

A.A. 2020/21

GRAFICA AL CALCOLATORE

ANDREA GIACCHETTI

FABS :)

NOTA

Questi appunti/sbobinatura/versione “discorsiva” delle slides sono per mia utilità personale, quindi pur avendole revisionate potrebbero essere ancora presenti typos, commenti/aggiunte personali (che anzi, lascio di proposito) e nel caso peggiore qualche inesattezza!

Comunque spero siano utili! 

QUESTA SBOBINA NELLO SPECIFICO:

È stata usata da me (30L), un collega (30L) e un altro collega (27), e contiene una buona sbobinatura di ciò che ha detto il prof + varie integrazioni da lo internet.

Consiglio caldamente la visione del video sull’equazione del rendering linkato nel paragrafo relativo. Il layout è pessimo, ma (non modestamente, lo so) credo che il contenuto sia ottimo,

so have fun 😊

Questa sbobina fa parte della mia collezione di sbobinature,
che è disponibile (e modificabile!) insieme ad altre in questa repo:

<https://github.com/fabfabretti/sboninamento-seriale-univR>

Table of Contents

NOTA	1
1 - Introduzione.....	6
» Storia	6
» Visual computing.....	7
2 – Dispositivi	8
» Display	8
» Stampanti	10
3 - Immagini e colore.....	11
» Immagini.....	11
» Grafica vettoriale.....	11
» Immagini raster	12
» Colore.....	14
4 – Geometria	16
» Ripasso: geometria euclidea.....	16
» Trasformazioni e matrici	17
» Trasformazioni problematiche: le rotazioni	19
» Macchina fotografica virtuale	21
5 - Modeling	23
» Mesh poligonali	24
» Mesh e rendering.....	28
6 - Rendering	32
» Paradigmi di rendering:.....	33
» Shading	33
» Equazione del rendering	35
» Modelli di illuminazione	36
» Implementazione del rendering	40
» Ricostruzione dei raggi: ray casting.....	40
» Ombre	42
» Trasparenza	42
» Simulazione più accurata in BSDF	42
7 – Pipeline di rasterizzazione	43
» Pipeline Grafica	44
» Prima parte: pipeline geometrica	45
» Seconda parte: rasterizzazione.....	47
» Output e input	52
» Schede grafiche	52

» Shaders.....	54
8 – Mapping	55
» Texture mapping.....	55
» Texture lookup.....	58
9 – Ombre	61
» Tecniche.....	61
10 - Animazione	64
11 - Visualizzazione scientifica.....	68
» Premessa: Principi di visualizzazione.....	68
» Visualizzazione scientifica.....	71
» Griglie 2D	73
» Dati multidimensionali.....	74

1 - Introduzione

GRAFICA AL CALCOLATORE: disciplina che studia le tecniche e gli algoritmi per la rappresentazione visuale di informazioni numeriche prodotte o elaborate dai computer.



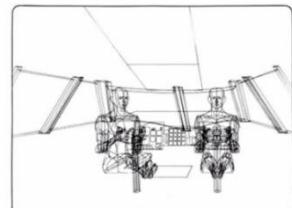
L'applicazione è creare la rappresentazione che nasce come dato nel computer e trasformarlo in modo da essere percepito in maniera visiva.

Nella grafica, in realtà, si parla anche di altri canali: per esempio la stampa 3D crea rappresentazioni che possono anche essere toccate.

STORIA

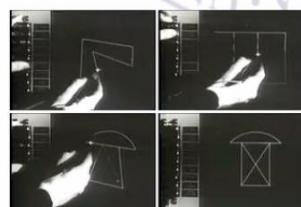
- **1960s**

- > Il termine "Computer Graphics" nasce negli anni '60, introdotto da William Fetter per descrivere la sua *ricerca ergonomica* in cui studiava come studiare la carlinga degli aerei Boeing in modo ergonomico - realizza il primo modello "umano". Nasce quindi l'idea della modellazione 3D che rappresenta una parte rilevante della moderna CG.
- > Negli anni '60 nascono anche i primi *terminali grafici* e i *giochi* nel 1961 nasce primo videogioco, Spacewar



- **1963**

- > Nasce la prima *Computer Grafica interattiva* (= uno schermo che non si limita a funzionare da schede perforate e la stampa di testo) il sistema sketchpad di Ivan Sutherland, un sistema di disegno su schermo che consentiva di tracciare linee che venivano corrette in disegni geometrici.



- **1970s**

- > Le interfacce grafiche evolvono con i PC fino ad arrivare alle *interfacce grafiche interattive* dei computer, le *WIMP* (windows icon menu pointers). La grafica interattiva 2D diventa parte integrante del sistema uomo macchina.
- > 1972: Primi *videogiochi commerciali*, come Pong.
→ I videogiochi sono una delle industrie che più ha spinto lo sviluppo di grafica fotorealistica! :D
- > 1972: prima *animazione di grafica 3D*, un modello della sua mano con 350 poligoni. Il creatore, Edwin Catmull, diventa cofondatore della Pixar.



- Gli algoritmi per disegnare linee raster, poligoni, e così via sono sviluppati negli anni 60-80. Si creano delle *pipeline di rendering*, per creare velocemente immagini sullo schermo a frame rate interattivi. Si creano standard e implementazioni di sistemi grafici e si arriva alla situazione attuale:



- > 1992 Silicon Graphics crea lo standard OpenGL
- > 1995 Microsoft rilascia Direct 3D

Le operazioni grafiche vengono *implementate su hardware specifico*. Inizialmente la grafica raster è calcolata sulla CPU, poi doppio buffer (per poterla vedere e creare altre, dato che il calcolo può essere lento rispetto al refresh dello schermo) per mantenere le immagini.

Poi vengono introdotte *schede con architetture specifiche*: nel 1985 esce il Commodore Amiga, uno dei primi home-computer con una GPU (Graphical Processing Unit), nel 1987 IBM ha le prime operazioni 2D nell'hardware, nel 1995 prima scheda video con pipeline per 3D, nel 2018 arriviamo a capacità incredibili come il ray tracing.

L'evoluzione storica ha fatto sì che per grafica si intenda soprattutto la *teoria del rendering e la pipeline delle schede grafiche*, dati da una serie di algoritmi che permettono di arrivare a immagini fotorealistica.

In realtà questo è un discorso estremamente ampio e nella realtà si utilizzano tecniche più avanzate. Però le idee di base ci sono.

Altre cose che ne fanno parte:

- > *Imaging*: image editing, video processing, to mapping
- > *Modellazione* 2D (grafica vettoriale) e 3D (CAD, scene editor)
- > *Rendering* 2D (rasterizzazione e disegno artistico) e 3D (immagini fotorealistiche)
- > *Animazione* 2D (sequenze, interpolazione) e 3D

Le applicazioni possono essere

- > *Non interattive*: impaginazione, stampa, video editing, cinema, digital manufacturing
- > *Interattiva*: interfacce utente, simulazione, gaming, visualizzazione scientifica, visualizzazione dell'informazione

VISUAL COMPUTING

Vista la convergenza dei due domini si parla spesso di visual computing, un termine generico per *tutte le discipline di informatica che si occupano di immagini e modelli 3D*: computer grafica, image processing, computer vision, VR e realtà aumentata, processing video ma anche pattern recognition, interazione uomo macchina, analisi e rendering di informazioni, videogiochi, effetti speciali.... Oggi, la disciplina del Visual Computing mette insieme tutte le discipline di CS che hanno a che fare con modelli 3D e immagini: computer graphics, image processing, visualization, pattern recognition, human computer interaction, machine learning and digital libraries.

Computer graphics vs vision

Computer Graphics

dato → immagine

Preso un input parametrico, voglio creare un'immagine: passo da numeri al calcolatore alla rappresentazione e interpretazione

Computer Vision

immagine → dato

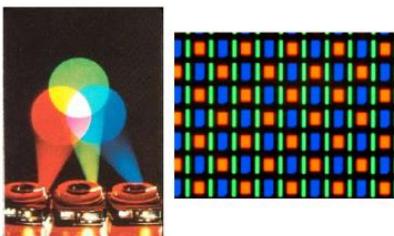
Prende immagini da un sistema di acquisizione e lo trasforma in nuova informazione, parametri e interpretazione, non-immagine.

- > Fotogrammetria
- > Identificazione facce da video

2 – Dispositivi

DISPLAY

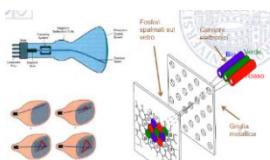
| Display raster



Fanno operazioni che trasformano il tutto in immagini raster, ovvero matrici di puntini luminosi che possono essere colorati con sintesi additiva sovrapponendo o mettendo molto vicini tre fasci di luce. Non è affatto banale: la visione umana non misura la frequenza di luce ma è tricromatica (ovvero la percezione dipende da tre sensori).

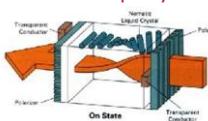
Nei dispositivi raster, il dato di uscita viene preparato su una memoria – in genere dedicata – detta *memoria di quadro* o *frame buffer*. Vi sono le informazioni utili a generare ogni singolo pixel.

» Display CRT (tubo catodico) [[video](#)]



Dietro lo schermo sono posizionate delle “pistole” in grado di emettere fasci di elettroni – uno per colore. Questi fasci vengono direzionati per colpire uno strato ricoperto di fosforo con varie composizioni (una per colore), le quali si illuminano con intensità proporzionale all’intensità del raggio. La pistola, come un pennello, si muove “dipingendo” l’intero schermo, un “pixel” per volta. Negli schermi vettoriali, anziché dipingerli tutti sequenzialmente può tracciare direttamente le linee richieste.

» Display TFT/LCD [[video](#)]



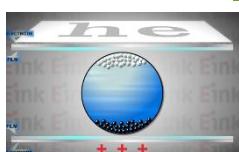
Lo schermo viene interamente illuminato con luce bianca. Dinnanzi a questa luce bianca sono posti due filtri di polarizzazione, con inclinazione di 90° l’uno con l’altro; in questo modo, la luce che passa per il primo viene completamente bloccata dal secondo. Fra i due filtri viene posto uno strato di cristalli liquidi, i quali sono inizialmente paralleli al primo filtro, e applicando un ΔV si torcono allineandosi al secondo; più voltaggio applico, più il raggio di luce sarà ruotato e passerà attraverso il secondo filtro. Per generare i colori RGB è sufficiente dividere lo schermo in pixels e aggiungere uno strato colorato su ciascun subpixel.

» Display OLED [[video](#)]



Sottile strato di materiale organico che, in corrispondenza del passaggio di una corrente elettrica si illumina in maniera puntuale. Gli elettrodi sono le due lastre all’interno delle quali è piazzato lo strato (almeno una delle due è trasparente per permettere l’emissione della luce)

» Electronic ink [[video](#)]



Sono quelli usati per gli ebook. Lo schermo è composto da “microcapsule” della dimensione di un capello, all’interno delle quali si trovano particelle nere (-) e bianche (+), caricate e sospese in un fluido trasparente. Esse sono racchiuse in un elettrodo, che caricandosi provoca lo spostamento delle particelle. Hanno un framerate molto basso ma riescono a riprodurre senza bisogno di illuminazione.

» Sistemi a proiezione

Sono caratterizzati dalla *grande dimensione* della superficie visibile.

Basati su tecnologia LCD o DLP (Digital Light Processing)

- LCD: come schermi TFT con proiezione a distanza
- DLP [[video](#)]: costituiti da un pannello di micro-specchietti, uno per ogni pixel, di cui si comanda la rotazione su un asse; riflettono la luce incidente in proporzione al loro orientamento; si ottiene maggiore luminosità e nitidezza delle immagini

» Display stereoscopici

Con un display stereoscopico si possono visualizzare due immagini diverse. Veniva già fatto a fine 19 secolo con lo stereoscopio.

- » *Anaglifo*: proietta due immagini sovrapposte che rappresentano il punto di vista dei due occhi, e uso degli occhiali con filtri cromatici o polarizzati per mostrare a ciascun occhio la sua.
- » *Head Mounted Display*: sono i VR; ho un display per occhio.
 - > Muovendomi, vorrei che la posizione della vista cambiasse di conseguenza! Si può ovviare se il programma che fa il rendering aggiorna continuamente il display in base alla posizione – come succede in VR : gli schermi seguono la posizione della testa all'interno della realtà virtuale.
- » *Barriere di parallasse*: sfruttano barriere per far vedere immagini diverse a ciascun occhio, come i 3Ds. La posizione deve essere fissa.
- » *LCD shutter glasses*: boh

Display vettoriali

Possono essere rappresentati da plotter; sono poco usati

Erano diffusi nei primi anni 60; tracciano direttamente linee e punti come un plotter a penna. Sono caratterizzati da primitive di disegno vettoriali e visualizzazione wireframe.

» Plotter



Muovendo una penna crea primitive di disegno. La rappresentazione al calcolatore deve essere trasformata in comandi appositi con protocolli specifici.

I plotter sono usati solo in applicazioni molto particolari poiché si preferiscono le stampanti.

STAMPANTI

Stampanti 2D

Compongono le immagini per via sottrattiva (non luce emessa ma pigmenti che assorbono luce!). Il DPI è molto più alto.

C'è una forte distinzione fra stampanti con unità di elaborazione e linguaggio di definizione della pagina, o stampanti che richiedono la descrizione della pagina come collezione di pixel.

In generale l'applicazione che prepara alla stampa crea una descrizione vettoriale del disegno.

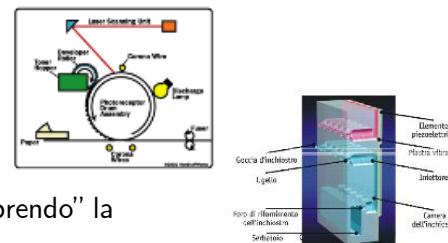
Non necessariamente arriva un dato raster in input. E' utile perché la risoluzione di una stampa dipende dalla stampante!

» Stampante laser

Usa un raggio laser e pesca toner di carica opposta (or smth)

» Stampante inkjet

La testina di stampa contiene un dispositivo piezoelettrico; l'impulso estende o contrae gli strade, deformando la piastra e "aprendo" la camera d'inchiostro.



Stampanti 3D

Stampanti a tecnologia sottrattiva

Le tecniche sottrattive richiedono il calcolo del percorso macchina, il percorso che l'utensile di taglio deve seguire.

È un compito complesso che tiene conto:



- » della *forma* dell'oggetto
- » delle *caratteristiche del braccio* robotico
- » della necessità di prevenire *collisioni*

L'accuratezza dipende dalla dimensione dello strumento di taglio (e del percorso macchina) e dal modello originale.

Spesso si preferisce effettuare una sgrossatura a macchina e una finitura manuale del prodotto.

Stampanti a tecnologia additiva

Le tecniche additive riproducono il modello per sezioni. Simula un processo in cui il modello digitale 3D è *suddiviso in sottilissime fettine* (risoluzione ordine di 0,1 - 0,2 mm). Ogni fettina è ricostruita dal dispositivo aggiungendo un sottile strato di materiale al modello in costruzione.

Il calcolo di ogni sezione consiste nell'"*affettare*" il modello, che deve essere completamente chiuso e il più possibile continuo.

Troviamo:

- » FDM
- » SL
- » Polyjet (inkjet su strati)
- » SLS

3 - Immagini e colore

IMMAGINI

Possiamo definire le immagini in modo diverso:

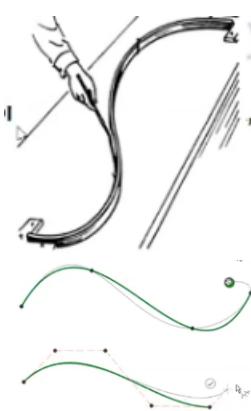
- » Definizione "matematica": l'immagine diventa la discretizzazione su una griglia di una funzione $f(x,y) \rightarrow$ immagine raster, è la definizione che usiamo nel corso di immagini
 - > In realtà, oltre alle immagini raster esistono anche le grafiche vettoriali (che sono usate anche per disegnare le altre) e i modelli 3D
- » Definizione intuitiva: Forma esteriore degli oggetti in quanto viene percepita attraverso il senso della vista; rappresentazione con mezzi tecnici o artistici della forma esteriore di cosa reale o fittizia.

Rappresentazione scena	Rendering	Output per display
Immagine raster	Resampling, trasformazione colore	Frame buffer
Immagine vettoriale	Comandi per il sistema di controllo di pennino/fascio	Comandi per display vettoriale (es. plotter)
Immagine vettoriale	Rasterizzazione	Frame buffer
Modello 3D	Slicing	Comandi per stampante 3D
Modello 3D	Rendering 3D fotorealstico/rasterizzazione	Framebuffer o immagine digitale per stampa o video

Visto che la definizione non è univoca, per dare una definizione generica che le includa tutte dobbiamo vederla come uno schema che adatta la definizione al tipo di immagine rappresentata. Quello che cambia è il tipo di dato e il tipo di output, che forza ad avere algoritmi ottimali diversi in base al tipo di immagine.

Possiamo rappresentare la grafica in un dato digitale in due modi: vettoriale e raster.

GRAFICA VETTORIALE



Prevede di rappresentare i disegni attraverso *primitive di disegno* come linee, curve, circonferenze.

Definire segmenti è semplice; per rappresentare le curve arbitrarie è usare le *splines*, ovvero vari *modelli matematici* che con pochi parametri codificano curve lisce. Lo spline è *l'oggetto fisico* (asticella flessibile e vincolata); in matematica sono funzioni vincolate da condizioni. Tipicamente si usano funzioni polinomiali e le si fanno passare per punti, oppure con funzioni approssimanti che minimizzano la distanza.

Esempi di applicazioni: disegno su display vettoriali, Plotter Input per stampanti... Tipicamente, i font!

Vantaggi

- + Rappresentazione codificata è più facilmente *comprendibile* a un essere umano
- + *Non perde risoluzione* quando viene scalata
- + Codifica più *compatta*

Limiti

- » Deve essere *convertito in raster* per poter essere mostrato sulla maggior parte dei display.
- » Non funziona bene per le *fotografie*

Rasterizzazione

È il processo che *trasforma un vettore in un raster*; può essere pensato come il campionamento dell'immagine vettoriale. La qualità della conversione dipende dalla risoluzione.

Se è troppo bassa c'è la scalettatura: effetto dell'aliasing delle alte frequenze.

Si può ridurre l'effetto dell'aliasing delle alte frequenze. Esso accade perché l'immagine è il campionamento di una *funzione 2D continua*.



IMMAGINI RASTER

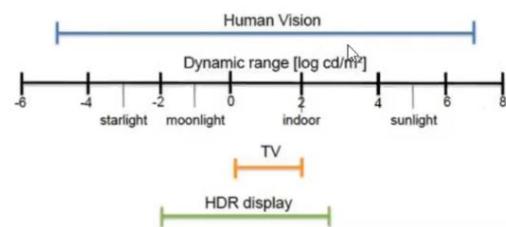
Sono le immagini digitali a cui siamo abituati con i monitor, e consistono in una matrice di elementi denominati pixel. Il nostro software scrive in un *frame buffer*, ovvero una memoria che contiene l'immagine (array di valori per pixel). La rappresentazione è semplice ma *occupa molta memoria*.

E' anche il *campionamento* della funzione luminosità che colpisce i pixel (oppure funzione vettoriale se si parla di rasterizzazione).

Le fotografie producono immagini raster: salvano per ogni pixel la luce arrivata in una certa regione del sensore.

Le caratteristiche sono:

- » Risoluzione: numero di *pixel* della griglia
- » Range dinamico e profondità di colore: bit di memoria per pixel; è il *rappporto fra minima differenza misurabile o rappresentabile e range di variabilità del segnale* (=luminosità o colore). I dispositivi hanno un range dinamico molto variabile.
 - » E' rappresentata più o meno espressivamente in base al numero di bit. Il valore codificato dovrebbe corrispondere alla luminosità del punto generata dal monitoro acquisita dal sensore, ma la cosa è un po' più complicata a causa dellanon-linearità della percezione umana.



Luminosità

Tralasciando momentaneamente il colore, i numeri nei pixel codificano l'intensità luminosa. Essa non può essere rappresentata in valore assoluto, poiché il colore viene displayato in maniera diversa su ogni schermo!

Andrebbe calibrato, ma la qualità dei monitor spesso è diversa (es. alcuni fanno neri più neri di altri).

Di conseguenza, al posto del valore assoluto ho una funzione di trasferimento che si basa sul valore *minimo e massimo rappresentabile* dal monitor.

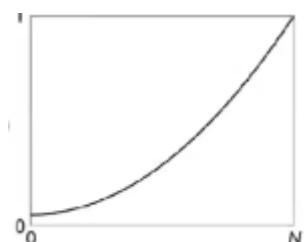
$$I = f(n) \quad f: [0, N] \rightarrow [I_{min}, I_{max}] \text{ con } I_{min} = \text{intensità del nero} \text{ e } I_{max} = \text{potenza massima}$$

In generale, questa funzione non è lineare.

$$R_d = \frac{I_{max} + I_{amb}}{I_{min} + I_{amb}}$$

Il range dinamico effettivo dipende dalla *quantità emessa dal display* e dalla *luce ambientale*.

Questo valore varia in base al display: se esso ha pochi livelli, è inutile avere una depth di immagine enorme perché tanto non sarei in grado di riprodurre differenze misurabili in quel display.



Desktop display in typical conditions: 20:1

Photographic print: 30:1

Desktop display in good conditions: 100:1

High-end display under ideal conditions: 1000:1

Digital cinema projection: 1000:1

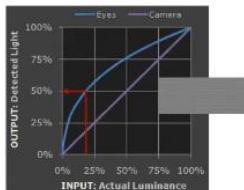
Photographic transparency (directly viewed): 1000:1

High dynamic range display: 10,000:1

Il principale motivo per cui la mappatura di luminosità sul numero non è lineare è che il *contrasto percepito* nell'uomo è a sua volta *non lineare*: aumentare la luminosità di x mi da una percezione totalmente diversa in base a se mi trovo in un livello di luminosità alto o basso.

la differenza percepita rimane costante che ho se *aumento in percentuale* (\rightarrow esponenziale logaritmico); empiricamente, la minima diffrenza è il 2%. In base a questa legge si può stimare il # di livelli necessario a non percepire posterizzazione.

Gamma encoding



La gamma encoding permette di vedere una diffrenza costante fra livelli di luminosità; serve poiché l'occhio umano percepisce la luce in modo non-lineare (anzi, segue un esponenziale), con una sensibilità maggiore fra la differenza fra toni più scuri. In questo modo si può ottimizzare l'utilizzo dei bit nella codifica di un'immagine: se un'immagine non è gamma-encoded, vengono allocati troppi bits per valori che l'uomo non è in grado di differenziare, e troppi pochi bits per valori ai cui l'occhio umano è molto sensibile.

Originariamente è stato sviluppato per compensare le caratteristiche di input-output dei display a tubo catodico: l'intensità della luce variava in maniera non lineare con il voltaggio dell'electron gun; con la gamma encoding si tornava ad avere un rapporto lineare. [\[source\]](#)

Gamma correction: Il numero che codifica la luminosità delle immagini viene trasformato (prima della rappresentazione sul display) con una correzione che permette di vedere una *differenza costante fra i livelli*.

$$I(n) = (n/N)^\gamma$$

Similarmente, le immagini acquisite dalle fotocamere sono corrette con il fattore inverso della gamma correction prima di essere quantizzate

$$n(I) = NI^{\frac{1}{\gamma}}$$

Quantizzazione

- Esempio: sfumature continue (gamma encoded)



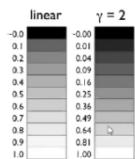
- Quantizzazione in spazio lineare 32 livelli



- Quantizzazione spazio gamma encoded



troppe luminose.



Tipicamente si quantizza il valore dell'immagine raster dopo aver trasformato la variazione di intensità; se non applico le correzioni in maniera corretta (ovvero $\neq 2.2$) arrivo ad avere *immagini scure o*

Immagini HDR (high dynamic range)

L'HDRI, sigla di high dynamic range imaging, è una tecnica utilizzata in grafica computerizzata e in fotografia per ottenere un'immagine in cui l'intervallo dinamico, ovvero l'intervallo tra le aree visibili più chiare e quelle più scure, sia *più ampio* dei metodi usuali.

Le tecniche per la creazione di una HDRI si basano sull'idea di effettuare scatti multipli dello stesso soggetto ma a diverse esposizioni, in maniera tale da compensare la perdita di dettagli nelle zone sottoesposte o sovraesposte di ciascuna singola immagine. La successiva elaborazione della serie di immagini consente di ottenere un'unica immagine con una corretta esposizione sia delle aree più scure che di quelle più chiare.

