice computo visto sopra e viene eseguito per ciascuna faccia.

Il vettore che si ottiene è detto **normale geometrica** del faccia, infatti corrisponde effettivamente alla normale del piano su cui giace il triangolo (che è costante) ed esiste sempre (non vale per facce quads – a meno che non siano planari) ad eccezione di triangoli degeneri (con 3 vertici allineati o, 2 vertici coincidenti, etc).

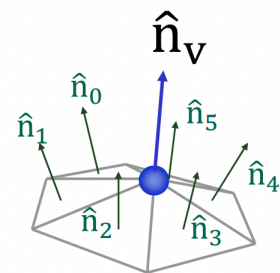
Questa normale è **orientata consistentemente** (nello stesso verso) solo se la mesh è ben orientata (ad es, sempre verso l'esterno in una mesh chiusa).

Calcolo delle normali per **vertice** di una mesh

Come si calcolano le normali per ogni vertice? Ovvero, come si ottiene un effetto smooth?

La normale di un vertice si ottiene dalla media delle normali delle facce/triangoli che condividono quel vertice (interpolazione).

$$\hat{n}_v = \frac{\hat{n}_0 + \dots + \hat{n}_k}{||\hat{n}_0 + \dots + \hat{n}_k||}$$



Dato che si normalizza il risultato, non c'è bisogno di divider per k la somma delle normali per faccia per calcolare la loro media.

La normale per vertice di una mesh non è definita in modo univoco per via geometrica. Dipende da quale superficie si intende rappresentare. Una strategia spesso utilizzata in pratica è usare una interpolazione delle normali delle facce adiacenti (per es, la media). Le normali (per vertice) fanno parte del modello e spesso vengono costruite insieme ad esso. Solo nei casi di una mesh sprovvista di normali, sarà necessario computarle dalla geometria (e dalla connettività).

Nel calcolo delle normali per vertice l'area e l'angolo di una faccia influenza la normale risultante, più e grande la faccia e/o l'angolo, più la normale verrà influenzata.

Wedge = angolo di una faccia.

Algoritmo per computo di normale per vertice con approccio gather

1. computo delle normali per faccia → per facce non-triangulari si possono mediare le normali costruite su ogni wedge della faccia. Si fa il prodotto cross dei due edge vectors adiacenti al wedge. Per trovarli, navigo gli half edge attorno alla faccia.
2. computo delle normali per vertice per $\forall$ vertice v → ciclo su tutte le facce adiacenti a v , sommando la loro normale in un vettore, e normalizzo questo vettore.

Problema: come trovo tutte le facce adiacenti ad un vertice dato? Se scandisco l'intero vettore della «lista-facce» il mio algoritmo diviene quadratico, se ho k vertici e $2k$ facce, richiede $\sim 2k^2$ operazioni. Nel mesh processing, gli algoritmi quadratici non sono feasible (non sono fattibili, k = molto grande)

Algoritmo per computo di normale per vertice con approccio scattered

1. $\forall$ vertice → azzero il suo vettore normale. Questo vettore servirà per mantenere le somme parziali.
2. $\forall$ faccia → calcolo la sua normale, e la sommo alla normale di ciascun vertice ai suoi corners. Alla fine di questo ciclo, su ogni vertice avrò accumulato un vettore da ciascuna faccia adiacente a quel vertice.
3. $\forall$ vertice → normalizzo il risultato.

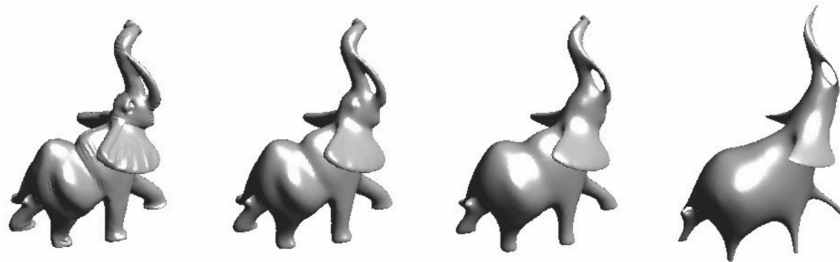
La strategia generale viene detta **scattering** → Al passo 2, invece che raccogliere (gather) su ogni vertice le normali dalle facce adiacenti, distribuisco (scatter) da ogni faccia le normali ai vertici adiacenti.

Questo algoritmo è **linare** (usa solo la struttura lista-facce), se ho k vertici e $2k$ facce, ci vogliono solo $\sim k + 2k + k$ operazioni.

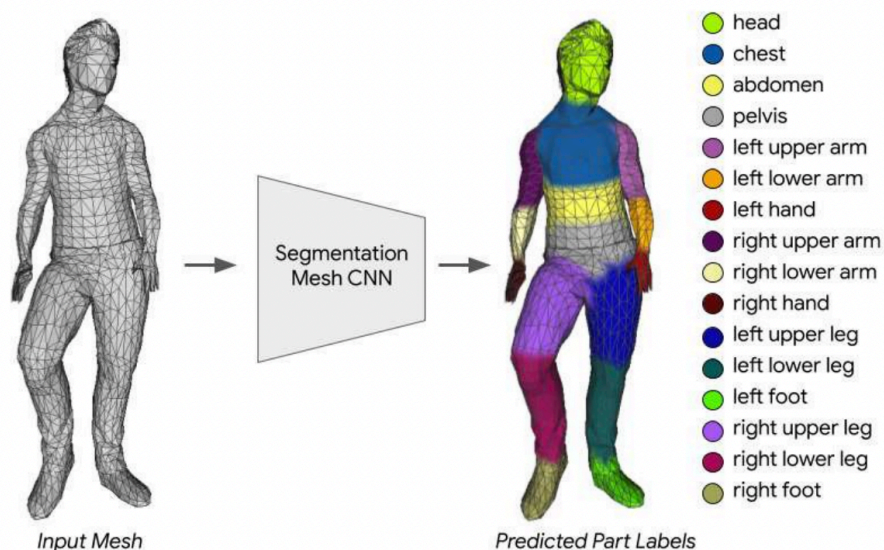
Nel geometry processing, gli algoritmi lineari sono feasible, sono veloci anche per mesh a risoluzione molto grande (per es, $K = 10^9$).

Il geometry processing può avere diverse applicazioni:

- **Mesh smoothing** → ha come obiettivo rendere la mesh più tondeggiante e liscia. Basta **spostare ogni vertice** nella posizione media dei vertici a lui adiacenti (vertici adiacenti = vertici connessi da un edge). La **connettività** della mesh rimane **inalterata**, si **modifica** solo la sua **geometria**. Il processo si può ripetere molte volte, cumulandone gli effetti (analogo concettuale del blur di un'immagine, in cui si sostituisce ad ogni pixel il valore medio dei pixel vicini, sfocando l'immagine; concetto matematico connesso: filtro Laplaciano, l'obiettivo è la minimizzazione del Laplaciano, cioè della distanza di ogni vertice dalla media dei suoi vicini). L'operazione può essere fatta anche sugli attributi (colore, etc)
- Effetti desiderati → si abbatta il rumore e si riducono le asperità della superficie.
- Effetti collaterali → spesso indesiderati, si abbattano anche i dettagli di forma utili, si perdono ad esempio gli spigoli vivi (crease angles) e di conseguenza la mesh tende a ridursi complessivamente in estensione. **Feature preserving smoothing** = Ottenere uno smoothing che rimuove il rumore e le asperità a piccola scala ma evita gli effetti collaterali sopra descritti (task studiato e più difficile)



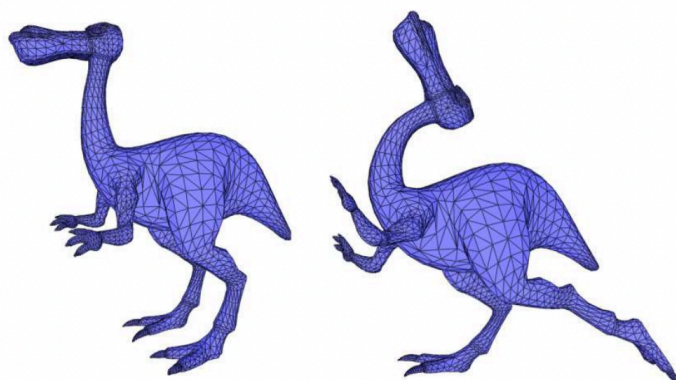
- **Mesh denoising**
- **Mesh cleaning / reparing** → insieme di task per la rimozione dei difetti della connettività di una mesh, cioè le situazioni non two-manifold, buchi (per es un poligono mancante) e orientamento non consistente delle facce (possibile, se mesh ben orientabile). Molte categorie di mesh, come le mesh scansionate (acquisizione 3D), presentano spesso molti di questi difetti.
- **Stima di proprietà geometriche** → Come nel caso della stima delle normali, sono solo un caso di proprietà di forma definite sulla superficie. Proprietà di cui si occupa la geometria differenziale.
- **Automatic mesh segmentation / labelling** → Data una mesh in input, identificare le zone semanticamente o strutturalmente distinte (tipicamente, come partizione delle facce o dei vertici in zone contigue). Questo processing è guidato da un'analisi della geometria, oppure data driven. Il mesh labelling assegna un'etichetta semantica ad ogni partizione.



- **Shape Retrieval** → data una grossa collezione di modelli 3D, individuare quelli simili ad una forma target richiesta. Spesso, la forma target è specificata attraverso un disegno semplificato 2D oppure: “trovare le mesh che rappresentano oggetti simili a quello rappresentato da una mesh target data”.



- **Procedural mesh generation** → nome generico per tutti quei procedimenti automatici che generano una mesh (geometria e connettività incluse). Tipicamente, guidato da parametri controllati dall'utente che determinano le caratteristiche ad alto livello delle forme generate (ad esempio: nella generazione automatica di mesh che rappresentano ingranaggi, un parametro può identificare «il numero di denti») e / o da scelte pseudocasuali. **Generative Networks** → L'approccio Machine learning alla mesh generation è data driven (per es: data una collezione di mesh che rappresentano forme di una certa classe come sedie, aerei..., produrre un insieme di varianti)
- **Shape correspondence** → Date due mesh che rappresentano oggetti di una stessa classe identificare su ciascuna di essa il punto corrispondente sull'altra ad esempio, sotto forma di un mapping (non biunivoco) fra i vertici. Basandosi su similitudine geometrica, topologica, e caratteristiche intrinseche della forma (ad esempio: la similarità di normale è poco significativa quella di curvatura intrinseca può esserlo di più)
- **Mesh Deformation** → Data una mesh iniziale questa viene soggetta a degli stimoli o vincoli esterni, computare una sua deformazione spaziale, cioè una nuova **geometria**, (la connettività della mesh è inalterata). La forma finale aderisce a principi fisici, geometrici (per es, conservazione dell'area, del volume, della forma dei triangoli, o semplicemente minimizzazione della distorsione subita dai singoli triangoli...). Esempio di vincolo esterno: una nuova la posizione xyz assegnata a solo alcuni vertici («questo vertice si sposta qui»). E' possibile effettuare questi computi in tempo reale.



- Mesh parametrization → texture mapping

Meshing = (task di geometry processing) dato un modello 3D, inizialmente non rappresentato come una mesh, costruire una sua rappresentazione mesh. È un esempio di **Surface**

Reconstruction = ricostruzione di una superficie (da dati che non sono una superficie). Detta anche poligonizzazione, o anche “segmentazione” in analogia con il 2D. Per es, «meshing di una nuvola di punti», come abbiamo visto.

Remeshing = data una superficie rappresentata come una mesh, costruire una rappresentazione mesh diversa. In ambiente industriale, è detto “retopology”, specie quando fatto manualmente da un’artista. La mesh di partenza differisce dalla mesh di arrivo per esempio:

- tri VS quad dominant VS quad («quad-remeshing»)
- (semi) regular VS irregular («semiregular-remeshing»)
- adaptive VS non adaptive resolution
- bad element shapes VS good element shapes

Spesso fatto per migliorare la qualità della mesh:

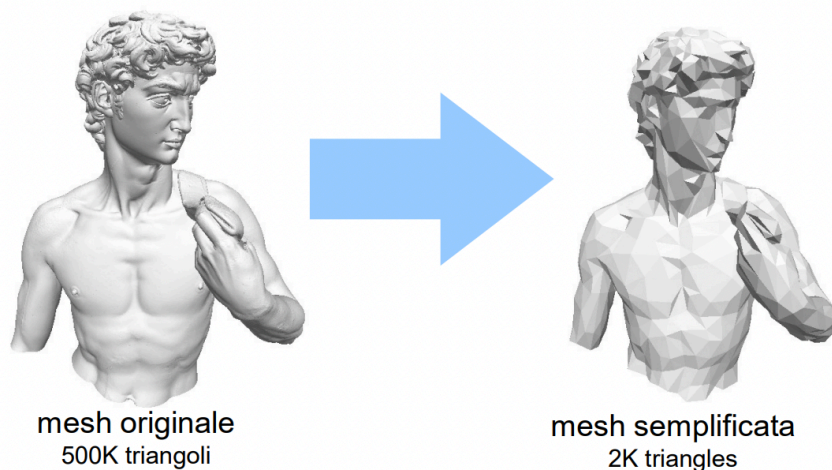
- rendere la risoluzione adattiva,
- rendere la risoluzione costante (dimensione facce costante, utile per es in una simulazione fisica),
- passare da triangoli a quads,
- rendere la mesh regolare.

semiregular remeshing = caso molto diffuso perchè le mesh catturate dal vero sono spesso irregolari ma le mesh semi-regolari sono più utili: più facili da editare, animare, etc

semplificazione automatica = riduzione della sua risoluzione. Non necessariamente un «remeshing» completo: alcune parti della mesh possono essere inalterate.

Semplificazione automatica di una mesh → da mesh hi-res a mesh low-res (detta anche low-poly mesh). Questo procedimento è chiamato anche: **Mesh coarsening**, Poly-reduction (in ambiente industriale) o Mesh decimation. Si vuole semplificare automaticamente una mesh perché la risoluzione deve essere adatta all’applicazione. Molti dei procedimenti su una mesh (rendering compreso) hanno una complessità lineare col numero di elementi.

Osservazione: in una nuvola di punti, bastava scegliere un sottoinsieme. Per mesh, è più complicato, non posso rimuovere elementi senza danneggiare le proprietà della mesh (es. chiusura)



L’obiettivo della semplificazione è ottenere un buon bilancio fra:

- costo (risoluzione della mesh risultante, cioè numero di elementi residui)
- qualità (errore geometrico introdotto rispetto alla mesh originale)

come definisco l’errore introdotto? misuro la distanza geometrica fra le due superfici descritte da mesh originale e mesh semplificata, cioè assumiamo mesh originale come ground truth.

come definisco la distanza fra due superfici? hausdorff distance = misura la distanza tra due sottoinsiemi di uno spazio metrico.

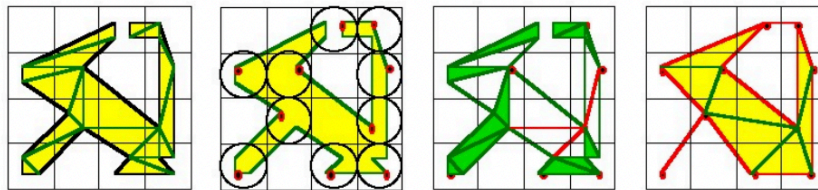
come la posso calcolare? task del geometry processing (Metro: measuring error on simplified surfaces)

Molte algoritmi, seguono approcci diversi le tecniche si differenziano in:

- Adattive e no → lasciano più triangoli dove ce n'è bisogno (es non nelle zone piatte) oppure riducono il numero di elementi ovunque.
- Con garanzie su errore massimo introdotto, o no → viene misurato? è limitato superiormente? oppure nessun limite / o nessuna misura.
- Incrementali o discrete → Ho una sequenza continua di mesh a risoluzione sempre inferiore? Oppure ho solo una singola mesh finale.
- Caratteristiche della mesh (2 manifoldness, chiusa) → sono sempre mantenute? oppure, possono essere perse?
- e molto altro...

Strategia a (vertex) clustering - non adattiva

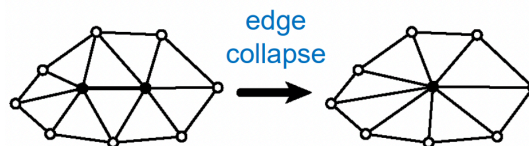
1. Dividi lo spazio 3D in una griglia regolare di cubetti.
 2. Tutti i vertici in un cubetto vengono "collassati" in un solo vertice "rappresentante" del cluster (a posizione interpolata)
 3. Ogni triangolo passa da indicizzare 3 vertici a indicizzare i 3 corrispondenti vertici rappresentanti
 4. rimuovere i triangoli che sono diventati degeneri (hanno solo 1 o 2 vertici distinti – non 3)
- L'errore introdotto e il numero di elementi finali dipendono dalla dimensione della griglia (parametro del metodo).



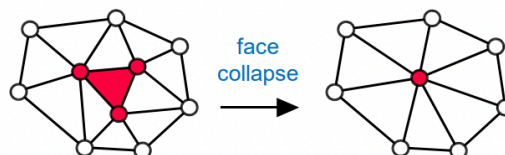
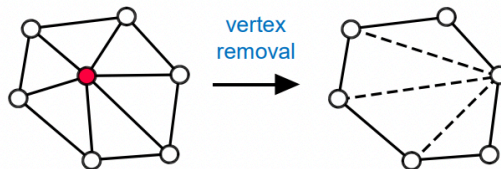
Strategia a edge collapse (operazioni locali iterate) - adattiva

Operazione locale (di coarsening) → modificare la mesh (connettività+geometria) solo in un intorno, lasciando il resto inalterato. Alcuni esempi:

- edge collapse



- vertex removal (si può considerare un caso particolare di edge collapse)
- face collapse (si può considerare una successione di due edge collapse)



repeat → scegli ed esegui una operazione locale (secondo un criterio)

until → obiettivo raggiunto (risoluzione target raggiunta o errore massimo superato)

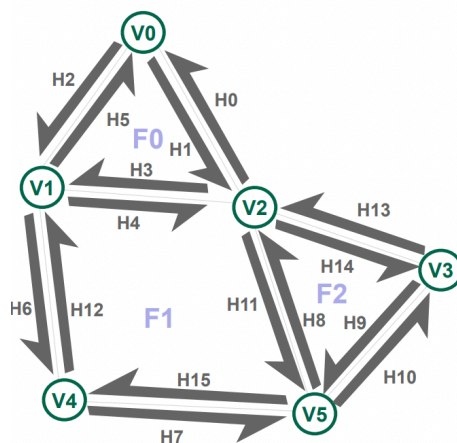
Le due strategie a confronto

Semplificazione ad operazioni locali	Semplificazione a clustering
adattiva	<u>non</u> adattiva
continua	discreta
possibile mirare a triangoli di forma buona	mira a facce di dimensione simile
possibile specificare errore massimo	
possibile specificare numero di facce target	possibile solo specificare dimensione griglia (solo indirettamente, errore o numero di facce)
può mantenere caratteristiche mesh $\Rightarrow$ two-manifoldness, chiusura, classe topologia	non mantiene caratteristiche mesh
spesso richiede mesh twomanifold in partenza	veloce, semplice da implementare, robusta (funziona su qualsiasi mesh)

La struttura lista facce della mesh indexed, usata per memorizzare la **connettività** non consente di effettuare le operazioni di mesh processing (se non attraverso una scansione dell'intero vettore delle facce, che richiede ovviamente un tempo lineare col numero di facce, eccetto: «data una faccia, elenca tutti i vertici che fanno parte di quella faccia»).

Per effettuare mesh processing, dobbiamo dotarci di strutture più adatte per memorizzare la connettività di una mesh che saranno più prolisse, onerose da mantenere durante le modifiche ma che consentono di navigare sulla mesh molto più agevolmente.

Struttura dati: half edges



	V	F	Next	Opp
H ₀ →	2	-	2	1
H ₁ →	0	0	3	0
H ₂ →	0	-	6	5
H ₃ →	2	0	5	4
H ₄ →	1	1	11	3
H ₅ →	1	0	1	2
H ₆ →	1	-	7	12
H ₇ →	4	-	10	15
H ₈ →	5	2	14	11
H ₉ →	3	2	8	10
H ₁₀ →	5	-	13	9
H ₁₁ →	2	1	15	8
H ₁₂ →	4	1	4	6
H ₁₃ →	3	-	0	14
H ₁₄ →	2	2	9	13
H ₁₅ →	5	1	12	7

Lista di half edge = La **connettività** è rappresentata da un vettore di Half Edge. Per ogni half edge memorizzo i campi:

- **Vertice** → l'indice da quale vertice parte
- **Faccia** → l'indice di quale faccia è un bordo
- **Next** → l'indice dell'half-edge che incontro proseguendo nella direzione dell'half edge (senza cambiare faccia o bordo della mesh)
- **Opposite** → indice all'altro half-edge che condivide lo stesso edge

Strutture a contorno → In ogni vertice, posso memorizzare l'indice di un Half-Edge che parte da quel vertice (uno qualsiasi). Posso avere un vettore di facce, per ogni faccia l'indice di un half edge appartenente a quella faccia (uno qualsiasi).

Lista facce indexed:

- Compatta (quindi, adatta a storing, ad esempio su disco o streaming)
- Sufficiente per ad alcuni task di processing (e in questo caso, preferibile)
- Il redering classico eseguito dalle GPU è pensato per questa struttura dati
- E' generale, non richiede ad esempio two-manifoldness o orientamento consistente delle faccie (quindi: capace di rappresentare strutture inconsistenti – è un vantaggio e uno svantaggio)
- Lista di elementi non omogenea, (alcune facce hanno un numero di vertici diverso da altre). eccetto che per tri-mesh o pure quad meshes: per loro, la lista facce è una matrice $3 \times N$ o $4 \times N$ (comodo)
- Navigazione complicata

Lista di half hedge:

- Prolissa (calcolo: quanti interi è necessario memorizzare in media, rispetto ad una lista facce? Ipotesi: in una tri mesh, ho vertici, facce, edge tipicamente in proporzione 1x, 2x, 3x. In una quad mesh: 1x, 1x, 2x)
- Più complicata da mantenere coerente durante le operazioni di modifica della connettività. Non adatta per il rendering su GPU
- Vantaggio: lista di elementi sempre omogenea: 4 elementi per half-edge (nella variante che abbiamo visto), anche su mesh poligonali miste
- Consente di «navigare sulla mesh» in modo semplice, con salti all'elemento adiacente in tempo costante (consentendo algoritmi di mesh processing in tempo lineare o pseudolineare piuttosto che quadratico)
- Richiede adattamenti se la mesh non è two-manifold e ben orientata

Nota: sono due rappresentazioni alternative di una stessa cosa (la connettività della mesh). Una si può costruire a partire dall'altra.

Texture

Spesso si vorrebbe campionare un attributo (un segnale: come colore, normale, lucentezza, temperatura etc) sulla superficie in modo molto più **denso** di quanto campioniamo la stessa superficie per catturare la sua forma geometrica. Per esempio:

- Forma → ci bastano 10 campioni per cm^2 (sufficiente per rappresentare una statuetta in dettaglio)
- Colore → vorremmo 100 campioni per cm^2 (per es, necessario per riprodurre il dettaglio pittorico su un vaso)

Memorizzando gli attributi per vertice su di una mesh poligonale, si ha:
 $1 \text{ campione di segnale} = 1 \text{ campione di forma} = 1 \text{ vertice}$

immagine raster = grigliato 2D regolare di pixel. In 2D, le immagini raster sono un ottimo modo di campionare segnali in modo molto denso.

Per ottenere un risultato simile al campionamento delle immagini raster sulle superfici 3D si applica un'immagine raster, detta **texture** (o tessitura), alla superficie di una mesh.

texture = grigliato regolare di **texel** = texture element.

Ogni punto sulla superficie 3D (S) viene associato ad un punto sulla rettangolo 2D (T).

NdR: le coordinate 2D su T sono indicate da (u, v) o a volte da (s, t) per distinguerle dalle coordinate 3D su S che sono (x, y, z) .

UV-map della mesh = (o UV-mapping in contesto industriale, o parametrizzazione in contesto accademico) mapping da S a T.

Ogni pixel della tessitura, chiamato texel, memorizza valori qualsiasi che variano sulla superficie (tipicamente: colori, normali, valori di trasparenza, coefficienti di lucidità o altri modi di descrivere un materiale).

La tessitura è sostanzialmente un'immagine raster, cioè un array 2D (una «matrice») di texel (texture elements). Come tale, è caratterizzata da:

- Risoluzione orizzontale **R_x** (di solito, potenza di due $\leq 2^{13} = 8192$)
- Risoluzione verticale **R_y** ($\leq 2^{13}$)
- Numero di canali per texel **Ch** (tipicamente, 1, 2 o 4)
- Numero di bit per canale **B** (spesso 8 ma anche 16, 24 ...)
- Spazio totale occupato in memoria (espressa in bit) = numero di texel · bit-depth = $R_x \cdot R_y \cdot Ch \cdot B$

In quanto immagine, può essere, per esempio, dipinta da un artista, catturata da foto, etc... Esistono anche molte tecniche per sintetizzarle automaticamente.

In quanto immagine raster, una tessitura si presta a semplici strutture multirisoluzione: basta ridurre la risoluzione dell'immagine per ottenere un «livello» diverso di una **piramide di livello di dettaglio**.

MIP-map = Piramide di livelli di dettaglio per texture/tessiture:

- Livello 0 → la tessitura originale, risoluzione massima
- Livello 1 → una tessitura ridotta ad $\frac{1}{2}$ della risoluzione in entrambe le dimensioni
- Livello 2 → riduzione del livello 1 (come sopra)
- Livello N → tessitura 1x1 (per tessiture quadrate)

Nota: in tutto sono $\log_2(\text{RES})$ livelli e lo spazio totale occupato da tutti i livelli oltre lo 0 è solo $\frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256} + \dots = \frac{1}{3}$ quello del livello 0



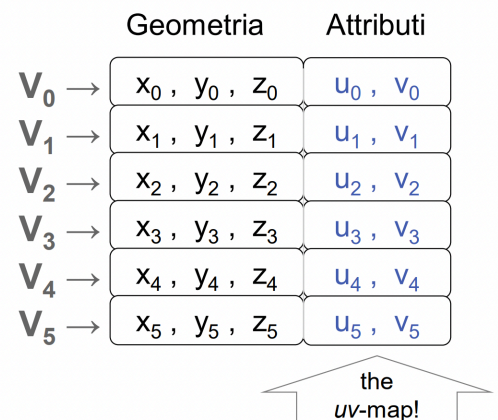
Le tessiture sono estremamente utilizzate nella maggior parte dei contesti applicativi in cui si usino mesh poligonali come: games, VR, movies, commercio elettronico, applicazioni architettoniche, visualizzazione scientifica, beni culturali... (ma spesso non: CAD, app. medicali). Parte della loro diffusione è dovuta al supporto HW: le GPU (il processore delle schede video) sono progettate per renderizzare mesh triangolari con tessitura associata.