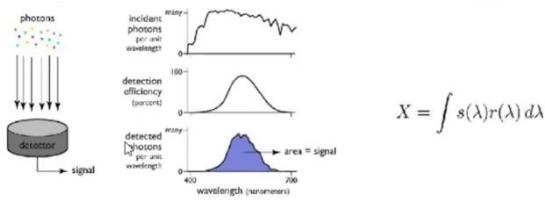
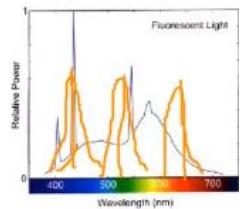
Prevede di acquisire la luminosità in *maniera lineare*, non limitati e trasformati \rightarrow *memorizza dati float*. E' reso possibile dalla grande quantità di memoria dei nostri giorni. Devono essere renderizzati per essere visualizzati sul display: esistono diversi modi. Ci pensa il frame buffer a rappresentare questa big info :)

Possono essere catturate da macchine fotografiche comuni con *esposizioni multiple* e blending.

COLORE

Generalmente lo si rappresenta con una **terna RGB**, per via di **sintesi additiva** (per i display) o **sottrattiva** (per le stampanti). Nella realtà, tipicamente, non ho solo tre colori ma uno spettro ricco di frequenze continue; lo schermo invece ha **tre semplici picchi**.

Il colore non esiste: nel mondo c'è solo la luce. :) Il nostro occhio percepisce l'onda e la trasforma in sensazione di colore.



I sensori negli occhi ci sono delle **cellule** (coni e bastoncelli)

- » Bastoncelli: rispondono a basse luminosità e rilevano contrasto e movimento
- » Coni: rispondono a tre tipi di frequenze (quindi rilevano il colore!)

Sono **sensibili a diverse frequenze**; Il sensore ad una frequenza risponde con un valore numerico.

Al cervello arrivano, di fatto, dei numeri; a fronte di uno spettro, produciamo tre valori (**comprimiamo lo spettro!** Buttiamo via un sacco di informazione).

Teoria tricromatica

Legge di Grassman: l'uomo è in grado di fare il match di 3+ colori, detti primari. Se la luce test T ha un certo colore $T = aP_1 + bP_2 + cP_3$, il match si verifica linearmente: quindi la somma di due luci è simulata dalla somma delle componenti nei colori primari delle due! (tipo i vettori...)

→ Ricavo **funzioni di matching**: data una terna di colori primari studio se i soggetti riescono ad associare colori a terne. Data una terna di colori di base possiamo studiare il matching dei colori sugli osservatori infunzione della frequenza. Componenti colore CIERGB ricavate dall'integrale delle frequenze dello stimolo su tutto il range.

Quali colori primari uso?

- » Dipende dal **device**
- » Risposte **non sempre ortogonali**
- » Non riesco a simulare tutto con una terna che somma! Avolte devo compensare con una **terna sottrattiva**.

Codifica/modello dei colori

Tipicamente si scelgono le tre frequenze base R,G,B riprodotte dal monitor, e ciascun colore sarà una terna di valori. Ma non è univoco:

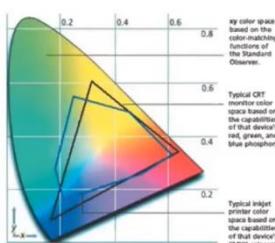
- » Dipende dalla risposta spetrale degli emettitori
- » Dipende dalla calibrazione del monitor (punto di bianco)
- » Dipende dalla risposta (non lineare)

Non tutti i colori visibili sono riprodotti fedelmente, poiché lo spazio di colore ha una gamut limitata.

- » RGB/sRGB:
 - cromaticità su R,G,B
 - punto di bianco su s
- » HSV (spazio percettivo)
Ciascuna dimensione corrisponde a una grandezza intuitiva

- Hue: tinta
 - Saturazione: distanza dal grigio
 - Illuminazione: quantità di bianco
- » CMYK (spazio sottrattivo)
- Usa i colori complementari di rosso, verde e blu più il nero. Lavora per sottrazione anziché per addizione.

Spazi di colore

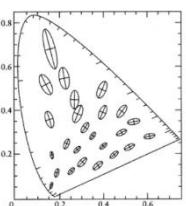


Uno spazio colore non è altro che una *mappatura dei colori che una periferica è in grado di riprodurre*, ovvero una *descrizione precisa di un colore e di come dovrebbe essere riprodotto*. Questi spazi colore definiscono esattamente come ciascun colore deve essere rappresentato dai suoi colori componenti, come dovrebbe apparire la miscela dei colori primari e quale luminosità dovrebbe produrre ogni schermo per determinato colore. Uno spazio colore si basa su un modello di colore (RGB, CMYK, etc...)

. Il modello colore è una *rappresentazione astratta del colore*, invece uno spazio colore lo perfeziona con specifiche regole adatte all'utilizzo che se ne andrà a fare. [\[source\]](#)

Gamut o gamma: si intende la gamma di colori riproducibili da parte di un dispositivo o di una tecnica. Per un display RGB è un triangolo, con i vertici sui colori primari emessi da un pixel

Spazio CIE XYZ: Lo spazio colore è standardizzato dal CIE del 1931, che consente di definire in *maniera oggettiva* il colore, *indipendentemente dai dispositivi*. Un monitor, comunque, non può riprodurre tutte le sensazioni di colore naturali.



Gli spazi visti *non* sono *percettivamente uniformi*: ad uguali variazioni di colore non corrispondono uguali variazioni degli stessi; ad esempio, i recettori del blu sono molto meno sensibili.

→ Per avere uno spazio colore percettivamente uniforme è stato definito dalla CIE lo *spazio L* a* b**, dove L indica la luminosità e a,b sono due colori.

Fenomeni particolari

La percezione del colore è media dal cervello, dunque ci sono fenomeni che (cit.) complicano la vita! Non percepiamo il livello, ma il contrasto. Ed esso *cambia in base ai colori adiacenti*.

- » *Adattamento luminoso*: riscaliamo mentalmente la luminosità adattandoci ai livelli
- » *Adattamento cromatico*: adattiamo la percezione del colore al colore della sorgente luminosa
 - > Evidente col bilanciamento del bianco!
- » *Metamerismo*: Dato che il colore è il prodotto di questa compressione, lo spettro che vedo guardando lo stesso oggetto potrebbe cambiare in base a da che tipo di luce è colpito. (Il problema è della vista, non del materiale!). Naturalmente la luce cambia molto nel mondo!



Trasparenza e canale alfa

Oltre ai colori e alla luminosità, le immagini raster possono avere anche una *matrice di valori alfa*. Alfa è un canale addizionale che solitamente definisce la *trasparenza*. È usato in compositing, operazione che facciamo per comporre più immagini in un solo buffer. (Una volta alfa era un valore necessario a tarare il display!)

4 – Geometria

La geometria ci serve per poter modellare oggetti 2d e 3d. Due problemi sempre da affrontare sono:

RIPASSO: GEOMETRIA EUCLIDEA

- » Punti: posizioni nello spazio; terne di numeri.
- » Vettori: differenze fra punti
- » Direzioni: vettori unitari

$$\mathbf{p} = (p_x, p_y, p_z) \quad \mathbf{v} = (v_x, v_y, v_z) \quad \mathbf{d} = (d_x, d_y, d_z)$$

- » Operazioni

» Somma e prodotto per uno scalare

$$\mathbf{u} + \mathbf{v} = \begin{bmatrix} u_x + v_x \\ u_y + v_y \\ u_z + v_z \end{bmatrix} \quad s\mathbf{v} = \begin{bmatrix} sv_x \\ sv_y \\ sv_z \end{bmatrix}$$

- **Prodotto scalare** o interno

Permette di calcolare velocemente se i due vettori sono ortogonali: sarà nullo.

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z = |\mathbf{u}| |\mathbf{v}| \cos \theta$$

- **Prodotto vettoriale** o esterno: fornisce un vettore ortogonale ai due.

- E' utilissimo per trovare le normali, in quanto servono a risolvere le equazioni del rendering e capire come la luce impatta l'oggetto.
- Permette anche di calcolare l'area del triangolo che formano i due vettori ab.

$$\mathbf{u} \times \mathbf{v} = \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix} \quad |\mathbf{u} \times \mathbf{v}| = |\mathbf{u}| |\mathbf{v}| \sin \theta$$

$$(\mathbf{u} \times \mathbf{v}) \cdot \mathbf{u} = 0 \quad (\mathbf{u} \times \mathbf{v}) \cdot \mathbf{v} = 0$$

• Lunghezza del vettore

$$l = \sqrt{v_x^2 + v_y^2 + v_z^2} = \sqrt{\mathbf{v} \cdot \mathbf{v}}$$

Il punto non è un vettore! (wooow)

- Non esiste la somma di punti, ma esiste la **combinazione affine**

➢ Presi tre punti casuali p, q, o consideriamo il punto

$$\mathbf{p}' = \alpha(\mathbf{p} - \mathbf{o}) + \beta(\mathbf{q} - \mathbf{o}) + \mathbf{o}$$

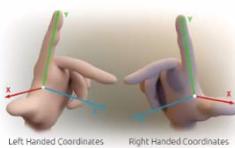


Se

$\alpha + \beta = 1$ allora \mathbf{p}' non dipende da \mathbf{o} ma solo da \mathbf{p} e \mathbf{q} (chiaramente \mathbf{o} si semplifica)

- \mathbf{p}' rappresenta un punto intermedio sulla retta p, q
- Se i due punti sono positivi, il punto sta sul segmento fra \mathbf{p} e \mathbf{q}
- Uso la combinazione affine per definire segmento PQ e retta pq .

Sistemi di riferimento (frame)



Data una base di tre vettori possiamo utilizzarli come base (ovvero rappresento i vettori come terne date dalle proiezioni del vettore sui tre elementi).

Le terne possono essere destrorse e sinistrorse:

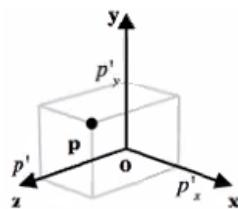
- > **Destrorse:** la rotazione attorno ad e₃ porta e₁ a coincidere su e₂ è antioraria se vista dalla parte positiva di e₃. Oppure fai la persona normale e usi la mano.

Posso specificare, oltre alla base, anche un punto O detto origine, quindi arrivo alla forma (e_1, e_2, e_3, O)

$$v = v_1 e_1 + v_2 e_2 + v_3 e_3$$

I vettori saranno espressi come $p = p_1 e_1 + p_2 e_2 + p_3 e_3 + o$

- > Un riferimento **cartesiano** è dato da un riferimento la cui base è ortonormale
- > Un riferimento è **destrorso** se lo è la sua base.



TRASFORMAZIONI E MATRICI

In 2D e 3D ci interessa modellare spostamenti e deformazioni di oggetti. Dobbiamo applicare ai punti trasformazioni geometriche quali:

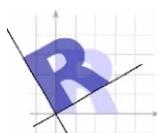
- **Traslazione**



$$T_t(\mathbf{p}) = \mathbf{p} + \mathbf{t}$$

$$T_t^{-1}(\mathbf{p}) = \mathbf{p} - \mathbf{t}$$

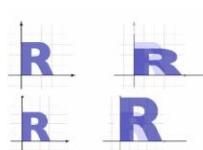
- **Rotazione**



$$R_\theta(\mathbf{p}) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} p_x \cos \theta - p_y \sin \theta \\ p_x \sin \theta + p_y \cos \theta \end{bmatrix}$$

$$R_\theta^{-1}(\mathbf{p}) = R_{-\theta}(\mathbf{p})$$

- **Scalatura** uniforme e non



$$S_s(\mathbf{p}) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} s_x p_x \\ s_y p_y \end{bmatrix}$$

$$S_s^{-1}(\mathbf{p}) = S_{\{1/s_x, 1/s_y\}}(\mathbf{p})$$

- **Riflessione**

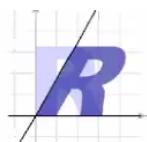
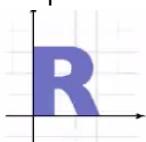


$$\begin{aligned} M_x(\mathbf{p}) &= \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} -p_x \\ p_y \end{bmatrix} & M_x^{-1}(\mathbf{p}) &= M_x(\mathbf{p}) \\ M_y(\mathbf{p}) &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} p_x \\ -p_y \end{bmatrix} & M_y^{-1}(\mathbf{p}) &= M_x(\mathbf{p}) \\ M_o(\mathbf{p}) &= \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} -p_x \\ -p_y \end{bmatrix} & M_o^{-1}(\mathbf{p}) &= M_o(\mathbf{p}) \end{aligned}$$

- **Deformazione affine**

Preservano parallelismo e distanza

Esempio: shear



$$Sx_a(\mathbf{p}) = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} p_x + ap_y \\ p_y \end{bmatrix}$$

$$Sy_a(\mathbf{p}) = \begin{bmatrix} 1 & 0 \\ a & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} p_x \\ ap_x + p_y \end{bmatrix}$$

Rappresentazione via matrici

Un modo comodo per rappresentarle è utilizzare il *prodotto di matrici*.

Una matrice è un array bidimensionale di elementi; per i nostri scopi sono numeri reali.

- » Una matrice può essere moltiplicata per uno scalare B

$$c_{ij} = \beta a_{ij} \quad \forall i, j$$

- » Due matrici possono essere sommate a patto che siano della stessa dimensione

$$c_{ij} = a_{ij} + b_{ij} \quad \forall i, j$$

- » Il prodotto di matrici è definito se il numero di colonne della prima matrice è uguale al numero di righe della seconda. Si tratta di *prodotto riga per colonna*.

$$c_{ij} = \sum_{l=1}^M a_{il} b_{lj}$$

- » La matrice *inversa* è M^{-1} tale che $MM^{-1}=M^{-1}M=I$

- » La matrice *trasposta* è $a_{ij}^T = a_{ji}$

- » Se una matrice è *ortonormale*, l'inversa è uguale alla trasposta.

Le matrici quadrate rappresentano quindi delle applicazioni lineari di uno spazio vettoriale in sé (formano un gruppo non abeliano). Tutte le applicazioni lineari di uno spazio vettoriale in sé sono esprimibili tramite matrici quadrate.

- » Matrici di *rotazioni* sono matrici per un vettore che lo ruota.

- » Es. asse Z

$$R = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- » Posso anche ruotare un sistema di riferimento, ovvero fare un *cambiamento di base*; in grafica sarà molto utile modellare gli oggetti (ovvero definire un oggetto via primitive geometriche espresse in un certo spazio; poi dovrò guardarle con una telecamera che avrà associato un sistema di riferimento della telecamera con una sua origine e dei suoi assi ortonormali).

Volendo rappresentare il punto di vista della telecamera, come minimo dovrò esprimere gli oggetti in quel sistema di riferimento.

Le matrici quadrate rappresentano applicazioni lineari di uno spazio lineare in sé stesso, e tutte le applicazioni lineari di uno spazio vettoriale in sé sono esprimibili con matrici quadrate.

Con matrici 2×2 e 3×3 scriviamo *rotazioni di vettori in 2D e 3D*, ma *non le generiche trasformazioni dei punti*. Per farlo dovremo un attimo cambiare le cose.

Trasformazione di un vettore:

$$Av = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} a_{11}v_1 + a_{12}v_2 \\ a_{21}v_1 + a_{22}v_2 \end{pmatrix}$$

Coordinate omogenee

Definisco coordinate omogenee delle quaterne definite rispetto al sistema di riferimento (e1,e2,e3,O) in questo modo:

$$\mathbf{v} = (v_1, v_2, v_3, 0) \quad \mathbf{p} = (p_1, p_2, p_3, 1)$$

In realtà i punti sono espressi da qualunque quaterna $\mathbf{p} = (\lambda x, \lambda y, \lambda z, \lambda w)$ con ultima coordinata diversa da 0, a patto che ci sia un *lambda comune* (ma posso dividere tutto per il quarto elemento e normalizzare la rappresentazione come $(x/w, y/w, z/w)$).

Trasformazione punti in coordinate omogenee

Utilizzare le coordinate generiche offre alcuni vantaggi:

- » Posso applicare operazioni fra punti e vettori in maniera “pulita” (ovvero con lo *stesso formalismo*)
- » Tutte le *trasformazioni affini* sono espresse da trasformazioni lineari o matrici fra oggetti in coordinate omogenee.
- » Posso applicare traslazioni, rotazioni e scalature con lo stesso formalismo: *traslazioni e notazioni sono esprimibili come moltiplicazioni per matrici* :)

$$T_t = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- > Traslazioni:
- > Scalatura
- > Cambio di riferimento

Con le matrici 4x4 possiamo mappare i sistemi fra di loro; questo equivale a una rotazione + traslazione.

$e_1=(1,0,0)$, $e_2=(0,1,0)$, $e_3=(0,0,1)$)

in un sistema arbitrario (O' , e'_1 , e'_2 , e'_3) sarà

$$\begin{pmatrix} e'_{11} & e'_{21} & e'_{31} & O_1 \\ e'_{12} & e'_{22} & e'_{32} & O_2 \\ e'_{13} & e'_{23} & e'_{33} & O_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- > Rotazioni:

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

!! Le rotazioni non sono commutative !!

Purtroppo, le rotazioni fatte così risultano scomode.

TRASFORMAZIONI PROBLEMATICHE: LE ROTAZIONI

Rotazione vs orientazione

La rotazione rappresenta un *cambio di orientazione*.

L’orientazione rappresenta la *posa* di un oggetto nello spazio

La relazione fra rotazione e orientazione è analoga a quella fra punto e vettore.

In 2D la rotazione è un angolo; in 3D esistono molte rappresentazioni equivalenti.

Teorema della rotazione di Eulero: lo spostamento di un corpo rigido su un punto fisso è una rotazione su un qualche asse.

- » Qualsiasi rotazione si può esprimere come rotazione di un angolo rispetto a un asse. Essa può essere rappresentata con:

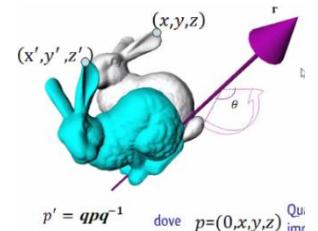
- > Angoli di Eulero e sequenza di rotazione XYZ
- > Asse/angolo
- > Quaternioni

Rotazioni asse-angolo

Esprimono la rotazione indicando un asse di rotazione (quindi un vettore con tre coordinate) e un angolo alfa di rotazione attorno a tale asse. Sono molto intuitive concettualmente, ma abbastanza scomode da usare e presentano interpolazione imprevedibile.

- » Ricordando le slide precedenti, posso descriverla semplicemente come matrice che mappa la nuova posizione in nel sistema di riferimento base

$$R(\alpha, \mathbf{r}) = \begin{pmatrix} (1 - c)r_1^2 + c & (1 - c)r_1r_2 - sr_3 & (1 - c)r_1r_2 + sr_2 & 0 \\ (1 - c)r_1r_2 + sr_3 & (1 - c)r_2^2 + c & (1 - c)r_2r_3 - sr_1 & 0 \\ (1 - c)r_1r_3 - sr_2 & (1 - c)r_2r_3 + sr_1 & (1 - c)r_3^2 + c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



$$p' = qpq^{-1} \quad \text{dove } p=(0, x, y, z)$$

Rotazioni come sequenza di rotazioni XYZ (Angoli di Eulero)

Una rotazione qualsiasi rispetto ad un asse passante per l'origine può essere decomposta nel prodotto di tre rotazioni rispetto agli assi coordinati; i tre angoli prendono il nome di angoli di Eulero.

La rappresentazione degli angoli di eulero non è univoca: a terne diverse può corrispondere la stessa rotazione : (E' tuttavia usato nei programmi di modellazione.

Funzionano bene con gli angoli di eulero, ed è una rappresentazione comoda quando si mappano su controlli come quelli degli aerei.

Problemi:

- » Rotazioni non univoche
Es: (z, x, y) [roll, yaw, pitch] = (90, 45, 45) = (45, 0, -45)
mandano entrambi l'asse x in direzione (1, 1, 1)
- » Gimbal Lock (blocco del giroscopio)
Alcune configurazioni sono problematiche
- » Interpolazione di rotazioni
Come calcolo il punto medio di una rotazione?

Quaternioni

Sono particolari *numeri complessi* che hanno 4 componenti anziché 2. Sono introdotti da Hamilton e hanno una loro algebra simile a quella dei numeri complessi.

$$w+ix+jy+kz \quad \text{where} \quad i^2=j^2=k^2=ijk=-1 \\ ij=k, jk=i, ki=j \\ ji=-k, kj=-i, ik=-j$$



Essi permettono di *passare da un'orientazione all'altra* in maniera semplice.

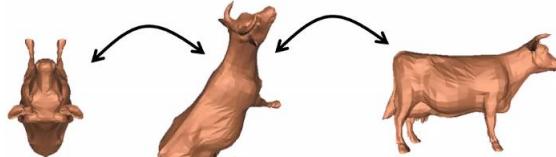
"Non spaventatevi, vediamo solo il formalismo". Il tono della voce del prof è come quello della mamma che ti dice "vieni qui, non ti faccio niente".

Questo è uno spazio quadridimensionale (quindi, che ha una dimensione in più rispetto alle rotazioni).

I *quaternioni unitari*, ovvero $w^2 + x^2 + y^2 + z^2 = 1$, possono *rappresentare le rotazioni* e possono essere rappresentati su una sfera. Si può esprimere come una coppia scalare e vettore. (w, v)

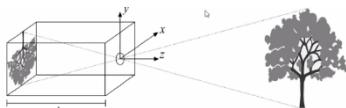
Vantaggi dell'uso dei quaternioni:

- » Operazioni più *rapide*

- » *Notazione più concisa* per le composizioni di rotazioni
 - » Numericamente *stabile* per piccole rotazioni
 - » Si può *convertire* all'espressione asse/angolo
 - » Principalmente è adatta ad *interpolazione* ed animazione:
 - A patto di dare un parametro che corrisponde alla velocità angolare - T rappresenta la velocità - posso usare la formula SLERP (spherical linear interpolation) che mi permette di rappresentare linearmente il passaggio tra due orientazioni. $\text{lerp}_t(q_1, q_2) = q_1 (q_1^{-1} q_2)^t + q_1 \exp(t \cdot \log(q_1^{-1} q_2))$
- L'interpolazione permette di fornire un'interpolazione che, intuitivamente, è il passaggio da una all'altra con meno strada.
- 
- Con gli angoli di eulero ci vuole più tempo, il risultato cambia in base agli assi considerati e le posizioni intermedie sono molto diverse da quella "più breve"!

Un programmazione è libero di scegliere quale formalismo utilizzare; la conversione è sempre possibile.

MACCHINA FOTOGRAFICA VIRTUALE



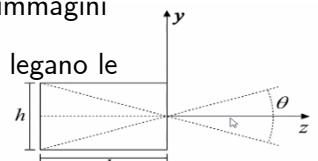
Macchina fotografica virtuale: Il modello che si usa per definire la "camera" è quello di *camera oscura* (camera *pinhole*).

La macchina fotografica virtuale è costituita da un parallelepipedo in cui la faccia anteriore presenta un foro di dimensioni infinitesime (pinhole camera) e sulla faccia posteriore si formano le immagini

Piazzando un sistema di riferimento sul buco posso facilmente trovare le equazioni che legano le coordinate xyz dell'oggetto a quelle delle proiezioni di quei punti sul piano.



Per semplificare il calcolo, si preferisce inserire un *piano immagine fra la scena e il centro della proiezione*; ne risulta il modello matematico della proiezione prospettica.



Sfruttando semplici ragionamenti sui triangoli simili posso calcolare la proiezione prospettica. In forma matriciale:

→ Non è quadrata!! Perché sto passando da uno spazio a un piano.

Proiezione prospettica e parallela

Proiezione prospettica: dal punto di vista geometrico, la proiezione è definita per mezzo di un insieme di rette di proiezione (i proiettori) aventi origine comune in un centro di proiezione, passanti per tutti i punti dell'oggetto da proiettare ed intersecanti un piano di proiezione.

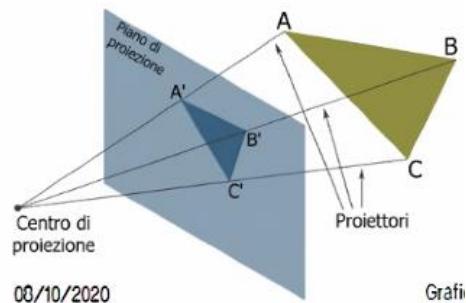
La proiezione di un segmento resta un segmento; dunque è sufficiente proiettare i segmenti.

Le proiezioni geometriche si dividono in:

» Proiezioni prospettiche

Distanza finita fra centro e piano di proiezione)

$$\mathbf{P}' = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix} = \begin{pmatrix} P_x \\ P_y \\ P_z/d \\ 1 \end{pmatrix} = \begin{pmatrix} d P_x/d \\ d P_y/d \\ 1 \\ 1 \end{pmatrix}$$

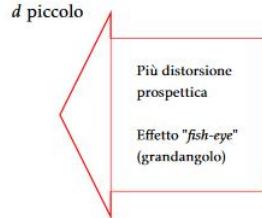
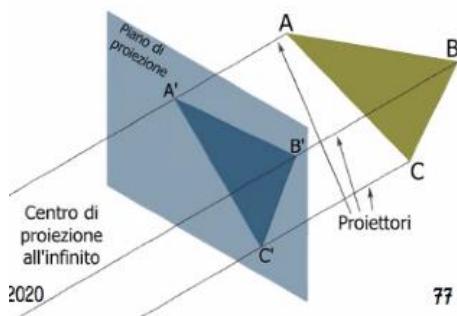


» Proiezioni parallele

Distanza infinita fra centro e piano di proiezione).

$$\mathbf{P}' = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Non corrisponde alla realtà perché ho un effetto di schiacciamento, ma in alcuni casi (oggetti molto lontani) si avvicina molto e mi permette di *semplificare notevolmente i calcoli*.



Trasformazioni prospettiche

In realtà, in ambito di renderizzazione non si usa davvero la proiezione prospettica ($3D \rightarrow 2D$) come vista fino ad ora, ma si preferisce la trasformazione prospettica. La differenza principale è che il risultato è comunque nello spazio 3D, dunque la matrice che rappresenta la trasformazione sarà una matrice quadrata 4×4 .

uale a d

$$\mathbf{P}' = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix}$$

Per passare alla rappresentazione 2D è sufficiente eliminare la z, che è sempre uguale a d.

In realtà la coordinata omogenea 2D che ricavo dall'applicazione della matrice è $P' = (P_x, P_y, P_z/d)$. L'operazione che trasforma in $P' = (P_x/d/P_z, P_y/d/P_z, 1)$ per avere la forma standard dei punti è la cosiddetta divisione. Prima della divisione i tre valori possono essere usati per rappresentare l'equivalenza dei diversi punti rispetto alla proiezione.

5 - Modeling

Ci sono molte tecniche di modellazione! Noi ci focalizziamo su modellazione 2D e rendering fotorealistico.

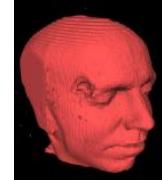
Potremmo farlo in molti modi:

- » Partizionamento spaziale (voxel)

La scena è suddivisa in una serie di cubetti; è una rappresentazione raster

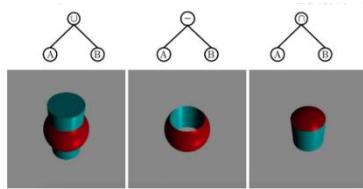
Non la si usa in quanto c'è grandissima complessità computazionale.

E' comunque molto diffusa in visualizzazione medica.



- » Primitive geometriche: definisco via mesh generate da poligoni

- > *Geometria costruttiva solida*:



E' molto utilizzata nel design CAD; si tratta di prendere delle forme primitive e costruire una scena usando operazioni booleane. Si usano le operazioni tipiche (unione, intersezione, differenza) che possono essere descritte tramite un albero. Ciascun nodo dell'albero contiene una delle tre operazioni e ciascuna foglia contiene una primitiva.

Per noi non è ottimale perché non è comoda per la trasformazione della scena in immagine.

- » Campionamento di punti sulle superfici (nuvole di punti)

Un modo per generare modelli 3D per mondi virtuali è acquisire dal vero. La scansione 3D, oggi tecnologia matura con diverse tecnologie Laser, luce strutturata, visione computazionale. Gli scanner (esattamente come le macchine fotografiche in 2D) di fatto non acquisiscono un modello del mondo, ma campionano la geometria (con eventuali attributi, es. colore) in una serie di punti discreti: si parla di nuvole di punti (point clouds)

- » Funzioni parametriche

Definiamo superfici lisce che definiscono qualunque forma; è l'equivalente delle splines in 2D.

Alcuni esempi sono *Bézier*, *NURBS*, *splines*...

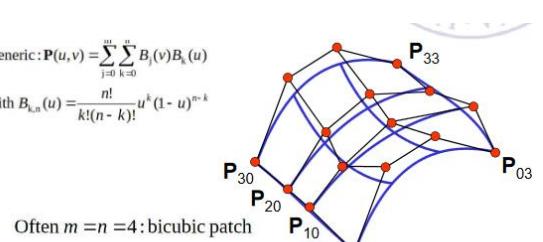
- > Esempio: *Superficie di Bézier* [[video](#)]

Blocco tutti i parametri e definisco univocamente una superficie liscia. Si usa parecchio in design grafico perché con un numero limitato di punti consente di definire superfici lisce.

Si usa anche in modellazione artistica, ma poi quando si fa rendering si passa in rappresentazione poligonale.

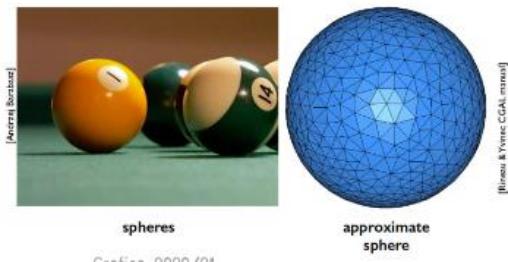
$$\text{Generic: } \mathbf{P}(u,v) = \sum_{j=0}^m \sum_{k=0}^n B_j(v) B_k(u)$$

$$\text{with } B_{k,n}(u) = \frac{n!}{k!(n-k)!} u^k (1-u)^{n-k}$$



Ci concentreremo sul *modeling poligonale* (*modellazione basata su approssimazione poligonale degli oggetti o boundary representation*), in quanto è supportato da tutti i sistemi di rendering ed è molto efficiente per fare i calcoli.

MESH POLIGONALI



Uso poligoni definiti da vertici e spigoli, in quanto nella pipeline di rendering delle schede grafiche si lavora in genere solo con facce triangolari. In particolare, la struttura dati più utile è la maglia o mesh di poligoni, che alla fine è un insieme di poligoni e triangoli che approssimano una superficie generica con una superficie fatta da poligoni che si intersecano tra loro sugli spigoli.

Upsides:

- + E' semplice, lineare a tratti (solo superfici piane)
- + Approssima bene superfici con qualunque topologia (indipendentemente da buchi e componenti connesse)
- + Permette di adattare localmente l'accuratezza
- + Il rendering è ottimizzato dalle CPU.
- + Si costruiscono *relativamente* facilmente con le nuvole di punti.

Downsides:

- » Esistono oggetti non modellabili con mesh di poligoni (fiamme, capelli, nuvole...)

DEF: un poligono è una *catena di punti chiusa in uno spazio 3D*. Generalmente essa è rappresentata da un vettore dove i vertici sono dati in ordine che mi dà anche il senso di percorrenza.

DEF: una mesh è un *insieme di poligoni le cui intersezioni sono esclusivamente vertici e spigoli dei poligoni*. I poligoni della maglia si dicono anche facce.

- > Connattività o topologia della mesh: indica come sono connessi i vertici. Si preferiscono superfici chiuse, ma di fatto questo non è sempre rispettato.
- > Bordo: il bordo di una mesh corrisponde agli spigoli che appartengono a un solo poligono. Una mesh senza bordo si dice chiusa.
- > Attributi: possiamo facilmente definire degli attributi per la mesh: posso codificarvi misure, colore, materiale, normali, coordinate UV map..

Il rendering lavora su mesh con poligoni triangolari, ma in modellazione capiterà di usare anche n-goni (ad esempio le quad mesh).

Manifoldness

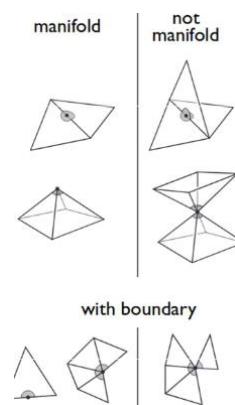
DEF: Manifoldness: ogni punto della mia mesh è isomorfo a un disco (=deve avere un intorno euclideo bidimensionale).



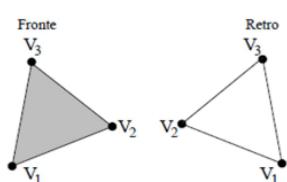
Questo non è scontato: la definizione di mesh consentirebbe situazioni non manifold!

La condizione di essere 2-manifold si traduce nei seguenti vincoli:

- » Uno spigolo appartiene al massimo a due triangoli
- » Se due triangoli incidono sullo stesso vertice allora devono essere raggiungibili l'uno dall'altro via percorso tra triangoli adiacenti, ovvero devono formare un ventaglio o un ombrello.



Orientazione



Il bordo della maglia consiste di uno o più anelli o loop. Se non esistono spigoli di bordo la maglia è chiusa.

DEF: L'orientazione di una faccia è data dall'ordine ciclico dei suoi vertici incidenti. L'orientazione determina il fronte e il retro della faccia.

Per poter dare un'orientazione definita devo avere tutte le facce orientate nello stesso modo. Questo non è sempre possibile:

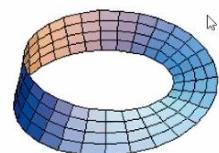
DEF: Una maglia si dice orientabile se esiste una scelta dell'orientazione delle facce che rende compatibili *tutte le coppie di facce adiacenti*.

DEF: Due facce adiacenti hanno orientazione compatibile se i due vertici del loro spigolo in comune sono in ordine inverso, ovvero *l'orientazione non cambia attraversando lo spigolo in comune*.

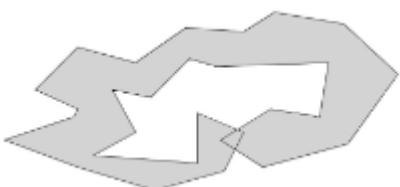
Di fatto le mesh che rappresentano oggetti sono sempre orientabili.

Non tutte le mesh 2-manifold sono orientabili:

- » Anello di Moebius: posso raggiungere qualsiasi punto della superficie da qualunque punto, quindi c'è un punto dove passo da una faccia "davanti" a una faccia dietro → orientazione non compatibile



Correttezza delle mesh



DEF: Una mesh è corretta se è *manifold, chiusa e orientata*. Deve avere anche *correttezza geometrica*, ovvero non avere autointersezioni.

Caratteristiche della mesh

- » Manifoldness
- » Risoluzione, ovvero numero di poligoni
 - !! Se la geometria è semplice, i triangoli extra sono inutili e costano risorse computazionali; esistono algoritmi di semplificazione automatica.
 - Nel tempo, il numero di poligoni gestibili dai software è ovviamente aumentato in maniera notevole
- » Curvature e normali, che influiscono sugli algoritmi di illuminazione.
- » Geometria: dove sono le superfici nello spazio
- » Topologia: come sono connesse



Memorizzazione

Per memorizzare abbiamo bisogno delle coordinate dei vertici, quindi 3 dati per ciascun vertice, e eventuali ulteriori attributi. Per lavorare sui triangoli ci serviranno le normali e le coordinate baricentriche.

Elementi di base:

- » Vertici: elementi 0-dimensionalni identificabili da tre coordinate; alle volte è utile associare anche altre caratteristiche (come il colore)
- » Spigoli: elementi 1-dimensionalni che rappresentano un segmento che unisce due vertici. Di solito non contengono altre informazioni.
- » Facce: poligoni bidimensionali formati da un certo numero di spigoli e vertici. Anche questi spesso contengono altre informazioni come il colore.

Normali

Da questi tre ricaviamo le normali come *prodotto vettore di vettori differenza di vertici*; per questioni di efficienza si pone come attributo sui vertici.

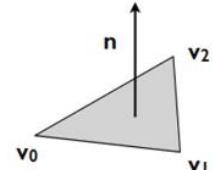
Ottengo anche l'area.

Coordinate baricentriche

I punti dei triangoli sono esprimibili anche come combinazione lineare delle coordinate dei vertici; i coefficienti sono detti coordinate baricentriche. Si usano per interpolare gli attributi.

$$\mathbf{n} = \frac{(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)}{|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)|}$$

$$A = \frac{|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)|}{2}$$



$$\mathbf{p} = w_0\mathbf{v}_0 + w_1\mathbf{v}_1 + w_2\mathbf{v}_2$$

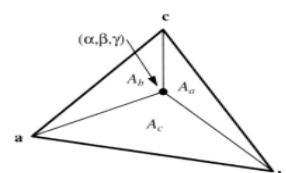
$$w_0 + w_1 + w_2 = 1$$

$$w_0 \geq 0 \quad w_1 \geq 0 \quad w_2 \geq 0$$

Algebraic view: linear combination

$$w_0 = 1 - w_1 - w_2 \Rightarrow$$

$$\mathbf{p}(w_1, w_2) = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v}_2 - \mathbf{v}_0)$$



Geometric view: relative areas

$$w_0 = A_0/A \quad w_1 = A_1/A \quad w_2 = A_2/A$$

Attributi delle mesh

Per rappresentare gli oggetti con le relative proprietà fisiche devo abbinare alla geometria dei valori di proprietà, che posso definire per vertice, faccia o wedge (vertice di faccia).

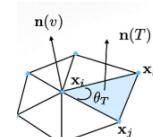
Alcuni esempi comuni sono:

- » Colore
- » Normali

E' fondamentale sapere quale è *l'esterno* della superficie, quale *l'interno* e quale *l'orientazione* locale. Non è calcolata costantemente per questioni di efficienza.

» *Normali sui vertici*: posso fare una media pesata delle normali e delle faccia. Come pesi posso usare pesi uniformi, le aree o l'angolo.

- » Coordinate texture



$$\mathbf{n}(v) = \frac{\sum_{T \in N_1(v)} \alpha_T \mathbf{n}(T)}{\left\| \sum_{T \in N_1(v)} \alpha_T \mathbf{n}(T) \right\|}$$

Materiali

Visto che dovremo creare le immagini illuminando gli oggetti e simulando la loro visione da parte di una telecamra, non basta memorizzare su vertici o facce il colore dell'oggetto – esso sarà *dinamico*!

Nel rendering si usa un algoritmo di shading che determina il colore in base alle proprietà del materiale, alle superfici (normali) e all'illuminazione incidente. Useremo un algoritmo di shading per il calcolo dell'immagine a partire da questi dati.



Texture mapping

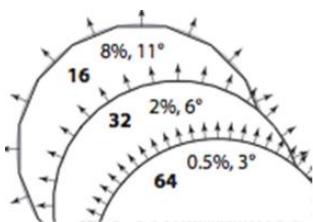
Si tratta di un trucco per limitare la complessità della mesh. Definisco una mappa con attributi e assegno le coordinate su tale immagine sui vertici della mesh

Per applicare l'attributo sui triangoli lo calcolo sui vertici in base alla mappatura, e poi si interpola tutto il triangolo. In questo modo posso applicare non solo il semplice colore, ma qualunque attributo – come per esempio perturbazioni della forma che creano un aspetto ruvido!

Le coordinate texture prevedono di associare coordinate u,v, nell'intervallo [0,1] a ciascun vertice, mappando i vertici in 3D e i pixel in 2D. Assegnare la mappa corrisponde a srotolare la superficie su un piano.. il che non è banale – esistono algoritmi e tool che se ne occupano, ma spesso si decade in discontinuità e distorsioni.

Inoltre, è molto comune mappare un oggetto “ripetutamente” a patto che la texture sia “tilable”, ovvero abbia un pattern che si ripete ai margini.

Approssimazione



L'errore di approssimazione è inversamente proporzionale al numero di facce. L'errore sulla posizione decresce di un fattore 4 ogni volta che raddoppiamo i segmenti, ma sono di un fattore 2 sulle normali!

Tendenzialmente vogliamo approssimare superfici lisce! Quindi sui vertici o sulle facce vorremmo approssimare la geometria differenziale (calcolo normali, piani tangenti, curvature...)

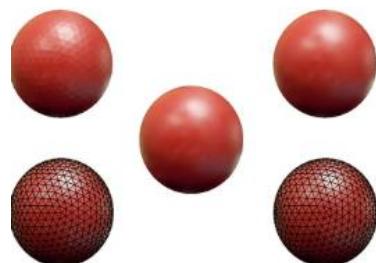
Il calcolo delle quantità differenziali è in genere influenzato dalla triangolazione: triangolo con angoli troppo piccoli risultano spesso problematici.

Shading

Il colore delle superfici illuminate dipende dall'angolo della direzione della luce incidente con le normali. Approssimare le normali con le normali delle facce fa sì che il colore sia costante sulla faccia, con effetto di tessellazione visibile :(

Soluzioni:

- » Aumentare la risoluzione della mesh (ew)
- » Calcolare le normali sui vertici e interpolare nel triangolo



Struttura dati della mesh

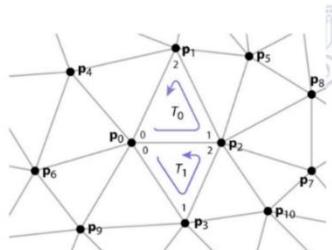
- » Vertici: danno informazioni di tipo posizionale
- » Spigoli: danno informazioni di tipo connettivo. Connettono i vertici introducendo un concetto di vicinanza e dando le informazioni di tipo topologico (=definiscono un grafo)
- » Facce: sono determinate una volta dati vertici e spigoli, quindi non hanno informazioni extra. Al più possono avere associati attributi ma è raro.

Ci sono vari modi per conservare informazioni su modelli. Nella applicazioni interattive le mesh dovranno essere caricate da file in memoria, e poi dalla RAM alla GPU. In GPU devono essere array di vertici con connettività, ma in memoria possono essere diverse.

Strutture semplici:

- » Lista di triangoli: potrei specificare tutte le facce della maglia come terne di triplette.. ma arriverei a duplicare le coordinate e rendere complesso il calcolo per trovare i vicini
- » Struttura indicizzata: lista dei vertici e lista delle face con puntatori ai vertici. E' la struttura normalmente utilizzata in OpenGL per il rendering, e permette di separare geometria e topologia.

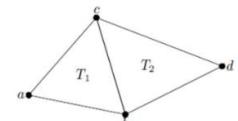
verts[0]	x_0, y_0, z_0
verts[1]	x_1, y_1, z_1
	x_2, y_2, z_2
	x_3, y_3, z_3
	\vdots
tInd[0]	0, 2, 1
tInd[1]	0, 3, 2
	\vdots



```
OFF
4 4 0
-1 -1 -1
1 1 -1
1 -1 1
-1 1 1
3 1 2 3
3 1 0 2
3 3 2 0
3 0 1 3
```



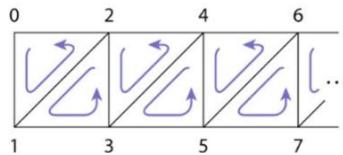
```
typedef struct {
    float x,y,z;
} vertice;
typedef struct {
    vertice* v1,v2,v3;
} faccia;
```



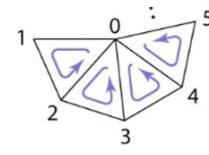
59

- » Strip e fans: comprimo parti di mesh eliminando le ripetizioni di indici

Triangle strip



Triangle fan:



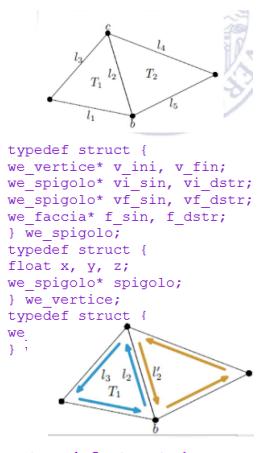
La struttura dati vista non è ottimale se devo cercare di trovare i vicini di punti, spigoli e triangoli. Quindi si creano strutture dati ad hoc per rendere veloci le ricerche.

- » Winged edge: aggiungo puntatori allo spigolo per rendere più semplice l'analisi delle incidenze. L'elemento base è lo spigolo (edge), con le sue due facce incidenti (wings)
 - Spigolo: 2 puntatori ai due vertici su cui incide e ai due spigoli uscenti da ciascun vertice
 - Vertice: puntatore a uno degli spigoli che incide su esso
 - Faccia: puntatore a uno degli spigoli che vi incide.

Ovviamente funziona solo con oggetti manifold.

- » Half edge: ogni spigolo viene diviso in due spigoli orientati in modo opposto.
 - Spigolo: contiene un puntatore al vertice iniziale, alla faccia a cui appartiene, al mezzo spigolo successivo (seguendo l'ordinamento) e al mezzo spigolo gemello.
 - Vertice: oltre a coordinate e attributi contiene un puntatore a uno qualsiasi dei mezzi spigoli che vi esce
 - Ogni faccia contiene uno dei suoi mezzi spigoli e altro, tipo la normale

Molto efficiente ed elegante!



```
typedef struct {
    we_vertice* v_ini, v_fin;
    we_spigolo* vi_sin, vi_dstr;
    we_spigolo* vf_sin, vf_dstr;
    we_faccia* f_sin, f_dstr;
} we_spigolo;
typedef struct {
    float x, y, z;
    we_spigolo* spigolo;
} we_vertice;
typedef struct {
    we_spigolo* spigolo;
} we_faccia;
```

```
typedef struct {
    he_vertice* origine;
    he_spigolo* gemello;
    he_faccia* faccia;
    he_spigolo* successivo;
} he_spigolo;
typedef struct {
    float x, y, z;
    he_spigolo* spigolo;
} he_vertice;
typedef struct {
    he_spigolo* spigolo;
} he_faccia;
```

La stessa applicazione può far uso di più di una struttura dati. La rappresentazione con la lista di vertici è tipica di file contenenti la geometria per gli oggetti, e si mappa direttamente con la struttura richiesta dall'HW grafico.

Le applicazioni grafiche che devono fare processing (ovvero modificare attivamente i modelli) in genere convertono i modelli in strutture più efficienti per gli algoritmi, come la half-edge.

MESH E RENDERING

Il rendering di oggetti descritti in mesh è reso semplice dal fatto che tutte le trasformazioni vengono eseguite sui vertici, ovvero si tratta di applicare trasformazioni affini su punti. E' vero anche per la

proiezione: per vedere come si proietta la forma di una maglia su un piano basta seguire la proiezione dei vertici.

La pipeline di rasterizzazione è stata creata e ottimizzata per modelli 3D sotto forma di mesh di triangoli, e hanno complessità computazionale proporzionale al numero di triangoli.

15/10/2020

Creazione di mesh

Modellazione artisica

Si modella manualmente utilizzando un software di modellazione

Modellazione procedurale

Si possono creare modelli in modo procedurale da una descrizione parametrica.

Scansione 3D

3D scanning: generalmente si catturano nuvole di punti.

La modellazione da scanner prevede di fare molto processing, con algoritmi studiati per consentire di processare questi modelli partendo dalla nuvola di punti:

Possono cercare di unire i punti generando dei triangoli (*ball pivot*) oppure prendere i punti e generare da questi una funzione implicita che mi dica quali punti sono esterni (what?) (*Poisson*). Si può partire anche da rappresentazioni volumetriche, con il partizionamento spaziale

Altre cose da fare:

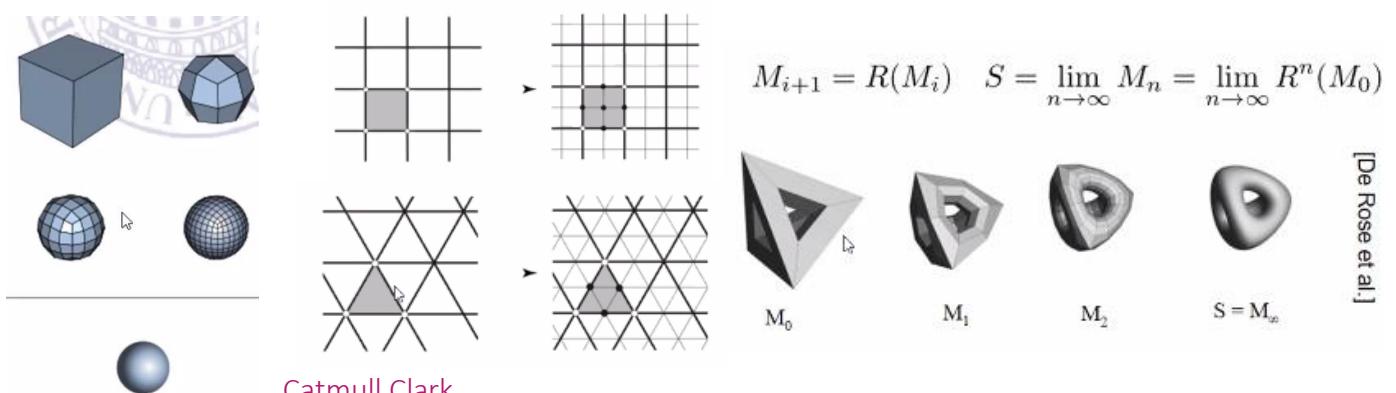
- » Smoothing: se parto da acquisizione dovrò eliminare il rumore
 - > Geometria computazionale permette di controllare se una mesh è corretta riguardo a topologia e geometria (le strutture delle mesh non garantiscono la correttezza)
- » Rimozione buchi e parti non manifold
- » Cambio la risoluzione: posso aumentare o diminuire i poligoni per ottimizzare il modello in base al suo utilizzo attraverso semplificazione (da un lato) e schemi di suddivisione (dall'altro).