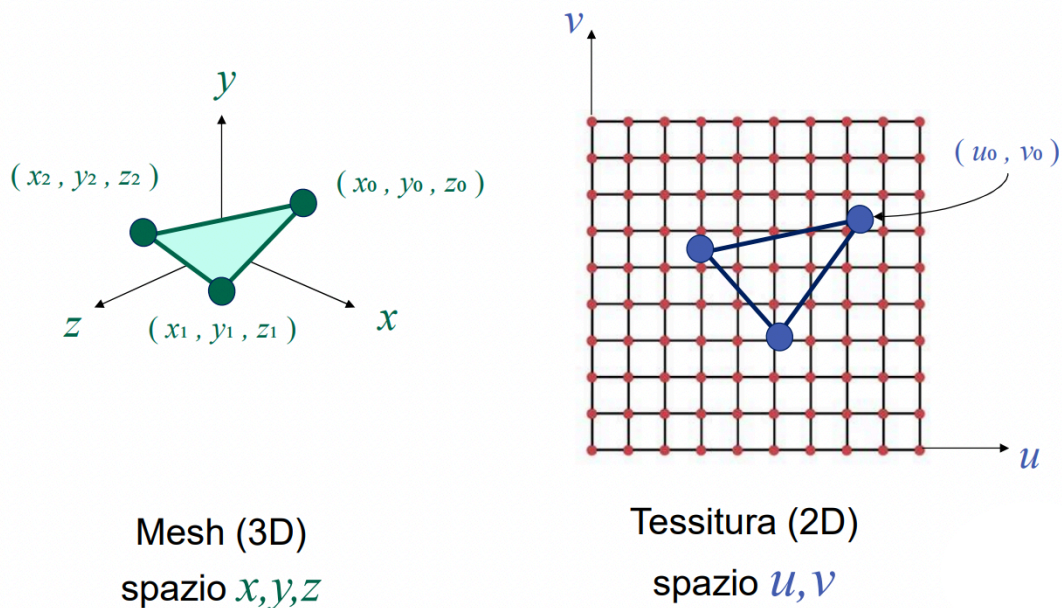
Le posizioni UV sono memorizzate come attributo per vertice. Come ogni attributo, si considera interpolato dentro alle facce della mesh (con le coordinate baricentriche).

Ogni triangolo della mesh corrisponde a due triangoli:

- il triangolo T_3 nello spazio in 3D «x,y,z»,
- il triangolo T_2 nello spazio 2D «u,v», dentro al rettangolo della texture

Ogni punto dentro al triangolo in T_3 è mappato nel punto con le stesse coordinate baricentriche nel triangolo T_2 (come tutti gli altri attributi per vertice, «u,v» è interpolato nelle facce). L'intera mesh è mappata sul «dominio parametrico» 2D).





Per fare UV-mapping di una mesh (o mesh parametrization) si devono assegnare le coordinate UV per ogni suo vertice. A volte, (raro) la mesh viene creata congiuntamente al suo UV map ma più spesso, la mesh viene creata (es. acquisita, o modellata) ed è sprovvista di un UV-map. Questo task non è semplice: la automatizzazione è tema molto studiato nel Geometry Processing. Spesso artisti digitali intervengono manualmente con software di 3D modelling.

Un buon UV map deve rispondere a molti criteri:

- **Inietività** → ogni punto della tessitura può essere mappato in al più un punto della mesh (no triangoli sovrapposti in spazio UV)
- **Bassa distorsione** → ogni triangolo di mesh T_3 deve essere mappato in un triangolo di tessitura T_2 di forma simile e di area proporzionale
- **Buon coverage** → le parti di tessitura non coperte da nessun triangolo rappresentano uno spreco di memoria e vanno minimizzate
- Limitato numero di texture seams

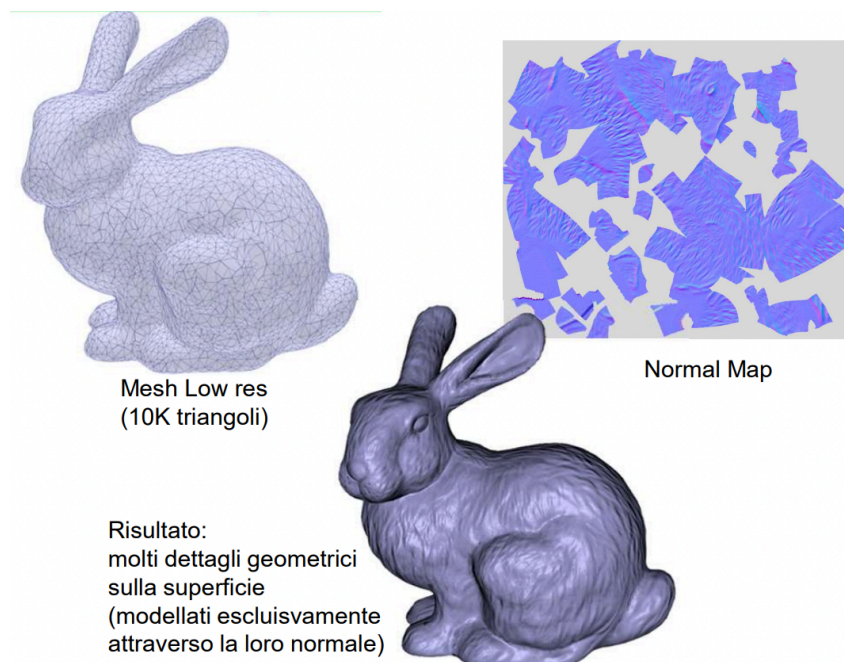
Un task che il geometry processing vuole risolvere è la mesh parametrization, ovvero costruire **automaticamente** un UV-map per una mesh data. Questo è un problema difficile da risolvere in modo soddisfacente (soprattutto la scelta sul posizionamento/numero dei tagli). Esistono un gran numero di approcci algoritmici diversi ma rimane un problema: gli UV-map prodotti automaticamente sono spesso di scarsa qualità.

Texture seams = (o cut) un UV-map richiede di introdurre dei tagli. Questi tagli consentono di aprire la mesh per stenderla (unwrap) su un piano. Avere più tagli significa avere meno distorsione (di solito).

Texture Atlas = (analogia con un atlante di mappe) approccio in cui mesh viene divisa in zone (patches), ciascuna parametrizzata in un "isola" di tessitura.

Normal-map = è una texture che campiona le normali della superficie. In questo modo, contribuisce a specificare la forma geometrica dell'oggetto (introducendo però una approssimazione). Tipicamente, la normal map riproduce dettagli geometrici minuti, ad alta frequenza (come gli avvallamenti su una buccia di arancia), mentre la mesh riproduce la forma generale dell'oggetto (come la forma sferica dell'arancia).

Quindi uno stesso oggetto può essere rappresentato, con simile accuratezza e una resa simile, attraverso una mesh ad alta risoluzione, oppure una mesh a bassa risoluzione ma provvista tessitura per dettagli «ad alta frequenza» (come una normal-map e/o una tessitura colore).



Hi-res mesh (con attributi)	Low res mesh con texture
densità di campionamento segnale = densità di campionamento geometria	risoluzione texture (colore, etc) indipendenti da risoluzione mesh (geometria)
aumentare i vertici richiede di incrementare anche la connettività (e quindi di processare più triangoli durante il rendering)	incrementare i campioni di tessitura ha un costo piccolo
non richiede vertex seams (se il segnale è continuo)	costo extra: UV-map sulla mesh. Va costruito (difficile); va mantenuto in memoria; richiede di introdurre alcuni vertex seams.
normale & posizione per vertex: la forma è catturata per intero, compreso nei suoi dettagli (questo permette di tenerne conto in tutti i tipi di processing)	in totale, meno oneroso in termini di spazio (in memoria), e tempo (rendering)
	normal-map: i dettagli di forma sono solo riprodotti visualmente attraverso il lighting

Texture baking = Sintesi automatica di una tessitura, per riprodurre un attributo definito su una mesh hi-res

- Input =
 - Mesh Hi-res M0, con attributi (per vertice, per faccia) che rappresentano un segnale che ci interessa riprodurre (colore, normale, materiale...)
 - Mesh Low-res M1, parametrizzata (cioè dotata di UV map)
- Output = una tessitura sintetizzata per M1 (baked texture) che riproduce su di essa l'attributo definito su M0

Un passaggio comune nel pipeline di creazione dei modelli 3D. Supportato per esempio da tutte le suite di modellazione 3D.

L'attributo definito per vertice nel modello Hi-Res (interpolato per triangolo) viene memorizzato sui texel della tessitura, secondo l'UV-mapping della mesh Low res.

In pratica ottengo il modello low-res con l'aspetto di quello High-res.

Una texture 2D è del tutto analoga ad un'immagine rasterizzata. Perché usare un termine separato per identificare questo tipo di immagine? Perché sono diversi in

- **uso atteso:** una texture è pensata esplicitamente per essere rimappata su una mesh (attraverso un UV-map)
- **contenuto:** i texel di una texture campionano qualsiasi segnale sulla superficie (normale, colore, coefficiente di lucidità... o altre proprietà del materiale, etc), laddove i pixel di un'immagine di solito sono campioni di colore (per es: rappresentati da una tripla di valori R,G,B)

Esistono formati file specifici per texture (come dds), diversi da quelli per immagini generiche (come png o jpeg)

- **numero di canali:** una texture standard ha tipicamente 4 canali (a volte: 1 o 2), un'immagine tipicamente ne ha 3
- **risoluzione:** per motivi storici inerenti al rendering, una texture ha risoluzione per lato come potenza di 2 (es: 512, 1024, 2048, 4096)
- **profondità di texel:** ogni canale di una texture ha un numero di bit per canale variabile da 4 a 16, molti formati per immagine prevedono solo 8bit per canale
- **compressione:** una tessitura deve essere random-accessible anche da compressa. Esistono schemi di compressione per texture, diversi dagli schemi usati per le immagini (che non hanno bisogno di essere random-accessible)
- **multi-risoluzione:** una tessitura spesso comprende un tipo di piramide di livelli di dettaglio detta «MIP-map»: ogni livello successivo dimezza la risoluzione del precedente (es: 1024x1024, 512x512, 256x256, ... , 2x2, 1x1)

Un formato per immagine generica (come png) può essere comunque usato come formato di interscambio per texture.

discontinuità C0 = is an actual break in the curve

continuità C0 = Curves share the same point where they join.

discontinuità C1 = is a change in the tangent (un nodo nella curva).

continuità C1 = Tangent vectors of the two segments are equal in magnitude and direction (share the same parametric derivatives)

continuità c-infinito =

Parametric Surfaces

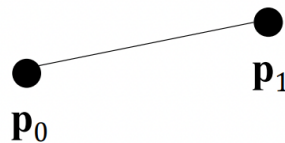
Modo di rappresentare superfici curve. Molto utilizzato per CAD/CAM.

Curve parametriche

Curva parametrica = Una rappresentazione per linee curve (usate in 3D per percorso di un'animazione, o 2D per la grafica vettoriale). Curva descritta come immagine di una funzione. La curva è quindi l'immagine di una funzione da $\mathbb{R}$ a $\mathbb{R}^2$ (o $\mathbb{R}^3$).

Esempio: segmento come "curva" parametrica (in questo caso, non è curva, cioè non ha curvatura). Il segmento fra i punti p_0 e p_1 è il luogo di punti raggiunti dalla funzione

$$f(t) = p_0 \cdot (1 - t) + p_1 \cdot t$$



In questo esempio, ed anche in generale i punti p_0 , p_1 sono **costanti** nella formula di f , sono i **punti di controllo** della curva: modificandoli, modifico la forma della curva.

In questo esempio:

- La curva è un polinomio lineare, di grado 1 (in generale, si usano polinomi di grado superiore)
- la curva passa attraverso i punti di controllo, i punti di controllo sono solo 2
- i punti di controllo sono all'inizio e alla fine della curva, la direzione tangente è costante per ogni t

Spline = curva polinomiale a tratti, di un certo grado n . Una spline è controllata attraverso un insieme di punti di controllo e si dice:

- **Approssimante** se è tangente ai segmenti che connettono i punti di controllo
- **Interpolante** se passa attraverso i punti di controllo, la curva interpola i punti di controllo

Una spline è divisa in archi, ogni punto su ogni arco sarà una certa interpolazione lineare dei punti di controllo. Sono presenti diversi tipi di spline, che differiscono in:

- la formula che definisce f
- se la spline è interpolativa o approssimativa
- il grado di continuità raggiungibile alle giunture fra due curve

Un tipo di Spline sono le **Curve di Bezier** =

- Sono una famiglia di splines di grado n qualsiasi
- Una curva di Bezier è composta da archi di Bezier (un arco di grado n è controllato da $n+1$ punti di controllo)
- L'arco è **interpolante** (passa attraverso) il primo e l'ultimo di questi punti, ma in generale **non** gli altri; di conseguenza per rendere la curva continua basta far coincidere l'ultimo punto di controllo di un arco col primo dell'arco successivo.
- Quelle di grado $n=3$ sono le più usate

Due metodi per definire un arco di curva di Bezier (di grado n):

- algoritmo di DeCasteljau;
- Riformulazione matematica, come polinomio

Le curve di Bezier sono usate per: design 2D, specifica della forma delle lettere in un font, ideale per path di animazione

Formulazione di De Casteljau: esempio per grado 3

Si hanno 4 punti di controllo p_0, p_1, p_2, p_3 .

Data una posizione parametrica t , trovo:

- il punto q_0 interpolando fra p_0 e p_1 (con t)
- il punto q_1 interpolando fra p_1 e p_2 (con t)
- il punto q_2 interpolando fra p_2 e p_3 (con t)
- il punto r_0 interpolando fra q_0 e q_1 (con t)
- il punto r_1 interpolando fra q_1 e q_2 (con t)
- il punto finale $f(t)$ interpolando fra r_0 e r_1 (con t)

Modo pratico di costruire (pro tip: e ricordare) la formula di Bezier di qualsiasi grado n

- Step 1: scomporre l'unità in t e $\bar{t}$: $\bar{t} + t = 1$
- Step 2: elevare alla n entrambi i lati $(\bar{t} + t)^n = 1^n = 1$
- Step 3: ottengo una scomposizione dell'unità in $n + 1$ termini

Esempi, per $n = 3$ $(\hat{t} + t)^3 = \hat{t}^3 + 3\hat{t}^2t + 3\hat{t}t^2 + t^3 = 1$

- Step 4: questi termini sono i pesi degli $n + 1$ control points

Esempi con grado 3 $f(t) = \hat{t}^3 p_0 + 3\hat{t}^2 t p_1 + 3\hat{t} t^2 p_2 + t^3 p_3$

Gli endpoints di arco di Bèzier coincidono con il primo e l'ultimo punto di controllo (ma non dagli altri) Gli endpoints sono con $t = 0, \bar{t} = 1$ e con $t = 1, \bar{t} = 0$

Come sono le direzioni tangenti nei due endpoints? Quanto vale la derivata f' nei punti parametrici $t=0$ e $t=1$? La risposta (che vale per tutti i gradi) calcolando le derivate è :

- all'inizio dell'arco, è sempre orientata come il vettore che connette il primo pt. di controllo al 2do
- alla fine dell'arco, è sempre orientata come il vettore che connette il penultimo pt. di controllo all'ultimo

Il grado 3 (curve di Bézier cubiche) rappresenta il grado più conveniente da usare. Ho quattro punti distinti che controllano la curva in modo intuitivo:

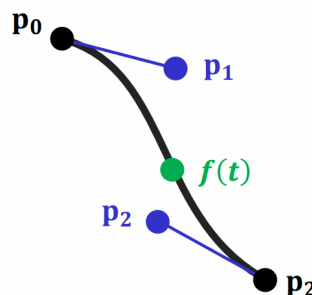
- Posizione e orientamento all'inizio della curva (i primi due)
- Posizione e orientamento alla fine della curva (gli ultimi due)
- Posso ottenere curve di Bézier con continuità $G1$ (e $C1$)

Per paragone:

- I gradi > 3 non sono altrettanto utili e più difficili da manipolare
- Coi gradi < 3 non posso manipolare le due direzioni tangenti in modo indipendente una dall'altra

Le curve cubiche sono ideali per fondere gli archi in una sola spline, detta «Bezièr curve» (o Bezièr path). Posso far coincidere le direzioni tangenti per ottenere una curva smooth.

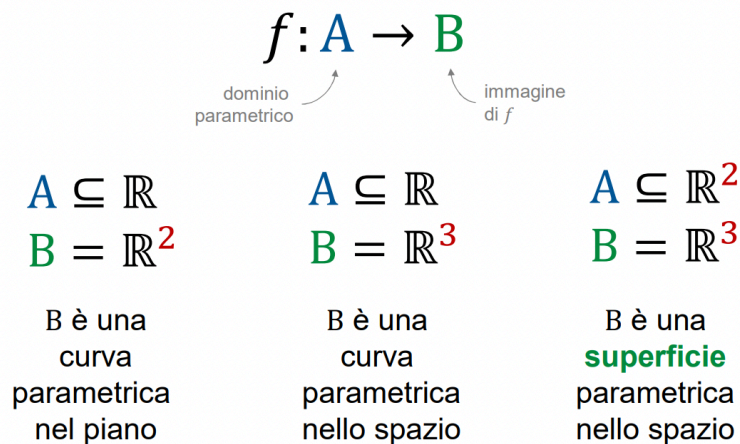
$$f(t) = (1-t)^3 p_0 + 3t(1-t)^2 p_1 + 3t^2(1-t) p_2 + t^3 p_3$$



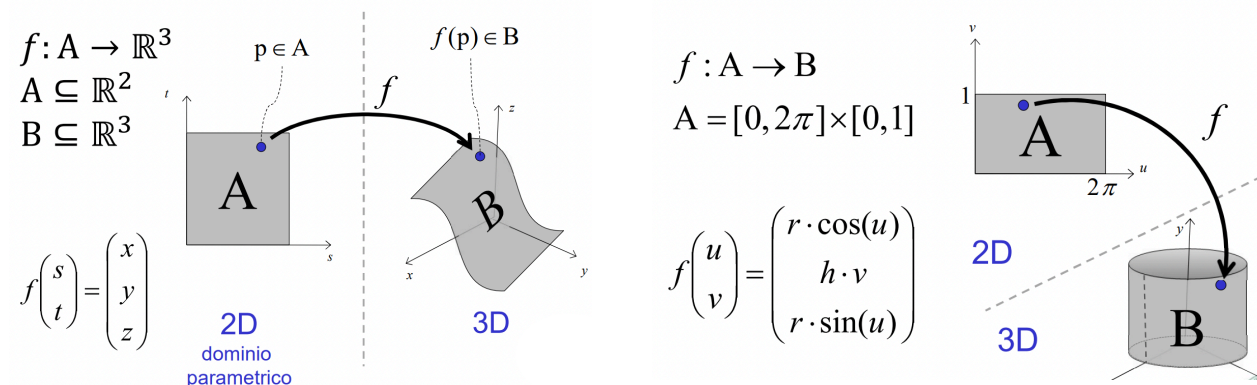
continua C0 = Se end-points coincidono

continua C1 = Se, inoltre gli end-points allineati si ha una curva continua con direzione di tangente continua (curva “smooth”- “liscia”)

Superfici parametriche



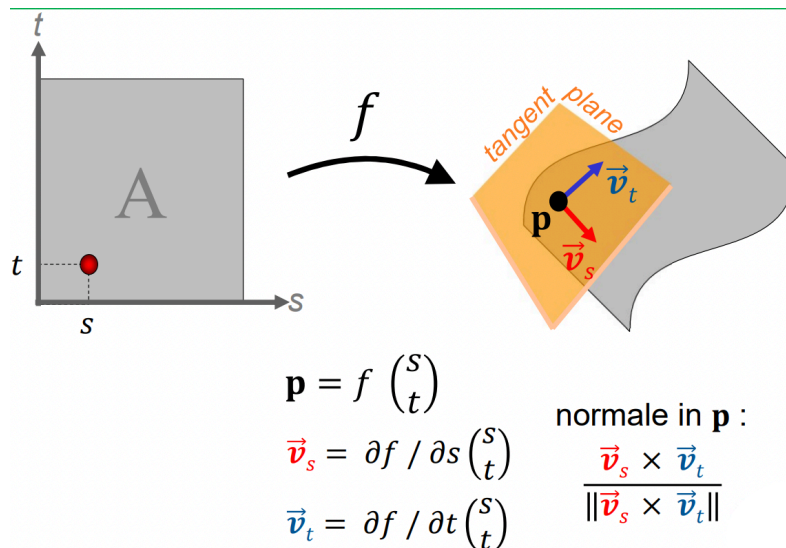
Superficie parametrica = immagine di una funzione da un dominio in $\mathbb{R}^2$ a $\mathbb{R}^3$. Per definirne una, scegliere una funzione (e il suo dominio parametrico A).



A = Dominio Parametrico = il dominio di f . Definito come regione di uno spazio bidimensionale. Spesso, per convenzione, definito come il quadrato $[0..1] \times [0..1]$. Per distinguere le sue coordinate da quelle usate come spazio oggetto (x,y,z), le chiamiamo s, t (a volte, u e v). Come per le tessiture.

B = immagine di f . Il luogo di punti raggiungibili da f a partire da A. Costituisce la superficie parametrica.

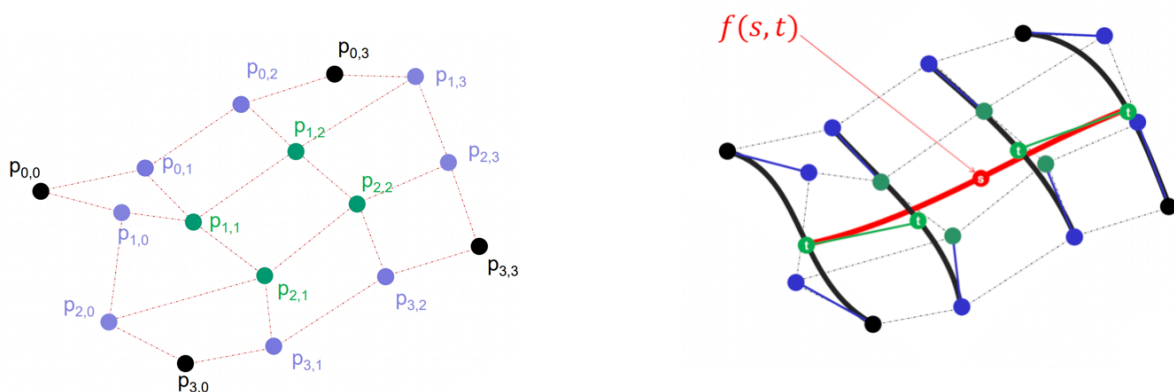
Posso derivare f sia rispetto a s , sia rispetto a t , le **derivate parziali** di f sono gli **spostamenti in 3D che effettuo incrementando** s o t . Si ottengono due vettori tangenti alla superficie. Quindi due vettori sul piano tangente alla superficie. La normale della superficie in un punto è data dal **prodotto cross** fra questi due vettori (rinormalizzato).



$f(s, t)$ è una interpolazione lineare di un certo numero di punti 3D di controllo. I pesi sono dati da polinomi (ora, in s e t), secondo una formula predeterminata.

Memorizzare la superficie consiste nel **memorizzare i punti**. Posso usare lo schema di una qualsiasi spline per costruire il corrispondente **patch** (una superficie). Quindi esistono: patch di Bézier, patch NURBS, ...etc. La superficie è un **tessuto di linee parametriche** come la trama e l'ordito in un tessuto.

Così come una curva (o un path) di Bézier è costituito da molti archi connessi, una superficie di Bézier è costituita da molti patch connessi sui loro lati. Nuovamente, il matching dei punti di controllo sul bordo garantisce che la superficie sia continua, senza gaps. Inoltre, imporre la co-linearità fra i punti di controllo interni garantisce che la superficie non abbia crease (discontinuità di normale), cioè che sia smooth (liscia).



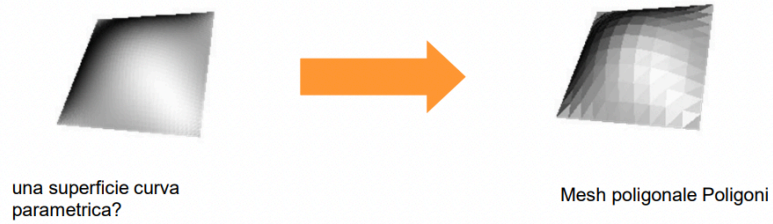
Vantaggi:

- Controllabili, intuitivi da modellare
- rappresentazione compatta ed espressiva
- la superficie nasce già con un dominio parametrico e un mapping incluso nella sua definizione
- sono intuitive da manipolare per il designer del modello

In molti contesti, la superficie parametrica è l'input o l'output diretto di task di geometry processing incluso:

- semplificazione: riduzione del numero di patch
- costruzione automatica

Per molti usi, tra cui il rendering, è necessario convertire la superficie parametrica in una mesh poligonale.



campiono il dominio parametrico A su una griglia regolare. Per ogni posizione campionata (s, t) :

- creo un vertice della mesh nel punto $f(s, t)$
- do a ogni vertice della sua normale, ottenuta con le derivate di f

Creo triangoli che connettono i vertici nel modo banale. Il risultato è una mesh regolare.

Osservazione: quanto densamente tassellare può essere deciso liberamente, anche in fase di rendering. La superficie rappresentata è curva: solo durante il rendering la approssimiamo con superfici poligonali.

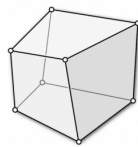
Rappresentazioni Volumetriche

I modelli 3D volumetrici possono essere:

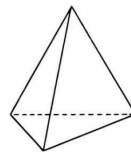
- Discreti & regolari → dataset voxelizzati, analogo di un'immagine rasterizzata, ma in 3D. Sono una griglia di voxel
- Discreti & irregolari → mesh poliedrali, analogo di una mesh poligonale (ma nel volume), insieme di poliedri adiacenti faccia a faccia.
- Continui → modelli impliciti, rappresentazione basata su funzioni volumetriche, superficie come luogo di zeri di una funzione

Mesh poliedrale

Una mesh poliedrale è il corrispondente volumetrico delle mesh poligonali, ed è composta da poliedri quali esaedri e tetraedri.



esaedro
(o "hexa" per brevità)



tetraedro
(o "tetra" per brevità)

Tipo degli elementi:

- hexahedra (anche detti "cuboidi")
- tetrahedra (piramidi a base triangolare)
- poliedri generici (raro)

mesh poliedrale = mesh composta da elementi poliedrici adiacenti faccia a faccia (hexahedron mesh, o hexa-mesh; tetrahedron mesh o tetra-mesh). È composta da:

- geometria → vertici (0D), con pos (x,y,z)
- connettività → poliedri (3D), facce (2D) e edge (1D) che connettono i vertici
- attributi → sui vertici, implicitamente interpolati dentro gli elementi
- Struttura dati simile alla mesh poligonale: indicizzata con una lista vertici e una di lista poliedri (esistono anche varianti per mesh poliedrali di struttura basata su half-edge)

Vengono usate principalmente per simulazioni fisiche (es. simulazione di carico, termodinamica, etc.) per verificare virtualmente le proprietà strutturali degli oggetti.

Tetraedri = struttura simpliciale del volume, un tetraedro è il luogo dei punti che sono l'interpolazione lineare fra i quattro vertici, cioè ogni punto P dentro un tetraedro T (superficie compresa) è esprimibile come una e una sola combinazione lineare dei quattro vertici di T. I 4 scalari sono detti le coordinate baricentriche di P dentro T. Ogni poliedro può essere scomposto in tetraedri.

Schema generale della soluzione per il computo delle «coordinate baricentriche» dentro un qualsiasi elemento simpliciale

1. Unire i 4 (3, 2) vertici $p_0 \dots p_n$ al punto p
2. Hai ottenuto 4 (3,2) nuovi sotto-tetraedri (-triangoli, -segmenti) che scompongono il tetraedro (triangolo, segmento) originale
3. Calcola l'estensione (cioè il volume, l'area, la lunghezza) di questi nuovi sotto-elementi
4. La coordinata baricentrica t è data dall'estensione dell'elemento opposto al vertice p diviso la somma delle estensioni (cioè diviso l'estensione dell'elemento originale)

dataset voxellizzati

voxel = volume element; elemento di una griglia regolare 3D.