Schemi di suddivisione

Permettono di avere un modo compatto per rappresentare una superficie con pochi poligoni e mi permette di passare a una rappresentazione ad alta risoluzione in live, senza doverla memorizzare

- » Aggiungo vertici cambiando la topologia
- » Muovo i vertici aggiunti per avere una struttura più liscia

Gli schemi tipici sono Catmull-Clark per mesh quad e loop per mesh triangolari



Ogni livello di suddivisione aumenta il numero di facce di un fattore 4.

- > Faccia: per ogni faccia viene creato un vertice faccia media di tutti i punti della faccia.

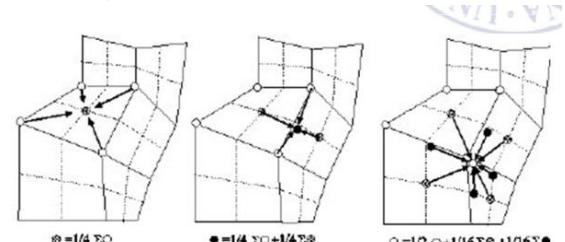
- > Spigolo: per ogni spigolo viene creato un vertice che è la media tra il centro del bordo e il centro del segmento realizzato con i punti della faccia delle due facce adiacenti
- > Vertici: sono aggiornati in base a
 - Vecchie coordinate
 - Media vertici faccia delle facce
 - Media centri degli spigoli a cui appartiene il punto.

Dato n numero facce cui appartiene un punto (n)

$$\text{new_coords} = (m1 * \text{old_coords}) + (m2 * \text{avg_face_points}) + (m3 * \text{avg_mid_edges})$$

Con $m1 = (n - 3) / n$; $m2 = 1 / n$; $m3 = 2 / n$

Nei vertici con più di 3 edges è un casino: ci saranno smoothing artifacts. Per evitarli servirà fare uno studio della topologia



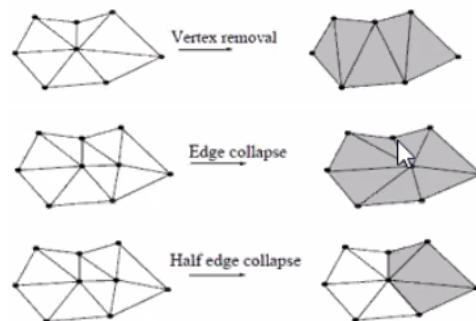
Semplificazione

Esistono algoritmi che invece diminuiscono il numero di vertici e spigoli.

- » **Quadratic edge collapse:** fonde due punti nel punto medio; elimino iterativamente gli edge che cambiano meno la geometria.
- » **Vertex removal:** rimuove un vertice e triangola la cavità
- » **Edge collapse:** rimuove uno spigolo fondendo due vertici
- » **Half edge collapse:** uno dei due vertici rimane fermo. (trascina un vertice sull'altro mashedolli)
- » **Contrazione iterativa degli spigoli:** ad ogni iterazione viene eliminato per contrazione lo spigolo di costo minore, ed i costi dei vicini vengono aggiornati. I vari metodi differiscono per la metrica di errore impiegata.

Esempio di metrica:

$$\varepsilon(P_1P_2) = \frac{\|P_2 - P_1\|}{|n_l \cdot n_r|}$$



I vari metodi differiscono per la metrica impiegata; un esempio è Garland Heckbert

>

Metodo di Garland-Heckbert: procede per contrazione iterativa degli spigoli. A ciascun vertice v è associato un insieme di piani; inizialmente sono i piani definiti dai triangoli incidenti sul vertice. Dopo una contrazione, al nuovo vertice si associa l'unione dei vertici che sono stati contratti.

A ciascun vertice è associato un errore delta. Per calcolare l'errore di un vertice basta mantenere la matrice Q .

Il costo di uno spigolo è l'errore

$$(\mathbf{v}_1, \mathbf{v}_2) = \mathbf{v}^T(Q_1 + Q_2)\mathbf{v}$$

- » Come scelgo v ? la posizione che minimizza la delta.

Multirisoluzione

A volte può essere utile avere modelli a vario livello di dettaglio, da utilizzare selettivamente nel rendering

Modelli composti e scene

Nelle applicazioni grafiche interattive i modelli interagiscono e quindi si muovono. Il moto rigido di singoli oggetti è facile da modellare e programmare; per oggetti composti di parti rigide la soluzione è quella di creare modelli gerarchici. Il moto delle foglie è la composizione delle trasformazioni geometriche date dalle matrici che si incontrano nella discesa del grafo. Le matrici descriveranno la posizione rispetto al nodo genitore.

Questo si realizza con:

- » Rigging – *inserimento dello scheletro*

Lo scheletro viene mosso in base alla cinematica, modellata da dati reali o motion capture.

Occorre definire:

- > Bones
- > Joints
- > DOFs (gradi di libertà)
- > Limits (limiti di variabilità)

Formalmente possiamo definire la posa come vettore di N numeri che mappa i parametri che controllano i gradi di libertà dello scheletro.

- » Skinning - *movimento della "pelle" in funzione dello scheletro*

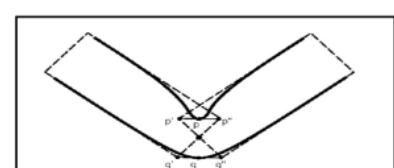
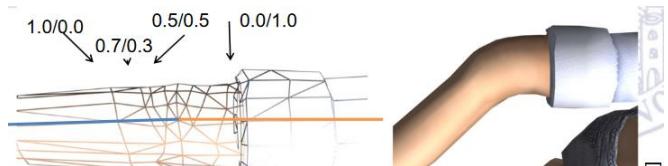
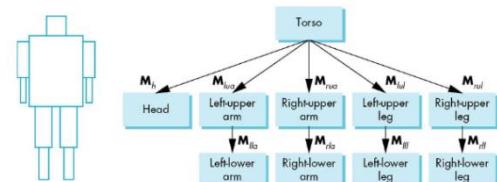
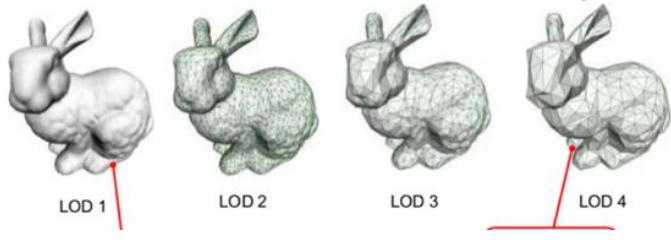
Il moto dei nodi delle mesh viene determinato dai punti vicini dello scheletro, con una somma pesata degli spostamenti. I pesi sono predeterminati.

Limiti:

- » Non rappresenta i reali movimenti causati dai muscoli
- » Non banale da controllare
- » Rischio artefatti se il rigging non è fatto ad opera d'arte

1. Calcola le matrici Q per tutti i vertici della maglia.
2. Per ciascuno spigolo $(\mathbf{v}_1, \mathbf{v}_2)$ calcola la posizione ottima per il vertice dopo l'ipotetica contrazione .
- L'errore $\mathbf{v}^T(Q_1 + Q_2)\mathbf{v}$ è il costo associato allo spigolo.
3. Costruisci uno heap con gli spigoli e chiave pari al costo.
4. Rimuovi lo spigolo $(\mathbf{v}_1, \mathbf{v}_2)$ di minor costo dalla cima dello heap, effettua la contrazione $(\mathbf{v}_1, \mathbf{v}_2) \leftarrow \mathbf{v}$
5. $Q_1 = Q_1 + Q_2$ e aggiorna i costi degli spigoli incidenti in \mathbf{v}_1
6. Ripeti da 4.
- Bisogna prestare attenzione a non creare incoerenze nella maglia, come p.es. il fold-over.

$$\Delta(\mathbf{v}) = \sum_{\mathbf{p} \in \Pi(\mathbf{v})} (\mathbf{p}^T \mathbf{v})^2 = \sum_{\mathbf{p} \in \Pi(\mathbf{v})} (\mathbf{v}^T \mathbf{p})(\mathbf{p}^T \mathbf{v}) = \sum_{\mathbf{p} \in \Pi(\mathbf{v})} \mathbf{v}^T (\mathbf{p} \mathbf{p}^T) \mathbf{v} = \mathbf{v}^T \left(\underbrace{\sum_{\mathbf{p} \in \Pi(\mathbf{v})} \mathbf{p} \mathbf{p}^T}_{Q} \right) \mathbf{v}$$

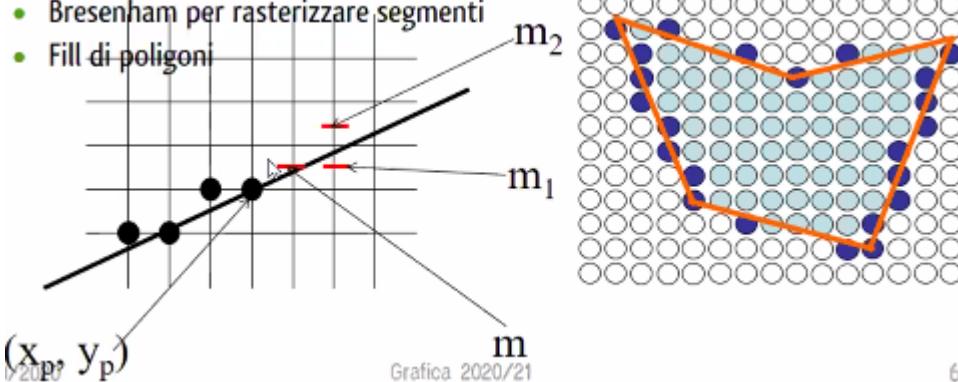


6 - Rendering

Chiamiamo rendering la parte che consiste nel calcolare l'output su display della visione della scena. Il tipo di rendering dipende dall'applicazione e dal modello geometrico del mondo utilizzato:

» Raster 2D

- > Parte da applicazioni di video o image editing
- > Da vettoriale a raster
 - Bresenham per rasterizzare segmenti
 - Fill di poligoni



Grafica 2020/21

6

Non è semplice: bisogna risolvere i problemi di aliasing e le scalettature.

» Vettoriale 2D

- > Animazione 2D o interfacce grafiche

» Vettoriale 3D

- > Tipico nelle applicazioni di visualizzazione scientifica, cinema...

» Raster 3D

- > Visualizzazione medica, per esempio.

Inoltre, i modelli del mondo possono essere:

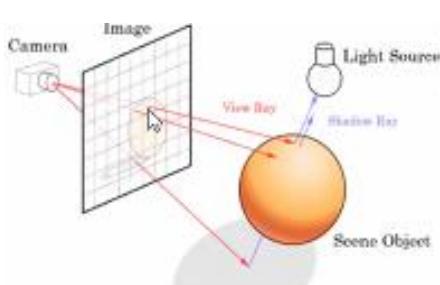
» Non interattiva

» Interattiva

- > La resa sullo schermo deve essere fatta in modo che l'utente possa interagire in tempo reale, quindi la resa deve essere fatta in un tempo breve ($<10^{-1}$)

Fotorealismo

Il rendering che ci interessa di più è quello fotorealistico di grafica 3D. E' una simulazione della fisica e della luce del mondo reale, ma si ricorre a delle semplificazioni del calcolo e della realtà

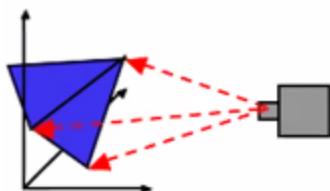


Il fotorealismo vuole simulare tutto ciò che potremmo catturare con una telecamera o con i nostri occhi: dobbiamo prendere i nostri oggetti e guardarli attraverso una telecamera.

Idealmente faccio *raycasting o raytracing* (anzi, il percorso inverso): simulo per ciascun pixel un raggio che seguiamo nella scena fino alla sorgente luminosa. Dovremmo simulare tutti i fotoni che attraversano la scena per ciascun punto calcolo lo shading che arriva dalla sorgente.

Questo è quello che fanno le tecniche più avanzate; normalmente si attuano delle versioni semplificate. Il problema di questa versione è che non si calcola la luce che arriva da eventuali altre direzioni (ad esempio la luce riflessa o rifratta), ovvero manca l'*illuminazione globale*.

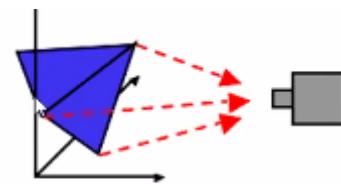
PARADIGMI DI RENDERING:



Ray casting:

Project image samples backwards

Si considerano i pixel dell'immagine da creare e si considera un raggio che parte dal centro ottico della telecamera e passa per ciascun pixel. Si trova l'intersezione con il primo oggetto nella scena e lì si calcola il colore da assegnare (shading). Posso considerare raggi riflessi e rifratti e iterare, sommando il colore (ray tracing). I motori di rendering moderni usano path tracing: si simula la fisica integrando moltissimi raggi che colpiscono il pixel (sempre tracciati all'indietro, ma anche bidirezionalmente) e risolvendo l'equazione del rendering



Rasterization:

Project geometry forward

Si considerano gli oggetti modellati della scena, per ognuno si calcola la proiezione sulla griglia dell'immagine e su tutti i pixel che intersecano questa proiezione si cerca di calcolare il colore da assegnare (shading).

SHADING

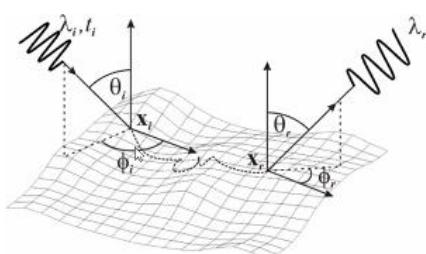
In entrambi i paradigmi dobbiamo calcolare il colore da assegnare al pixel.

Non si tratta di applicare semplicemente un colore associato ai triangoli o agli oggetti in qualche modo. Il termine shading si usa proprio perché si tratta di riprodurre l'effetto chiaroscuro.

Interazione luce-materia

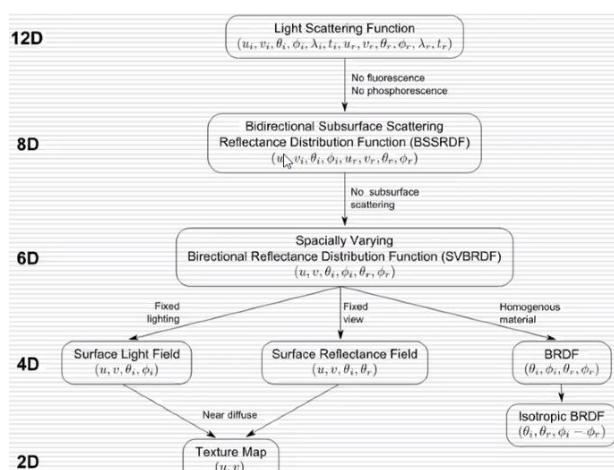
Non posso assegnare un colore fisso agli oggetti, poiché esso cambierà e andrà modellato sulle proprietà del materiale, che simula quelle del materiale reale.

L'interazione luce-materia è molto complessa!



- » Ottica geometrica: i raggi luminosi percorrono linee rette.
- Modello **emissione, riflessione, assorbimento, rifrazione**.
- » Ottica ondulatoria: permette di modellare **interferenza** e **polarizzazione**
- » Ottica quantistica: permette di modellare **fluorescenza** e **fosforescenza**.

Noi ci affidiamo alla geometria <3



Diventa molto difficile da modellare!

La riduzione a 4 parametri (**BRDF**) è quella tipicamente usata in grafica, anche se poi con trucchi vari si cerca di simulare, se necessario, quegli effetti che ho buttato via.

Nella semplificazione al calcolatore buttiamo via la **fisica dello spettro** della luce naturale; rappresentando tutto con tre fasci in RGB butto via vari fenomeni di composizione degli spettri.

Radiometria e grandezze utili

La radiometria si occupa di radiazioni estese sull'intero intervallo delle possibili lunghezze d'onda e non considera gli effetti sull'osservatore.

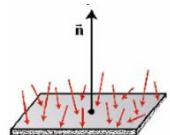
Essendo interessati ad un modello oggettivo, definiremo le grandezze radiometriche. Assumendo che non ci sia interazione tra le diverse lunghezze d'onda, si può misurare l'energia indipendentemente per un certo numero di lunghezze d'onda campione che servono a rappresentare l'intera distribuzione spettrale. Di solito se ne usano 3, per motivi legati al sistema visivo umano, corrispondenti al rosso, verde e blu (RGB).

La luce è un fenomeno fisico ben determinato, e ci sono grandezze fisiche che ci servono per capire il fenomeno :) Tutte le grandezze fisiche, dunque, le figuriamo alla singola onda (quindi, *facciamo finta che i vari fasci hanno tutte le stesse caratteristiche*).

» Potenza

È la velocità a cui la luce viene emessa o assorbita.

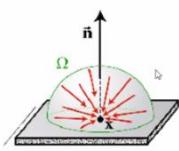
$$P = dQ/dt$$



» Irradianza / radiosità:

$E(x) = \frac{d\Phi}{dx}$ *E(x) è il rapporto tra il flusso ricevuto da un elemento infinitesimo di superficie in x e la sua area dx; è la potenza per unità di area.*

$B(x) = \frac{d\Phi}{dx}$ *B(x) il rapporto tra il flusso emesso da un elemento infinitesimo di superficie in x e la sua area dx; è la potenza per unità di area.*



Luce che arriva in un'unità di superficie (e non in un punto senza dimensioni!), quindi caratterizziamo il rapporto fra il flusso ricevuto in superficie e la sua area. Irradianza e radiosità sono la stessa grandezza (una densità superficiale di flusso) e si misurano in [W/m²]. La differenza è che l'irradianza è energia ricevuta, la radiosità è energia emessa. In entrambi i casi, l'energia ricevuta/emessa si considera da/verso tutte le direzioni.

» Angolo solido sotteso: *È pari all'area della proiezione dell'oggetto su una sfera unitaria in P.*

Il termine $dx \cos\theta$ rappresenta l'area proiettata di dx lungo la congiungente y e dx.

$E(x) = \frac{d\Phi}{dx} = \frac{I \cos\theta}{r^2}$ Se poniamo in y una sorgente di luce puntiforme con intensità radiante I, allora l'irradianza nel punto x vale come la formula

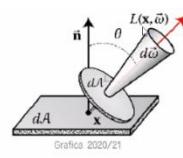
» Radianza. *L(x, \omega)* nel punto x in una direzione ω è la densità superficiale della intensità radiante in x lungo la direzione ω , considerando l'area della superficie proiettata. Intui

La radianza nel punto è la densità superficiale dell'intensità radiante lungo una direzione considerando l'area della superficie proiettata.

Intuitivamente è la luce intorno a una direzione. Per rappresentare la direzione si rappresenta l'angolo solido: la luce emessa sarà una frazione di un angolo solido

La radianza da x verso y è uguale a quella che raggiunge y dalla direzione di x : non si attenua con la distanza.

$L(x, \omega)(\omega \cdot n) = \frac{d^2\Phi}{d\omega dx}$ Non si considera è l'attenuazione della luce in grande distanza; noi usiamo un modello tale per cui l'energia trasportata rimane sempre invariata.



$$L(x, \omega) = \frac{dI(\omega)}{d\omega (\omega \cdot n)}$$

Scattering

Dovremo calcolare la riflessione dei raggi e i contributi delle riflessioni dei raggi provenienti da tutte le direzioni. Il surface scattering va bene per le superfici comuni; per quelle più complesse servono subsurface scattering (superficie traslucide tipo pelle) e volumetric scattering (fumo, nebbia...)

EQUAZIONE DEL RENDERING

Se voglio sapere tutta la luce che arriva in un punto devo fare la *somma di tutti i raggi* che rimbalzano in un certo punto. Aggiungendo la radianza emessa ottengo l' equazione del rendering o equazione della radianza, che esprime la radianza totale che il punto x lascia in una certa direzione ω .

$$L(x, \omega_0) = L_e(x, \omega_0) + \int_{\Omega} p(x, \omega_i, \omega_0) L(x, \omega_i) (\omega_i \cdot \vec{n}) d\omega_i$$

eq. rendering

BRDF (modello di materiale)

Somma in tutte le direzioni

PER CAPIRLA BENE, ILLUMINANTE QUESTO VIDEO:

https://www.youtube.com/watch?v=eo_MTI-d28s

Il problema è che dentro il termine a destra – ovvero quello che dovrei sommare – ci sono anche altre incognite: non so quanta luce arriva da ciascuna direzione!

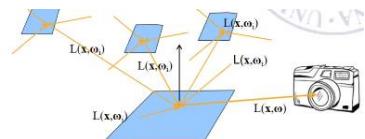
Nei programmi che siamo abituati ad usare sappiamo che definiamo anche delle sorgenti luminose. Una prima approssimazione consiste nel definire sorgenti semplici e puntiformi.

Si può fare (e si dice illuminazione locale), ma non considera il fatto che in scene relativamente complesse la luce può rimbalzare anche su altre parti della scena: ci sarebbe una *ricorsione infinita* (ma obv si approssima).

Ci sono due problemi da risolvere:

- » Modellare la ricorsione della luce che arriva
- » Calcolare la BRDF di ogni tipo di materiale

Per rendere trattabile il problema si applicano delle semplificazioni.



1 – Problema della ricorsione

Si vuole riuscire a calcolare un numero potenzialmente infinito di riflessioni.

» Algoritmi locali

I modo che si è utilizzato all'inizio – quando non c'erano le risorse di oggi – è stato di eliminare del tutto la ricorsione eliminando i rimbalzi sulla scena. LOL WEAK FLEX

Considero la radianza entrante solo lungo le direzioni corrispondenti ai raggi provenienti direttamente dalle sorgenti luminose.

Dà un risultato molto poco realistico, ma tutto sommato utilizzabile.

» Algoritmi globali

Tengo conto della natura ricorsiva dell'equazione, ma trascuro alcuni fenomeni di interriflessione per rendere il problema trattabile. Comunque:

- > Ray tracing: nella forma più semplice solo per le riflessioni speculari path tracing più accurato. Inizialmente funziona solo fra superfici a specchio, ma ora si è evoluto in path tracing che permette di modellare statisticamente molti più raggi.
- > Radiosity: modella solo le interriflessioni fra superfici diffuse; È un'applicazione del metodo degli elementi finiti per risolvere l'equazione di rendering di scene composte di superfici perfettamente diffuse.

2 – Modellazione dei diversi materiali

Si vuole riuscire in tempi ragionevoli a calcolare tutti i materiali.

Ho degli strumenti che mi permettono di *simulare* i diversi materiali (riflettometri) per vedere quanta luce viene riflessa. Nella pratica non viene fatto perché dovrei fare tantissime rilevazioni; si preferiscono funzioni parametriche che riproducono comportamenti simili a quelli reali.

Devo introdurre delle BRDF che simulino bene questi comportamenti dei materiali!

Materiali opachi

la luce penetra all'interno e viene *ridiffusa uniformemente* in tutte le direzioni.

La BRDF si semplifica notevolmente dato che non ho riflessione fra oggetti.

» Aspetto *matte*

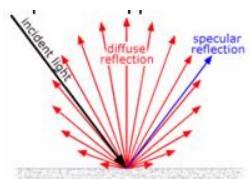


Materiali lucidi

la luce *rimbalza* in superficie.

Vedo delle parti più luminose, e così come nello specchio esse si muovono quando mi muovo.

» Aspetto *glossy*

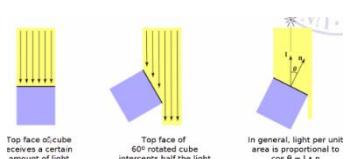


Materiali "misti"

Molti materiali hanno comportamenti intermedi.



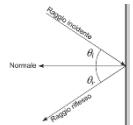
Diffusione pura (lamberdiana):



Dipende *solo dalla direzione di incidenza* della luce, e uso la legge del coseno (l'intensità della luce riemessa dipende dalla quantità di luce che arriva in superficie e viene riemessa. In base all'angolo formato con la superficie.

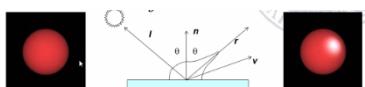
Legge di Fresnel

Rappresenta lo specchio perfetto: tutta l'energia che arriva viene riemessa in una direzione singola, tale per cui l'angolo di incidenza è uguale all'angolo di riflessione. Vale solo per gli specchi perfetti.



Riflessione speculare

L'idea che funziona – pur non avendo basi fisiche – è quella di modellare la riflessione speculare con un comportamento simile a quello dello specchio ma modificato: aggiungo alla componente diffusiva una riflessione speculare che non esiste soltanto nella direzione della riflessione speculare pura, ma diminuisce proporzionalmente alla distanza angolare dalla direzione di riflessione speculare pura: la luminosità è massima dove la normale alla superficie è parallela alla luce incidente, e diminuisce tutto attorno.



A questa aggiungo un highlight vero e proprio che funziona come funzione speculare pura.

MODELLO DI ILLUMINAZIONE

BRDF

Simulare tutte le interazioni fisiche è troppo dispendioso in termini computazionali: quello che si fa è modellare il comportamento su soli *quattro parametri* che esprimono la *direzione di ingresso e di uscita* della luce, attraverso una funzione chiamata Bidirectional Reflectance Distribution Function.

$$\rho(x, \omega_i, \omega_r) = \frac{L_r(x, \omega_r)}{L_i(x, \omega_i)(\omega_i \cdot \mathbf{n})d\omega_i}$$

La quantità che rappresenterà il nostro raggio sarà la *radianza* (densità superficiale dell'intensità radiante in x lungo la direzione omega considerando l'area della superficie proiettata).

Calcoliamo il rapporto tra la radianza in una direzione e la radianza modulata dal fattore di schiacciamento .

Modello di illuminazione di Phong

È l'esempio di come sia stato introdotto lo shading in grafica 3D, ed è stato introdotto nei primi anni 70.

È stato introdotto in un contesto di illuminazione locale – anche se poi come funzione è utilizzabile anche in un contesto globale in cui calcolo iterativamente le riflessioni.

È tutt'ora utilizzato nella pipeline di rasterizzazione.

Semplifica il modello fisico con una serie di semplificazione:

- » Solo sorgenti puntiformi
- » Solo locale (no riflessioni reciproche)
 - > Trucchetto per simulare l'effetto della *illuminazione globale* senza calcolarla
- » Non calcola la rifrazione: non calcola i materiali *trasparenti* o semitrasparenti
- » Non calcola le ombre
- » Luce considerata come RGB
- » Intensità luminosa = radianza (La assegnamo al raggio)
- » Funzioni parametriche che esprimono le componenti riflessiva e speculare
 - > *Componente diffusiva*
Si approssima la funzione di riflessione come una costante k_d dipendente dal materiale.

$$I_d^{out} = I k_d \cos \theta \doteq I k_d (\mathbf{n} \cdot \mathbf{l})$$

Si applica solo per angoli fra 0 e $\pi/2$ (= no se arriva da dietro la faccia). Devo illuminare e considerare valide solo le superfici che sono dalla parte giusta e nel giusto range!

- > *Componente speculare: formula di Blinn-Phong*

$$I_s^{out} = I k_s (\mathbf{n} \cdot \mathbf{h})^n \quad \text{dove } \mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|}$$

Ora si usano modelli più raffinati ma coincide bene col comportamento vero dei materiali. Poco intuitiva; viene da una formula empirica. Risultato: se n è alto l'highlight è stetto; altrimenti si allarga.

- !! le due n sono cose diverse: quella all'interno è la normale mentre l'altra un'esponente dello scalare.
- \mathbf{H} è l'halfway vector; è un vettore che divide a metà l'angolo fra \mathbf{l} e \mathbf{v}

- > *Componente ambientale*

Nel modello di Phong si è trovato un trucco per generare una simulazione dell'illuminazione globale, rinunciando a modellare le interriflessioni vere e proprie. Si introduce una *componente ambientale*: si assegna alle componenti di luce riflessa una certa luce costante. Aggiungo una costante che mi aggiunge un valore proporzionale alla luce. Posso aggiungere anche un parametro I_a che calcola questa costante in base alla luce ambientale presente nella scena.

$$I = I_a k_a$$

- > I_a = radiazione luminosa totale della scena
- > k_a = riflettività del materiale
- > I_a = costante per tutti i punti di tutti gli oggetti

Sommando tutti i contributi su ogni componente della luce ottengo

$$I^{out} = I_a k_a + I (k_d (\mathbf{n} \cdot \mathbf{l}) + k_s (\mathbf{n} \cdot \mathbf{h})^n)$$

BRDF di Phong

Non è basato su una simulazione fisica, ma funziona molto bene!

Ci permette di semplificare notevolmente la formula di partenza:

- » La luce emessa è sempre zero in quanto abbiamo introdotto le sorgenti di luce puntuali (esterne agli oggetti della scena).
- » L'elemento a destra nella formula sarà sempre zero tranne che nella direzione della sorgente puntiforme, quindi posso eliminare l'integrale e tenere solo quella direzione

Arrivo alla seguente forma:

$$L(x, \omega) = L(x, \omega_L) \rho(x, \omega_L, \omega) (\omega_L \cdot \mathbf{n}) d\omega$$

Phong è dato dalla formula

$$I^{out} = I \left(k_d + k_s \frac{(\mathbf{n} \cdot \mathbf{h})^n}{(\mathbf{n} \cdot \mathbf{l})} \right) (\mathbf{n} \cdot \mathbf{l})$$

I_{out} corrisponde alla radianza uscente $L(x, \omega)$; posso dunque ricavare che la *BRDF di Phong* è

$$\rho(x, \omega_i, \omega_r) = k_d + k_s \frac{(\mathbf{n} \cdot \mathbf{h})^n}{(\mathbf{n} \cdot \omega_i)} \quad \text{dove } \mathbf{h} = \frac{\omega_i + \omega_r}{\|\omega_i + \omega_r\|}$$

Questo modo era usato intorno agli anni 70 ed è ancora parte della pipeline di rendering.

Miglioramenti apponibili

Nella grafica anni '70 si introduce questo modo che è tutt'ora usato, e in certi contesti persino sufficiente. Per avrere maggior fotorealismo bisognerà aggiungere altri effetti

- » Attenuazione della luce: assorbimento dei mezzi e *attenuazione in profondità*. Inserisco un semplice *fattore di attenuazione* con la distanza.
- » Emissione: simulo un materiale che *emette luce*. Il problema è che se poi non faccio i calcoli ricorsivi non ho illuminazione globale e la *componente emissiva non illumina effettivamente la scena*.

Nei programmi di modellazione scientifica ci si limita al modello di Phong, in quanto è abbastanza fedele e non è necessario avere un risultato fotorealistico.

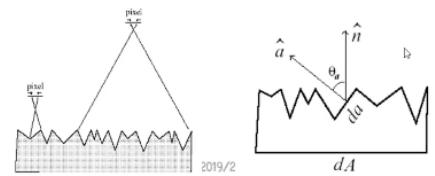
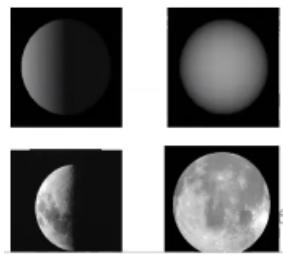
Limiti:

- » Il risultato è accettabile per alcuni programmi, ma non sufficiente per modellazioni avanzate.
 - > Nella realtà, i materiali opachi non rispettano la *legge del coseno*
 - > Il *colore* non è davvero indipendente dal viewpoint
 - > Gli *highlight* si comportano diversamente e possono persino variare nel colore.
 - > Negli oggetti opachi non vediamo mai bordi completamente *scuri e neri* come succede con il modello lambertiano (as an artist i can confirm lol)

Modelli basati su microfacet

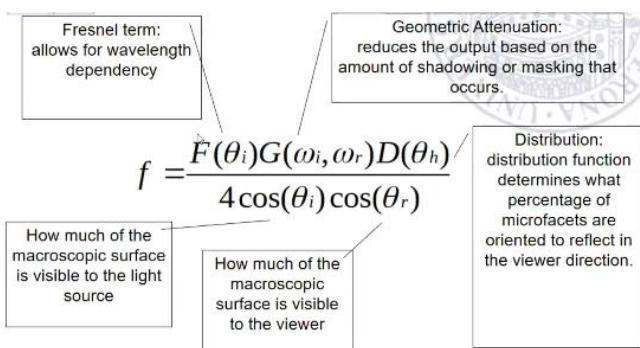
Tentano di fare una *simulazione fisicamente plausibile*. Ci si basa su microfaccette che non rendono il modello direttamente liscio, ma hanno delle inclinazioni

statisticamente determinabili che definiranno particolari comportamenti di riflessione.



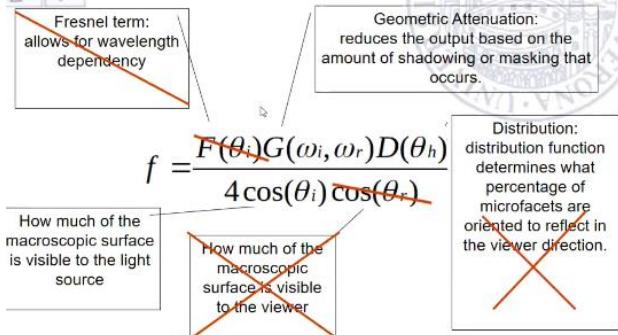
Altri modelli aggiungono altri parametri alle formule per poter esprimere maggiori nuance nelle immagini:

» Torrance-Sparrow BRDF



- > Fresnel: cambio di colore
- > Attenuazione: fenomeno di masking nelle riflessioni, in base alla distribuzione di faccette statistica.
- > Distribution: percentuale di faccette orientate verso la vista
- > Fattori globali: es. quanta superficie è visibile a osservatore e luce

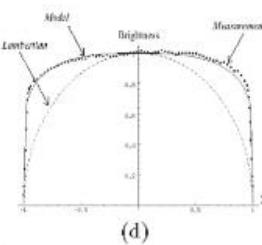
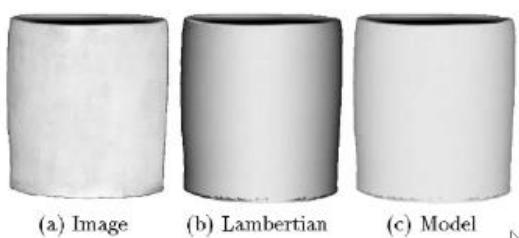
» Oren-Nayar

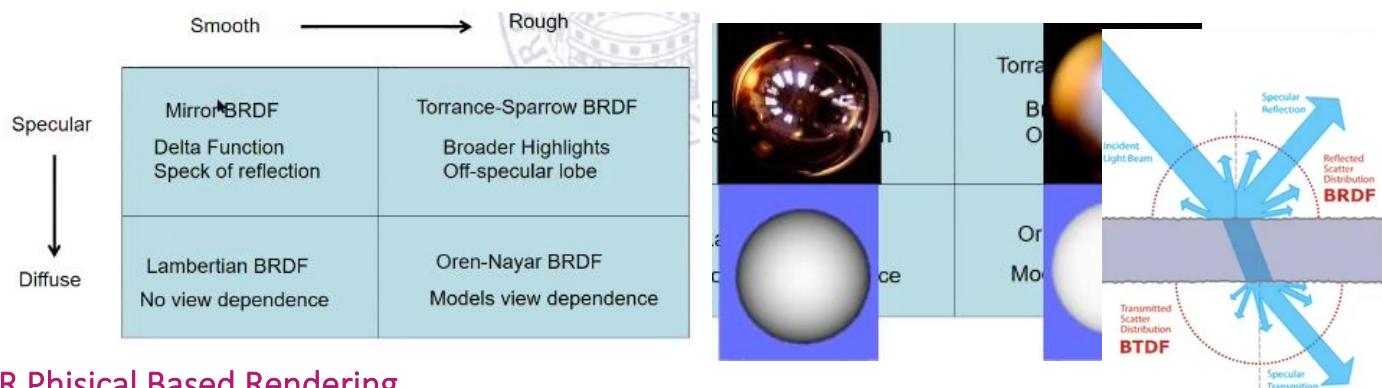


» Solo diffusivo: eliminiamo la visibilità a livello macroscopico e lasciamo solo l'attenuazione geometrica

» È quella che definisce l'appiattimento labertiano.

Confronto: con i modelli più complessi riesco a modellare la vera ceramica. I modelli usati in Blender hanno una parametrizzazione che aggiunge più info rispetto a "speculare/diffuso": usano quei modelli li più altri parametri – come la rugosità, che cambia la dimensione delle faccette. Con questi modelli definisco comportamenti più complessi, come la vera ceramica.





PBR Physical Based Rendering

Nel rendering fotorealistico si tende a sostituire la modellazione di Phong con funzioni molto più complesse derivanti dalla fisica; inoltre, si usano modelli physical-based, introducendo:

- » Vincoli relativi alla conservazione dell'energia e della componente trasmessa
- » Microfaccette
- » Traslucenza
- » Se possibile, illuminazione globale

In blender abbiamo due modelli di rendering; uno adotta l'approccio di rasterizzazione tipico delle schede grafiche che usa i trucchetti senza usare la vera illuminazione globale; il secondo è Cycles, che fa il path tracing e cerca di risolvere davvero le equazioni del rendering.

Riassumendo: siamo in grado, anche solo col modello di Phong, di risolvere il problema del rendering via ray casting, anche senza ricorsione :D

IMPLEMENTAZIONE DEL RENDERING

Ci sono due principali approcci:

- » Uno accurato che ricostruisce i raggi al rovescio
 - > Luce globale ammessa
- » Uno tipico delle applicazioni real time che lavora sulle mesh ed è implementato in hardware (rasterizzazione, pipeline di rasterizzazione)
 - > Non è ovvio fare illuminazione globale.

RICOSTRUZIONE DEI RAGGI: RAY CASTING

L'implementazione è di fare una scansione di pixel dell'immagine e per ciascuno traccio un raggio; se interseca un oggetto della scena allora il colore del pixel sarà quello proiettato da quell'oggetto della scena. Posso calcolare il modello poiché so la normale, la sorgente luminosa e i parametri del materiale.

L'unica difficoltà è calcolare l'intersezione raggi-oggetti: problema del tutto geometrico. In passato non era implementato perché il calcolo è notevole (benché il ragionamento sia semplice).

Esempio: intersezione raggio-sfera

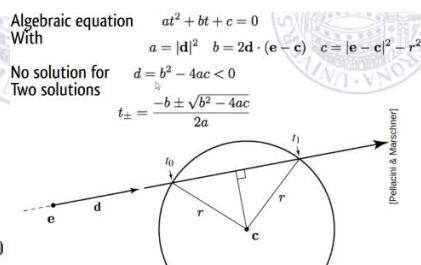
- Enforce point on a ray and on a sphere
- Leads to system of equations

$$\begin{cases} \mathbf{p}(t) = \mathbf{e} + t\mathbf{d} \\ |\mathbf{p} - \mathbf{c}|^2 = r^2 \end{cases}$$

- By substitution

$$\begin{aligned} (\mathbf{e} + t\mathbf{d} - \mathbf{c}) \cdot (\mathbf{e} + t\mathbf{d} - \mathbf{c}) &= r^2 \\ |\mathbf{d}|^2 t^2 + 2\mathbf{d} \cdot (\mathbf{e} - \mathbf{c}) t + |\mathbf{e} - \mathbf{c}|^2 - r^2 &= 0 \end{aligned}$$

punto



» Devo risolvere un sistema di equazioni di II grado; in base al discriminante so se e quante intersezioni ci sono.

Dovrei calcolare la normale: questo è semplice in quanto calcolato il punto mi basta seguire la direzione centro-

Esempio: interezione col triangolo

Every point on the triangle can be as

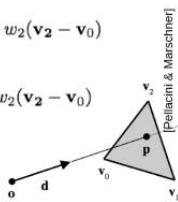
$$\mathbf{p}(w_1, w_2) = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v}_2 - \mathbf{v}_0)$$

Set up a system of equations

$$\begin{cases} \mathbf{p}(t) = \mathbf{e} + t\mathbf{d} \\ \mathbf{p}(w_1, w_2) = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v}_2 - \mathbf{v}_0) \end{cases}$$

By substitution

$$\mathbf{e} + t\mathbf{d} = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v}_2 - \mathbf{v}_0)$$



Linear system of 3 equations for 3 unknowns

$$\mathbf{e} + t\mathbf{d} = \mathbf{v}_0 + w_1(\mathbf{v}_1 - \mathbf{v}_0) + w_2(\mathbf{v}_2 - \mathbf{v}_0)$$

$$w_1(\mathbf{v}_0 - \mathbf{v}_1) + w_2(\mathbf{v}_0 - \mathbf{v}_2) + t\mathbf{d} = \mathbf{v}_0 - \mathbf{e}$$

$$\begin{bmatrix} \mathbf{v}_0 - \mathbf{v}_1 & \mathbf{v}_0 - \mathbf{v}_2 & \mathbf{d} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ t \end{bmatrix} = \begin{bmatrix} \mathbf{v}_0 - \mathbf{e} \end{bmatrix}$$

Solve with Cramer's rule

$$t = \frac{(\mathbf{v}_2 \times \mathbf{e}_1) \cdot \mathbf{e}_2}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \quad w_1 = \frac{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{v}_2}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1} \quad w_2 = \frac{(\mathbf{v}_2 \times \mathbf{e}_1) \cdot \mathbf{d}}{(\mathbf{d} \times \mathbf{e}_2) \cdot \mathbf{e}_1}$$

with $\mathbf{v}_2 = \mathbf{e} - \mathbf{v}_0$, $\mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0$, $\mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0$

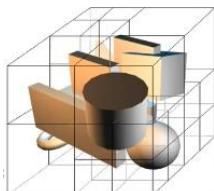
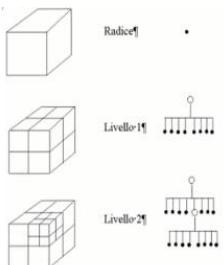
Check for constraints on barycentric coordinates and ray distance

$$t_{min} \leq t \leq t_{max} \quad 0 \leq w_1 \leq 1 \quad 0 \leq w_2 \leq 1 \quad w_1 + w_2 \leq 1$$

Riduzione della complessità

Concettualmente è molto semplice, ma avendo anche milioni di triangoli diventa estremamente oneroso: il costo è lineare nei vertici, che sono spesso milioni. Questo può essere ridotto in molti modi:

- » Evito di fare un ray-caster se posso farne a meno :D
- » Organizzo la struttura dati in maniera furba, per renderlo slineare



» *Octree*: divido lo spazio in partizionamenti/celle successive. Tutto lo spazio di un parallelepipedo (ad esempio) è *diviso ricorsivamente* finché in una parte ho un numero di primitive fissato. Così ho una *struttura dato che approssima bene il mio modello*.

Trovare l'intersezione avviene intersecando il cubo che racchiude la scena e scorrendo l'albero dei cubi all'interno. Solo le primitive che sono dentro il cubo intersecato saranno calcolate.

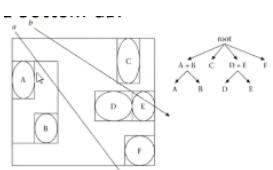
KD-tree

Divido lo spazio in modo che *ogni piano divida a metà il numero di oggetti rimanenti*.

L'attraversamento può avvenire sequenzialmente o ricorsivamente.

- » Tecniche come pruning, culling e broad-phase per diminuire il numero di primitive che hanno bisogno di essere controllate.

» *Volumi di contenimento*: Es. se so che i triangoli sono contenuti in una regione geometrica può grande, posso testare quella esterna anziché i molteplici triangoli interni. Posso usare alberi dei volumi di contenimento gerarchici: calcolo i volumi esterni e passo agli "interni" solo se necessari (= se ho avuto intersezione).



» *Partizionamento spaziale*: Divido lo spazio in celle

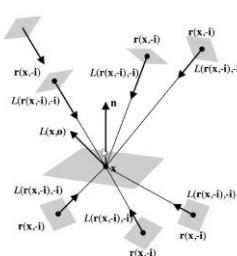
OMBRE

Sono molto facili da implementare nel ray casting; basta controllare se c'è occlusione tra punto sorgente e luminosa.

TRASPARENZA

Anche questa può essere introdotta facilmente prolungando i raggi quando si incontra una superficie trasparente. (Invece nella rasterizzazione è un bordello!)

SIMULAZIONE PIÙ ACCURATA IN BSDF



» Devo aggiungere bene la *traslucenza*, ovvero devo considerare anche la semisfera sotto la superficie

$$L_r(\mathbf{x}, \mathbf{o}) = \int_{H_x^2} L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{n}_x \cdot \mathbf{i}| d\omega_i$$

» *Path tracing*: è il seguire i raggi per poter calcolare l'illuminazione globale. (Il ray tracing segue solo gli highlight speculari!)

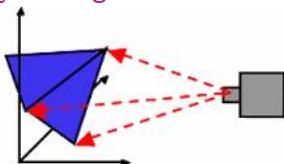
L'idea è che risolvo l'equazione con metodi statistici, calcolando un insieme di raggi e tracciando all'indietro quel numero di raggi, facendoli andare in modo diverso in base alla probabilità che la riflessione avvenga in un certo modo.

Blender usa path tracing o branched path tracing. Possiamo configurare il numero di sampling. Più ne metto e meno le mie immagini sono rumorose.

7 – Pipeline di rasterizzazione

Evita di calcolare l'intersezione su tutti i raggi tracciati all'indietro: i calcoli possono essere significativamente semplificati, ma sono comunque calcoli niente male. E' la pipeline che si è affermata negli ultimi anni per l'idea è che anziché fare raycasting (tracciare i raggi dalla telecamera alla scena) posso proiettare i triangoli verso la telecamera.

Ray tracing

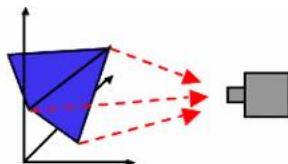


Ray casting:

Project image samples backwards

- » *Molto intuitivo*: è lo stesso meccanismo dei pittori che dipingono :)
- » Funziona con qualunque tipo di rappresentazione (non per forza triangoli!)
- » Visibilità gestita automaticamente (i raggi che non arrivano non sono visibili ☺)
- » Incorpora facilmente ombre e trasparenze
- » Complesso calcolare intersezioni

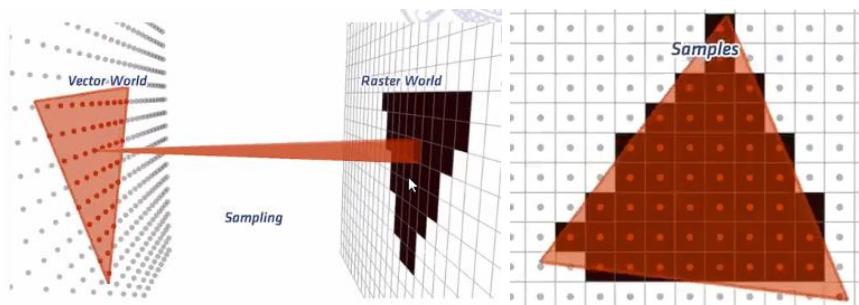
Rasterization



Rasterization:

Project geometry forward

- » *Meno intuitivo*
- » Ci vogliono vari algoritmi
- » Solo per rappresentazione poligonale
- » Più complesso capire cosa è visibile
- » Non gestisce oggetti di illuminazione globale e trasparenza senza ricorrere a trucchi
- » Permette di gestire l'antialiasing
- » Facilmente parallelizzabile
 - » Ho due procedure parallele – una sui vettori e una sui punti .
- » Hardware ottimizzato



Quello che faccio è cercare di capire come colorare i puntini in base alle proiezioni della scena!
Tutte queste operazioni sono ora estremamente efficienti ed implementati in hardware.