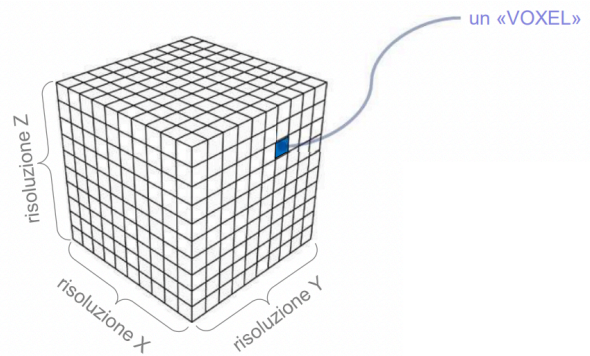
In Java: `Voxel[][][] data = new Voxel[resX][resY][resZ]`

1 Voxel = 1 Boolean(1 bit) → ogni voxel può essere pieno o vuoto.

Lo spazio è cubico con la risoluzione lineare. E' di solito un prezzo troppo alto, anche nel caso di 1 solo bit per voxel.

1 texel = 1 voxel → volumetric textures; E' supportata dall'hardware come ogni altra tessitura. Modella il segnale dentro al volume (per esempio come gli oggetti sono colorati all'interno)

Non richiede alcuna parametrizzazione della superficie.



Griglia regolare 3D
(o lattice)

Isolinea = Su un piano 2D:

- ogni punto del piano ha un valore scalare (es. pressione, altezza, etc.)
- prendo tutta la regione con valore scalare $>\sigma$ (sigma), il bordo della regione è costituito da punti di valore $=\sigma$ (isolinea)

Isosuperfici = Su un volume 3D:

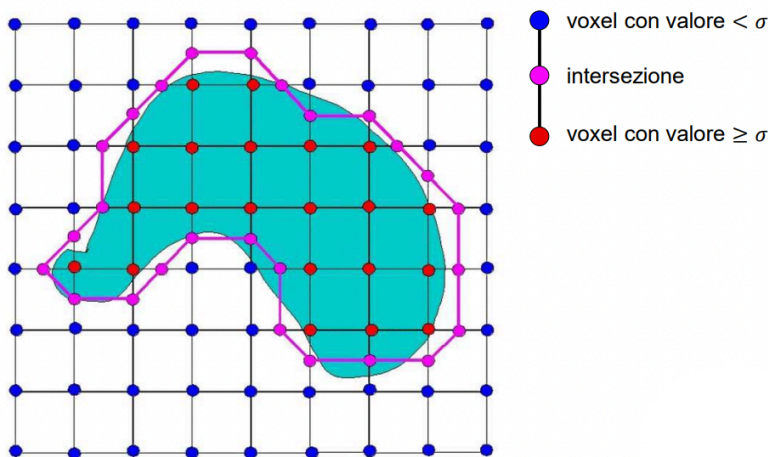
- ogni punto dello spazio ha un valore scalare (densità, pressione, etc.)
- prendo tutta la regione 3D con valore scalare $>\sigma$, il bordo della regione è costituito da punti di valore $=\sigma$ (isosuperficie)

Algoritmo marching squares (2D)

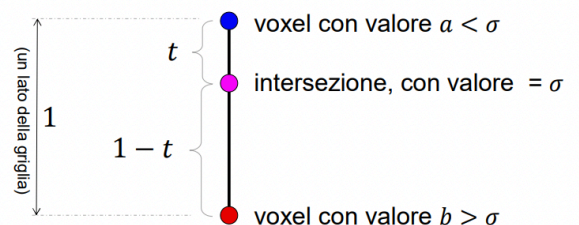
parto con una griglia di voxel e una regione delimitata da un'isolinea.

I passi da compiere sono :

- sogliare i voxel
- trovare le intersezioni tra voxel con $val < \sigma$ e voxel con $val \geq \sigma$
- unire le intersezioni creando dei segmenti (oppure trovare intersezioni con l'isolinea e unirle con dei segmenti)



✓ Come trovare le intersezioni

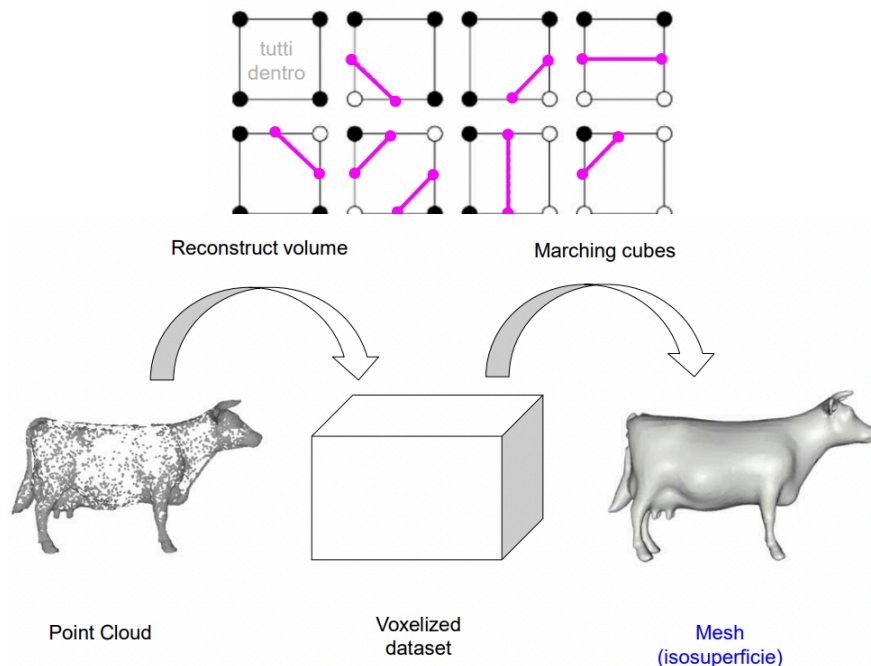


✓ Ipotesi: segnale interpolato linearmente:

$$a(1-t) + bt = \sigma \quad \Leftrightarrow \quad t = \frac{\sigma - a}{b - a}$$

Come trovare i segmenti che connettono le intersezioni?

Consideriamo un quadrato fra 4 voxel. Ogni suo vertice è «dentro» $< \sigma$ o «fuori» $\geq \sigma$. Per ogni combinazione, (e sono solo $2^4 = 16$) decido, una volta per tutte, i segmenti da costruire per unire le intersezioni. Alla fine si ottiene un tabella come questa:



Ogni voxel della griglia è dentro o fuori (valore $>$ o $<$ della soglia prescelta). Ogni edge della griglia (orientato lungo la X, Y o Z), che connetta un vertice dentro ad uno fuori, ha una intersezione con l'isosuperficie cercata:

- la si trova trovata come nel caso 2D
- per ciascuna intersezione, creo un vertice della mesh
- ho ottenuto la geometria della mesh!

Come ottengo la connettività?

- Scompongo la griglia in cubetti $1 \times 1 \times 1$
- Ogni cubo ha 8 vertici, ciascuno dei quali è dentro o fuori ($2^8 = 256$ casi)
- Uso una tabella per analizzare, in ciascun caso, quali poligoni creare per connettere i vertici sui lati della griglia

Il problema del Marching cubes è l'efficienza, infatti ci sono n^3 cubi da analizzare. Una possibile soluzione possibile è non processare i cubi vuoti (senza intersezioni sugli spigoli, né facce all'interno). Si seguono questi passi:

- trovo un cubo non vuoto e lo processo;
- passo a processare i cubi non vuoti vicini;
- itero finché non completo la mesh.

Passare da modello voxelizzato a Mesh poligonale:

- Output: mesh chiusa, two-manifold, ben orientata
- Contro: cattivo meshing con triangoli lunghi e stretti o anche degeneri

Strutture volumetriche come dataset intermedio → Un modo comune di convertire una point cloud (e altro) in una mesh è passare attraverso a una struttura volumetrica a voxel, in questo modo:

1. Stendere una griglia volumetrica nel volume coinvolto.
2. Memorizzare una «signed distance function» nel volume (1 valore scalare per voxel, che memorizza una stima della distanza dalla superficie, negativo se il punto è dentro la superficie).
3. estrarre isosuperficie (marching cubes)

Nel passo 2: ogni punto della nuvola di posizione p e normale n «vuole» che:

- il valore della funzione in p sia 0
- il gradiente della funzione in quella posizione sia allineato a n

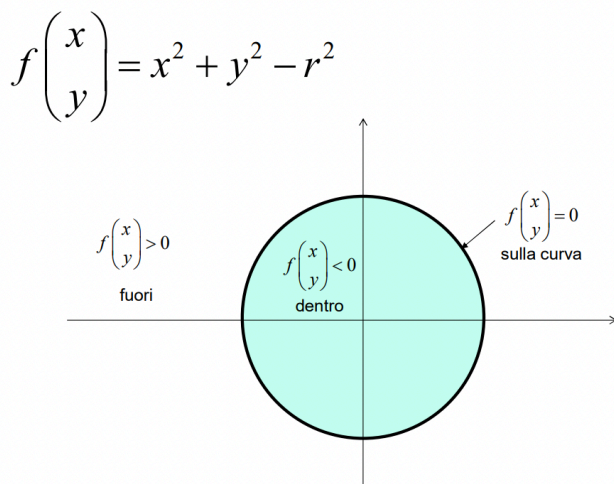
Modelli impliciti

oggetto definito da una **funzione continua** che passa **da punti 3D a valori scalari**.

Il valore di f definisce, per un punto dato p , se è dentro oppure fuori dall'oggetto:

- $f(p) < 0 \iff p$ dentro → 1 o valore negativo
- $f(p) > 0 \iff p$ fuori → 0 o valore positivo
- $f(p) = 0 \iff p$ sulla superficie → $\frac{1}{2}$ o 0

Il gradiente della funzione è invertito nei due casi di interpretazione semantica dello scalare (primo caso: gradiente verso il dentro. Secondo caso: verso il fuori)

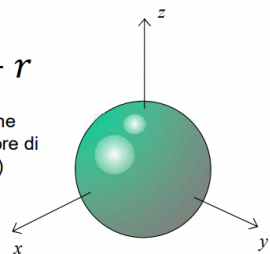


$$f\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \sqrt{x^2 + y^2 + z^2} - r$$

cioè

$$f(p) = \|p - \mathbf{0}\| - r$$

Origine
(vettore di
 $0,0,0$)



Sfera centrata nell'origine

La superficie che delimita il modello implicito (volumetrico). E' dato il luogo degli zeri di una funzione $f : \mathbb{R}^3 \rightarrow \mathbb{R}$

Ci sono diverse categorie di modelli impliciti:

- Superfici algebriche: $f()$ è un polinomio
- Superfici quadriche: $f()$ è di grado 2 → classe importante: equazioni semplici, ma buon potere espressivo (per esempio, sfere perfette)
- Superfici cubiche: $f()$ è di grado 3
- Superfici quartiche: $f()$ è di grado 4
- etc

Vantaggi delle sup. implicite:

- è una rappresentazione volumetrica (facile determinare se p sia dentro o fuori dall'oggetto)
- è compatta

- rappresenta superfici curve
- buon modello per superfici che variano nel tempo
- consentono potenti operazioni di editing

Da modello implicito a mesh → il modo più comune (ma non l'unico) è passare da voxel:

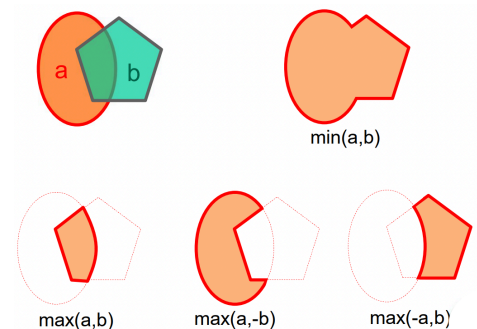
- campionare f ad (una data risoluzione res_x, res_y, res_z) ottenendo un campo scalare voxelizzato
- estrarre una mesh poligonale M attraverso Marching Cubes (con soglia 0)
- calcolare le normali nei vertici di M , con attraverso i gradienti di f calcolati nelle posizioni dei vertici

(preferibile al calcolo delle normali dei vertici attraverso le normali dei triangoli, come abbiamo visto per una mesh generiche)

Siano A e B modelli impliciti con funzioni primitive f_A e f_B .

Posso definire (come modelli impliciti) la funzione primitive della loro:

- inversione → $-f_A$
- intersezione → $\max(f_A, f_B)$
- unione → $\min(f_A, f_B)$
- scavo di B da A → $\min(f_A, -f_B)$
- Scavo di A da B → $\min(-f_A, f_B)$



Modellare oggetti solidi 3D (mesh) a partire dalle forme più semplici (primitive, modelli impliciti) che li compongono. Utile per modellare oggetti meccanici / artificiali (Contesto CAD).

Il modello è rappresentato da una struttura ad albero:

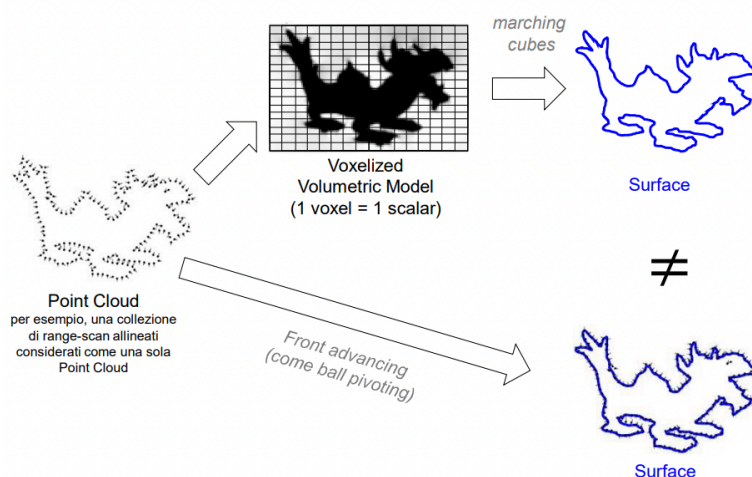
- radice = l'intero oggetto
- nodi interni = operazioni booleane dei nodi figli
- foglie = forme basilari – associate a funzioni primitive (esempio: sfere, cilindri, semispazi, cubi...)

E' l'albero di parsing di un'espressione di operazioni booleane.

Da nuvola di punti a mesh

Abbiamo visto due metodi per convertire una nuvola di punti in una mesh

- Modo **diretto** → con algoritmi di Front Advancing (come Ball Pivoting) si aggiunge una connettività che connettere i vertici della nuvola di punti, completando la mesh
- Modo **indiretto** → si converte la nuvola di punti in un dataset volumetrico voxellizzato (che campiona nel volume una stima della signed distance function) e si estrare da questo dataset una mesh poligonale (con algoritmo Marching Cubes con soglia 0)



Nel metodo **diretto**, si mantengono i punti della point cloud inalterati, come vertici della mesh. Questo è un problema quando la point cloud presenta difetti come: **rumore**, outliers, difetti di allineamento (se la nuvola di punti è ottenuta allineando e unendo due nuvole separate, come spesso è il caso), densità diverse non volute o accidentali, parti mancanti. Sono difetti comuni nelle point cloud generate da acquisizioni 3D.

Nel metodo **indiretto**, i vertici nella mesh finale sono il risultato di stime diverse di posizione questo è uno svantaggio, quando la point cloud è di ottima qualità (accurata, ben distribuita, completa). La regola generale è: ricampionare un qualsiasi dataset introduce sempre un deterioramento del segnale. È invece un grosso vantaggio in altri casi, perché consente di:

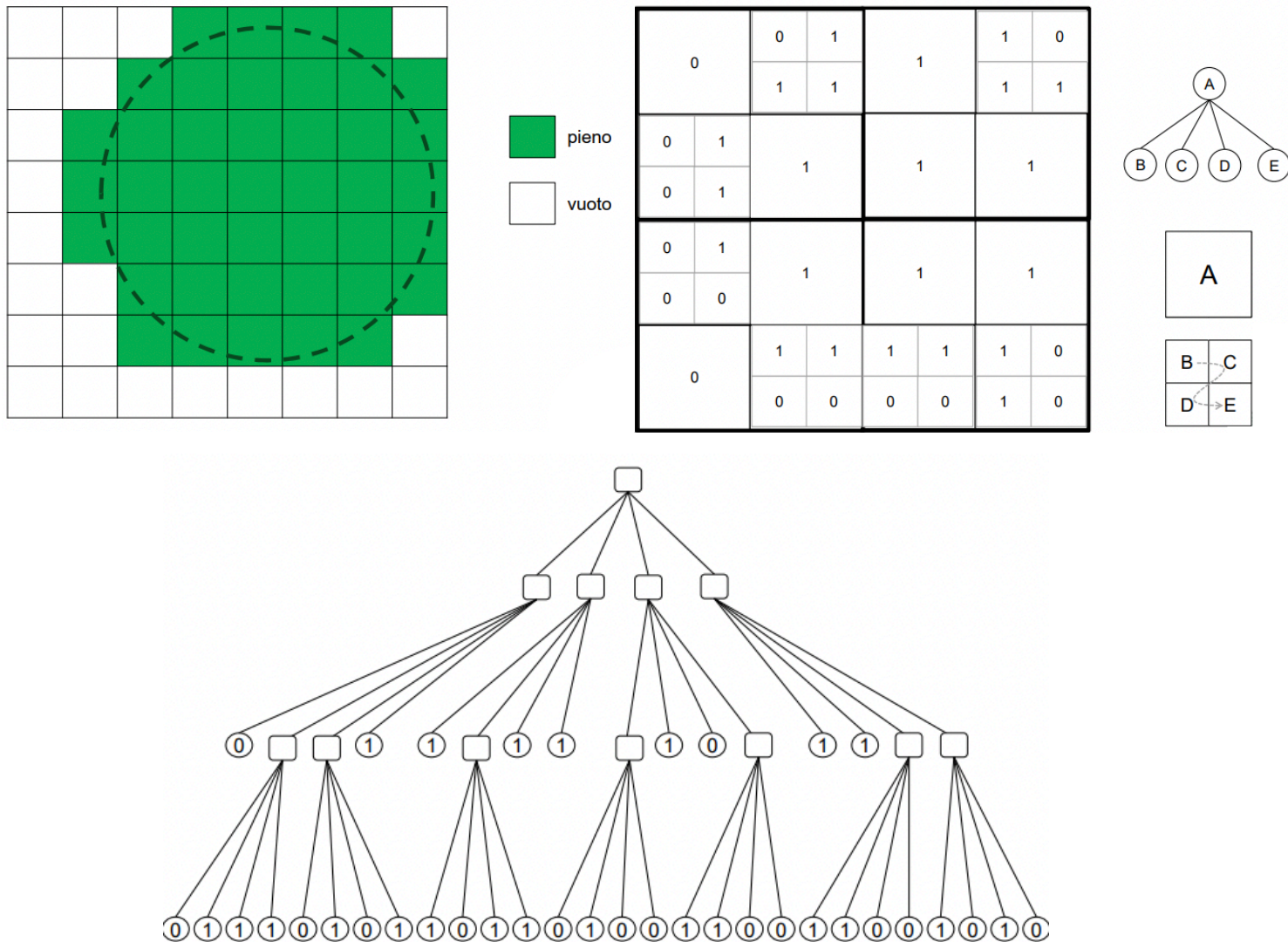
- abbattere il rumore, fondendo stime diverse (di posizione e normale della superficie) in una sola stima congiunta, più accurata
- completare la superficie, estendendola in modo naturale su zone mancanti
- garantire la two-manifoldness, chiusura, e orientamento consistente del risultato – caratteristiche del Marching Cubes

Alcuni problemi ulteriori:

- La mesh risultante è molto irregolare e presenta triangoli di forma sfavorevole (lontano da equilatera) – altre caratteristiche del Marching Cubes
- Il dataset volumetrico è affetto dalla **curse of dimensionality**, $O(n^3)$ memoria: usando alte risoluzione, diviene molto oneroso, unfeasible usando basse risoluzioni, si perdono molti dettagli geometrici

Il problema della «curse of dimensionality» è dovuto all'uso di **risoluzione** di un dataset di voxel non è adattiva. Esistono strutture dati più compatte, perchè dotate di risoluzione adattiva. Una delle più diffuse è l'**oct-tree**.

Vediamo prima una sua versione 2D: il **quad-tree** → in questo primo esempio, ipotizziamo di voler rappresentare in forma di quad-tree un dataset “voxelizzato” (ma in 2D) con 1 bit per voxel (1 = pieno, 0 = vuoto)



Come quad-tree: solo 28 foglie (+9 nodi interni)
come volume di voxel equivalente: $8 \times 8 = 64$ voxel

Oct-tree:

- Struttura ricorsiva, ad albero
- Un'alternativa molto più efficiente a dati una voxelizzati
- Ogni nodo è associato ad un cubo nel volume
 - Radice: associato all'intero dataset
 - Hanno 8 figli, uno per ciascun ottavo del padre
- Foglie: memorizzano il dato da memorizzare nei voxel (ad esempio, 1 bit – pieno o vuoto oppure un valore di signed distance field)
- **Numero totale nodi:** circa proporzionale all'estensione del bordo dell'oggetto rappresentato, è quindi **quadratico**, non cubico, con la risoluzione.

Dataset superficiali regolari

Sono tre strutture dati **superficiali** che si basano su un concetto simile, ovvero il campionamento della superficie. I campioni sono memorizzati su una **griglia regolare 2D**.

Connettività implicita = ogni campione è implicitamente connesso ai propri vicini sulla griglia.

Vantaggi comuni di queste strutture:

- La connettività è implicita (non deve essere memorizzata)
- Anche le relazioni di adiacenza fra elementi sono implicite
- Multi-risoluzione semplice da ottenere (come per le immagini 2D rasterizzate – cioè di pixel)
- Geometry Processing può essere parallelizzato: molto facile da ottenere (usando la GPU)
- Machine learning su forme 3D: reso semplice dalla regolarità (è come Machine Learning su immagini)
- Analogia con immagini 2D (molti vantaggi – es compressione, editing, filtering)

Alcuni svantaggi:

- La risoluzione non è adattiva
- L'espressività è **limitata** (solo alcune forme possono essere rappresentate)

height-field

```
var height[resX][resY]float;
```

Campo di altezze (o mappa di altezze, o height map, o modello 2.5D o mappa/immagine di profondità o depth map o depth-image).

array 2D di valori scalari, il valore ad indice x,y rappresenta il punto 3D con quelle coordinate.

Solo la z va memorizzata.

Range scan

```
float[][] rangeScan=newfloat[resX][resY]
```

array 2D di misurazioni di profondità.

range scanning = tecnica di acquisizione 3D che produce range-scans.

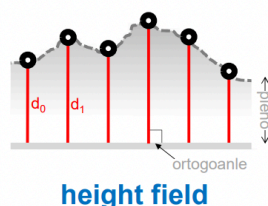
depth camera = macchina fotografica che cattura un range scan. Ogni valore misura la distanza di una superficie dall'osservatore.

```
var rangeScan [resX][resY]float
```

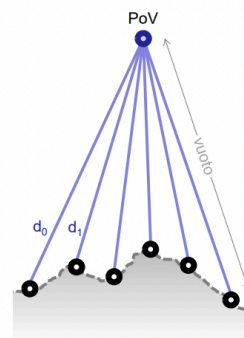
La range scan presenta molti difetti di misurazione: rumore, outlier, missing data...

La connettività implicita la rende una Triangular Mesh, completamente regolare, ben orientata, 2 manifold.

Stessa struttura,
ma la semantica
è differente



height field



range scan

Geometry images

`var geoimage [resX][resY] Point`

una griglia regolare che memorizza direttamente campioni come punti 3D. Memorizza la geometria di una mesh, la connettività è implicita, gli attributi possono essere memorizzati in griglie come la geometria.

Non molto usato(perché richiede parametrizzazioni difficili da ottenere e distorte).

PARTE II: 3D rendering

Subdivision Surfaces (cenni)

Una linea curva può essere approssimata da una linea spezzata. Suddividere (subdivide) una linea spezzata C serve per ottenere una versione più ad hi-res di C (detta curva limite), costruendo per ogni segmento di C dei segmenti più piccoli.

Esistono *molti* procedimenti diversi, uno dato **schema** prescrive come ottenere la nuova:

- **connettività** (quali nuovi vertici aggiungere, come connettere vecchi e nuovi vertici con nuove facce, se mantenere o no i vertici originali)
- **geometria** (e attributi)

Esistono molti schemi di suddivisione per curve. Ad ogni passaggio, i vertici pre-esistenti vengono:

- lasciati inalterati → schemi detti **interpolativi** (perché la curva limite passa da i vertici di partenza)
- spostati in una nuova posizione → schemi detti **approssimativi** (perché la curva limite si avvicina solo, approssima, i vertici di partenza)
- rimossi → schemi detti **duali**

Uno schema di suddivisione prescrive:

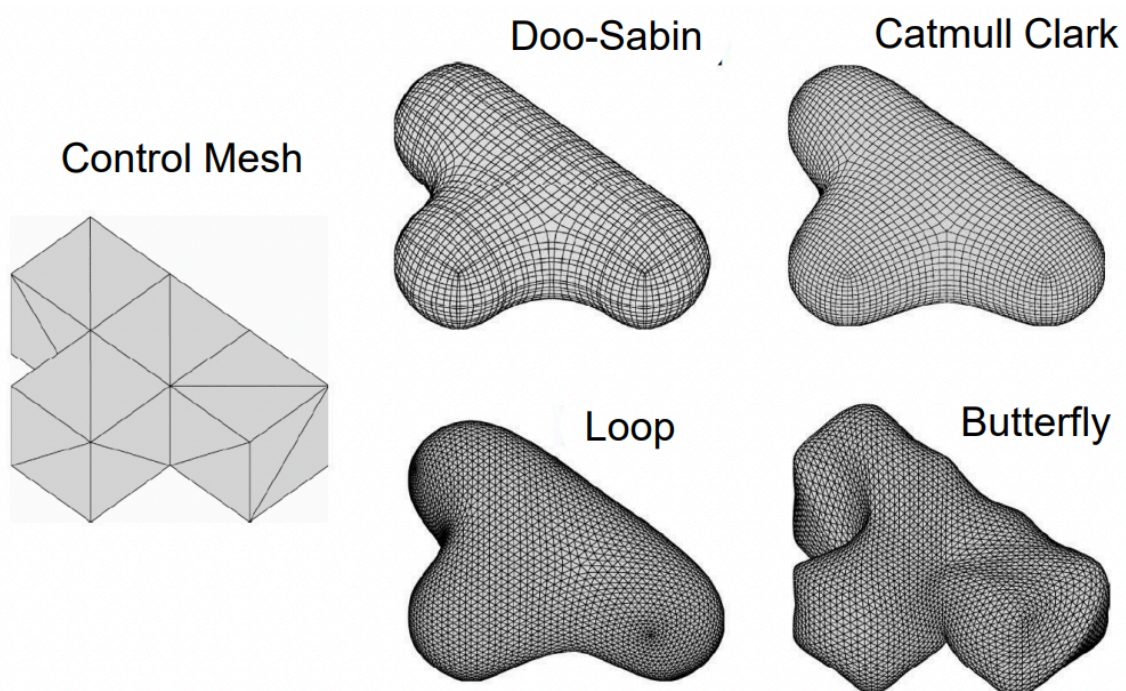
- Dove posizionare i nuovi vertici (come funzione dei vertici vicini)
- Se mantenere i vecchi vertici; se sì: se e dove ri-posizionarli (come funzione dei vertici vicini)

Suddividere (subdivide) una mesh M → ottenere una versione più ad hi-res di M , suddividendo ogni poligono di M in poligoni più piccoli. Attraverso un procedimento automatico, adottando cioè uno schema di suddivisione.