Problemi

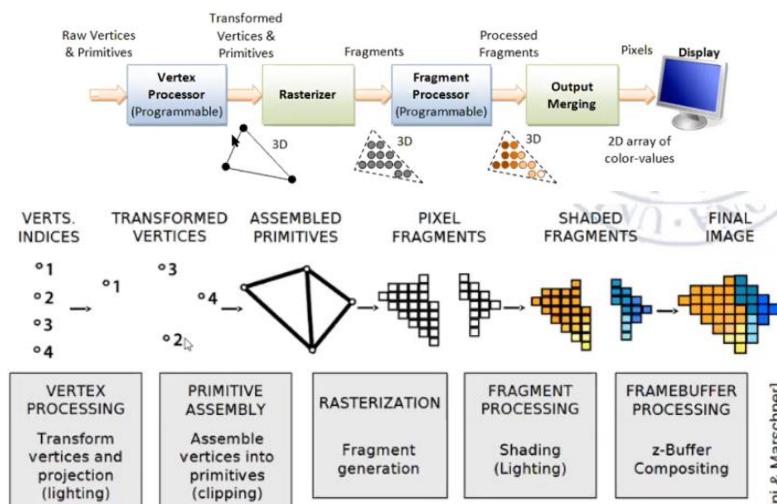
- » Superfici nascoste: non so quali sono visibili e quali no
- » Clipping: decidere quali oggetti sono all'interno dei volumi di vista
- » Scan conversion: passare da disegno vettoriale a disegno raster (ciascun triangolo si sovrappone alla matrice di pixel, e devo capire quali sono i pixel che si accendono!)
- » Shading interpolato: se gli attributi si trovano sui punti dovrò calcolare il risultato dei parametri sui punti interni ai vertici.
- » Effetti globali: non avendo più i raggi perso illuminazioni reciproche e calcolo esplicito per le ombre. O li discardo direttamente :)

PIPELINE GRAFICA

Nell'evoluzione della grafica il ray casting è stato riassunto con questa sequenza:

- » Proiettiamo i triangoli
- » Eliminiamo i triangoli non visibili
- » All'inizio si calcolava il colore sui vertici; ora lo si fa in base agli attributi dei vertici, che vengono passati al rasterizzatore
- » I triangoli vengono rasterizzati
 - > Discretizzazione di ogni triangolo
- » Si assegna ai pixel (frammenti) un colore o attributi derivati da quelli del triangolo
 - > Dato che magari tanti triangoli si proiettano sullo stesso punto devo fare un po' di processing per capire quale frammento è quello visibile.
- » Eventualmente texture mapping
- » L'immagine è passata al framebuffer

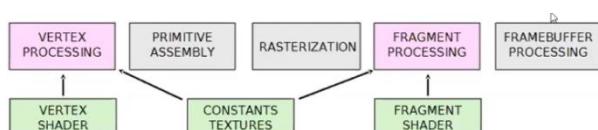
Ora è tutto programmabile: i programmatore possono divertirsi a modificare gli shader per dare effetti più specifici.



Tutto questo è fatto per milioni di triangoli a framerate interattivi grazie al progresso hardware.

A basso livello, la macchina grafica è una macchina a stati divisa in una metà geometrica e una raster. Si possono poi gestire le operazioni in maniera fine implementandole direttamente.

- » La prima parte è geometrica: applica trasformazioni geometriche che mettono i triangoli nel sistema di riferimento della telecamera e li proiettano. Alla fine delle operazioni geometriche si riassemlano le primitive
- Bisogna poi aggiungere:
 - > Clipping: decidere se gli oggetti sono dentro o fuori dal volume di vista, e eventualmente eliminare le parti fuori
 - Input: lista di vertici a cui è associata la connettività; i vertici sono processati con le trasformazioni geometriche
- » La seconda parte deve trasformare segmenti e triangoli in forme rasterizzare, per poi processare i pixel/frammenti ottenuti per ricavare il colore e metterli tutti insieme.



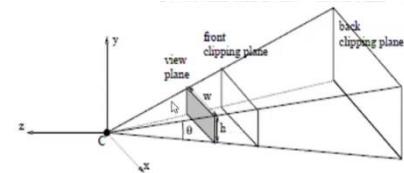
Tutto questo è, normalmente, gestito dalla scheda grafica.

PRIMA PARTE: PIPELINE GEOMETRICA

Trasformazione prospettica e divisione prospettica

- » Metto tutti gli oggetti in riferimento alle coordinate della scena (riferimento mondo)
- » Trasformo le coordinate mondo nelle coordinate della telecamera: xy sono le coordinate del piano immagine e z è la profondità del piano di vista, convenzionalmente rappresentata in direzione opposta alla vista.
 - > Oltre al piano immagine, nel caso della rasterizzazione definisco anche due piani: *front clipping plane* e *back clipping plane*, che determinano la distanza massima e minima. Sto, dunque, definendo un vero e proprio *volume di vista*, a forma di un tronco di piramide, nel quale mapperò gli oggetti attraverso una proiezione prospettica.
- » Trasformazione prospettica

→ Mappo il frustum in un parallelepipedo retto, e poi mappo questo sul piano immagine con una proiezione ortografica (ovvero elimino la terza coordinata). Così ottengo un *volume di vista canonico*, mappato nel range di coordinate -1 ... 1 fra i piani fronte-retro e laterali.



Volendo fare proiezione ortogonale, basta sostituire la trasformazione prospettica con una trasformazione affine che mappa il volume di vista (parallelepipedo in questo caso) nel volume canonico. Praticamente è una scalatura :)

Il motivo per cui non faccio direttamente la proiezione è che si semplificano le operazioni di *clipping*.

- » Proiettare sul piano immagine (con proiezione prospettica o affine!)
 - > Schiacciamento della matrice di proiezione parallela, ovvero butto via la Z
 - > Trasformo le coordinate:
 - Ripristino l'aspect ratio corretto (distorto dalla trasformazione prospettica)
 - Scalo e traslo l'immagine alla risoluzione richiesta.

Riepilogando,

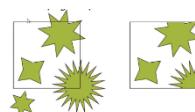
$$\mathbf{P}_{\text{screen}} = M_{\text{viewport}} M_{\text{proj}} M_{\text{view}} M_{\text{instance}} \mathbf{P}_{\text{local}}, \text{ dove}$$

Coordinate oggetto modellato Trasformo in coordinate mondo Trasformo in coordinate camera (eye) Trasformo in coordinate canoniche $\mathbf{p}_{\text{canonical}} = (M_{\text{persp}} \text{ or } M_{\text{ortho}}) \mathbf{p}_{\text{eye}}$ Trasformo in coordinate pixel	$\mathbf{P}_{\text{local}}$ $\mathbf{P}_{\text{world}} = M_{\text{instance}} \mathbf{P}_{\text{local}}$ $\mathbf{P}_{\text{eye}} = M_{\text{view}} \mathbf{P}_{\text{world}}$ $\mathbf{p}_{\text{canonical}} = (M_{\text{persp}} \text{ or } M_{\text{ortho}}) \mathbf{p}_{\text{eye}}$ $\mathbf{P}_{\text{screen}} = M_{\text{viewport}} \mathbf{p}_{\text{canonical}}$
---	--

Nelle schede grafiche reali, le *trasformazioni* sono scritte esplicitamente negli *shader*, mentre clipping, mappatura etc avvengono automaticamente secondo la pipeline.

Clipping

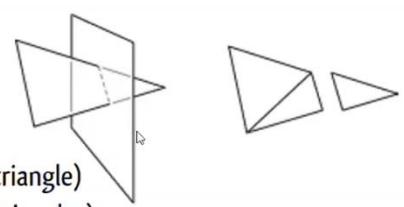
- » Algoritmi per individuare se le primitive sono dentro o fuori
- » Algoritmi per spezzare le primitive che si trovano sia dentro che fuori



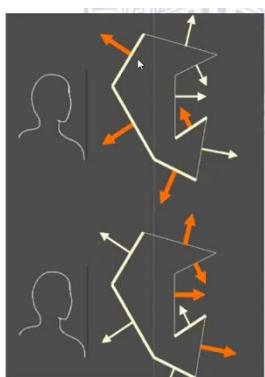
Lo faccio con algoritmi che devono essere il più possibile efficienti. Posso farlo facilmente dopo la trasformazione prospettica: qui mi basta fare il clip sui piani x,y,z = 1,-1.

Clip triangolo rispetto a piano

- 4 casi
 - all in (keep)
 - all out (discard)
 - one in, two out (one clipped triangle)
 - two in, one out (two clipped triangles)



Back-face culling



Concerne la rimozione delle superfici posteriori.

Non so a priori cosa è visibile o meno, ma se ho i triangoli orientati e so quali sono avanti/dietro in base alla normale, posso facilmente evitare di vedere quelle che di fatto sono il dietro – lo faccio solo con chi ha la faccia anteriore diretta verso la visione, ovvero se l'angolo formato è superiore a $\pm 90^\circ$. *Se forma angoli inferiori, non è visibile.*

Posso farlo calcolando il prodotto scalare fra normale e vettore di visualizzazione. Ma se le coordinate sono quelle della vista canonica mi basta vedere il segno della coordinata z delle normali! Quindi a un segno positivo della z delle normali nello spazio calcolato corrispondono facce front-facing, e viceversa.

Ragionevolmente, avrò circa la stessa quantità di davanti e dietro: di solito *dimezzo il tempo di calcolo*.

Rimozione delle superfici occluse e algoritmi HSR

Di solito, il grosso del problema non è il culling ma proprio le facce frontali di *oggetti ostruiti di altri oggetti*. Dovrò aggiungere un algoritmo apposito HSR, hidden surface removal, che determini le superfici non visibili dall'osservatore. Se non faccio ciò, posso solo usare rendering wireframe.

Posso avere due tipi di algoritmo HSR:

In realtà, si preferisce procedere a queste operazioni nella seconda parte della pipeline, attraverso il z-buffering. È molto più efficiente e fa questo processo a livello di pixel, nella fase di raster della pipeline. Lo fa con un meccanismo furbo che non calcola la profondità, ma sfrutta un buffer (= no ordinamento).

Possiamo avere:

- » Algoritmi che operano in object-space determinano, per ogni primitiva geometrica della scena, le parti della primitiva che non risultano oscurate da altre primitive nella scena. Gli algoritmi operano nello spazio di definizione delle primitive
 - > Esempio: algoritmo del pittore ordina le primitive per distanza e disegna da lontano a vicino

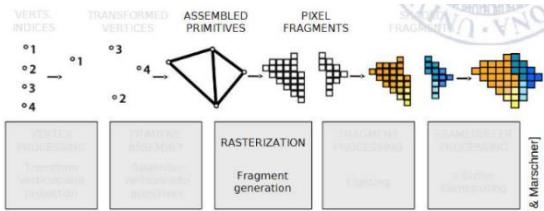
Dovrei calcolare l'ordinamento della profondità delle primitive... e potrebbero esserci situazioni ambigue. Più primitive incrociate: quale calcolo prima? Il calcolo, tra l'altro, può cambiare al movimento di oggetti e telecamera, dovendo ricalcolare ad ogni refresh. Questo è molto pesante.



- » Algoritmi che operano in image-space determinano, per ogni punto "significativo" del piano di proiezione (ogni pixel del piano dell'immagine), la primitiva geometrica visibile "attraverso" quel punto. Gli algoritmi operano nello spazio immagine della scena proiettata.
 - > Esempio: z-buffering (trattato in seguito)

A questo punto abbiamo mappato i triangoli sul piano immagine nella regione del sensore e abbiamo eliminato le parti esterne del volume. Ora dobbiamo sovrapporre i triangoli ai pixel dell'immagine discretizzata e colorare i pixel corrispondenti.

SECONDA PARTE: RASTERIZZAZIONE



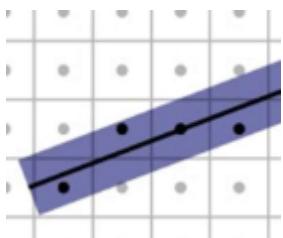
Si intende il processo di discretizzazione che trasforma la primitiva geometrica 2D in una rappresentazione discreta di un insieme di pixel. Spesso si usa come sinonimo dell'intera pipeline.

Passi:

1. Scan conversion

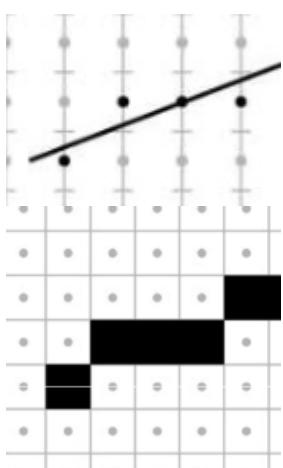
Stabliamo per ciascun triangolo su che pixel esso si proietta. Per ogni pixel interessato da un triangolo si genera un *frammento*, ovvero la più piccola unità in cui viene discretizzato un poligono. Normalmente si genera un frammento per ciascun pixel, ma potrebbero essere più di uno. La scan conversion attribuisce gli attributi dei vertici al frammento, interpolandoli.

PROBLEMA: Vorremmo rappresentare una linea, che però non si sovrappone perfettamente ai pixel! Bisogna scegliere quali pixel "illuminare". Esistono diversi algoritmi:



> Point sampling

Coloro tutti i pixel il cui centro ricade nel rettangolo.. E' semplice ma dà origine all'aliasing.



> Midpoint algorithm (Bresenham)

```
for (x : ceil(x0) ... floor(x1)) {
    auto y = b + m*x;
    output(x, round(y));
}
```

Considero la direzione orizzontale o verticale (= decido se è più in altezza o in lunghezza) in base al coefficiente angolare.

Scandisco i pixel e coloro il pixel più vicino alla linea, ovvero lascio solo il pixel più vicino in ogni colonna o riga.

```
x = ceil(x0);
y = round(m*x + b);
d = m*(x + 1) + b - y;
while(x < floor(x1)) {
    if(d > 0.5) {
        y += 1;
        d -= 1;
    }
    x += 1;
    d += m;
    output(x, y);
}
```

Moltiplicare e arrotondare sono operazioni lente. Miglioro l'efficienza: ad ogni passo posso solo accendere E o NE, e decidere quale delle due in base a

$$d = m(x + 1) + b - y$$

$d > 0.5$ decide tra E and NE

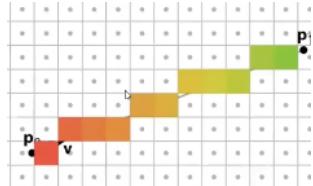
Da lì per magia posso eliminare le moltiplicazioni. Posso avere un algoritmo più efficiente con delle operazioni iterative: parte dalla prima posizione e le accende mano mano per via euristica. Calcola il punto successivo e istanza il valore che permette

di scegliere l'istanza successiva (in alternativa se accendere quello sopra o quello accanto!)

Posso inoltre stimarlo incrementalmente ad ogni passo con semplici addizioni e moltiplicazioni.

Esistono algoritmi simili anche per rasterizzare i triangoli:

2. Interpolo i valori delle variabili note sui vertici sui pixel stesso, ad esempio colori e coordinate texture.



> Lineare (proporzionale alla distanza dai vertici).

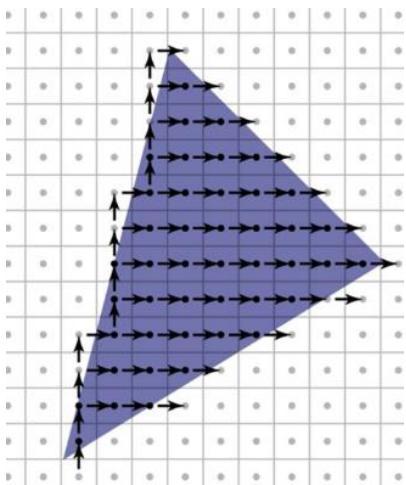
Anche qui, per sveltire il calcolo posso percorrere pixel per pixel via scansione di Bresenham e avere un algoritmo che aggiunge passo passo il valore interpolato. Per i triangoli è più complicato, ma per fortuna è tutto già implementato!

Interpolazione lineare

- 1D: $f(x) = (1 - \alpha) y_0 + \alpha y_1$
- dove $\alpha = (x - x_0) / (x_1 - x_0)$

In 2D possiamo prendere α pari alla frazione di distanza tra (x_0, y_0) e (x_1, y_1)

3. Rimane solo da stabilire quali punti sono interni, e su quali accendere il punto e assegnare i valori degli attributi interpolati! Lo faccio undendo i due algoritmi precedenti



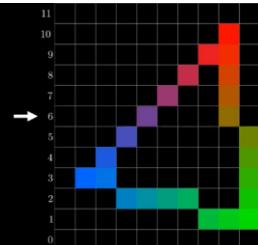
> Pixel-walk / Pineda [[video](#)]

- > Usiamo l'algoritmo Bresenham per disegnare le linee fra i tre vertici
- > Usiamo l'interpolazione lineare per decidere come colorare i vertici

A questo punto possiamo usare l'algoritmo

- > Salviamo tutti i valori di x pieni su ciascuna linea
- > Facciamo interpolazione lineare sulla riga

```
y11 = [ ]
y10 = [8, 8]
y9 = [7, 8]
y8 = [6, 8]
y7 = [5, 8]
y6 = [4, 8]
y5 = [ ]
y4 = [ ]
y3 = [ ]
y2 = [ ]
y1 = [ ]
y0 = [ ]
```



- > Posso prendere una regione intorno al triangolo e, quando faccio il ciclo di scorrimento, verifico se il centro del pixel è dentro o fuori attraverso il calcolo delle coordinate baricentriche (= esprimendo i valori di un punto come coordinate baricentriche del triangolo, ovvero come combinazione affiine delle coordinate dei tre vertici. Il punto interno ha tre coefficienti fra 0...1, altrimenti sono punti esterni)

Input:

- Tre punti 2D (i vertici nel pixel space)
 - $(x_0, y_0); (x_1, y_1); (x_2, y_2)$
- Attributi definiti per vertice
 - $q_{00}, \dots, q_{0n}; q_{10}, \dots, q_{1n}; q_{20}, \dots, q_{2n}$

Output: una lista di frammenti con

- Le coordinate intere nella griglia dei pixel (x, y)
- I valori degli attributi interpolati



Basta trovare i coefficienti c_x, c_y, c_k che definisce la funzione che ha i valori noti ai 3 vertici

Sistema lineare di 3 equazioni in 3 incognite:

$$\begin{aligned} c_x x_0 + c_y y_0 + c_k &\rightarrow q_0 \\ c_x x_1 + c_y y_1 + c_k &= q_1 \\ c_x x_2 + c_y y_2 + c_k &= q_2 \end{aligned}$$

In forma matriciale:

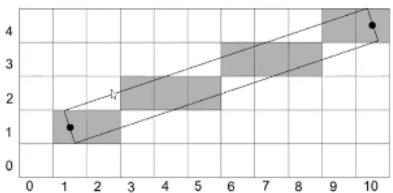
$$\begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} c_x \\ c_y \\ c_k \end{bmatrix} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \end{bmatrix}$$

Aliasing



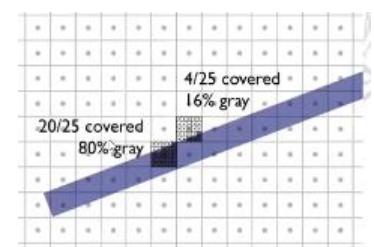
Se la risoluzione non è elevata ho il problema dell'aliasing: le scalettature sono molto evidenti.

La soluzione è sfumare le righe: se la linea deve essere spessa 1 pixel, calcolo l'area e la percentuale di linea contenuta in un pixel. E' molto pesante...



Supersampling:

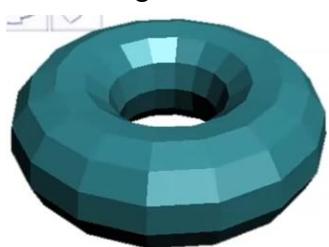
- > Faccio una *clusterizzazione a una risoluzione più alta*, e poi faccio una *riduzione della soluzione nella quale assegno il colore proporzionale alla percentuale*.
- > Posso migliorarlo ancora facendo sì che la distanza nel calcolare questo peso tenga conto non solo della percentuale di copertura ma anche della *distanza della linea*. Ma rallento.



Shading

Lo shading è il processo che permette di assegnare il colore ai frammenti di un triangolo. È calcolabile con il modello di illuminazione – come per esempio il modello di Phong.

1. Flat shading:



- Vertex stage (input: posizione come attributo vertice; parametri materiale e normali attributi triangolo)
 - trasformare posizione e normali (object to eye space)
 - Calcolare colore shading per ogni triangolo usando le normali
 - trasformare posizione (eye to screen space)
- Rasterizer
 - Interpolazione parametri: z' (screen z)
 - Colore assegnato uguale a tutti i frammenti
- Fragment stage (output: colore, z')
 - Assegnare il colore solo se interpolated $z' < \text{current } z'$

Calcolo il colore faccia per faccia, e assegno lo stesso colore all'intero triangolo

Quando ricostruisco i triangoli dai vertici calcolo lo shading e posso avere una normale calcolata che vale per tutto il triangolo e il colore è uguale per tutti i frammenti. Ovviamente, alla fine, lo z buffer permette di confrontare la distanza attuale con quella attualmente già memorizzata nel buffer – se è minore, la sostituisce con la corrente. Il problema del flat shading (= 1 colore per ciascuna faccia) la tessellazione è molto evidente soprattutto perché l'occhio è molto sensibile al contrasto.

- > Posso usare più facce, ma si vedono comunque le discontinuità...

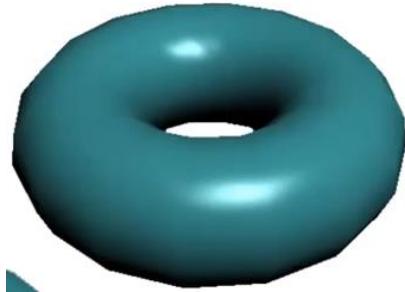
2. Gouraud shading: calcolo i colori sui vertici e interpozi sui triangoli.



- Vertex stage (input: posizione, parametri materiale e normali per vertice)
 - trasformare posizione e normali (object to eye space)
 - Calcolare colore shading per vertice
 - Transformare posizione (eye to screen space)
- Rasterizer
 - Interpolazione parametri: z' (screen z); colore
- Fragment stage (output: colore, z')
 - Assegnare il colore solo se interpolated $z' < \text{current } z'$

Fa cagher

3. Phong shading (per pixel shading):



- Vertex stage (input: posizione, parametri materiale e normali per vertice)
 - trasformare posizione e normali (object to eye space)
 - Transformazione posizione (eye to screen space)
 - pass through materiali
- Rasterizer
 - Parametri interpolati: z' (screen z); parametri materiali; normali
- Fragment stage (output: colore, z')
 - Calcolare shading con parametri materiali e normali interpolate
 - Assegnare il colore solo se interpolated $z' < \text{current } z'$

Interpolo i parametri del materiale e calcolo il colore su ciascun pixel.

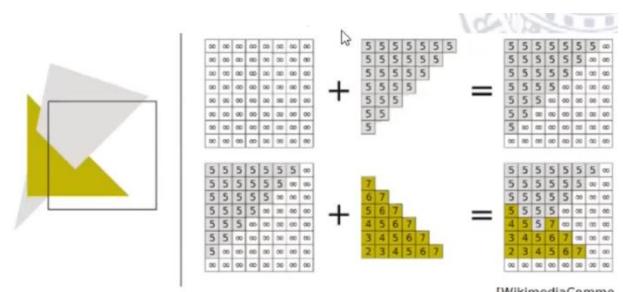
Interpolo le normali linearmente e calcolo l'equazione di illuminazione in ogni pixel.

E' tutto implementato in hardware: il programmatore deve solo passare le variabili.

5/11/2020

Z-buffer

E' l'algoritmo standard per la rimozione di superfici occluse nella pipeline: solo una piccola parte dei triangoli ho ho processato farà parte dell'immagine finale, e dovrò *eliminare i pixel dei triangoli che non si vedono*. Devo assicurarmi che solo i pixel più vicini vengano copiati nel frame buffer.



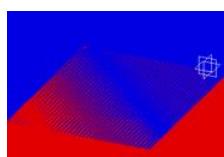
[Wikimedia Commons]

- + Non serve riordinare le primitive
- + È già implementato in firmware
- + Complessità circa costante

Depth buffer: si usa un buffer-matrice della dimensione dell'immagine, ma con *una risoluzione maggiore* (di solito 24-bit) per codificare il *valore della profondità* (voglio poter discriminare anche piccole differenze).

L'idea è che durante la rasterizzazione io so la profondità di ciascuna primitiva. Inizializzo il mio buffer a infinito. Via via che processo i triangoli, quando devo inserire il risultato lo inserisco solo e solo se la distanza della primitiva è minore del valore attualmente memorizzato nel buffer.

Z fighting



Può succedere che alcuni punti abbiano la stessa distanza. Questi punti sono critici, in quanto il valore memorizzato cambierà in base all'ordine con cui vengono processati i triangoli. Questo crea il fenomeno visto a laboratorio, detto z-fighting.

Trasparenza e blending

Possiamo simulare – ma in maniera molto diversa da quella del raytracing – anche la trasparenza. Lo faccio con i trucchi, ovvero con il cosiddetto alpha blending:

- » I colori hanno *4 componenti* (R,G,B,alfa)
- » Quando arriva un nuovo frammento che sopravvive al depth test, anziché sovrascriverlo direttamente *uso la formuletta*

$$(r, g, b)_{\text{finale}} = (r, g, b)_{\text{vecchio}} \cdot (1 - \alpha) + (r, g, b)_{\text{nuovo}} \cdot (\alpha)$$

"alpha blending"

Dove il vecchio è quello dietro e il nuovo è quello davanti.

Anche qui è importante l'ordine di rendering: devo necessariamente processare i triangoli in ordine inverso di distanza, o faccio casino. Questo è molto complicato! Si cerca di farlo solo quando serve e cerco di minimizzare il problema.