Subdivision surface → Iterando il processo di suddivisione, si converge ad una «superficie limite»

Differenze principali fra schemi:

- Interpolativi VS Approssimativi VS Duali (come visto per le curve)
- Poligonalità della mesh richiesta in input, e prodotta in output Tri-mesh? Quad-mesh? Mesh poligonale in generale?
- Caratteristiche della superficie limite, in termini di smoothness (continuità di normale e/o curvatura)



Alcuni schemi di suddivisione comuni per mesh:

- Catmull-Clark → da mesh-poligonale (generica) a pure-quad mesh, approssimativo
- Doo-Sabin → da mesh-poligonale a mesh poligonale, duale
- Loop → da tri-mesh a tri-mesh, approssimativo
- sqrt(3)
- Butterfly → da tri-mesh a tri-mesh, interpolativo
- Kobbelt

Connettività:

- quali nuovi vertici produrre?
- come connetterli con nuove facce ed edge?

Geometria:

- come ottenere le coordinate (e gli attributi) dei nuovi vertici (a partire dai vertici originali?)
- solo per schemi approssimativi: dove spostare i vertici originali?

Due modi di usare uno schema di suddivisione:

- Come modo per **rappresentare superfici curve**
 - La mesh poligonale iniziale, detta **control mesh**, o base mesh serve come rappresentazione della **limit surface**
 - Come fase del rendering, la base mesh viene suddivisa (per n passi) e mandata a schermo (come mesh poligonale)
 - Il modellatore manipola la control mesh come sostituto della limit surface
- Come **strategia per costruire mesh poligonali hi-res**
 - Modellazione low-poly di una mesh iniziale
 - il modellatore richiede un passo di suddivisione ogni volta che voglia incrementare la risoluzione
 - la mesh risultante può essere modificata e/o suddivisa ulteriormente
 - nota: la mesh iniziale può essere poi scartata

Immagini digitali in breve

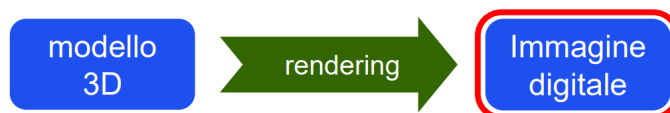


Immagine = Un assegnamento di un colore ad ogni punto di una regione piana rettangolare. Quindi una funzione f da (x,y) a (colore).

Lo studio dei colori come fenomeno fisico, della sua misurazione e rappresentazione è il soggetto di alcune discipline quali:

- **Colorimetria** = studia la misurazione e rappresentazione del colore
- **Radiometria / Fotometria** = ramo della fisica che studia della radiazione elettromagnetica (che comprende la luce)

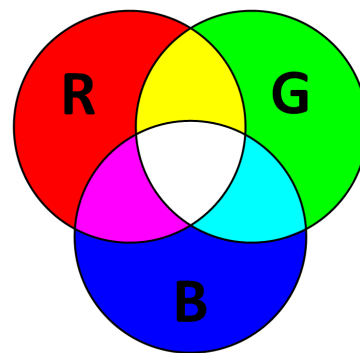
In questo corso, risponderemo ad una quesito più semplice: Cosa devo mandare ad un monitor per per riprodurre un dato colore in uno dei suoi pixel?

Sintesi additiva del colore → Il pixel di un monitor riproduce un dato colore sovrapponendo 3 luci ad 3 date intensità:

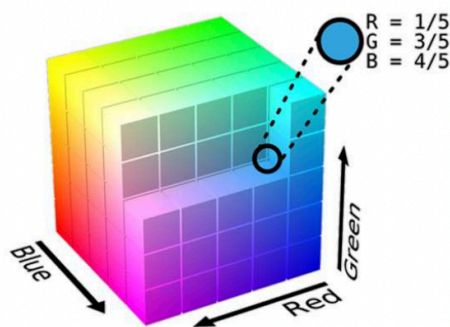
- una rossa (R)
- una verde (G)
- una blu (B)

Questa soluzione è giustificata da considerazioni sulla percezione umana (che ricadono nel dominio della colorimetria) la **teoria del tristimolo**.

Per determinare quale colore riprodurre dobbiamo scegliere tre valori di intensità (per R, G, B), ciascuno scelto fra un massimo e un minimo (es fra 0.0 e 1.0, o fra 0 e 255). I valori min e max corrispondono rispettivamente alla massima e minima intensità della luce prodotta da tre emettitori (che insieme costituiscono un pixel sul monitor)



Un colore (che vada riprodotto su un monitor) è dunque rappresentabile da una tripletta di valori scalari. Questa struttura dati somiglia a quella per un punto 3D / vettore 3D: tre componenti scalari (seppur chiamati R,G,B invece che X,Y,Z). I valori di R,G,B sono interpretabili come le coordinate di un punto di uno **spazio colore 3D**: in cui ogni punto è un colore diverso (lo spazio colore «RGB» è solo un esempio di spazio colore).



Immagini digitali:

- Rappresentazioni vettoriali → un set di primitive 2D sovrapposte, quali: curve parametriche, triangoli, poligoni (2D), cerchi, zone contornate da curve di Bèzier (2D), testo (in ascii, ad una pos, associato ad un font). Ciascuna primitiva associata ad un colore.
- Rappresentazioni rasterizzate → una griglia regolare 2D di campioni di colore, detti “pixel” (da “picture element”).

Ogni «canale» (R, G, B) è rappresentato da un **numero intero a n bits** dove n è detta la **profondità del canale**. Quindi, come numero da 0 a $(2^n)-1$. Caso più diffuso: $n = 8$ (1 byte per canale).

Conseguenze:

- Livello massimo esprimibile: = 255.
- In decimale, un colore può essere espresso come, ad es (150,77,103)
- In esadecimale, il colore è espresso come, ad es, 0x964D67

Ogni «canale» (R, G, B) è rappresentato da **uno scalare da 0 a 1**. Si può usare la stessa struttura dati utilizzata per punti e vettori. Per es, nel linguaggio GLSL il tipo vec3 si usa per rappresentare punti, vettori, o colori e le sue tre componenti scalari possono essere chiamate intercambiabilmente.

Caratteristiche delle immagini Raster

- Resolution = numero di linee e colonne. Esprimibile anche in MegaPixel.
- Ogni pixel ha k canali, 1 canale = 1 valore scalare.
- Immagine RGB = 3 canali: Red, Green, Blue.
- Immagine toni di grigi (gray-scale) = 1 canale (gray level)
- Ogni canale è memorizzato con un dato numero di bit. se 8: ho 28 = 256 valori possibili, min = 0 and max = 255. Immagine B & W = il solo canale ha un 1 bit (0 = B e 1 = W)
- **image depth** = (profondità) bit totali per un pixel. per es: RGB con 8 bits per channel («true color») = 24 bits. per es: B & W image (una «bitmap»): = 1 bit

raggio dinamico = rapporto fra luminosità del punto più luminoso e quello più buio

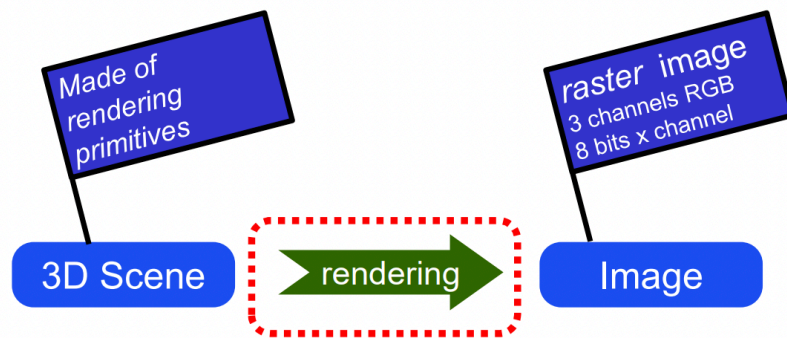
HDRI – High Dynamic Range Images = immagini con raggio dinamico elevato. Richiedono più bit per canale.

Immagini Raster e RAM, totale memoria = $\text{resX} \cdot \text{resY} \cdot \text{imageDepth}$

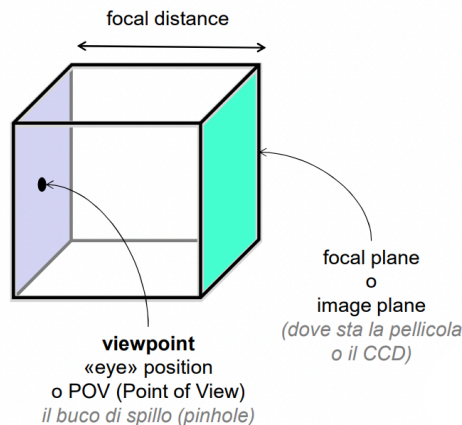
Compressione spesso necessaria, molti schemi di compressione sono usati. Tipi di schemi di compressione:

- Lossless : non deteriorano il dato
- Lossy : più aggressivi (es 1:10) deteriorano il dato
- Texture compression : una compressione lossy di tipo speciale utilizzabile dalle immagini di texture

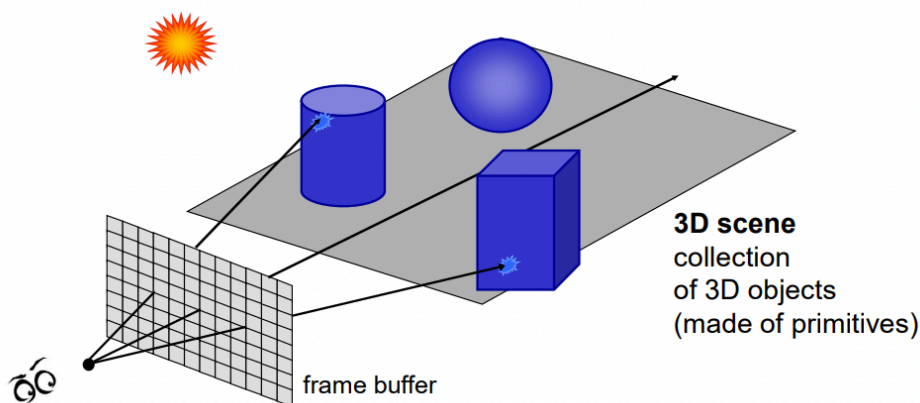
Rendering: ray-tracing



Molti algoritmi di rendering prendono diretta ispirazione dal modo in cui un dispositivo reale di sintesi di immagini produce un'immagine. Dispositivi come: una macchina fotografica (analogica o digitale), una telecamera, un occhio umano. A questo fine, possiamo rimuovere tutti gli elementi non necessari di tali dispositivi, per ridurli ad un modello più semplice possibile. Questo modello è detto Pin-hole camera.



Pin-hole camera = cattura tutta la luce (= i fotoni) che passa per il POV. La pin-hole camera misura quanta di questa luce arriva da ogni direzione: ogni pixel rilevato $p[i,j]$ = quantità di luce che è passata dal POV in direzione (simile a) $(POV - P(i,j))$ durante il tempo di esposizione. La luce è partita da degli emettitori (fonti di luce) prima di arrivare al POV, ha interagito con la scena.



Ray-Casting = (backwards" ray-tracing) seguire a ritroso i fotoni che raggiungono il POV.

Per ogni pixel sullo schermo:

1. costruisco un raggio (detto il raggio primario di quel pixel)
2. trovo le sue intersezioni con gli oggetti della scena
3. scelgo l'intersezione più vicina al POV

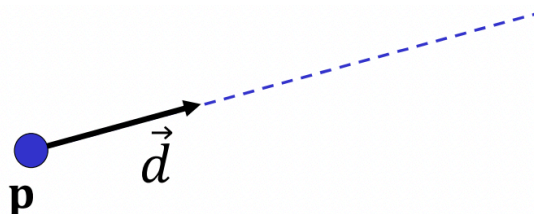
Implementazione: basata sul computo dell'intersezione raggio-primitive (che quindi deve essere ottimizzata). Numero di intersezioni da calcolare = numero di pixel.

Con un algoritmo di Ray-casting possiamo renderizzare qualsiasi cosa per la quale siamo in grado di computare l'intersezione con un raggio. Vale anche per qualsiasi algoritmo di Ray-tracing. Esistono quindi ray-caster per il rendering di: Mesh poligonali, Campi d'altezza, Modelli impliciti, Superfici parametriche, come patch di Bezier. In ciascun caso, serve di sapere implementare la procedura di intersezione raggio-oggetto.

Un raggio (ray) è una semiretta. La possiamo modellare come:

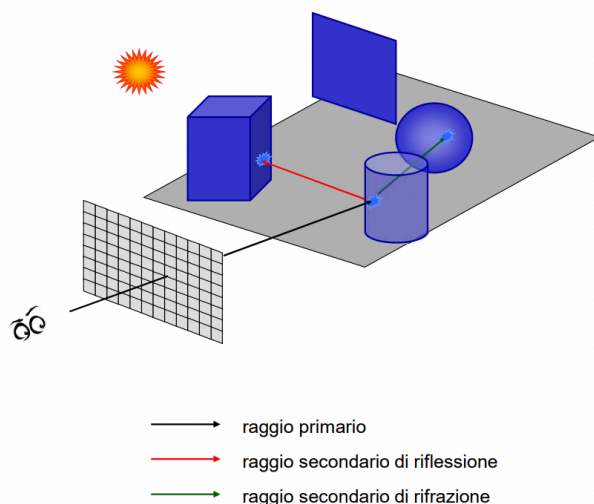
- Punto $\mathbf{p}$ di partenza del raggio
- Vettore $\vec{d}$ direzione (unitario o no, è una scelta)

I punti sul raggio sono i punti $\mathbf{p} + k\vec{d}$ per un qualche scalare $k \geq 0$ cioè i punti raggiungibili a partire da $\mathbf{p}$ facendo k passi in direzione $\vec{d}$.



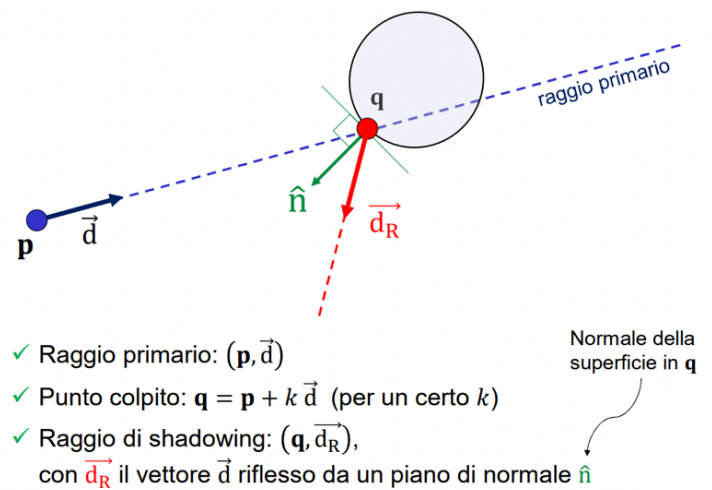
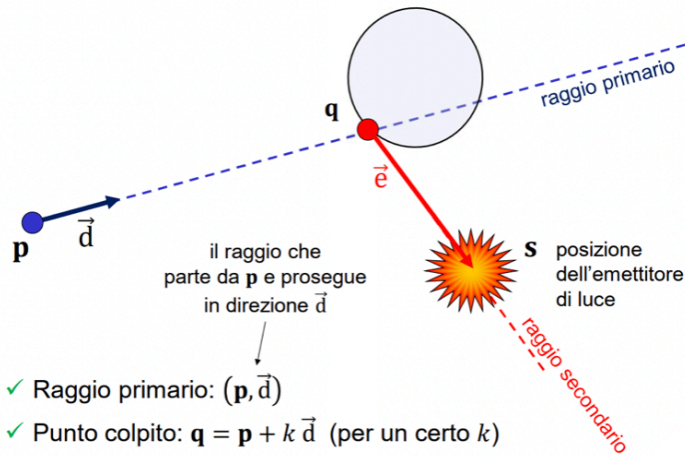
Intersezione raggio con più primitive → Basta prendere la primitiva che interseca col k minore fra k_0, k_1, k_2

Ray Tracing = (classe di algoritmi). Seguendo la stessa idea per un altro passo, **tracciamo** a ritroso il percorso dei fotoni, non solo dal POV ad un punto su un oggetto 3D, ma dall'oggetto 3D all'emettitore della luce che ha emesso quel fotone. Questo richiede di computare le intersezioni su un ulteriore raggio, detto di **shadowing**, è un esempio di raggio secondario (è la stessa procedura, intersezione raggio-primitiva, eseguita più volte per uno stesso pixel). Algoritmi basati su tracciamento di raggi sono detti di Ray-Tracing (compreso il semplice ray-casting).



Come computare la direzione raggio riflesso?

- Input: Direzione del raggio primario, Normale della superficie nel punto colpito dal raggio primario
- Output: Direzione del raggio secondario (insieme alla posizione colpita dal raggio primario, questo modella il raggio secondario di riflessione)



Algoritmi

Molti algoritmi e varianti si basano su principi simili. Alcuni di questi:

- **Ray-casting** = un solo raggio (primario) viene “mandato” (cast) per pixel
- **Ray-tracing** = Ogni raggio viene “tracciato” (traced) attraverso vari fenomeni successivi riflessione o rifrazione, attraverso raggi secondari. Nota: in ordine **inverso** rispetto alla realtà, come al solito (è anche usato come nome della classe generale di questi algoritmi)
- **Path-tracing** = Per ogni pixel, lanciare molti raggi, che seguono i path pseudo-casuali (“monte carlo sampling”) con una distribuzione che riflette quella reale. Mediare i risultati per ottenere il pixel finale. Più raggi vengono mandati, maggiore il realismo.
- **Ray-marching** = Una classe di algoritmi per accelerare l'intersezione raggio-primitiva. Principio: raggio viene spezzato in una serie di passi (segmenti) successivi, per ciascuno dei quali si devono testare solo le primitive a lui vicine (invece di tutte).

Rendering: Rasterization

Algoritmi di rendering si dividono in due categorie:

- basati su **Ray-tracing** → per ogni pixel → si lancia un raggio, si trovano le intersezioni del raggio con le primitive; si determina il colore del punto colpito (per es., calcolando le luci da cui è raggiunto, l'illuminazione...)
- Basati su **Rasterization** → per ogni primitiva → la si proietta sullo schermo, in 2D (trasformazione). Si converte la forma 2D in pixel (rasterizzazione). Si determina il colore di questi pixel.

Gli algoritmi di rendering basati su **rasterizzazione** sono stati resi molto popolari dal supporto di Hardware specializzato come la **GPU**. Sono gli algoritmi per cui questo Hardware è stato pensato e progettato.

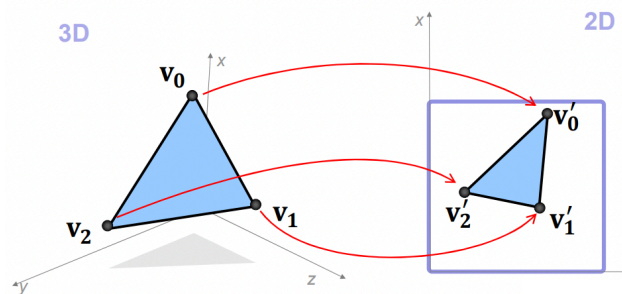
Rendering accelerato con GPU: necessario per rendering in tempo reale (per es, videogame)

Recentemente: anche algoritmi basati su ray-tracing vengono spesso accelerati su GPU.

La principale (e quasi unica) primitiva di rendering supportata è il **triangolo**. Cioè, la GPU è in grado di **trasformare** e, soprattutto, **rasterizzare** triangoli. Dunque, l'Hardware-supported rendering in tempo reale è soprattutto **rendering di Triangular meshes**.

Come trasformare un triangolo da 3D a 2D?

1. trasformare ciascuno dei suoi tre vertici
2. connetterli in un nuovo triangolo



Come rasterizzare, ovvero convertire in pixel, un triangolo?

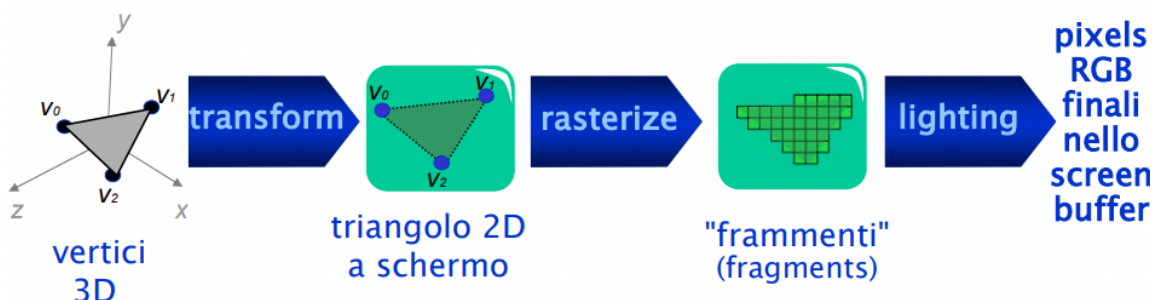
Bisogna trovare quali «frammenti» generare: uno per ogni pixel dell'immagine contenuto dal triangolo.

«**Frammento**» = un piccolo pacchetto di dati per il quale vogliamo generare un pixel (esempio: normale, colore di base...) Lo distinguiamo da «pixel» = il colore RGB finale generato per un frammento.

Questo processo è anche detto **T&L = Transform & Lighting**.

Transform = trasformazioni di sistemi di coordinate. scopo: portare le primitive in spazio schermo

Lighting = computo illuminazione. scopo: calcolare il colore RGB finale di ogni pixel dell'immagine finale



Rasterization-based rendering ha come input un insieme di vertici che formano triangoli.

L'algoritmo a pipeline (catena di montaggio) consiste in diverse fasi:

1. Fase per **vertice** → ogni vertice viene portato in una posizione 2D sullo schermo . Si tratta di una "trasformazione spaziale".
2. Fase per **triangolo** → ogni triangolo 2D viene rasterizzato a schermo. Viene cioè identificato un "frammento" per ogni pixel coperto dal triangolo 2D.
3. Fase per **frammento** → Per ogni frammento, si computa il colore RGB del pixel corrispondente (tipicamente, calcolando l'illuminazione).

//Considerazioni generali sul Pipeline.

Ogni fase del pipeline avviene in parallelo. Ogni vertice, triangolo, frammento può essere processato in contemporanea con ciascun'altro.

Le 3 fasi sono in cascata. Il pipeline va tanto veloce quanto la sua fase più lenta, detta il collo di bottiglia. Se il collo di bottiglia avviene:

- per vertice, l'applicazione si dice «transform limited»
- per frammento, l'applicazione si dice «fill limited»

Primitive di rendering

Q: quali **primitive di rendering** usando **rasterization**?

A: qualunque cosa si sappia:

- 1) proiettare (da 3D a 2D)
- 2) «rasterizzare» (in 2D)

convertire
in pixel

per le GPU

Quindi, allo stato attuale:

1. PUNTI
2. SEGMENTI
3. **TRIANGOLI**

per es, point splatting
per point clouds

Per es,
per triangle mesh

Il caso principale!
(di gran lunga).
Le GPU sono
ottimizzate per
questo caso

E' il motivo
principale
della popolarità
delle tri-mesh
come prim

Q: quali **primitive di rendering** usando **ray-tracing**?

A: qualunque cosa si sappia:

- 1) intersecare con un raggio

efficacemente!

Quindi:

Per es,
per triangle mesh

⇒ Triangoli?

Anche, ma anche

primitive più complesse, come:

⇒ Implicit surfaces

Per es,
per CSG

⇒ Height fields

⇒ Voxelized dataset

⇒ ...

Il metodo
principale
per «direct
volume
rendering»

Vantaggi del ray-tracing:

- E' una classe di algoritmi concettualmente semplice
- E' facile ottenere effetti visuali complessi e realistici (incluso rifrazione, riflessioni, ombre...)
- Simula in modo abbastanza accurato la sintesi di un'immagine da parte di una macchina fotografica (con molte semplificazioni)
- E' possibile renderizzare direttamente qualsiasi «primitiva di rendering»: basta implementare la funzione di computo di intersezione con un raggio
- Parallelismo implicito: ogni pixel può essere implementato in parallelo (modello SIMD = single instruction, multiple data)

Vantaggi degli approcci basati su rasterizzazione:

- Complessità lineare con numero di primitive
- Per ogni primitiva, si processano solo i pixel coinvolti
- E' l'algoritmo per il quale le GPU sono state pensate, dunque supporto HW garantito

Visione storica / tradizionale (il luogo comune)

Ray-tracing

- Lento, ma **accurato**
- Gli algoritmi standard per il rendering offline
- Per esempio, usato nella movie industry

Rasterization

- **Veloce**, ma approssimato
- Gli algoritmi standard per il rendering online
- Per esempio, usato nella game industry



Render farms = Il rendering di entrambi i tipi sono massicciamente parallelizzabili (sono dotati di Implicit parallelism). Per avvantaggiarsene, abbiamo bisogno di hardware parallelo. Anche il algoritmi di ray-tracing sono parallelizzato.

La realtà: **raytracing non** è necessariamente **lento**

- È intrinsecamente parallelizzabile → alto livello di parallelismo implicito e quindi, implementabile con HW parallelo (per es., render farms)
- Le primitive possono essere forme più **complesse** → Questo riduce il numero di primitive necessarie a comporre una data scena virtuale
- Ottimizzazioni: usando apposite strutture dati, è possibile ridurre fortemente il numero di primitive da testare per ogni raggio → Strutture di indicizzazione spaziale. Nota: è più arduo per scene animate. Le complessità degli algoritmi di Ray-Tracing può essere resa sublineare col numero di primitive.

Real-time raytracing oggi è comune.

La realtà: **rasterization based** supporta effetti di luce **compessi e ralistici**

- Esistono molti algoritmi basati sul rasterization per simulare, o approssimare:
 - Ombre portate
 - Diffrazioni
 - Rifelssioni speculari
 - Riflessioni multiple (illuminazione «globale»)
 - Rifrazioni
 - E altri effetti tradizionalmente presenti solo negli algoritmi di raytracing
- Si tratta di algoritmi specializzati, per ciascuno di questi effetti, usati per es nei videogames

Lo stato dell'industria di intrattenimento

L'industria cinematografica si avvale quasi esclusivamente di algoritmi basati su ray-tracing

- Per il rendering finale (offline)
- Ma usa algoritmi basati su rasterization per: effettuare preview (real time), per costruire asset (ad esempio, modellare le mesh), etc

- Rendering finale: parallelizzato massicciamente su render farms, ma comunque spesso lento (ore o anche decine di ore per fotogramma)

L'industria dei videogames si avvale quasi esclusivamente di algoritmi basati su rasterization

- Accelerati su GPU consumer level
- Dunque i modelli 3D usati sono sempre tri-meshes
- Trend recente: passare invece ad algoritmi di ray-tracing oppure misti

*"Rasterization is fast, but needs cleverness to support complex visual effects.
Ray tracing supports complex visual effects, but needs cleverness to be fast."*

David Luebke (NVIDIA)

Trasformazioni Spaziali

Lista completa degli effetti geometrici ottenibili con trasformazioni affini

✓ Rotazioni
⇒(qualsiasi)



✓ Traslazioni



✓ Scalature
⇒uniformi o no
⇒compreso ribaltamenti



✓ Shearing

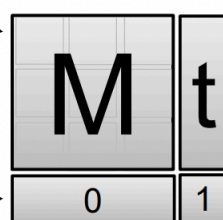


✓ ... e tutte le composizioni di queste operazioni



Questa sottomatrice 3x3 si applica alle coord cartesiane sia dei **punti** che dei **vettori**

Ultima riga: (0,0,0,1)
Così che i **punti** vengano mappati sempre in **punti**, e i **vettori** sempre in **vettori**



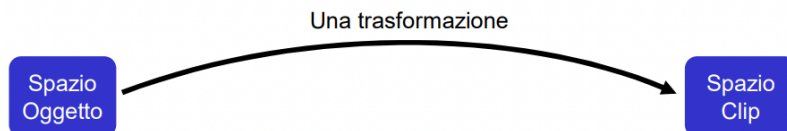
Matrice 4x4

Questo vettore t si somma al risultato quando trasformo **punti** ma viene moltiplicato per zero quando trasformo **vettori**.

La sequenza di trasformazioni nel rendering: visione di insieme

La trasformazione che stiamo cercando va dallo spazio in cui sono definite inizialmente le nostre mesh: lo «spazio oggetto» (iniziale) ad uno spazio definito sullo schermo in cui vengono proiettate: lo spazio clip (finale).

Il rasterizzatore si aspetta triangoli da rasterizzare definiti nello spazio clip! Vediamo questi due spazi iniziali e finali. Per definire uno spazio occorre definire: l'origine e 3 assi.



Spazio oggetto = E' il sistema di riferimento in cui è definito ciascun modello 3D. Cioè lo spazio in cui esprimo, per esempio:

- In una mesh: le posizioni dei vertici, le normali (vettori unitari),
- i punti di controllo di un bezier patch,
- l'input della funzione che definisce un modello implicito,
- Le posizioni (interi) dei voxel in un modello voxellizzato
- Le coord. (x,y: implicite; z: memorizzata) dei punti in un campo di altezza
- etc.

Ogni modello è provvisto del suo spazio oggetto. il suo sistema di riferimento «personale» in cui il modellatore / lo scanner etc ha definito i punti e i vettori che lo costituiscono. E' il sistema di riferimento in cui sono espressi, ad esempio, le coordinate dei vertici e nelle normali in un file OBJ.

// Per noi ora: è lo spazio di partenza della sequenza di trasformazioni!

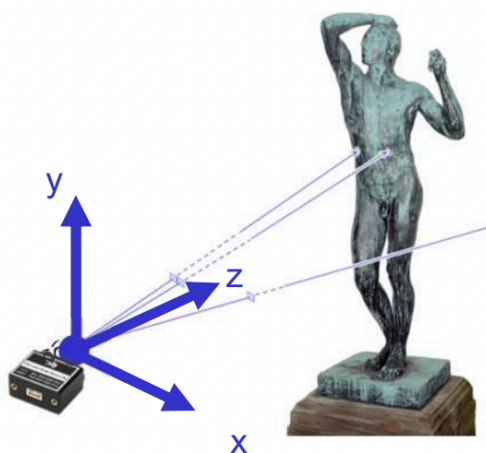
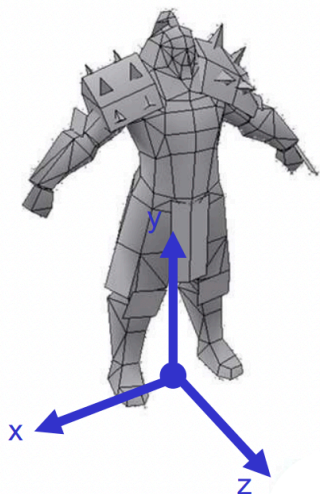
Lo spazio oggetto è scelto arbitrariamente del modellatore (o del software) che produce il modello 3D.

Per esempio, una convenzione per una mesh 3D che rappresenti un personaggio di un videogioco è spesso:

- Origine: un punto sul terreno localizzato fra i piedi del personaggio
- Asse X: l'asse dalla sinistra alla destra del personaggio
- Asse Y: l'asse dal basso all'alto del personaggio
- Asse Z: la direzione avanti del personaggio

Per esempio, una convenzione per una range scan:

- Origine: punto di partenza dei raggi (posizione dello scanner)
- Asse X: l'asse dalla sinistra alla destra dello scanner
- Asse Y: l'asse dal basso all'alto dello scanner
- Asse Z: l'asse centrale del cono dei raggi che vengono mandati dello scanner



Spazio Clip = (o «Normalized Device Coordinates», «NDC») Uno spazio allineato allo schermo / all'immagine prodotta dal rendering

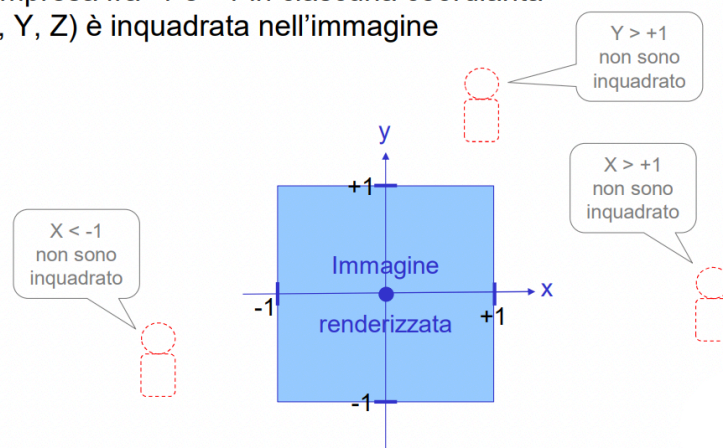
- Origine: centro dello schermo / dell'immagine
- Asse X: orizzontale allo schermo / all'immagine, da sx a dx
- Asse Y: verticale allo schermo / all'immagine, verso l'alto

In realtà, anche questo spazio è tridimensionale (utile per computare correttamente gli effetti di occlusione). Si aggiunge quindi:

- Asse Z: ortogonale allo schermo

Lo spazio clip è lo spazio finale in cui dobbiamo portare tutte le primitive (nel rendering basato su rasterizzazione).

- ✓ In spazio clip vale una convenzione: solo la parte di scena compresa fra -1 e +1 in ciascuna coordinata (X, Y, Z) è inquadrata nell'immagine



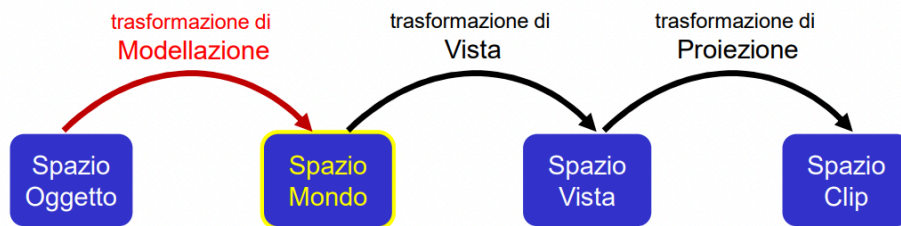
I modelli che solo parzialmente inclusi fra -1 e +1 sono spezzati (**clipped**) in una parte da rasterizzare e in una parte non rasterizzata. Questo, storicamente, dà il nome allo spazio ("lo spazio in cui avviene il clipping").

In spazio clip l'immagine renderizzata / lo schermo ha dimensione 2 (da -1 a +1) sia verticalmente che orizzontalmente. Questo è indipendente dalla risoluzione in pixel o dall'aspect ratio dell'immagine rasterizzata (o dello schermo). Per questo motivo, le coordinate in spazio clip sono anche dette NDC (**Normalized Device Coordinates**, non dipendono dal dispositivo di output, cioè dal monitor).

Anche nella direzione Z, vale che la scena viene renderizzata solo nella porzione che va da -1 a +1. Anche le parti della scena troppo lontane o troppo vicine vengono scartate.

La trasformazione dallo spazio oggetto (iniziale) allo spazio clip (finale) viene scomposta (almeno, concettualmente) in una piccola sequenza di passaggi dal significato standard. Anche ciascun passaggio può essere visto come un cambio di sistema di riferimento (o spazio). Ogni passaggio cattura alcune scelte da effettuare durante il rendering. Vediamo gli spazi intermedi, in sequenza.

Per definire uno spazio occorre definire l'origine e 3 assi.



Una scena 3D è spesso costituita da molti modelli 3D. Ciascuno provvisto del suo spazio oggetto.

Spazio Mondo = è il **sistema di riferimento globale della scena**: è comune a tutta la scena (a differenza dello spazio oggetto, che è «personale» di ogni oggetto). Questo spazio è definito (in modo arbitrario) da chi compone la scena (mettendo insieme, ad esempio, diverse mesh). Ad esempio: un level design di un video-game.

Trasformazione di modellazione = Trasforma i punti / vettori da spazio oggetto a spazio mondo. Riflette in che modo ciascun oggetto è **disposto fisicamente nella scena**. Per comporre la scena, dotiamo ogni suo oggetto che la compone della sua trasformazione di modellazione. Per modellare un interno, dovremmo mettere insieme modelli 3D separati di: mobili, pareti, lampadari... Determinare queste trasformazioni di modellazione, per ciascun oggetto, è il modo di modellare (costruire un modello di) la scena. Da cui, il nome della trasformazione.

Un modo facile ed intuitivo di determinare la trasformazione di modellazione è definirla come una sequenza (matrix duplication) arbitraria di trasformazioni in cascata di (matrici)

- rotazione,
- traslazione,
- e scaling

che riflettono

- l'orientamento,
- la posizione,
- e la dimensione

desiderata per quell'oggetto nella scena (cioè in spazio mondo)

In una computer animation, posso spostare un oggetto nella scena, ridefinendo in ogni fotogramma la sua *trasformazione di modellazione*. Cambiando così in un colpo solo le coordinate dei suoi punti e vettori in spazio mondo, ma non in spazio oggetto!

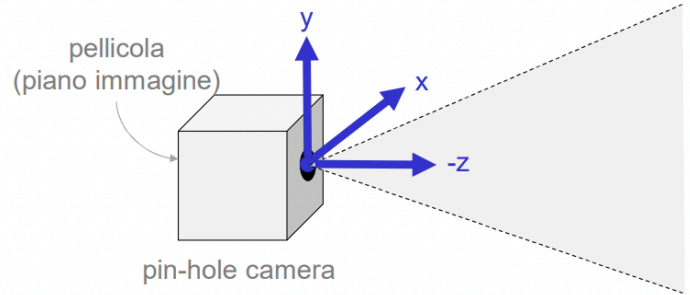
La struttura dati della mesh – che è memorizzata in RAM in spazio oggetto - non cambia!

multi-instancing = disegnando, in uno stesso screen buffer, una stessa mesh più volte, ogni volta con una diversa matrice di modellazione, (cioè: disegnando diverse istanze dello stesso oggetto), ottengo diverse copie dello stesso oggetto nella scena (Nota: ogni istanza ha punti e vettori di coordinate diverse, in spazio mondo, ma identiche, in spazio oggetto).

Mantenendo in memoria una sola mesh (con coordinante oggetto univoca), mostro diversi oggetti a schermo (ciascuno con coordinante mondo diverse). Ad esempio, possono essere istanze di una stessa mesh: tutte le sedie della mia scena di interno, tutte le forchette sul tavolo, le 4 ruote del modello di una macchina, Etc.

spazio vista = Lo spazio vista è posizionato in modo standard rispetto alla pin-hole camera che inquadra la scena

- Origine: point of view (l'hole della pin-hole camera)
- Asse X: asse orizzontale della camera
- Asse Y: dal basso all'alto della camera
- Asse Z: dal davanti al dietro della camera



Trasformazione di vista = Trasforma i punti / vettori da spazio mondo a spazio vista.

Dipende dalla posizione e dall'orientamento della camera che inquadra la scena, detti i **parametri ESTRINSECHI della camera**. Se la telecamera si sposta, cambia la trasformazione (la matrice) di vista. Questo causerà ovviamente uno spostamento nel piano immagine degli oggetti inquadrati, anche se la loro posizione nella scena (in spazio mondo!) non è cambiata. Tipicamente è una trasformazione rigida: rotazione + traslazione.

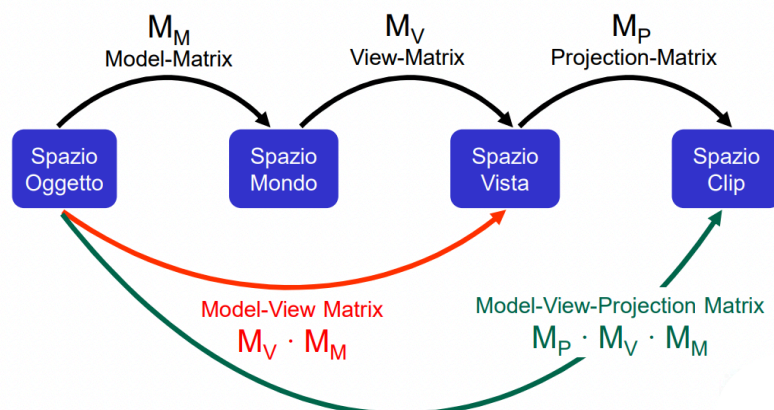


Trasformazione di proiezione = Trasforma lo Spazio Vista in Spazio Clip.

E' determinata da caratteristiche interne della camera come

- le dimensioni del box della camera
- la posizione del «buco di spillo» della pin-hole camera
- soprattutto dalla lunghezza focale (determina se grandangolo, teleobiettivo...)
- Questi sono detti i **parametri INTRINSECHI** della camera

Responsabile anche degli effetti prospettici come oggetti più vicini appaiono più grandi. Cioè: oggetti che hanno la stessa dimensione in spazio mondo, ma sono ad una Z diversa in spazio vista, appaiono più grandi in spazio clip (più grande, quando la Z è più vicina allo 0).



Domande

Quali sono le coordinate omogenee:

- Di un pt che appare in mezzo dallo schermo, in spazio clip? $(0,0,0.5,1)$
- Della posizione del POV, in spazio vista? Il POV è l'origine quindi $(0,0,0,1)$
- Del un punto in alto a destra del monitor, in spazio Clip? $(1,1,0.5,1)$
- Della direzione di vista, in spazio vista? $(0,0,-1,0)$

Date le matrici di Modellazione M, Vista V, e proiezione P, come si trovano le coord omogenee:

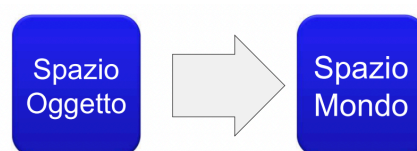
- Dell'origine dello spazio oggetto, in spazio mondo? Con M
- Dell'origine dello spazio mondo, in spazio oggetto? Con M^{-1}
- Della direzione di vista, in spazio oggetto? Con $M^{-1} \cdot V^{-1}$
- Della posizione del POV, in spazio mondo? Con V^{-1}
- Di un punto che appare in mezzo allo schermo, in spazio oggetto? $M \cdot V \cdot P$

Trasformazioni nel rendering: Model

Trasformazione di modellazione:

- Da: Object Frame
- A: World Frame

Detta di modellazione perché serve a modellare il contenuto della scena. Decidendo la posizione / dimensione / etc di ciascuno oggetto che la compone.



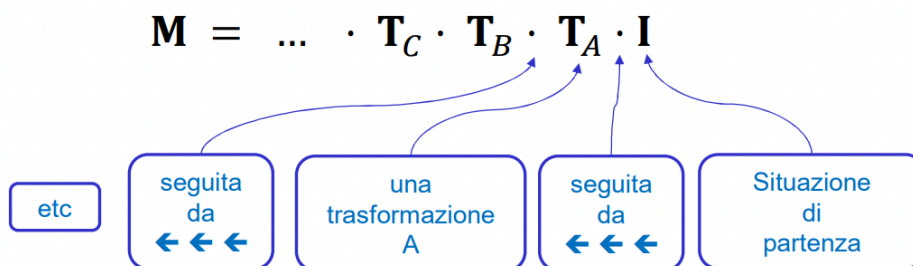
Matrice di Modellazione = Serve a identificare uno spazio univoco per tutta la scena. Ad esempio, il problema di allineare diverse range scan può essere interpretato come il problema di trovare una trasformazione di modellazione per ciascuna di esse, che porta le due superfici rappresentate a coincidere (nei punti di overlap)... nello spazio mondo.

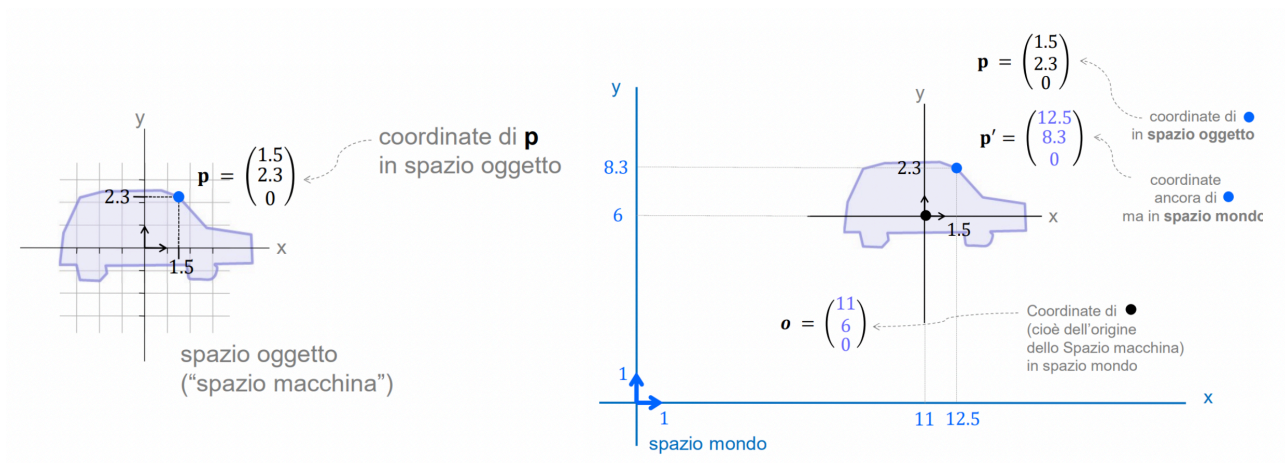
Quando è l'identità, allora lo spazio oggetto coincide con lo spazio mondo. Questo è utile quando un unico oggetto deve essere ispezionato (ad esempio, durante una sessione di sculpting).

Quindi l'oggetto ha la propria origine nell'origine del mondo ed è orientato in modo da avere l'asse delle X (per es, dalla propria sinistra alla propria destra) coincidente con l'asse delle X del mondo (per es, da Ovest a Est), etc.

Come ogni matrice di trasf. spaziale affine, posso vedere (e costruire) la matrice di modellazione **M** in due modi equivalenti:

- Come una descrizione dello **spazio oggetto**: origine e assi, espressi nello **spazio mondo**, costituiscono le colonne di **M**
- Una sequenza di trasformazioni spaziali *A*, *B*, *C* ... (come traslazioni, rotazioni, etc) che modificano la situazione "di partenza" vista sopra



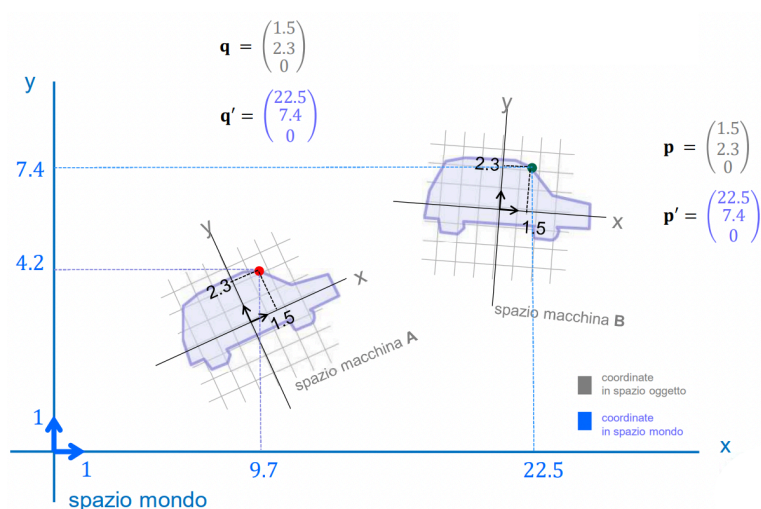


Quale **matrice di modellazione** ha la macchina, nel disegno qui sopra?

$$\begin{pmatrix} 1 & 0 & 0 & 11 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Nota che questa matrice può essere interpretata come

- La descrizione dello spazio macchina (espresso in spazio mondo)
- La trasformazione di traslazione che la macchina subisce per passare dalla posizione originale, nell'origine, alla posizione mostrata



Due istanze nello stesso spazio mondo → nota che i due punti indicati, nelle due macchine, hanno le stesse coordinate p e q in **spazio oggetto** sono lo stesso vertice della mesh!

Ma coordinate diverse p' e q' in **spazio mondo** perché le due istanze di macchina sono associate a due matrici di modellazione diverse.

Trasformazioni nel rendering: View

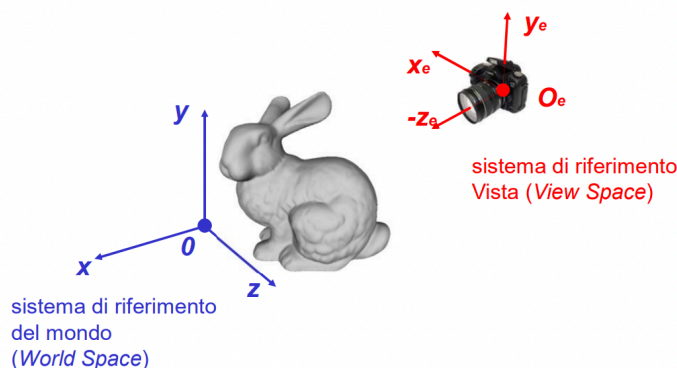
Matrice di vista dipende da **dove e come è orientata la camera** – cioè dai suoi parametri **estrinseci**.



Trasformazione di vista:

- Da: World Frame
- A: View Frame

La Trasformazione di Vista porta tutti gli oggetti da renderizzare in uno spazio di riferimento riferito alla telecamera.



Dipende interamente dai «parametri estrinseci» della macchina fotografica (virtuale) cioè da dove è, e come è orientata (nel mondo) (per es: un tipico task di Computer Vision: «registrare una foto» = evincere i parametri estrinseci della camera al momento del suo scatto).

E' un cambio di sistema di riferimento, cioè una trasformazione affine...Matrice di Vista = la Matrice che fa questa trasformazione.

Un modo per esprimere i parametri estrinseci:

1. Posizione dell'osservatore (POV) → cioè, dell'occhio (la pupilla), della macchina fotografica (il punto di fuoco) etc
2. Posizione di un punto target osservato → si richiede che questo punto compaia in mezzo alla foto oppure, equivalentemente, una direzione di vista
3. Vettore "alto" ("up-vector") → Descrive una direzione che, nella foto, deve apparire come direzione verticale, verso l'alto. Distingue, ad es, una foto "portrait" da una "landscape" (e da un "campo obliquo")

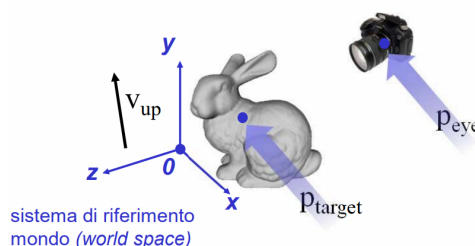
Nota: si tratta di punti e vettori espressi nel Sistema di coordinate Mondo. Descrivono la posizione / orientamento della camera nella scena.

Input:

- 1) **camera position:** P_{eye}
- 2) **target position:** P_{target}
- 3) **vettore di alto:** V_{up}

nb: punti e vettori espressi in spazio mondo!

un esempio di descrizione esaustiva dei parametri estrinseci della camera



Matrice di modellazione M di un oggetto qualsiasi:

- da spazio di quel dato oggetto,
- a spazio mondo

Matrice di Vista V :

- da spazio mondo
- a spazio Vista (cioè lo spazio oggetto... della camera).

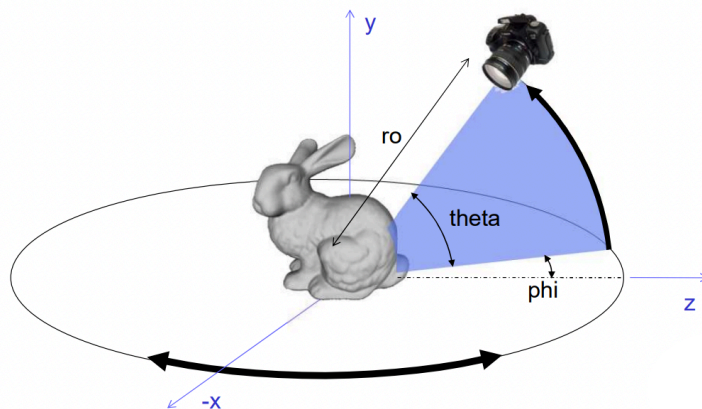
Quindi, se l'oggetto in questione è la camera: V è l'inversa di M .

la trasf. di **modellazione** che localizza un **oggetto** in una certa posizione & orientamento

è l'**inversa**

della trasf. di **vista** necessaria per piazzare la **camera** nella stessa posizione & orientamento

Esempio: una semplice "Trackball":



Trackball = interfaccia usata per consentire all'utente di selezionare in modo semplice i parametri estrinseci e dunque la matrice di vista.

Una semplice trackball: posizione & orientamento della camera controllata da soli tre parametri:

- angoli (ϕ e θ)
- distanza (ro) (es. mappati su assi X Y mouse)
- mousewheel

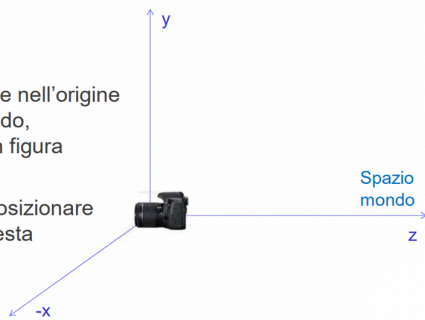
Utile per visualizzare un piccolo oggetto e quindi permettere alla camera di orbitare intorno all'origine.

Passo 0

Se la matrice di vista è l'identità, allora lo spazio vista coincide con lo spazio mondo

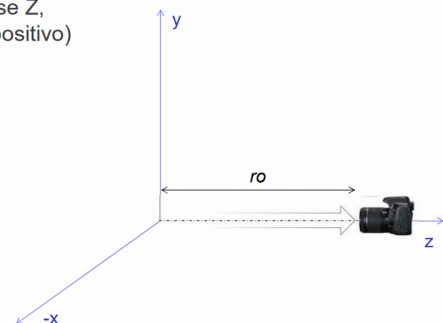
La camera è dunque nell'origine dello spazio mondo, orientata come in figura

Immaginiamo di riposizionare la camera da questa situazione.



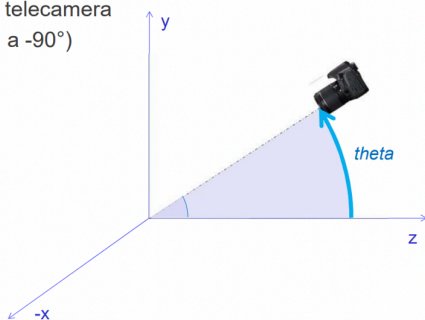
Passo 1

Spostare la macchina all'indietro di ro unità all'indietro (dunque, sull'asse Z, con ro numero positivo)



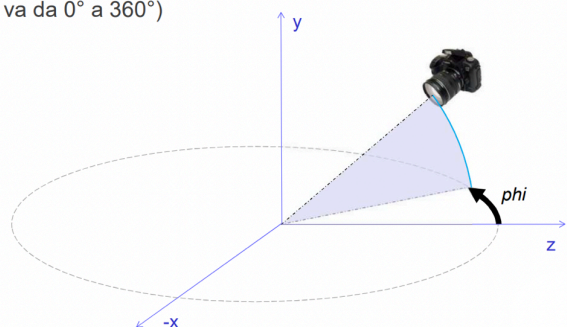
Passo 2

Ruotare di $theta$ gradi attorno all'asse delle X, quindi alzando la telecamera (theta va da $+90^\circ$ a -90°)



Passo 3

Ruotare la camera di phi gradi attorno all'asse delle Y (phi va da 0° a 360°)



Realizzazione della Trackball

✓ Matrice che sposta la telecamera nel posto voluto:

$$R_Y(phi) \cdot R_X(theta) \cdot T(0,0,ro)$$



✓ Questa è la matrice di modellazione dell'oggetto camera
⇒ va da spazio camera a spazio mondo!