- » Divido la scena in *parti opache e trasparenti*
- » Parto *renderizzando le cose opache* (che posso renderizzare con il z-buffer semplice)
- » *Disattivo lo zbuffer* per poter visualizzare le cose in trasparenza
- » *Disegno le primitive semitransparenti* in ordine di distanza decrescente.

Se non lo faccio, rischio di generare piccoli bug grafici come il seguente →



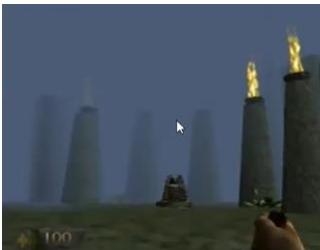
Per scene statiche è fattibile; nelle scene dinamiche può essere problematico, e di certo è molto pesante.

Compositing

Il compositing è il mescolare il risultato della passata di rendering con altri valori, oppure con altre passate di rendering. Questo consente di ottenere effetti come nebbia, oscurità, sfocatura o motion blur.

Depth cueing

Voglio simulare la nebbia in distanza; mi limito a fare che il risultato del frame buffer viene mischiato a un valore di fondo in maniera proporzionale alla distanza.



Se lo shading fornisce un colore C_s , esso verrà interpolato linearmente col colore di nebbia C_f nel seguente modo

$$C_s = (1 - z_s)C_s + z_s$$

C_f nero mi darà effetto di lontananza, e C_f bianco effetto nebbia. Per definire una nebbia non uniforme posso impiegare le fog map, ovvero un'immagine in cui RGB sono il colore e alfa la densità.

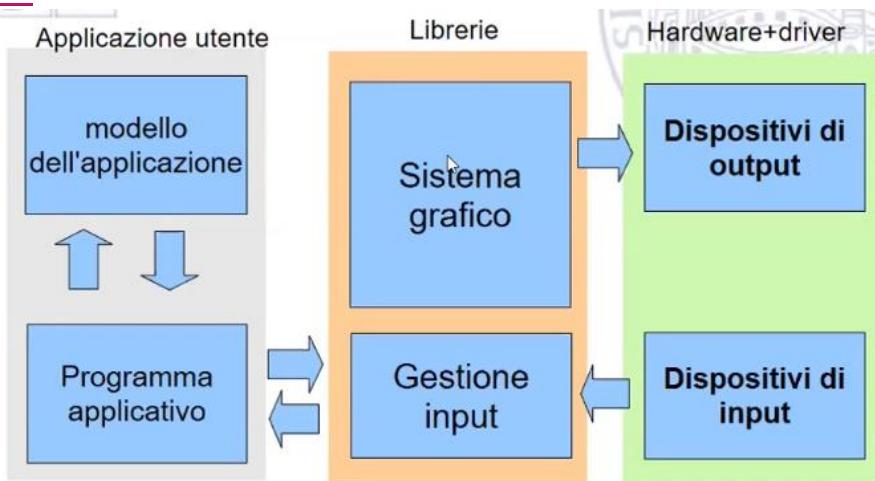
Questa tecnica è usata per:

- » Senso di profondità
- » Nebbia
- » Mascherare il fenomeno di pop-in, ovvero l'"apparizione" nel frustum di vista.

Tecniche multipass (più passate di rendering)

- » **MOTION BLUR:** simulo l'effetto di sfocamento dato dall'oggetto che si muove velocemente. Posso fare il rendering dell'oggetto in più posizioni diverse e accumulare i risultati.
- » **DEPTH OF FIELD:** Accumulo immagini diverse prese muovendo la telecamera leggermente oppure cambiando la dimensione del piano immagine in modo che i punti sul piano $z=zf$ sono fermi. In pratica muovo il punto di vista e aggiusto il frusto di vista.
- » **OMBRE:** faccio più passate e quella delle ombre applico tecniche particolari or smth

OUTPUT E INPUT

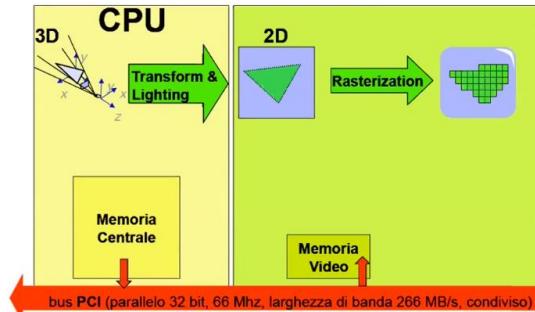


SCHEDE GRAFICHE

Tutta la pipeline è demandata all'HW: le schede grafiche gestiscono nei sistemi moderni interattivi tutta la parte di pipeline del sottosistema raster e geometrico. Sono sfruttate anche per altri particolari contesti dove è importante il calcolo parallelo.

Evoluzione HW

» 1995-1997: 3DFX Voodoo

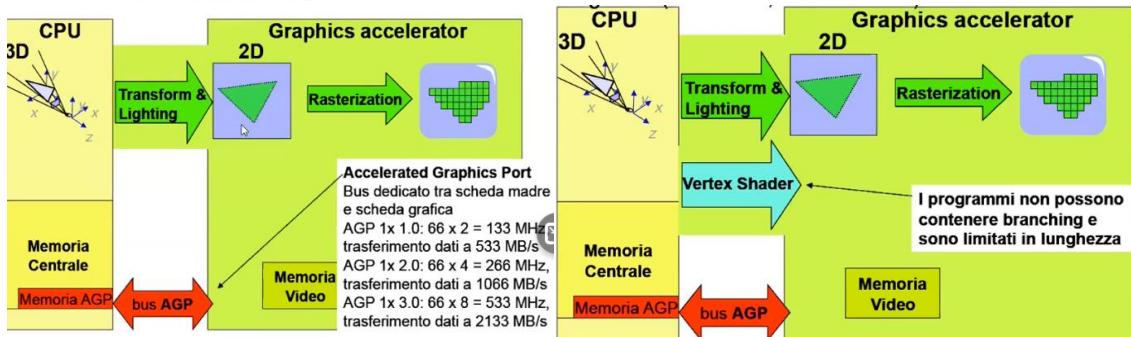


Scrivo su un buffer, lo swappo (= lo metto come quello visibile) e scrivo sull'altro, e così via.

All'inizio erano riservate solo a megacalcolatori fighissimi professionali, poi sono state portate sui portatili domestici. Le prime schede in realtà facevano solo la rasterizzazione; il resto era fatto in CPU. Il passaggio sul BUS è ovviamente problematico.

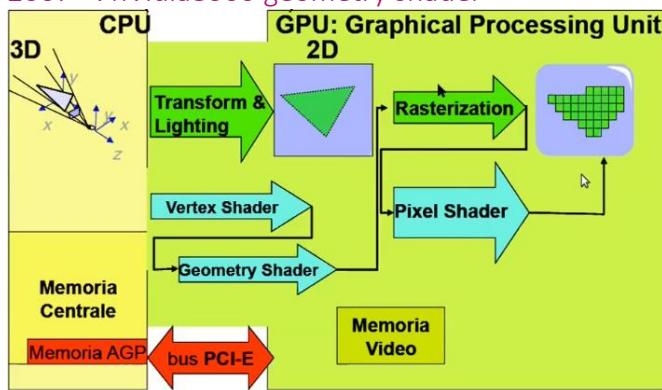
» 1998: nVidia, TNT, ATI , 2001: vertex shader

■ 1998: NVidia TNT, ATI Rage



Si iniziano a spostare sull'altro lato tutti i calcoli e velocizzare i buffer di trasmissione. Le operazioni sono ancora fissee vincolate a delle operazioni standard (Phong, interpolazione lineare). Abbiamo le cose di base e possiamo abilitarle o meno, ma non ho ulteriore customizzazione.

- » 2007+ : nVidia8000 geometry shader

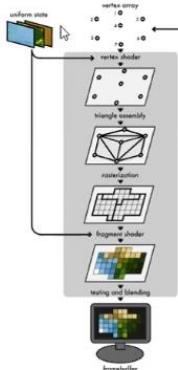


Tutto parametrizzato, non ho vincoli, super ottimizzato, programmabile. La memoria è usata in maniera indipendente.

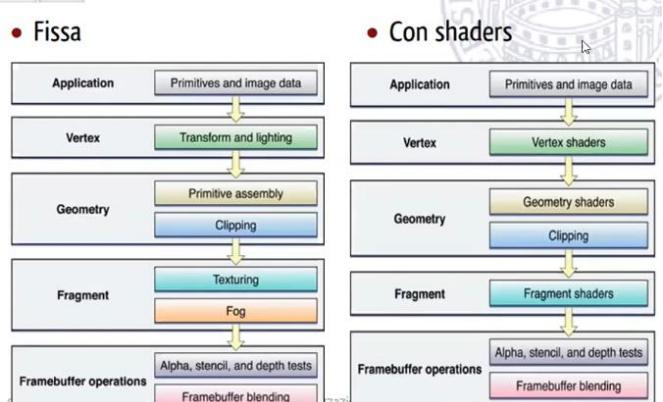
Per programmare queste pipeline servono delle interfacce standardizzate: si sviluppano delle API per la grafica 3D. Essa gestisce come diamo gli ordini a questa pipeline programmabile delle schede grafiche, ed è usata anche per fare programmazione general purpose parallela.

- » OpenGL: discende dai primi standard della grafica (come dalla silicon graphics)
- » DirectX: versione indipendente di Microsoft
- » Recenti: WebGL, Vulkan, Metal (Apple)...

Pipeline grafica standard:



- » Passo la lista dei modelli da renderizzare come array di vertici + attributi più informazioni su texture o variabili di calcolo
- » Processing geometrico sui vertici
- » Ricostruzione dei triangoli sul piano immagine e eventualmente shading
- » Rasterizzazione e composizione dell'immagine bitmap



All'inizio della grafica 3D le funzioni di processamento ai vari livelli erano fisse, ovvero l'utente poteva al massimo cambiare i flag o la geometria della telecamera, decidere se abilitare il backface culling, illuminazione fissa, ... ma non molto altro.

Nella versione programmabile abbiamo aggiunto info:

- » Non necessariamente l'illuminazione avviene a livello di vertici e non per forza è fissa..
- » Posso aggiungere altre operazioni che modifichino la geometria, aggiungono vertici e così via.

Il fatto che oggi sia programmabile implica che si possano implementare molti modi di simulare la formazione delle immagini con effetti complessi

Nella versione iniziale era tutto predefinito

- Solo alcune opzioni della pipeline che abbiamo visto erano modificabili (disegno wireframe o pieno, tipo di shading interpolativo, flat o gouraud, attivazione depth buffer, ecc.)
- Oggi si possono programmare gli “shader” cioè i programmi eseguiti nella pipeline che processano i vertici prima della rasterizzazione dei triangoli e poi processano i “frammenti” generati dalla rasterizzazione
- I programmi possono accedere alla memoria e usare variabili passate dal programma in esecuzione in CPU

SHADERS

Sono piccoli programmi scritti in GLSL, simile al C ed eseguiti su hardware grafico. Sostituiscono la pipeline a funzioni fisse.

- **Vertex Shader (VS):** per vertex operations
- **Tessellation Shader (TS)** da 4.3
- **Geometry Shader (GS):** per primitive operations (da 3.2)
- **Fragment shader (FS):** per fragment operations

8 – Mapping

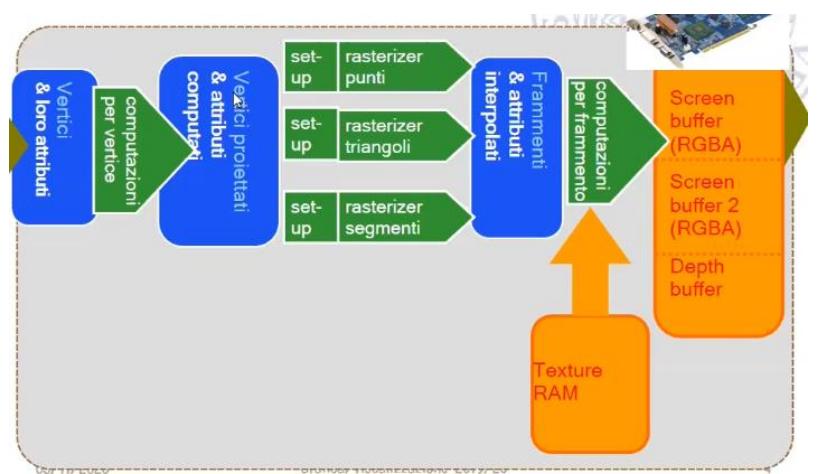
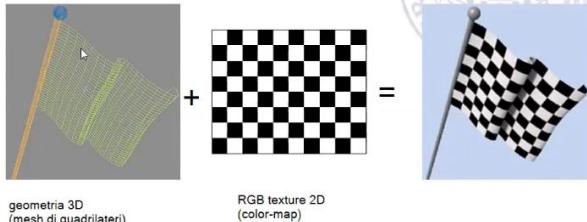
Il modello di illuminazione di Phong è abbastanza versatile, ma è limitato: non posso simulare i dettagli di un materiale a meno di introdurli nella geometria, aumentando il numero di poligoni. Il rendering ha complessità proporzionale al numero di vertici, quindi ma anche no.

Invece di incrementare la complessità del modello, si preferisce mappare il triangolo su un'immagine (*usata come lookup table*), e quando faccio lo shading vi leggo dei valori che riescono a simulare risultati fotorealistici senza aumentare troppo la difficoltà di calcolo. *E' ora implementato direttamente in HW.*

TEXTURE MAPPING

La prima idea è quella di mappare un pattern di colore. Il texture mapping consiste nell'applicare una immagine su una superficie, come una *decalcomania*.

Una texture map è dunque una matrice bidimensionale di dati indirizzati da due coordinate s,t comprese fra 0 e 1. Il dato contenuto, tipicamente, è il colore.. ma potrei memorizzare qualunque altro dato. Il texture mapping consiste nel fare il *lookup dei triangoli nelle sue coordinate all'interno della lookup table*.



Textures comuni:

- » *Color-map*
- » *Alpha-map*
- » *Normal-map o bump-map*
- » *Shininess-map*

Rimappare immagini sulla geometria richiede un processo complesso, di solito di lavoro artistico oppure di processing.

Storia:

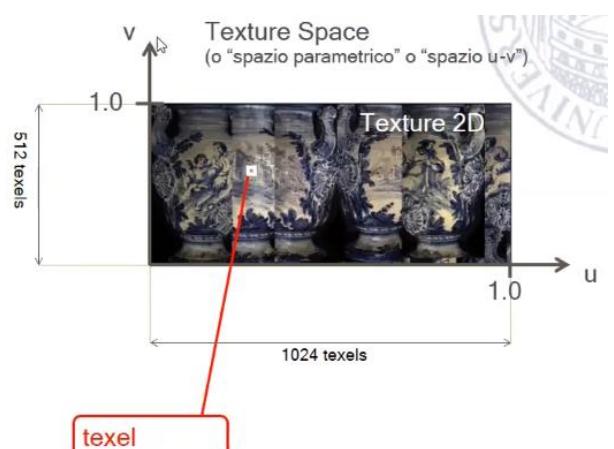
- » 1974: nato da *Ed Catmull* e introdotto nella sua tesi di dottorato
- » 1992: primo texture mapping *in hardware*

Da allora ha avuto un aumento rapidissimo della diffusione ed è una delle fondamentali tecniche di rendering.

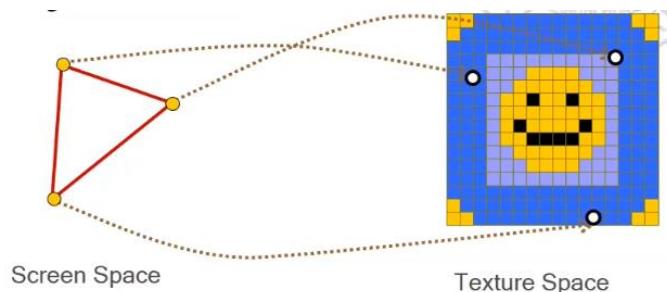
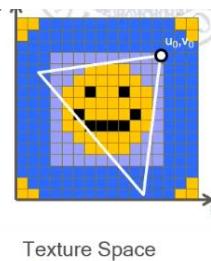
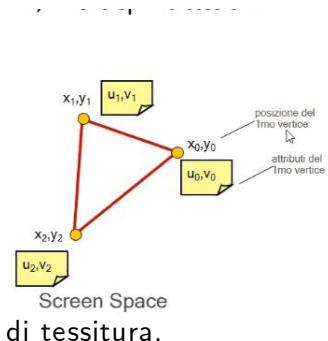
Nomenclatura

- » Texture si riferisce all'immagine.
- » Il texel è il pixel dell'immagine che uso come texture.

Vogliamo una funzione che *mappi il texture space sul mio spazio continuo 3D*. Quindi trasformo la texture in coordinate normalizzate nel range 0,1, e le chiamo U,V per non confondermi con X,Y.

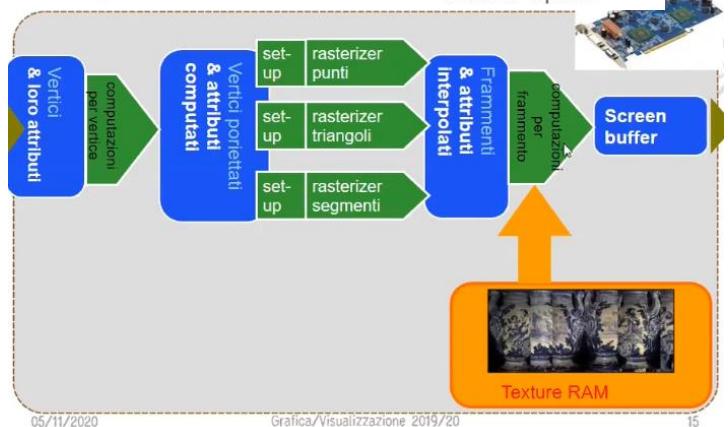


Ad ogni vertice dei triangoli assegno ulteriori attributi, ovvero le sue coordinate u, v nello spazio



Quando faccio la rasterizzazione prendo i miei frammenti generati; avendo interpolato le coordinate u, v anziché colorarlo in base allo shading vado anche a leggere il colore della posizione corrispondente nella mia mappa texture.

Questo si fa nello stadio di shader della pipeline.



Insidie:

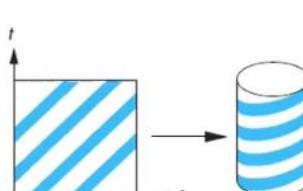
- » L'interpolazione lineare delle coordinate texture può portarmi ad artefatti. Questo succede perché la prospettiva cambia la scalatura della texture, quindi interpolare linearmente solo lungo le coordinate x, y fa scalare differentemente le coordinate u, v . Il trucco è che è necessario interpolare in tre dimensioni, ovvero *si basa sulle coordinate baricentriche* .
- » Inoltre, le texture devono essere già caricate in memoria, o rischio problemi dato che i trasferimenti da memoria secondaria sono molto lenti. Di certo è infattibile farle in ambiti real time o per animazioni.
 - > Calcolare le coordinate textures “on the fly” durante il rendering
 - > Precalcolare la texture e salvarla insieme alla mesh.

Non esiste soluzione ideale: *dipende dall'applicazione che stiamo progettando*.

Parametrizzazione della superficie

Occorre definire una parametrizzazione W che mappi un punto s, t della texture ad un punto P della superficie dell'oggetto 3D, ovvero che spalma la texture sulla superficie. E' facile trovare una corrispondenza se la superficie è a sua volta espressa in maniera parametrica; è facile esprimerla in questo modo se si tratta di una superficie semplice come un cubo o un cilindro.

W dovrebbe essere invertibile, ovvero si vorrebbe avere una corrispondenza biunivoca.



$$W : (\theta, z) \rightarrow (s, t) = \left(\frac{m}{2\pi} \theta, \frac{n}{h} z \right)$$

FIGURE 7.13 Texture mapping with a cylinder.

S/O mapping

Una tecnica generale è la mappatura in due passi.

- » S-mapping: mappo la texture su una *superficie intermedia semplice*, in modo che la parametrizzazione sia facile. ###look on google la ha spiegata da culo
- » O-mapping: mappo ogni punto della superficie intermedia in un *punto della superficie in esame*.
- » Ho varie scelte possibili:

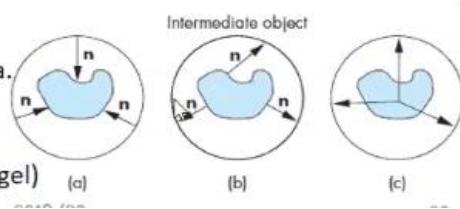
Esempi di O-mapping.

(a) Usando la normale alla superficie intermedia.

(b) Usando la normale

dalla superficie dell'oggetto.

(c) Usando i raggi dal centro dell'oggetto. (©Angel)



Più l'oggetto si discosta dall'oggetto intermedio, tanto più troverò distorsioni o continuità.

La concatenazione dei due mapping genera la corrispondenza W fra pixel e textel.

Coordinate extra-range 0,1

Le coordinate uv dovrebbero essere fra 0,1, ma può essere una buona idea consentire di andare oltre.

Questo permette di realizzare diverse modalità:



GL_CLAMP_TO_EDGE



GL_REPEAT



GL_MIRRORED_REPEAT



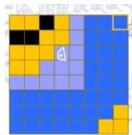
GL_CLAMP_TO_BORDER

Quando sei scazzato e non sai che fare...

GL_repeat: con texture piccole copro grandi superfici ripetute. Ovviamente funziona solo se le texture sono tileable.

TEXTURE LOOKUP

Problema: la corrispondenza fra texel e pixel non è intera: è una funzione data dalla combinazione della funzione che mappa. Essa può mappare in maniera non regolare ed intera. Quindi, se consideriamo la parte di immagine rasterizzata essa potrebbe finire in una regione molto più piccola o molto più grande.

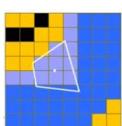


Magnification:

Il pixel mappato sulla texture è più piccolo di un texel.



- » Prendo il texel in cui cado: prendo il texel con centro più vicino, o arrotondare all'intero più vicino.
 - > Problema: viene pixelato
 - > Va bene se è proprio importante che i bordi siano belli evidenti
 - > Più veloci
- » Interpolazione bilineare: prendo la media pesata dei quattro texel più vicini
 - > Di solito viene meglio
 - > Più lento
 - > Viene un po' sfocato



Minification:

il pixel mappato sulla texture racchiuderebbe più texel

Purtroppo le due soluzioni precedenti non risolvono il problema dell'aliasing a distanza.

Nearest Filtering

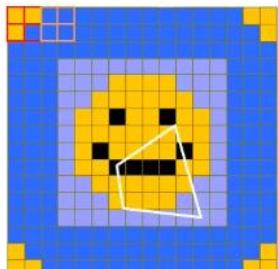
Bilinear interpolation non risolve il problema



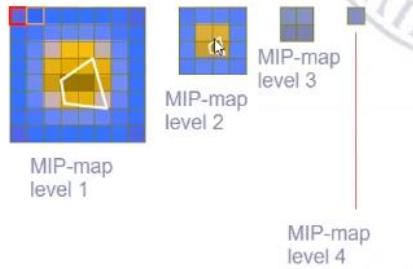
- » MIP-mapping: multum in parvo (latino???? In my computer science???)

L'idea è che rimpicciolisco la mia texture fino a che la dimensione del texel è paragonabile a quella del pixel.

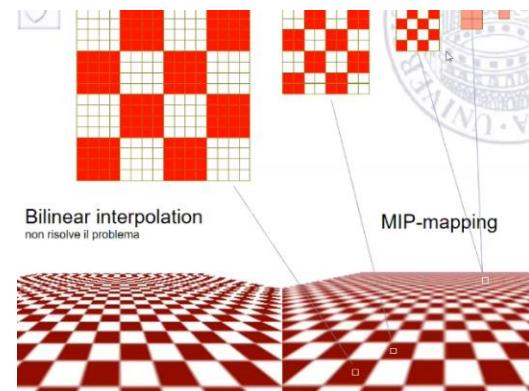
Esiste un algoritmo che permette di ricavare un *fattore di scala r=texel/pixel* (prendo il valore massimo fra x e y), che può variare nello stesso triangolo, derivato dalle matrici di trasformazione. Si può dimostrare che il livello di mipmap idoneo è $\log_2 r$.