✓ La **matrice di vista** è dunque la sua *inversa*, e cioè:

$$T(0,0,-ro) \cdot R_X(-theta) \cdot R_Y(-phi)$$

Le inverse, in ordine inverso

Trasformazioni nel rendering: Projection

in arte, architettura progettazione come riportare oggetti 3D su un piano 2D?

Trasformazione di proiezione ortogonale

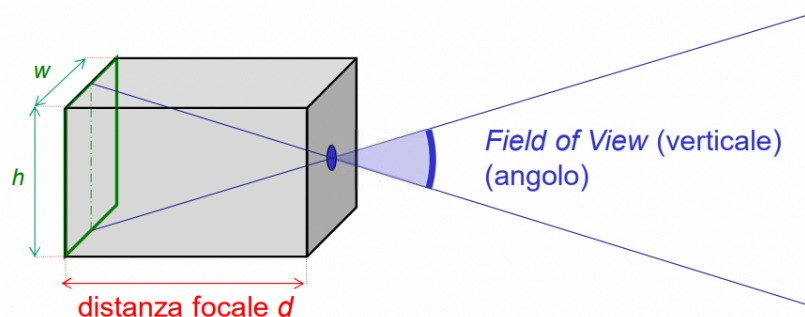
Modo 1: da 3D a 2D? facile: ignoriamo la z (per ora)

matrice corrispondente:

$$P_Z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Ho ottenuto una proiezione **ortogonale** → non c'è alcuna prospettiva: la dimensione (a schermo) degli oggetti non dipende dalla loro distanza dall'osservatore. Le linee parallele (in spazio vista, mondo, oggetto) rimangono parallele anche a schermo. La direzione di vista è costante in ogni pixel.

simula: il punto di vista all'infinito; con un cannocchiale molto potente; lunghezza focale infinita.



Parametri intrinseci della camera: (solo i principali)

- dimensioni image plane (w , h) (in spazio vista)
- distanza focale (d) (in spazio vista) oppure Field of View (FoV)

Si possono ottenere uno dall'altro, sapendo h (come?).

Il valore d è facile da usare, ma FoV è intuitivo da determinare:

- FoV grande >60° (dist foc. piccola): «grandangolo»
- FoV piccolo <45° (dist foc. grande): «teleobiettivo»

Matrice di proiezione prospettica

matrice di trasformazione
per la
proiezione prospettica
(nota: non è affine)

$$\mathbf{P} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$\mathbf{P} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} d \cdot x \\ d \cdot y \\ d \cdot z \\ -z \end{pmatrix}$$

...che esprime il
punto di coordinate
cartesiane...
(divido per w)

$$\begin{pmatrix} -x \cdot d/z \\ -y \cdot d/z \\ -d \end{pmatrix}$$



Proiezione Prospettica: che effetto fa



d piccolo

$$\mathbf{P} = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

d grande

Più distorsione
prospettica.

Effetto "fish-eye"
(grandangolo)

Proporzioni
più mantenute

Effetto "zoom"
(eg. vista dal
satellite)



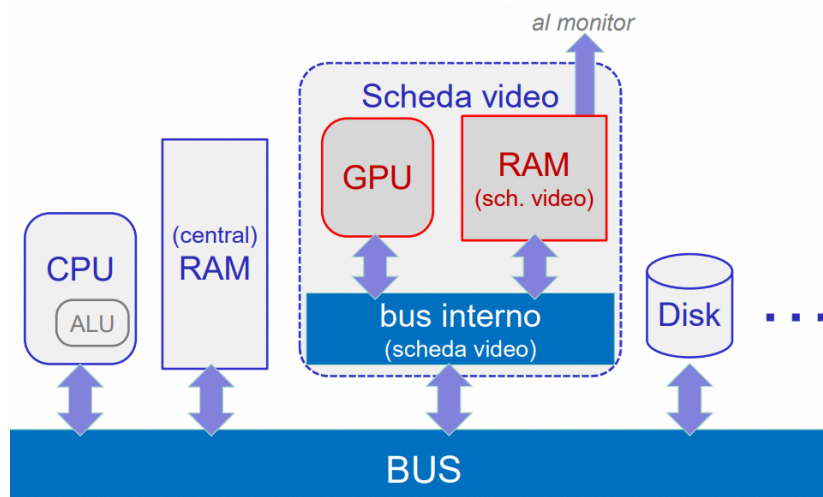
La trasf di proiezione deve portare l'intervallo Z (spazio vista) $[-z_{\text{near}} .. -z_{\text{far}}]$ nell'intervallo Z (spazio clip) $[-1..+1]$.

		Spazio Vista	Spazio Clip	Spazio Schermo	
x	da:	«left clipping plane»	-1	0	pixel coordinates
	a:	«right clipping plane»	+1	res_x	
y	da:	«bottom clipping plane»	-1	0	pixel coordinates
	a:	«top clipping plane»	+1	res_y	
z	da:	$-z_{min}$	-1	0	pixel depth
	a:	$-z_{max}$	+1	1	

Graphic Hardware, Drivers, API e librerie (cenni)

L'hardware specializzato per il rendering è la **GPU** = Graphics Processing Unit:

- è la CPU della scheda video
- Possiede Instruction Set specializzato
- Ha una architettura a pipeline, sfrutta l'alto grado di parallelismo
- Possiede la propria memoria RAM



Vantaggio: **efficienza** →

- instruction set specializzato (computazioni più comuni sono hard-wired)
- computazioni in parallelo:
 1. fra CPU e GPU → rendering demandato alla scheda grafica mentre il resto dell'applicazione è libera di utilizzare la CPU e RAM base.
 2. a volte fra GPU distinte (es. più schede sullo stesso BUS)
 3. a volte fra le fasi del pipeline (vanno tutte in parallelo)
 4. dentro ogni fase del pipeline (più sotto-processi per fase)
 5. instruction level: operazioni operano su vettori di 4 operandi.

Svantaggio: **rigidità** →

- scelta quasi obbligata dell'approccio al rendering utilizzato

Come tutte le architetture pipelined, la velocità (efficienza) della GPU grafica è determinata dallo step più lento, detto "il collo di bottiglia", o **bottleneck**. Le fasi:

- successive vengono "starved", sono spesso inattive in attesa di input
- precedenti vengono "choked", sono spesso inattive in attesa che la fase seguente sia pronta a ricevere i propri output

Efficienza del pipeline di rendering rasterization based. Dove è il collo di bottiglia?

- Nella fase per **vertice**
 - l'applicazione è detta **transform-limited** sinonimo: geometry-limited
 - "C'è troppa geometria, la trasformazione geometrica di tutti questi vertici impiega troppo tempo"
- Nella **rasterizzatore**, o nella fase per frammento
 - applicazione è **fill-limited**
 - "L'applicazione non riesce a riempire (fill) lo screen buffer di pixel a abbastanza in fretta"

Conseguenza: le prestazioni (frame per secondo):

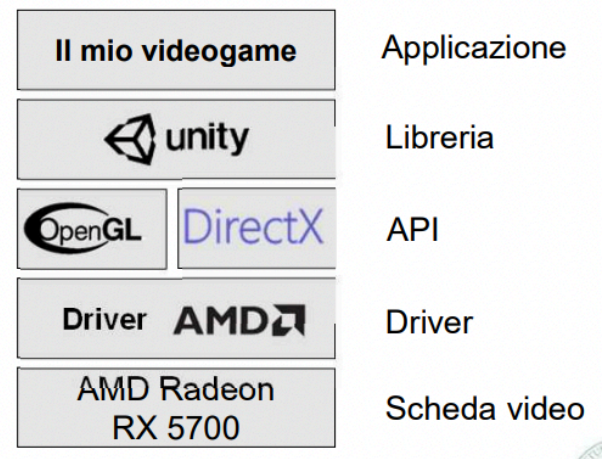
- delle applicazioni geometry limited migliorano (ad esempio) riducendo il numero e/o la risoluzione delle mesh renderizzate (o delle point cloud, etc)

- delle applicazioni fill limited migliorano (ad esempio) riducendo la risoluzione dello schermo

Api grafiche diffuse: Direct3D, Vulkan, Metal

Livelli di astrazione:

- Applicazione (es: videogame) - Librerie/SDK (es: unity)
- API (es: openGL/DirectX)
- Driver (es: AMD)
- Scheda video / grafica
- Monitor



Rasterizing primitives

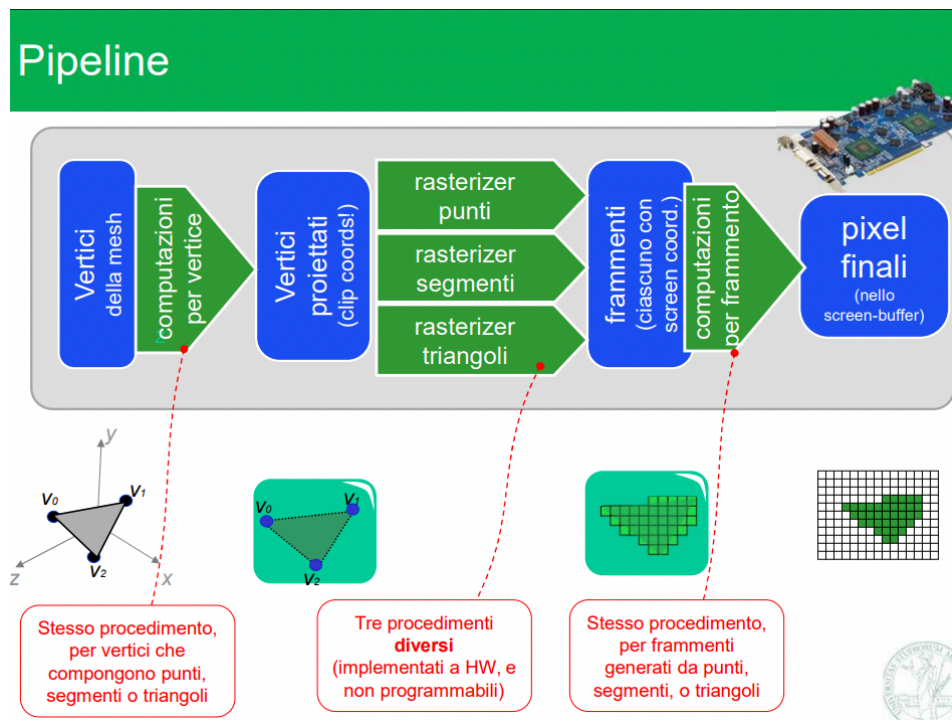
Rasterizzazione = Individuare i **frammenti** che compongono una primitiva, uno per ogni pixel coperto dalla primitiva.

I frammenti prodotti vengono passati alla fase successiva (che determina il colore schermi). E' un procedimento puramente 2D, estremamente parallelizzabile. Per sfruttare questo, si adotta un hardware parallelo. Sulle GPU, è implementato in hardware (architettura specializzata per lo scopo). E' una fase dell'hardware molto efficiente e poco flessibile (non è programmabile dall'utente).

Gli hardware GPU attuali sono progettati per rasterizzare solo tre tipi di primitive:

- Triangoli 2D – 3 vertici
- Segmenti 1D («linee») – 2 vertici
- Punti 0D («point splats») – 1 vertice

Il primo caso è particolarmente utile ed ottimizzato. I vertici sono passati al rasterizzatore in coordinate clip omogenee. Per prima cosa, vengono convertite in coordinate schermo cartesiane. Il rasterizzatore usa solo le x e la y (in spazio schermo, cioè le coordinate pixel), la z (in spazio schermo, cioè la depth del pixel) viene ignorata.



Pipeline della rasterizzazione:

1. Vertici (punti in R^3) → (computazioni per vertice)
2. Vertici proiettati (punti in R^2) → (RASTERIZZAZIONE)
3. Frammenti (candidati pixel) → (computazioni per frammento)
4. Screen buffer

Rasterizzare triangoli → Individuare i frammenti che compongono i triangoli

- Input: 3 vertici (punti 2D in screen space)
- Output: i frammenti interni al triangolo

Ogni frammento ha coordinate pixel intere. Devono essere prodotti tutti e soli i frammenti il cui centro è interno al triangolo.

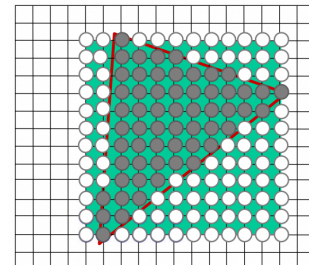
Il rasterizzatore si occupa anche di **interpolare gli attributi** in ogni frammento prodotto

- Input ulteriore: un set di attributi per ogni vertice (qualsiasi insieme di vettori, scalari...)
- Output ulteriore: un set di valori interpolati, per ogni frammento prodotto

Come sappiamo, l'output è dato dalla interpolazione degli input avente come pesi le coordinate baricentriche del frammento nel triangolo renderizzato. Gli attributi interpolati sono passati al processing per frammento.

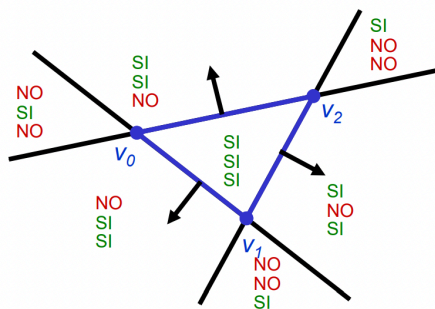
Un algoritmo di rasterizzazione per triangoli parallelizzato

1. Trovo un rettangolo (di coordinate intere) che contiene il triangolo
2. Processo tutti le posizioni interne (nota: questo è parallelizzabile)
3. Per ogni posizione interna: se è dentro al triangolo, allora produco un frammento



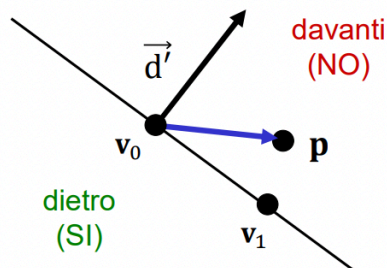
Test di appartenenza ad un triangolo →

Triangolo = intersezione di 3 semipiani, un punto è interno al triangolo sse appartiene a tutti e tre i semipiani



Test di appartenenza ad un semipiano →

basta verificare se $(p-v_0) \cdot \vec{d}' < 0$



Da quale parte della retta passante per v_0 e v_1 si trova il punto p ?

Soluzione:

$$\vec{d} = (v_1 - v_0) = \begin{pmatrix} d_x \\ d_y \end{pmatrix}$$

$$\vec{d}' = \begin{pmatrix} -d_y \\ d_x \end{pmatrix}$$

il vettore $\vec{d}'$ è $\vec{d}$ ruotato di 90°

la cosiddetta *Edge function* dell'edge da v_0 a v_1

Basta verificare se:

$$(q - p_0) \cdot \vec{d}' < 0$$

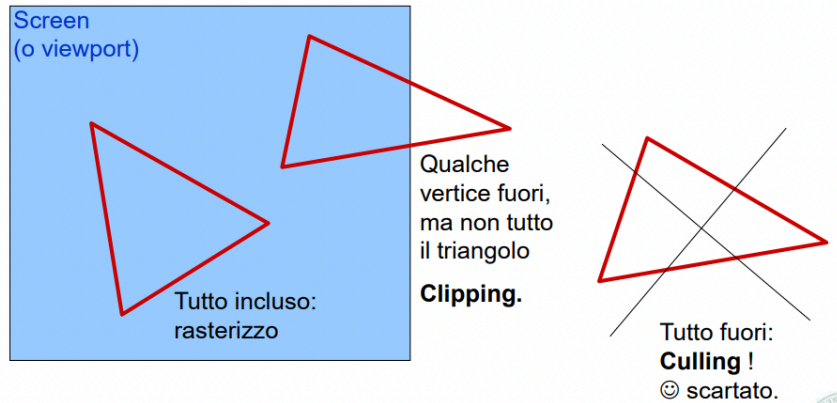


Dettaglio: cosa succede se il frammento è “proprio sulla **linea**”? cioè se la edge function fa proprio 0?

Ci sono molte possibili soluzioni, ma una regola deve sempre valere (è compito dell’implementazione hardware farla valere): un frammento su una linea “a cavallo” fra due triangoli deve essere rasterizzato da esattamente **uno dei due**,

- non entrambi, vanno evitati gli “overdraw”
- non nessuno dei due: perché vanno evitati i gap (buchi) fra i due

Cioè: se il frammento è scartato dall’edge-test dell’edge “da V0 a V1” allora NON deve essere scartato dall’edge test del edge in senso opposto “da V1 a V0”, e viceversa.



Clipping = prendendo uno screen/viewport e un triangolo, qualche suo vertice rimane fuori dallo schermo, ma non tutto il triangolo. Come mi comporto col triangolo tagliato?

- Soluzione → Come si interseca un bounding box con lo screen?
 - trovo un **bounding Box**,
 - interseco bounding Box con il viewport
 - rasterizzo nel bounding box come normale.

Vantaggi:

fortemente parallelizzabile, clipping diventa facile, adatta alla implementazione HW.

Svantaggi:

overhead grandi, si testano molti frammenti inutilmente

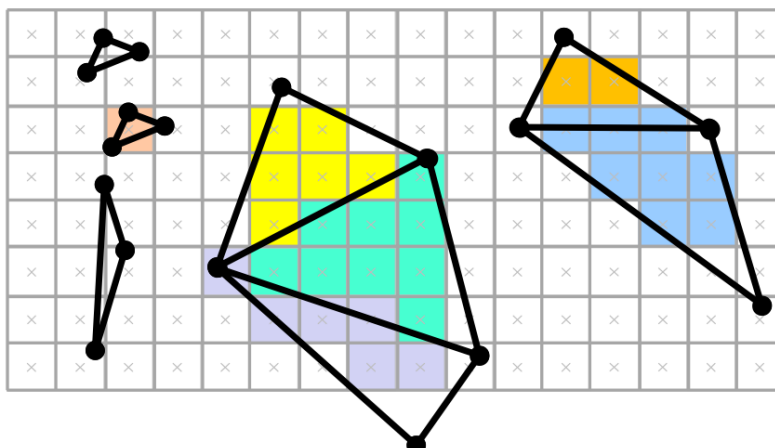
Caso sfavorevole: i triangoli lunghi e stretti, orientati lungo la direzione diagonale dello schermo (in spazio clip) necessitano il testing di ogni triangolo sull'intero schermo

Culling = tutto il triangolo rimane al di fuori dello screen (niente raster da fare).

Test che compie il rasterizzatore:

- Il centro del pixel ricade dentro al triangolo 2D?
- considerare il triangolo come intersezione di 3 semipiani

La rasterizzazione di 2 triangoli che condividono 1 edge genera 1 frammento per ogni pixel nella loro unione → non avremo sovrapposizioni di frammenti o gaps



Triangoli della stessa forma ma posizioni diverse possono generare insieme di frammenti diversi. Un triangolo può generare qualsiasi numero di frammenti, anche nessuno o anche tutti quelli sullo schermo.

Se un triangolo è parzialmente fuori al viewport, vengono generati solo i frammenti interni al viewport (clipping).

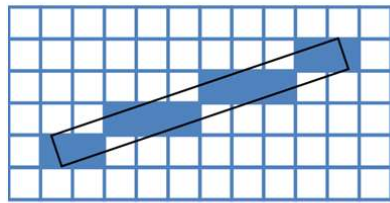
Rasterizzazione Linee →

utilizzata nel rendering a wireframe. Algoritmo di **Bresenham** (non più utilizzato):

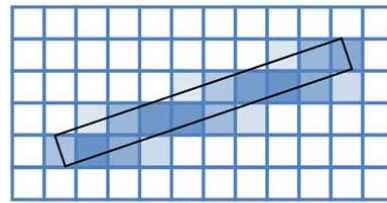
- Vantaggi: richiede solo computazioni fra interi, è efficiente.
- Svantaggi: sequenziale (non parallelizzabile), l'iterazione del ciclo dipende dalla precedente.

Per rasterizzare linee si può anche fare uso di point splat.

Senza:



Con:



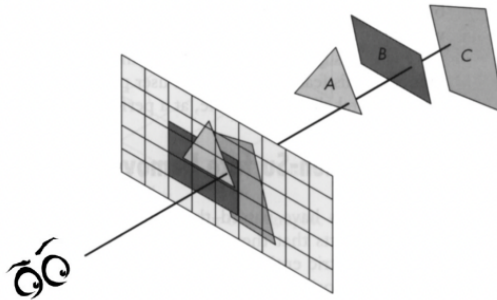
Qua sopra si può veder un esempio di rasterizzazione di una linea senza e con **anti-aliasing**.

Rasterizzazione triangoli: SINTESI→

- Bisogna individuare i frammenti che compongono i triangoli.
- Presente anche l'interpolazione degli attributi (dati in input 3 set di attributi ricevo in output l'interpolazione di ciascun attributo per ogni frammento prodotto).
- I vertici sono passati al rasterizzatore in coord. clip omogenee.
- Avviene in 2D (input tre punti in 2D, output frammenti). Viene generato un frammento per ogni pixel il cui **centro** ricade dentro il triangolo 2D.

Rimozione di superfici nascoste

Gli oggetti più vicini devono coprire quelli più lontani (ovvero gli oggetti lontani sono “occlusi” da quelli più vicini).



Negli approcci basati su ray-tracing, questo era banale da ottenere: basta prendere, per ogni raggio primario, la primitiva intersecata avente la distanza k minore.

Negli approcci basati sulla rasterizzazione, non è banale. Se i pixel prodotti dalla rasterizzazione di una primitiva sovrascrivono sullo screen buffer quelli generati alla stessa posizione dalle primitive precedenti allora il valore finale sullo screen buffer sarà semplicemente il valore dei pixel rasterizzati dall'ultima primitiva rasterizzata.

basata sull'orientamento

le primitive rasterizzate sovrascrivono nel frame buffer quelle rasterizzate in precedenza. Quindi basta rasterizzare le primitive nell'ordine giusto (“back to front”).

Esempio: **Algoritmo del pittore (depth sorting)**: utilizzato quando è facile ordinare preventivamente le primitive back-to-front (cioè in ordine crescente di Z nello spazio vista) in fase di pre-processing. E' usato raramente perché implementato dalla CPU.

Problemi:

- ordinamento costa $O(n \cdot \log(n))$ con n =numero primitive (troppo costoso)
- quando effettuare l'ordinamento? le coordinate in spazio di vista sono note solo dopo la trasformazione
- Non esiste sempre un ordine corretto (si possono creare cicli o intersezioni)
- una primitiva non ha un'unica Z (ogni vertice ha una Z diversa), quale usiamo?
- si complica il rendering usando un ordine prefissato.

Back-face Culling

il **back-face culling** è una tecnica (**order independent**) che determina se uno dei poligoni dai quali è composto un oggetto solido è visibile dall'angolo di visuale dell'inquadratura scelta (si applica a un caso particolare per mesh chiuse e ben orientate).

Ipotesi: superficie **chiusa opaca e ben orientata**, non vedrò mai l'interno, qualunque faccia si veda da dietro (back-facing triangles) finirà per forza occlusa dalla vista da (almeno) un layer di facce “viste da davanti” (front-facing triangles).

Se la camera è dentro la mesh → scartare le facce front-facing

Se la camera è fuori dalla mesh → scartare le facce back-facing

Il facing (front o back) di una faccia è il **segno del prodotto dot** fra **normale** della faccia e **direzionale** di vista V (V = direzione VERSO l'osservatore, cioè $V = (0,0,1)$ in spazio Vista). Se prodotto dot negativo = front facing (assumendo normali verso esterno).

Equivalentemente, il facing può essere determinato dall'**orientamento** orario oppure antiorario dei sui **vertici** in spazio clip (oppure schermo). Dipende da come è modellata la mesh ma per convenzione:

- Antiorario → front-facing
- Orario → back facing

Questo può essere determinato nello stesso test di appartenenza del frammento al triangolo:

- Se il frammento supera tutti e tre gli edge test: frammento interno a triangolo anti-orario
- Se il frammento non supera nessuno dei tre edge test: frammento interno a triangolo orario (scartare, se il back-face culling è attivato)
- Ogni altro caso: frammento esterno (scartare in ogni caso)

Il processo di back-face culling avviene nella **fase di setup che precede la rasterizzazione** di ciascun triangolo.

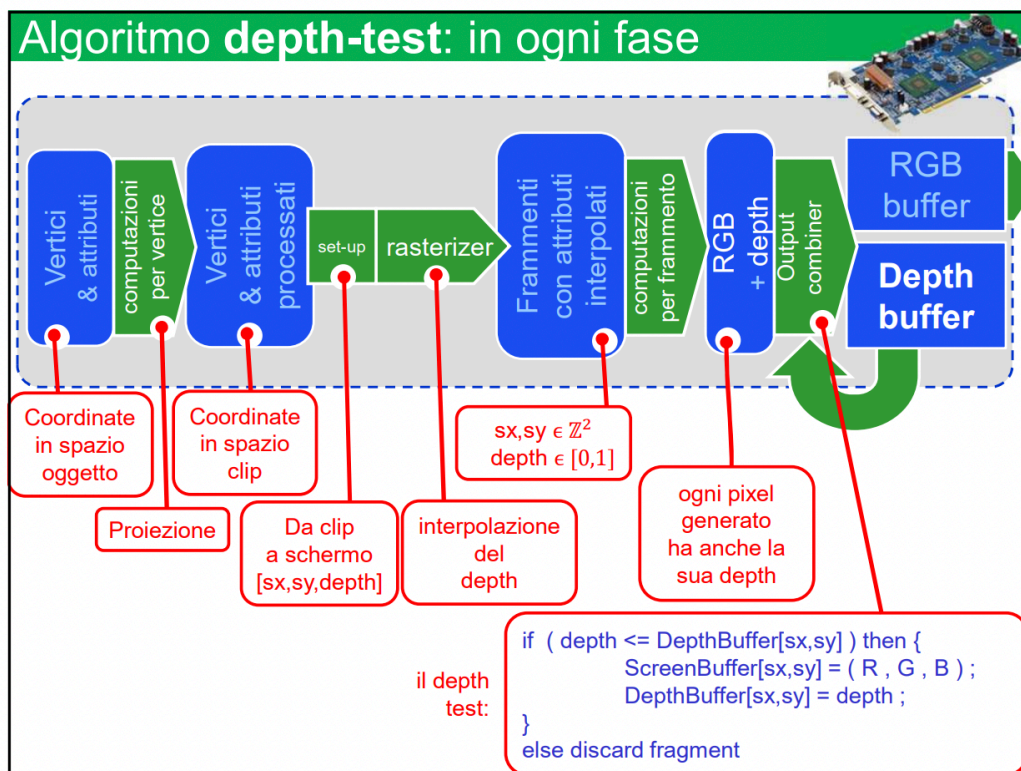
Quando usare il backface culling:

- Se valgono le ipotesi dette, allora il back-face culling non cambia le immagini generate
- Quando è possibile usarlo, è molto conveniente perché migliora le prestazioni delle applicazioni fill-limited, riducendo di circa il 50% il numero di frammenti prodotti
- Non impatta però le applicazioni geometry-limited, dato che lo scarto della primitiva avviene a valle del processing per vertice
- contribuisce all'hidden-surface removal, dato che rimuove solo facce auto-occluse
- Non è però sufficiente, nel caso di mesh concave, perché non rimuove tutte le facce auto-occluse
- E' un test eseguito dall'HW della GPU (non viene implementato)

Il back-face culling non basta a rimuovere tutte le superfici nascoste.

Algoritmo Depth test e Depth Buffer

Idea base: di tutti i frammenti che insistono su un pixel tengo solo quello di profondità minore. Serve di conseguenza una struttura per memorizzare la profondità attuale di ogni pixel.



Depth buffer = (o z-Buffer) un buffer 2D, con la stessa risoluzione dello screen buffer (è l'output finale del rendering, mostrato a schermo), che per ogni posizione [x,y] memorizza la depth del frammento attualmente presenta in quella locazione.

Valore di depth (profondità) di un frammento è...

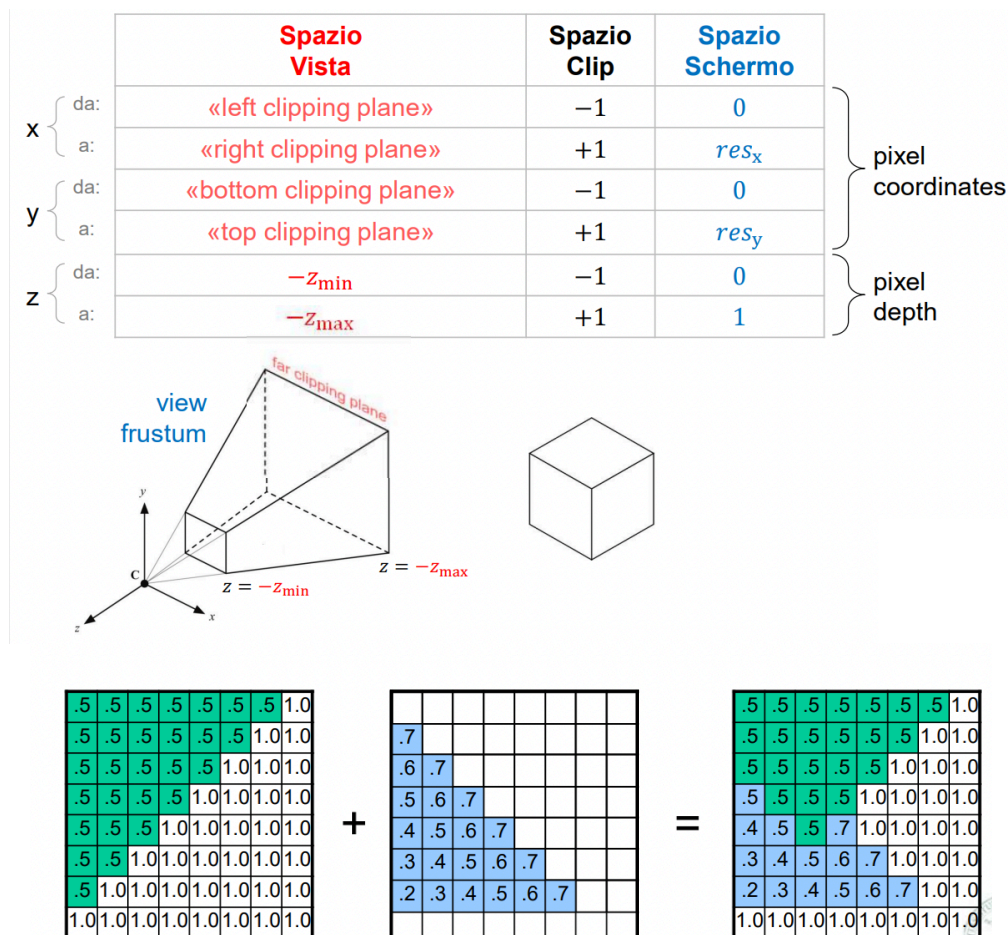
... un valore da 0 a 1:

- 0 → pixel più vicino possibile
- 1 → pixel più lontano possibile

...determinato (per vertice) a partire dalla Z dello **spazio clip**

- Se $Z = -1 \rightarrow \text{depth} = 0$
- Se $Z = +1 \rightarrow \text{depth} = 1$
- $\text{depth} = (Z+1) / 2$

...interpolato, dentro ogni primitiva, per ottenere il valore dei frammenti. Come qualsiasi altro valore attributo definito sui vertici, attraverso coordinate baricentriche.



Questo **algoritmo depth-test** funziona anche in situazioni in cui sono presenti intersezioni e cicli. Il rendering diventa **order independent** → se disegno prima la primitiva davanti, i frammenti di quella dietro verranno scartati; viceversa i frammenti della primitiva davanti sovrascriveranno i frammenti della primitiva dietro.

Problema dello **Z-fighting** = con superfici parallele con la stessa Z o Z molto vicine, abbiamo problemi con la visualizzazione della superficie. Non viene mai completamente evitato.

Limiti dell'algoritmo:

- costa un po' di memoria (GPU)
- problemi di aliasing sulla z

- frammenti vengono scartati solo alla fine della pipeline (dopo aver effettuato inutilmente tutta la computazione)
- assume superfici opache (problemi con superfici opache)
- dati condivisi in lettura e scrittura (problemi per chi implementa HW)
- precisione

Effetto collaterale dell'algoritmo di depth test → dopo il rendering, ho prodotto non solo un'**immagine** (RGB per pixel) ma anche un **depth-buffer** (profondità per pixel)

Una **depth image** con un «bassorilievo» della scena renderizzata → si tratta di una **range scan**.

Il color-buffer (RGB) viene mandato a schermo, il depth-buffer è solo di uso interno e normalmente viene scartato ma alcuni algoritmi di rendering lo sfruttano invece per gli scopi più vari.

Il vantaggio del depth test è che il rendering è reso **order independent**:

- se disegno prima la primitiva davanti: i frammenti di quella dietro verranno **scartati**
 - se disegno prima la primitiva dietro: i frammenti della primitiva davanti li **sovrascriveranno**
- il risultato finale: lo stesso! (eccetto i rarissimi casi di pareggio). L'efficienza può essere però diversa: scartare è più efficiente di scrivere-per-poi sovrascrivere. Quindi, ordine ottimale è **front-to-back**.

Depth test: considerazioni pratiche sull'efficienza

- scartare è più efficiente di scrivere-per-poi-sovrascrivere, quindi: meglio disegnare prima le cose davanti poi quelle dietro: rendering front-to-back
- l'impatto è accentuato dalle ottimizzazioni HW, quindi: il depth-sorting è ancora utile, come ottimizzazione (non è necessario, il risultato è sempre corretto)

Il depth test è hard-wired affinché sua implementazione sia ottimizzata in GPU, quindi nell'output combiner, non nel fragment shader programmabile! Necessario, perché il depth-test richiede lettura + scrittura in memoria condivisa (per il Depth Buffer): problema che impatta l'efficienza e il parallelismo. Molte ulteriori ottimizzazioni sono fatte in HW (silenziosamente) per consentire di anticipare il test e di testare gruppi di frammenti alla volta.

Per completezza ci sono altri per-fragment tests effettuati nell'output combiner:

- Scissor test → scarta i frammenti esterni ad un dato rettangolo
- Stencil test → scarta i frammenti usando una maschera arbitraria

Participating media - fog

L'effetto reale che vogliamo simulare è il **participating medium**.

participating medium = il mezzo attraverso il quale viaggia la luce influenza la luce stessa, assorbendone una parte (su alcune frequenze). Maggiore è il percorso fino alla camera, più forte è l'effetto. E' facile simulare questo effetto anche nel rendering basato su rasterizzazione.

Idea base: tanto più un punto è lontano dall'occhio più il suo colore è impattato da nebbia, foschia, etc.

La Z del vertice in spazio vista determina il **fattore di nebbia**.

Dato un punto, in spazio vista, di coordinata z :

$$fog = clamp \left(\frac{-end - z}{-end + start}, 0, 1 \right)$$

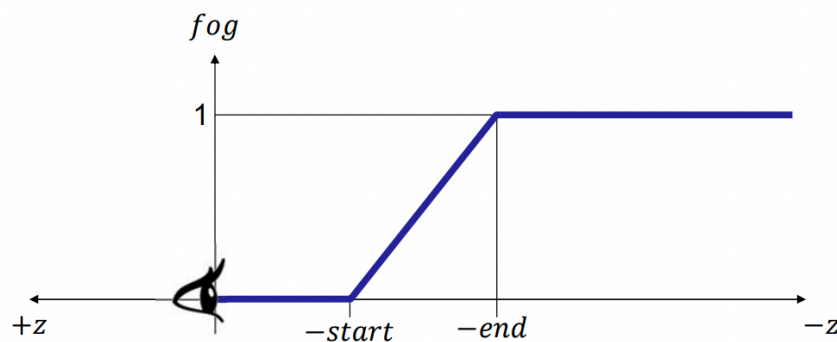
opacità finale della
nebbia
0=niente nebbia
1=si vede solo nebbia

funzione
che porta a 0 se <0
e a 1 se >1

Calcolo del coefficiente della nebbia

start e *end* sono parametri:

- *start*: la distanza (positiva) dall'osservatore prima della quale non si ha alcuna nebbia
- *end*: la distanza (positive) dall'osservatore, dopo la quale si ha 100% nebbia



Spazio vista

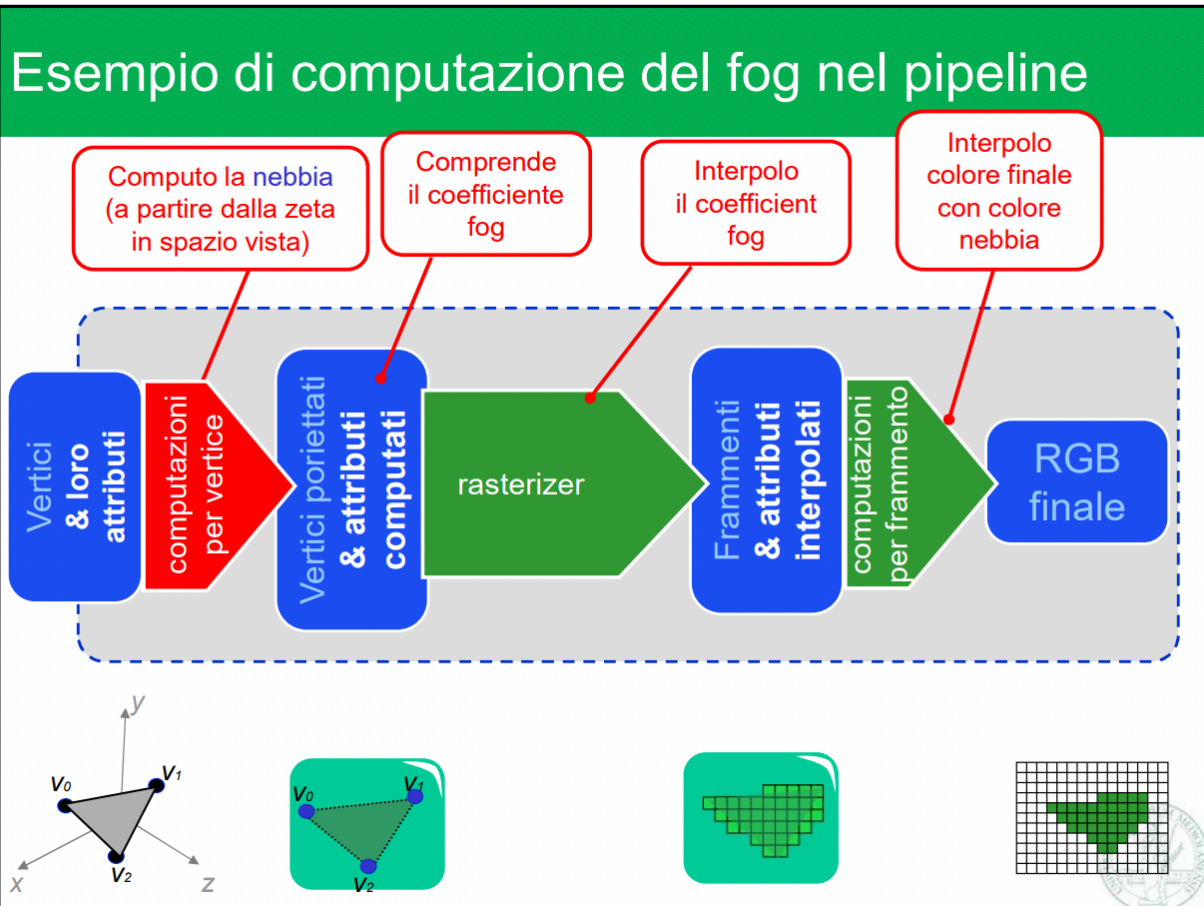
Dopo aver calcolato l'opacità, la applico al **colore** dei frammenti. Il colore finale e' una interpolazione fra il colore nebbia e il colore originale.

$$colore_{finale} = fog \cdot colore_{nebbia} + (1 - fog) \cdot colore_{originale_frammento}$$

il colore
nebbia
(è un altro
parametro)

a parole:

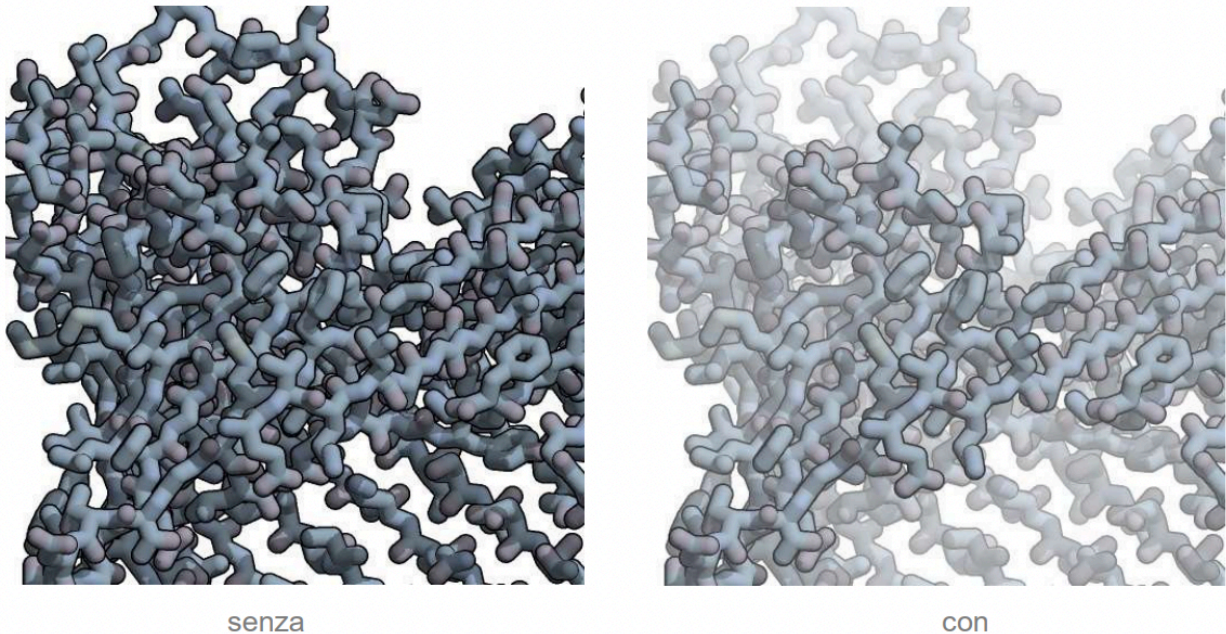
"il colore finale e' una interpolazione
fra il colore nebbia e il colore originale"
(l'ennesima l'uso di interpolazione
che incontriamo)



Usi dell'effetto fog in CG:

1. Simulare la presenza di participating media come nebbia, foschia, acqua sporca (rendering subaquei) etc;
2. Mascherare gli artefatti di popping dovuto al clipping contro il far-clipping-plane.
 - Senza: le cose «spariscono» di colpo quando attraversano il far-clipping plane
 - Con: le cose hanno un fade out, e si dissolvono nella nebbia con continuità, diventando color-cielo man mano che si avvicinano al far clipping plane. Per ottenere questo effetto (Quando spariscono):
 - (a) il colore del fog deve coincidere con il colore sfondo
 - (b) la distanza far del fog deve coincidere con la distanza far della matrice di proiezione
 - Usato nei videogames (che spesso hanno bisogno di far clipping plane più vicini di quanto si desidera, per motivi di efficienza)
 - Se color fog (e sfondo) = nero: effetto oscurità (finto)
3. Suggestire visivamente la profondità degli oggetti:
 - Nel contesto della visualizzazione scientifica (e altri) il fog consente di leggere la profondità relativa nei rendering 3D, e produrre immagini intuitive e facili da interpretare. Particolarmente utile in assenza di altri indizi come la prospettiva (per es, usando proiezione ortografica).

Particolarmente utile in presenza di forme 3D complesse e non intuitive. Il fog usato per questo scopo è detto **depth cueing** («dar spunti di profondità»).



Esempio di depth cueing nella visualizzazione molecolare

Il compute del fog è implementato dal **programmatore** a livello dell'app grafica. Non è una funzionalità prevista dalla GPU o dalle API. Nessun particolare supporto HW è necessario, a differenza di come abbiamo visto per depth-test o back-face culling.

Nelle API a basso livello (come OpenGL o WebGL) l'effetto fog è quindi implementato programmandolo nel:

- **vertex shader** (il programma utente eseguito su ogni vertice)
- **fragment shader** (il programma utente eseguito su ogni frammento)

Nelle librerie ad alto livello, come three.js, l'effetto fog è implementato a livello di libreria, e va solo abilitato.

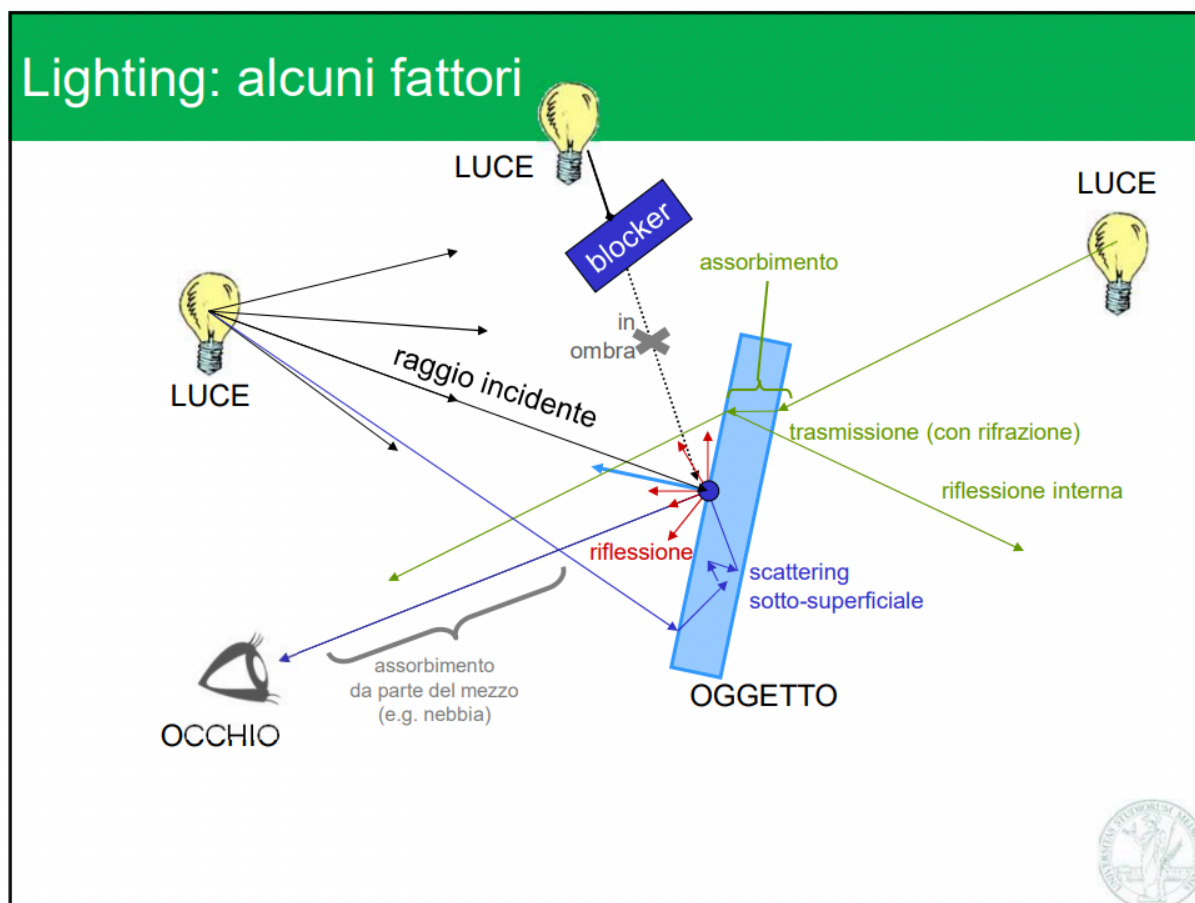
Illuminazione

L'altra metà del rendering è quella di determinare la luce (quanta e di che colore) che arriva (da un punto alla scena, all'occhio/la camera)

Input:

- caratteristiche dell'oggetto illuminato ("materiale"),
- caratteristiche della luce che illumina
- forma degli oggetti/della scena (ovvero l'orientamento delle superfici)

Output: colore percepito per un dato punto.



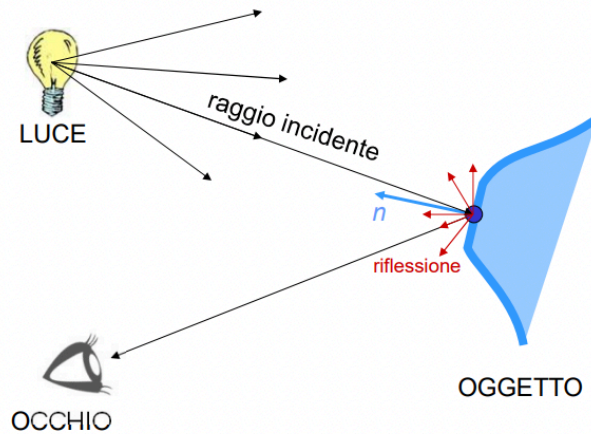
Illuminazione locale (torna molto più facile da fare con il nostro HW) tiene conto solo di:

- Luce → condizioni di luce (n. luci, loro pos, loro colore)
- Oggetto → pezzetto della superficie da illuminare (orientamento (la normale), caratteristiche del materiale per es: colore, lucentezza)
- Occhio → Posizione osservatore (POV)

Illuminazione globale:

- riflessioni multiple
- ombre
- scattering sottosuperficiale
- rifrazione
- Etc..

Lighting locale: emitter → oggetto → occhio (e nient'altro!)



materiale = in CG è una modellazione che descrive in che modo una data superficie reagisce alla luce e comprende cose come:

- Quanto è chiaro oppure scuro
- Di che colore si presenta
- Quanto è lucido oppure opaco (privo di riflessi lucidi, o highlight)
- Quanto è trasparente, semitrasparente, traslucido, o permeabile alla luce
- Se e quanto il materiale è “cangiante”
- Se è fosforescente o emette luce propria (oltre a riflettere quella di cui è investito)

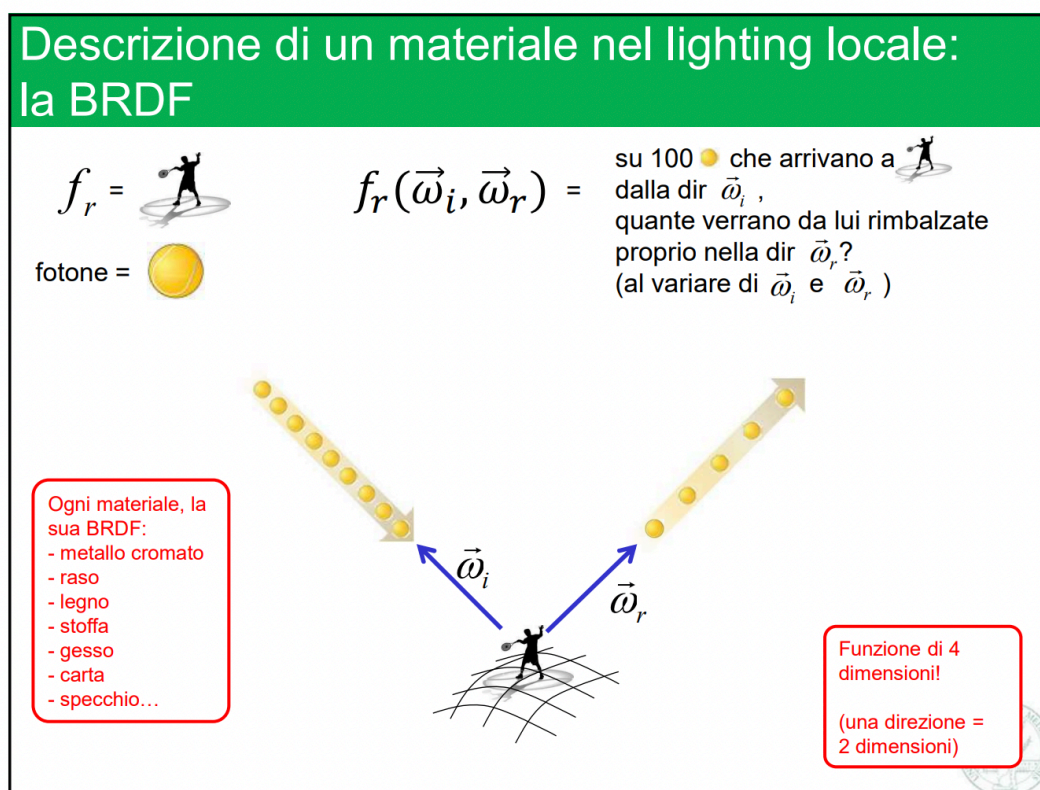
Per quello che riguarda il lighting locale, il materiale è esaustivamente descritto da una funzione che descrive in che modo la superficie riflette la luce che la raggiunge direttamente, cioè quanta della luce che investe un pezzetto di superficie viene riflessa dalla superficie in ogni possibile direzione. Tecnicamente, questa funzione prende in input:

- Una possibile direzione di **provenienza** della luce $\vec{\omega}_i$ (vettore unitario)
- Una possibile direzione di **uscita** della luce $\vec{\omega}_r$ (vettore unitario)

e restituisce in output:

- Quanta della luce proveniente da $\vec{\omega}_i$ verrà mandata in $\vec{\omega}_r$.

Questa funzione viene detta **Bidirectional Radiance Distribution Function**



Questa funzione viene detta BRDF:

- Bidirectional (perché prende due direzioni)
- Reflectance (riflettanza, abilità di riflettere la luce)
- Distribution (perché è una distribuzione di riflettanza)
- Function

La BRDF descrive in modo esaustivo un materiale (per quello che riguarda il lighting locale), ogni materiale è caratterizzato da BRDF diverse.

Per ora occupiamoci della più semplice classe delle BRDF: quelle **costanti**, dette puramente diffuse, o **Lambertiane**. I materiali che esibiscono questa BRDF sono detti diffusivi o Lambertiani.

Esempio di materiali che si avvicinano a questo: gesso, carta ruvida (in realtà è sempre solo un'approssimazione). Un materiale lambertiano riflette la stessa percentuale di luce in tutte le direzioni possibili, indipendentemente da quale sia la direzione di arrivo della luce. Quindi è privo di riflessi, esibisce lo stesso colore visto da tutti gli angoli.

Il lighting è view-independent: non dipende dalla direzione di vista. Dobbiamo occuparci di rispondere ad alcune domande:

- Quanto è questa percentuale di luce costante? (dipende dal materiale)
- Quanta luce raggiunge il punto sulla superficie? (dipende dalle luci)

Materiale lambertiano = (detto anche puramente diffusivo) è definito un materiale in cui la BRDF è **costante**

$$f_r(\vec{\omega}_i, \vec{\omega}_r) = \text{cost}$$

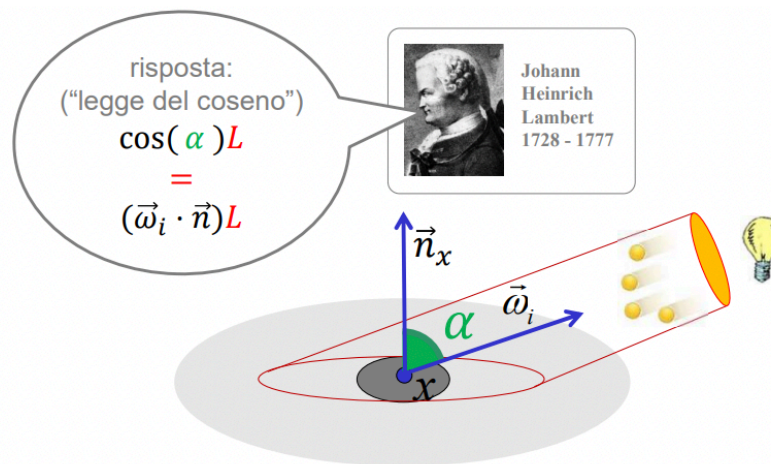
È anche materiale (puramente) diffusivo → Una superficie di questo materiale riflette la luce da cui è raggiunta diffondendola **uniformemente**.

La costante è detta l'**albedo (del materiale)** il rapporto fra la luce incidente e la luce riflessa va:

- Da 0, materiale perfettamente nero, **assorbe** tutta la luce
- A 1, materiale perfettamente bianco, **diffonde** tutta la luce

I materiali lambertiano (o diffusivi) reali si verificano quando a livello microscopico, la superficie presenta micro-sfaccettature disposte in modo molto **irregolare**, caotico.

Sottoproblema : Se dalla direzione di **provenienza** della luce $\vec{\omega}_i$ arrivano L lumens, quante ne riceve un intorno di x con normale $\vec{n}_x$?



Caso particolari:

- luce perfettamente radente: $\vec{\omega}_i \cdot \vec{n}_x = 0$ infatti nessun fotone colpisce l'intorno
- luce da dietro: $\vec{\omega}_i \cdot \vec{n}_x < 0$ allora la risposta è 0 (non certo un numero negativo) perché l'intorno si trova nella sua stessa ombra)

Equazione di lighting per materiali lambertiani (o puramente diffusivi)

✓ L'equazione di lighting risultante è

$$P = (\vec{n}_x \cdot \vec{\omega}_i) D L$$

Prodotto dot, ma zero se negativo
 Luminosità finale del pixel
 normale della sup
 direzione della luce (vettore unitario che va verso la luce)
 Intensità della luce
 albedo del materiale (quanto è chiaro o scuro il suo aspetto)

Anche se il modello di materiale diffusivo (o di Lambert) è basato sulla fisica reale, possiamo anche darne una interpretazione geometrica intuitiva: Il fattore $(\vec{n}_x \cdot \vec{\omega}_i)$, il coseno dell'angolo fra i due vettori, è anche una misura della **similarità** fra la direzione $\vec{n}_x$ normale alla superficie e la direzione $\vec{\omega}_i$ «verso la luce».

Quindi, la legge del coseno di Lambert dice: «tanto più la superficie è orientata verso la luce (cioè, tanto più la sua normale è simile alla direzione di luce), maggiormente chiara ci apparirà.»

L'equazione di lighting vista (che è basata sui fenomeni fisici ben compresi) sarebbe utilizzabile per immagini in bianco e nero. Per passare ad immagini a **colori**, un modo semplice è applicare la stessa equazione separatamente sui canali rosso, verde, e blu (è una approssimazione brutale, ma molto utilizzata).

Lighting... a colori: una riscrittura

$$\begin{pmatrix} P_R \\ P_G \\ P_B \end{pmatrix} = (\vec{n}_x \cdot \vec{\omega}_i) \begin{pmatrix} D_R \\ D_G \\ D_B \end{pmatrix} \otimes \begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix}$$

Scalatura del vettore RGB (nessun simbolo)
 pixel finale
 Diffuse color o Lambertian color o Base color.
 In pratica, la risposta alla domanda "di che colore è l'oggetto" (nel punto x) (a prescindere dalla luce usata per illuminarlo)
 Qui usiamo questo simbolo per donotare il "prodotto componente per componente"
 Intensità / colore della luce (luci potenzialmente colorata)

L'equazione di lighting vista è molto semplice, consiste esclusivamente di una componente di riflessione diffusiva, però è **physically based** (il fenomeno ottico modellato è accuratamente riprodotto, eccetto che per la tricromia RGB). E' un esempio di modello di **illuminazione locale**. In questo modello:

- il «materiale» è descritto unicamente dal **base color** (detto anche diffuse color -- o albedo se in bianco e nero)
- L'unico aspetto geometrico che è rilevante è la normale del punto illuminato. In particolare, la direzione di osservazione non conta.
- Per questo, il modello è un modello «view independent».
- Il chiaroscuro non dipende dalla direzione di osservatore: non abbiamo **riflessi** (infatti materiali come gesso ne sono privi)

Lighting con più sorgenti di luce

- ✓ Abbiamo supposto di avere una sola sorgente di luce
⇒ di direzione $\vec{\omega}$ e intensità (L_R, L_G, L_B)
- ✓ Aumentare il numero di luci è molto semplice:
basta calcolare e sommare i contributi di ciascuna luce
- ✓ Esempio con due luci

$$\begin{pmatrix} P_R \\ P_G \\ P_B \end{pmatrix} = (\vec{n}_x \cdot \vec{\omega}_1) \begin{pmatrix} D_R \\ D_G \\ D_B \end{pmatrix} \otimes \begin{pmatrix} L_{1,R} \\ L_{1,G} \\ L_{1,B} \end{pmatrix} + (\vec{n}_x \cdot \vec{\omega}_2) \begin{pmatrix} D_R \\ D_G \\ D_B \end{pmatrix} \otimes \begin{pmatrix} L_{2,R} \\ L_{2,G} \\ L_{2,B} \end{pmatrix}$$

pixel
finale

Direzione
luce 1

Intensità / colore
della luce 1

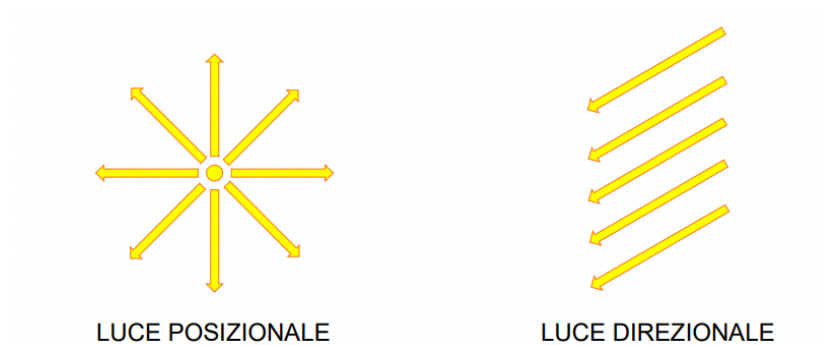
Direzione
luce 2

Intensità / colore
della luce 2



L'aggiunta di ogni luce comporta un aggravio di computazione del lighting. Dobbiamo computare un altro termine, su ogni pixel della scena (o ogni elemento da calcolare).

- In assenza di qualsiasi fonte di luce, il lighting degli oggetti ha zero addendi, e il colore risultante è (0,0,0) al buio, tutti gli oggetti sono neri.
- All'aggiungere fonti di luce, il lighting si satura verso il bianco, dato che il pixel finale al massimo è (1,1,1). Quindi se vogliamo aggiungere molte luci, ciascuna di loro deve essere più fiavole.



Possiamo modellare due tipi di sorgenti di luce:

direzionali = modella fonti di luce molto distanti (per es, il sole). ha una direzione di luce incidente $\vec{\omega}$ costante su tutta la scena (il vettore unitario $\vec{\omega}$ definisce la geometria della luce). Una luce direzionale ha un'intensità/colore (L_R, L_G, L_B) costante su tutta la scena.

posizionali = modella fonti di luci vicine (per es., lampadine). È descritta da una posizione nello spazio P_L . La direzione della luce è diversa per ogni punto da illuminare. Un punto in posizione Q riceverà la luce dalla direzione:

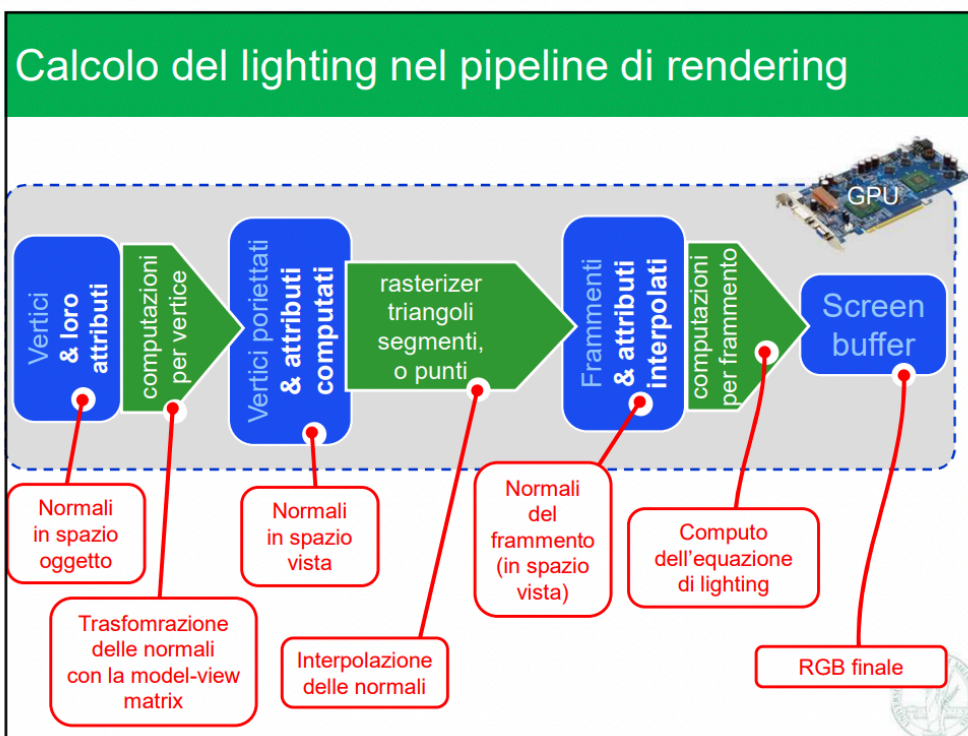
$$\vec{\omega} = \frac{P_L - Q}{||P_L - Q||}$$

In ogni caso, sia il vettore $\vec{\omega}$ che il punto Q sono tipicamente espressi in spazio mondo.

In una luce posizionale, l'intensità/colore può essere attenuato, per ogni punto illuminato Q , da un fattore di affievolimento che varia in funzione della sua distanza dalla luce $d = ||P - Q||$. La formula fisica è

$$f_{affievolimento} = \frac{1}{||P_L - Q||^2}$$

a volte si usa un esponente minore di 2 per rendere artificialmente meno decisa l'attenuazione della luce con la distanza.



il computo del **lighting** tipicamente viene svolto nella **fase per frammento**: tecnica nota come “per-pixel lighting”, calcolo dell’illuminazione per ogni pixel (ma esistono anche varianti e ottimizzazioni). Dunque, il lighting avviene dopo la trasformazione dei vertici. L’equazione di lighting è calcolata in un **fragment-shader**.

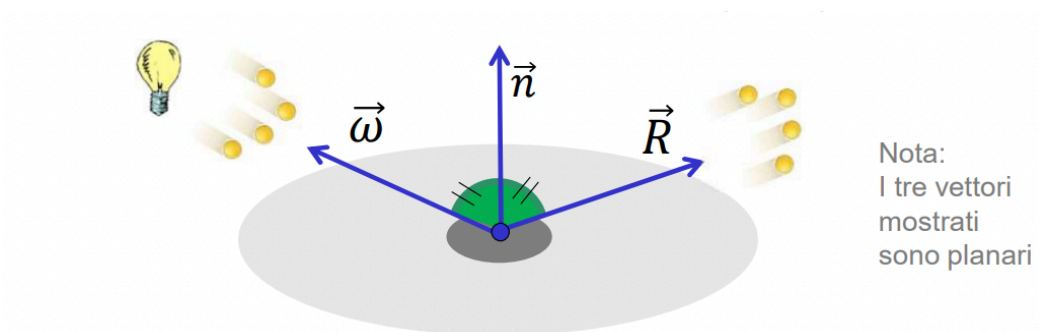
I punti e vettori usati nell’**equazione di lighting** devono, ovviamente, essere tutti espressi nello stesso spazio (direzione luce, direzione vista, normale, posizione luce...) spesso si sceglie lo **spazio vista** oppure lo **spazio mondo**: a questo scopo, le normali della mesh (originalmente espresse, ovviamente, in spazio oggetto) devono essere trasformate in spazio vista o mondo, nella fase di trasformazione per vertice, usando la matrice model-view o di model (nota: non in spazio clip: la matrice di proiezione prospettica non può essere usata per trasformare vettori, ma solo punti). Le direzioni/posizioni della luce vanno espresse nello stesso spazio.

Limiti del modello di Lambert → Un materiale Lambertiano riproduce solo l’aspetto materiali opachi (dull) privi di riflessi luminosi ma lo fa in modo fisicamente corretto.

Vediamo il modello di **materiale di Phong**, capace di riprodurre anche l’aspetto di superfici **lucide** (come quelle bagnate, levigate, incerate, etc...) aggiungendo riflessi lucidi (specular reflections, o highlights).

modello di Phong = non è ispirato da considerazioni fisiche, non è confermato da misurazioni radiometriche. È basato su una costruzione geometrica molto approssimata ma efficace.

In presenza di un materiale molto levigato / liscio / lucido, i fotoni rimbalzeranno sulla superficie (riflessione) in modo simile a quello di una pallina da ping-pong su un tavolo.

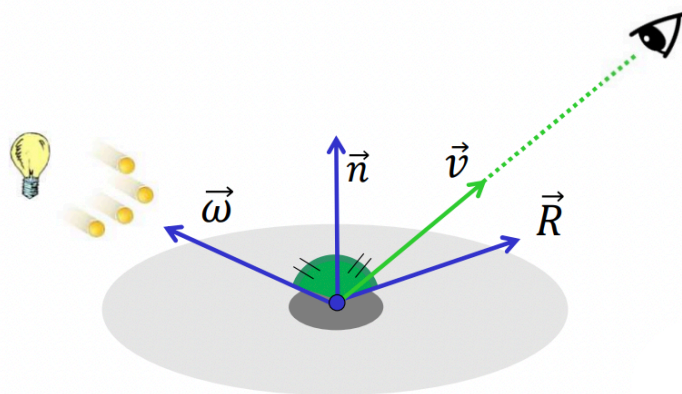


Intuizione → la BRDF di un materiale lucido manderà un numero **maggiore** dei fotoni provenienti dalla direzione $\vec{\omega}$ nella direzione riflessa $\vec{R}$, che non nelle altre direzioni (a differenza di quella di un materiale diffusivo / lambertiano, che li **rimbalza** in qualsiasi direzione con uguale probabilità)

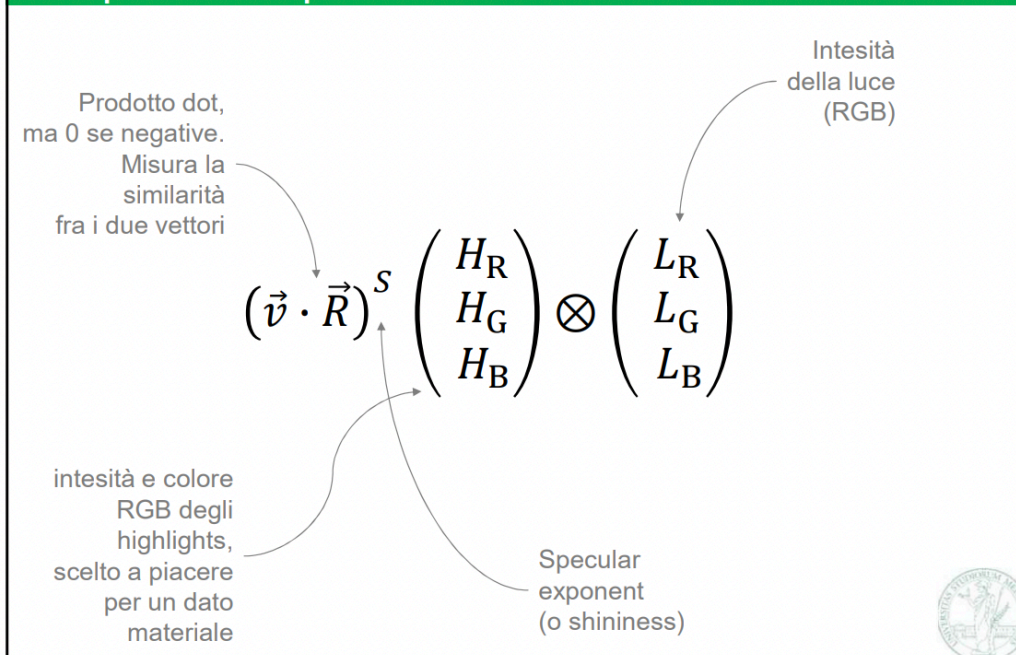
Il modello di Phong somma alla componente di riflesso **diffusiva** (quella vista) una componente di riflesso **speculare**, che sarà tanto maggiore tanto più la direzione di vista $\vec{v}$ sarà simile alla direzione di luce riflessa $\vec{R}$.

calcolo di $\vec{R}$ a partire da $\vec{\omega}$ e $\vec{n}$ (vedi lez algebra punti e vettori)

$$\vec{R} = -\vec{\omega} + 2 (\vec{\omega} \cdot \vec{n}) \vec{n}$$

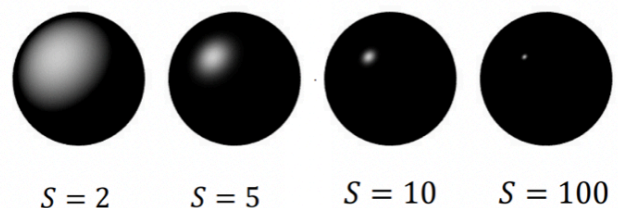
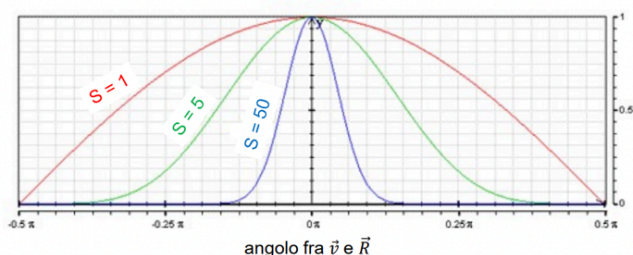


Modello di illuminazione di Phong: componente speculare: calcolo



shininess → (o specular exponent) La formula $(\vec{v} \cdot \vec{R})^S$ (fra 1 e 0) produrrebbe riflessi troppo grandi, perché decresce troppo lentamente al discostarsi di $\vec{v}$ da $\vec{R}$. Per ovviare, basta elevare ad una potenza $S > 1$ (elevando un numero minore di 1 ad un $S > 1$, lo si avvicina allo 0). Il fattore S è detto **specular exponent**, o shininess, o glossiness. Tanto maggiore vale S , tanto più simile dovranno essere $\vec{v}$ e $\vec{R}$ per avere un riflesso visibile, cioè tanto più piccoli e concentrati saranno gli highlights (questo passaggio non ha nessun preciso significato fisico, ed è solo un espediente si usa per ottenere l'effetto desiderato).

✓ Elevando il coefficiente ad una potenza $S > 1$, si ottengono highlights più piccoli e concentrati



Modello di illuminazione di Phong: riassunto degli input del calcolo

Input:

✓ la direzione di luce $\vec{\omega}$

✓ la direzione di vista $\vec{v}$

è un modello *view-dependent* !

✓ Intensità-colore della luce $\begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix}$

✓ Base-color o diffuse-color $\begin{pmatrix} D_R \\ D_G \\ D_B \end{pmatrix}$

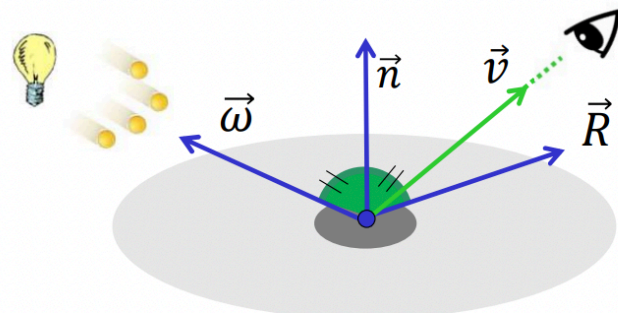
✓ Intensità-colore degli highlights $\begin{pmatrix} H_R \\ H_G \\ H_B \end{pmatrix}$

✓ Esponente di specularità,
o shininess, o glossiness S

Nuove caratteristiche dei
materiali che concorrono a
differenziare
un materiale da un altro



Blinn-Phong = è una variante (quasi equivalente) per calcolare la similarità fra $\vec{v}$ e $\vec{R}$ (la componente speculare) senza dover calcolare esplicitamente $\vec{R}$.



$\vec{v}$ coincide con $\vec{R}$ sse la normale $\vec{n}$ è esattamente a metà strada dei due. Invece che calcolare $\vec{R}$ (a partire da $\vec{w}$ e $\vec{n}$) e poi chiedersi quanto sia simile a $\vec{v}$, la formula Blinn-Phong calcola la direzione media fra $\vec{w}$ e $\vec{v}$, detta **half-way vector**, e si chiede quanto sia simile ad $\vec{n}$. Questo produce un risultato molto simile ma meno oneroso calcolare.

Modello di illuminazione di Blinn-Phong: componente speculare: calcolo

Prodotto dot, ma 0 se negativo. Misura la similarità fra i due vettori

Half-way vector: la media (rinormalizzata) fra direzione di vista e direzione di luce

Intensità della luce (RGB)

$$(\vec{n} \cdot \vec{h})^s \left(\begin{pmatrix} H_R \\ H_G \\ H_B \end{pmatrix} \otimes \begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix} \right)$$

intensità e colore RGB degli highlights, scelto a piacere per un dato materiale

Specular exponent o shininess (come in precedenza)



modello completo → il modello di illuminazione di Phong aggiunge, (per ogni luce) la componente diffusiva (di Lambert) alla componente Speculare (qui, calcolata con Blinn-Phong)

$$\begin{pmatrix} P_R \\ P_G \\ P_B \end{pmatrix} = (\vec{n}_x \cdot \vec{\omega}) \begin{pmatrix} D_R \\ D_G \\ D_B \end{pmatrix} \otimes \begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix} + (\vec{n} \cdot \vec{h})^s \begin{pmatrix} H_R \\ H_G \\ H_B \end{pmatrix} \otimes \begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix}$$

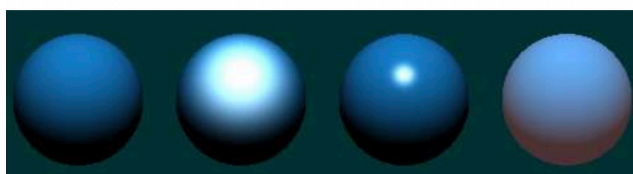
Diagram illustrating the Phong shading model equation. Labels include: Base color del materiale, Shininess, Specular color del materiale, Half-way vector: $\frac{\vec{v} + \vec{\omega}}{\|\vec{v} + \vec{\omega}\|}$, Direzione luce, Intensità / colore della luce 1, Direzione vista, and pixel finale.

Un problema coi modelli di illuminazione visti è che le superfici non raggiunte da alcuna luce (ad esempio, i cui prodotti dot fra normale e dir luce o half-way siano zero) sono **perfettamente neri**. In una scena realistica, questo è un caso molto raro.

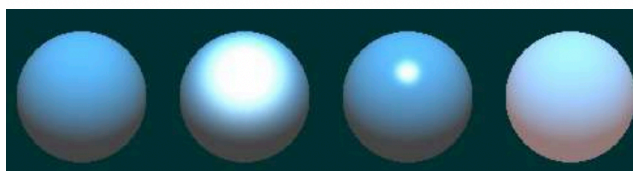
Nella maggior parte dei casi, si può assumere che un minimo di **luce ambientale** raggiunga qualsiasi superficie da qualsiasi direzione. Per simulare questo stato di cose, si può semplicemente aggiungere al modello di illuminazione una **componente ambient**, una piccola costante additiva (in R, G e B) data dal prodotto:

- Tra l'intensità di una apposita fonte luce di tipo «ambientale» (una luce pervasiva, globale, che non ha ne' una direzione, ne' posizione)
- e il colore di una apposita componente «ambient» del materiale.

Componente *ambiente*



senza



con

Modello di illuminazione di Phong con luce ambiente

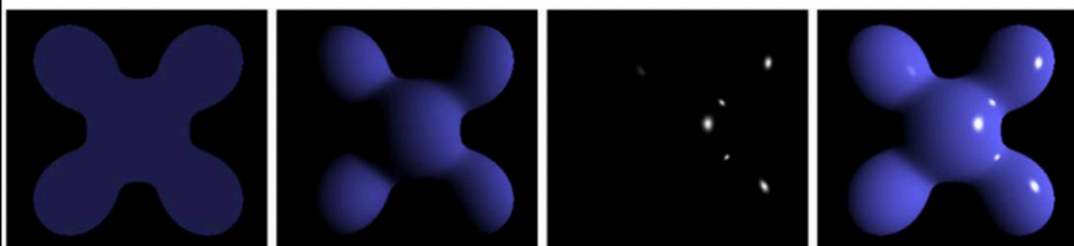
✓ Modello di illuminazione risultante

$$\begin{array}{c}
 \text{ripetuto per ciascuna luce direzionale e posizionale} \qquad \text{per la luce ambiente} \\
 \hline
 \begin{array}{ccc}
 \text{termine diffusivo} & \text{termine speculare} & \text{termine ambiente} \\
 \hline
 \begin{pmatrix} P_R \\ P_G \\ P_B \end{pmatrix} = (\vec{n}_x \cdot \vec{\omega}) \begin{pmatrix} D_R \\ D_G \\ D_B \end{pmatrix} \otimes \begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix} + (\vec{n} \cdot \vec{h})^s \begin{pmatrix} H_R \\ H_G \\ H_B \end{pmatrix} \otimes \begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix} + \begin{pmatrix} A_R \\ A_G \\ A_B \end{pmatrix} \otimes \begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix}
 \end{array}
 \end{array}$$



Modello di phong esempio (con una luce e componente ambiente)

Il risultato finale è ottenuto sommando tutti i termini



ambiente + diffusivo + speculare (highlights) = totale



Il modello di Phong (visione semplificata del problema del lighting) descrive un materiale come:

- Un colore diffusivo
- Un colore ambient, di solito corrispondente al colore diffusivo
- Un colore speculare
- Uno specular coefficient (da 1 a 100)

La scelta di un qualsiasi materiale (velluto, oro, etc) si effettua spaziando fra i possibili valori da assegnare a queste variabili.

Questo modo di definire i materiali lascia **pochi gradi di libertà**, e consente di riprodurre solo materiali in una gamma ristretta di possibilità.

Inoltre molte delle scelte possibili risultano in materiali dall'aspetto «sbagliato» ritenuti, secondo gli standard attuali della CG, **poco realistici**.

Anche la descrizione dell'ambiente di illuminazione vista (cioè, una collezione di luci direzionali o posizionali) è una **semplificazione** eccessiva, che non consente di riprodurre fedelmente ambiente di illuminazione realistici.

Modelli di materiale più complessi, che richiedono però computi di lighting più onerosi, consentono di definire materiali in modo più completo, e rendono anche più facile e intuitiva la scelta di parametri atti a riprodurre materiali reali.