MIP-map
level 0

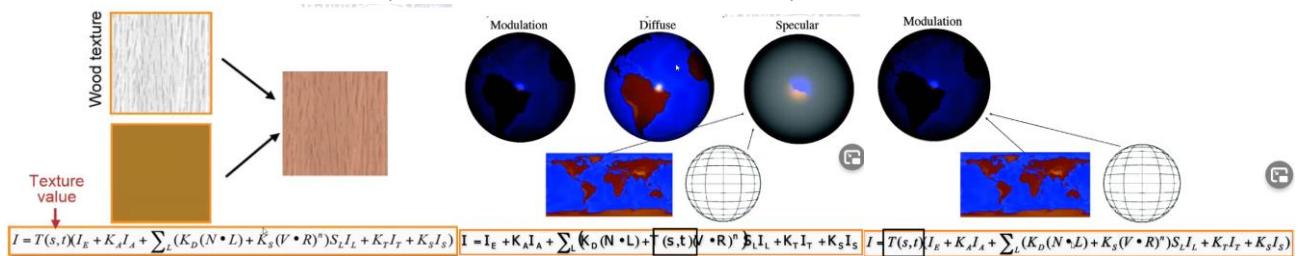


MIP-map
level 4
(un solo texel)



Uso delle texture: modulazione del colore

Generalmente, non si fa puro color mapping (= attacco la texture e basta): il color map viene poi modulato, ovvero moltiplicato (oppure unito in qualche maniera) per il valore risultante dallo shading.



- » Tipicamente assumo bianco e grigio come componente ambientale, diffusa e speculare
- » Problema: perdo gli highlights

In generale, per le texture moderne, si hanno tanti parametri che regolano l'illuminazione e se voglio controllarli in maniera più fine devo usare il mapping per scopi molteplici.

<ul style="list-style-type: none"> » Bump map 	<p><i>Perturba la normale in un unico punto con il valore corrispondente nella texture.</i> Altero lo shading senza alterare la geometria!</p> <p>Sul bordo diventa evidente l'assenza di una vera variazione di geometria.</p> <p>» Matematicamente: se io ho una mappa $B(u,v)$ che specifica la perturbazione, la normale localmente sarà $P'(u, v) = P(u, v) + B(u, v) \mathbf{n}$</p> <p>($B$ = fattore di perturbazione; P = normale "normale"). Se non voglio calcolare il punto intermedio del triangolo posso ricavarlo dai vertici</p>
<ul style="list-style-type: none"> » Displacement map: 	<p>Se volessi farlo "davvero" dovrei farlo con una mappa che <i>modifica davvero la geometria spostando i punti</i>. Computazionalmente è super pesante.</p> $P_{new} = P_{old} + h \cdot \vec{N}$
<ul style="list-style-type: none"> » Normal map 	<p><i>Specifico la normale nei punti.</i> Dovrei rappresentare tutte e tre le componenti (dato che do direttamente le normali e non le variazioni) quindi è molto più pesante.</p>
<ul style="list-style-type: none"> » Environment map: 	<p>Usa una texture per dare l'impressione di <i>riflettere l'ambiente circostante</i>.</p> <p>E' stata introdotta nelle pipeline di rendering in maniera piuttosto semplice: <i>la texture rappresenta la luce che arriva su quell'oggetto</i></p> <p>Tipicamente è una texture fatta in O-mapping, spesso sferica. Ricavo la coordinata texture incrociando la direzione del versore di direzione di vista rispetto alla normale fino a incrociare la texture. Funziona ma chiaramente non è realistico: per esempio manca il riflesso della teiera su sé stessa. Posso poi aggiungere ulteriori trucchi: per esempio applicando dei filtri oppure "sfocando" la envmap per fare un'idea di leggera ruvidezza del</p>

	<p>materiale.</p> <p>Per riflettere anche la luce emessa da altri oggetti devo introdurre delle cubemaps attorno agli oggetti. Se gli oggetti si muovono devo poi aggiornare questa cubemaps</p>
» Image-based lightning	<p><i>Si tratta di un'estensione dell'envmap;</i> serve a illuminare in modo realistico un oggetto sintetico. Si catturano immagini dal vero (creando una sfera detta "light probe")</p>
» Light map	<p>Contiene il risultato del <i>calcolo dell'illuminazione fatto offline</i>: Tipicamente il calcolo del valore di illuminazione è view independent (come radiosity) e viene salato come immagine a livello di grigio. In fase di rendering si aggiunge la lightmap con una certa modulazione.</p>
» Mappa di trasparenza:	<p><i>Modula l'opacità dell'oggetto.</i> Spesso questa trasformazione è accorpata al colore</p>
» Mappa di emissione	

Screen-space reflections

Per ottenere la riflessione si possono usare le screen-space-reflections: *sfrutto l'idea del rendering già fatto per calcolare le riflessioni.* Per ciascun frammento:

- › Calcolo il raggio di rigresso
- › Traccio sul raggio di riflessione e scopro il colore del punto da cui viene la luce
- › Uso quel colore e lo blendo con quello precedentemente in quel punto :P



Attenzione! Per i punti che riflettono cose "nascoste", chiaramente tutto ciò non funziona. :(

Impostori planari

Posso usare le *texture bidimensionali per simulare oggetti 3D*.

Molto usato nei videogiochi: se un oggetto è visto da lontano non mi serve che sia modellato in maniera accurata! Ad esempio, se ho degli alberi lontani mi basta aggiungere una texture sul terreno che mi dia l'impressione che sia ricoperto da foreste.

In generale l'immagine ha anche un canale alpha che mi permette di rappresentare l'oggetto senza che sia visibile il bordo della texture.

Al solito, se cambio il punto di vista rischio che l'oggetto risulti errato o addirittura sparisca

- » Billboard: posso fare in modo che i poligoni ruotino sul loro asse in modo da fronteggiare sempre la telecamera. In questo modo non rischio che venga inquadrato di lato (e sparirà). Posso anche usare più immagini che cambiano in base al punto di vista.

Funziona anche per simulare le *particelle*, come esplosioni o gli effetti atmosferici quali neve o pioggia.



9 – Ombre

Tutto ciò che abbiamo fatto fino ad ora (in pipeline) non permette di calcolare le ombre. Eppure le ombre sono chiarificanti:

- » *Chiariscono la percezione della posizione* (es. se si muove sull'asse z o solo sull'asse y)
- » La definizione dei bordi mi fa capire che *tipo di illuminazione* sto osservando (strong/diffused)
 - > Nella realtà, le ombre sono sempre diffuse (sfumate).

Il meccanismo di proiezione del cono d'ombra è del tutto analogo a quello della proiezione prospettica usata nel rendering. Non si vede alcuna ombra se la sorgente luminosa coincide con il punto di vista (perché saranno nelle parti occluse). Quindi posso facilmente usare delle tecniche di rimozione di superfici nascoste per calcolare le ombre. Per le sorgenti puntiformi, le ombre hanno bordo netto (non-soft).

Le ombre geometriche possono essere precalcolate per oggetti statici, e poi aggiunte alle lightmap (molto usato nei videogiochi).

Proprietà delle ombre proiettate

- » L'ombra che il poligono A getta sul poligono B a causa di una sorgente luminosa puntiforme si può calcolare proiettando il poligono A sul piano che contiene il poligono B con centro di proiezione fissato in coincidenza della sorgente.
- » Non si vede alcuna ombra se la sorgente luminosa coincide con il punto di vista.
- » Per scene statiche le ombre sono fisse. Non dipendono dalla posizione dell'osservatore
- » Se la sorgente (o le sorgenti) è puntiforme l'ombra ha un bordo netto

TECNICHE

Light map

Le ombre geometriche statiche possono essere precalcolate per oggetti statici e aggiunte alla lightmap.

Ombra sul piano (projective shadows) (Outdated)

Calcola l'ombra calcolata da un oggetto su un piano. L'idea è di *dipingere l'ombra come un oggetto piatto (nero) sul piano*.

Per disegnare l'oggetto effettuo il rendering della scena con colore nero, telecamera centrata nel punto luce e usando il piano che voglio calcolare come piano di immagine.

Problemi:

- » Z-fighting: I triangoli d'ombra sono sullo stesso piano del piano su cui si proiettano.
 - > Risolvo spostando leggermente il piano d'ombra
 - » Solo ombre nette
 - » Se il piano ha una texture il risultato è brutto
 - » Se uso un'ombra semiopaca, laddove ci sono poligoni multipli ho una sovrapposizione
 - » Le ombre vanno ridisegnate al movimento della camera, anche quando l'oggetto è fermo
 - > Se ho altri oggetti in giro, possono "tagliare" l'ombra (dato che su di essi non viene calcolata)
- Disegniamo il piano
 - Si incrementa lo stencil buffer dove il piano è renderizzato
 - Disabilitiamo depth buffer
 - Calcoliamo l'ombra dell'oggetto
 - Disegniamo l'ombra dove lo stencil buffer vale 1
 - Si evita di ridisegnare
 - Riabilitiamo depth test
 - Disegniamo oggetti

Shadow maps

Questa tecnica si rifà a una certa proprietà delle ombre, ovvero alla rimozione di superfici nascoste. Il problema di calcolare l'ombra diventa quello di *calcolare la visibilità* di qualcosa.

- » Circondiamo la luce con un cubo; *per ogni faccia implementiamo uno z buffer*.
- » Se lo z del punto è più grande di z della z del buffer significa che c'è un oggetto che blocca la luce, che c'è un'ombra in quel punto e che lo si può colorare di conseguenza. Altrimenti è colpito dalla luce e si può fare lo shading normale.

In caso di luci multiple avrò bisogno di più buffer.

Poiché questo è implementato in maniera molto efficiente (salvo i soliti problemi degli zbuffer, come il fatto di rifare più volte gli stessi calcoli), posso calcolare la distanza dal punto alle luci molto velocemente.

Problemi

- » Mi servono tanti z buffer
- » La discretizzazione delle mappe può crearmi *artefatti di discretizzazioni*; a seconda della prospettiva i pixel della shadow map possono coprire molti pixel dell'immagine e arrivo ad avere artefatti a blocchi
 - > Risolvo trasformando lo spazio della shadow map affinché sia proporzionale giusto.

19/11/2020

Shadow volume

E' un metodo introdotto da Crow e Bergeron. Considera il volume d'ombra (shadow volume) di un poligono rispetto ad una sorgente luminosa è la parte di spazio che il poligono occlude alla vista della luce. Produrrà regioni di spazio che sono in ombra: quindi per ciascuna primitiva *genero dei tronchi di piramide* che generino l'ombra.

Può funzionare solo se ho un numero limitato di primitive. Può funzionare meglio se uso delle gerarchie.

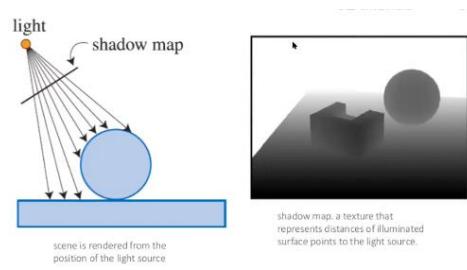
Le facce della piramide prendono il nome di shadow planes; vengono rappresentate in modo che *il fronte sia verso la luce e il retro verso l'ombra*.

Algoritmo

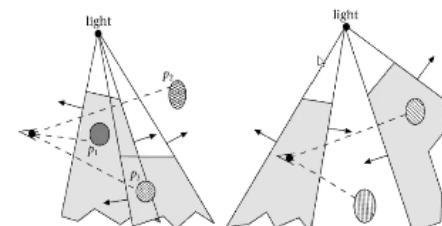
Dato un punto che sto rasterizzando voglio capire attraverso i coni d'ombra se è illuminato o meno.

E' un algoritmo molto efficiente, ma è *più complicato* da settare rispetto alle shadow map.

- » Posso dimostrare che se considero i punti da valutare e traccio la congiungente fra centro ottico della telecamera ai punti della scena, il segmento intersecherà un po' di volte i lati dei coni d'ombra. Posso contare il numero di attraversamenti in maniera molto efficiente.
- » Se il punto di vista è fuori dai coni d'ombra, il punto è illuminato se e solo se la differenza fra entrata e uscita dai coni d'ombra (che posso valutare in base all'angolo con la normale) è zero.
- » Se la vista è a sua volta in un cono d'ombra dovrei contare in quanti volumi d'ombra si trova.

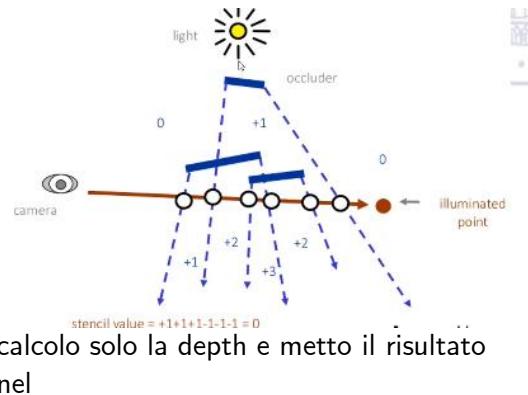


Nella pipeline grafica ho un metodo semplice:



- Usare i buffer nella procedura di rasterizzazione per il contatore
 - Creare shadow volumes per ogni occlusore (convesso)
 - Fare rendering con componente ambientale, tenere la depth
 - Per ogni sorgente
 - Inizializzare buffer (stencil buffer) al numero di volumi contenenti i punti di vista
 - Usando z buffer test disabilitando il suo aggiornamento
 - Fare rendering solo front facing dei poligoni degli occlusori incrementando buffer (stencil) per tutti i frammenti che superano il depth test
 - Fare rendering con solo back facing decrementando il buffer/contatore
 - Solo sui pixel col buffer a 0 si rifà il rendering con l'illuminazione accesa (sorgenti puntiformi)

» Prima passata di rendering in cui

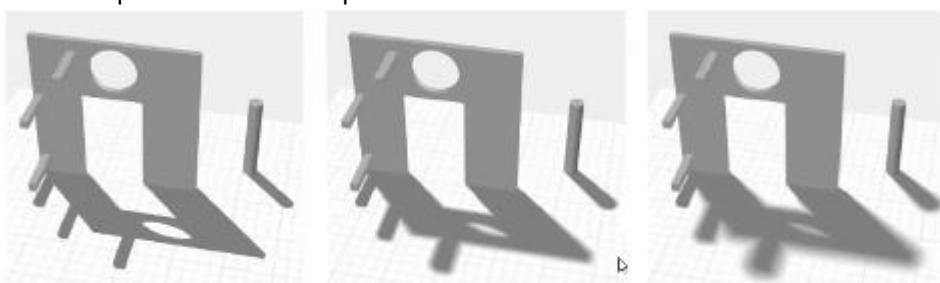
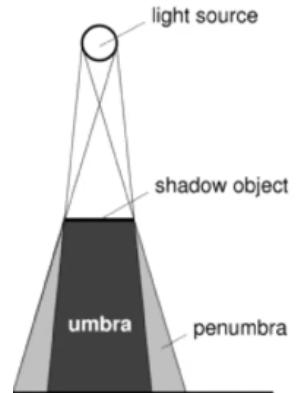


Ombre sfumate

In generale si cerca di rendere le ombre più realistiche sfumandole; facendo delle ombre con gli algoritmi visti, infatti, i contorni sono solamente neri e netti. Tipicamente questo non è quello che accade enella vita reale.

Metodi:

- » Sposto leggermente la sorgente luminosa e faccio più passate di rendering. Permangono delle bande di colore costante che si *sfumano in post processing*.
Moltiplico per n (numero di sorgenti che uso) il *costo computazionale*.
- » Sposto leggermente gli oggetti e faccio più passate di rendering, mediando i risultati. Molto meno usato!
- » Posso ottenere la sfumatura via sfumatura locale, in maniera proporzionale alla distanza da dove viene gettata l'ombra. E' molto lontano dalla realtà fisica ma replica visivamente quello che accade.



Ambient occlusion

È il nipote di SSAO (IM GONNA HAVE A FUCKING STROKE)

SSAO realizzava delle *occlusion map* calcolando la relazione di ciascun pixel rispetto ai suoi vicini. Prende in considerazione solo i pixel visibili!



È un metodo di illuminazione globale (circa) che prende in considerazione la geometria della scena e quanto la luce riesce a raggiungere ciascuna zona. Simulo il fatto che la luce ambientale arriva meno laddove si trova occlusione (calcolo quanto dell'angolo solido di un punto è effettivamente visibile).

10 - Animazione

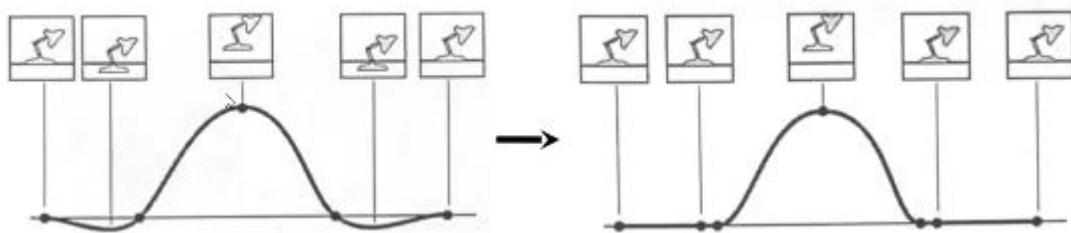
Sappiamo che fare animazione significa generare immagini multiple che viste velocemente simulano il movimento, così come simuliamo il colore su tre colori primari.

Sono sufficienti **10 frame/sec**, ma l'occhio gradisce meglio 30-60.

Dal punto di vista pratico mi basta un sistema che faccia il **rendering di diverse immagini** in maniera mentre cambio qualcosa nella scena:

- » Moti rigidi
- » Deformazioni di superfici
- » Variazioni materiali
- » Spostamenti o variazioni di camere
- » Variazioni di ambiente.

Molte cose possono essere fatte a mano sfruttando la keyframe animation: definisco delle posizioni e calcolo, interpolando, quelle intermedie. Al calcolatore mi basta decidere i keyframe e il metodo di interpolazione



Le funzioni interpolanti, similmente alle spline, cercano di **fittare sui punti un polinomio a tratti** creando delle onde – talvolta non desiderate! Tendenzialmente si risolve aggiungendo punti.

Movimenti rigidi

$$\mathbf{t}(t) = \mathbf{t}_0(1-t) + \mathbf{t}_1 t$$

- » Traslazioni: interpo linearmente senza problemi.
- » Rotazioni: posso avere problemi!
 - > Interpolare gli elementi di matrici non dà le rotazioni che mi aspetterei,
 - > con gli angoli di euler ho problemi (gimbal lock, traiettorie strane).
 - > Con asse-angolo non ottengo sempre quella orretta: l'angolo è corretto ma se gli assi non sono uguali ho un pasticcio
 - > L'interpolazione giusta è quella sulla sfera unitaria, ovvero quella per quaternioni applicando la spherical linear interpolation.

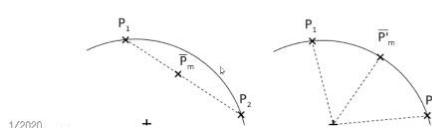
L'interpolazione giusta è sulla sfera unitaria

Spherical linear interpolation

Creo base ortonormale dagli assi \mathbf{e}_0 e \mathbf{e}_1

Interpo angolo tra vettori

$$\begin{aligned} \mathbf{e}_1^P(t) &= \frac{\mathbf{e}_1 - (\mathbf{e}_0 \cdot \mathbf{e}_1)\mathbf{e}_0}{|\mathbf{e}_1 - (\mathbf{e}_0 \cdot \mathbf{e}_1)\mathbf{e}_0|} & \psi = \arccos(\mathbf{e}_0 \cdot \mathbf{e}_1) \\ \mathbf{e}(t) &= \cos(t\psi)\mathbf{e}_0 + \sin(t\psi)\mathbf{e}_1^P \end{aligned}$$



• Media pesata tra i vettori originali

$$\begin{aligned} \mathbf{e}(t) &= w_0(t)\mathbf{e}_0 + w_1(t)\mathbf{e}_1 \\ \psi &= \arccos(\mathbf{e}_0 \cdot \mathbf{e}_1) = \alpha(t) + \beta(t) \\ \sin \psi &= \sin \alpha(t)/w_0(t) \\ \sin \psi &= \sin \beta(t)/w_1(t) \\ \alpha(t) &= (1-t)\psi \quad \beta(t) = t\psi \\ w_0(t) &= \sin t\psi / \sin \psi \\ w_1(t) &= \sin t(1-\psi) / \sin \psi \end{aligned}$$

Voglio che al variare del parametro t ci sia una velocità costante.

- » La formula vale in tutte le dimensioni

$$\mathbf{e}(t) = \frac{\sin t\psi}{\sin \psi} \mathbf{e}_0 + \frac{\sin t(1-\psi)}{\sin \psi} \mathbf{e}_1$$

- » Formulazione in termini di quaternioni unitari

$$\psi = \text{acos}(\mathbf{q}_0 \cdot \mathbf{q}_1)$$

$$\mathbf{q}(t) = \frac{\sin t\psi}{\sin \psi} \mathbf{q}_0 + \frac{\sin t(1-\psi)}{\sin \psi} \mathbf{q}_1$$

- » Asse fisso velocità angolare costante

> Posso calcolarla anche su asse angolo or smth

Problemi: Se l'angolo va a 0 ho problemi di stabilità ma i quaternioni risolvono perché ho sempre 2 quaternioni che rappresentano lo stesso angolo (or smth)

Blend shape animation

Riguarda il controllo della deformazione delle forme; posso interpolare linearmente la posizione dei vertici fra le pose chiave

$$\mathbf{p}'_i(t) = \sum_j w_j(t) \mathbf{p}_{ij}$$

Pro

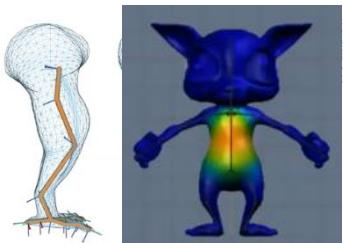
- » Fatte le posizioni è *facile interpolare*
- » Facile *implementare l'interpolazione*

Contro

- » Consuma *molta memoria*
- » Difficile estrapolare bene verso pose non previste fra quelle presenti
-> *Posso mettere controller separati per ogni parte della faccia*
- » Non funziona per il corpo di un personaggio perché dovrei interpolare *troppe posizioni!*

Linear blend skinning

Lego le trasformazioni della superficie a una serie di punti di controllo.

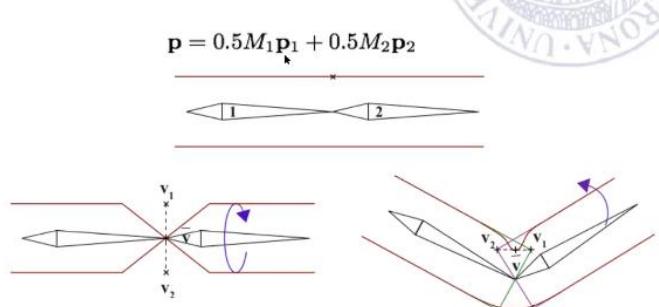
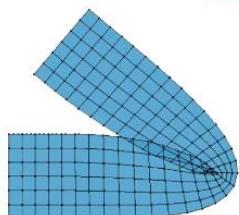


- » Definisco la *posizione dell'osso come trasformazione rigida* (frame)
- » Definisco quale osso influenza quale vertice, dando dei *pesi* a ciascun vertice
- » Definisco le posizioni di interpolazione come *media ponderata* delle trasformazioni

$$\mathbf{p}'_i(t) = \sum_j w_{ij} M_j(t) \mathbf{p}_i$$

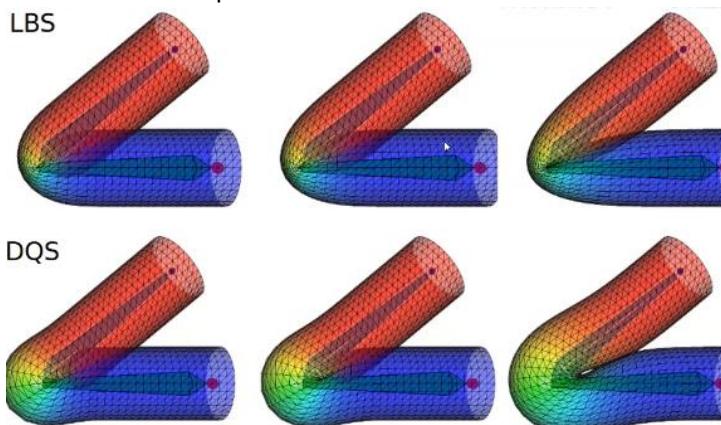
$$\sum_j w_{ij} = 1$$

Possono generarsi degli artefatti, tipicamente *vicini ai giunti* dello scheletro



Problemi:

- » Stesso problema delle rotazioni con le interpolazioni.
 - > Si risolve con un *doppio quaternione* ma lasciamo perdere la matematica :) Esso permette di unire nella stessa notazione rotazione e traslazione.
 - Inoltre permette di limitare gli artefatti! Abbiamo dei *rigonfiamenti* ma sono *meno evidenti* dell'interpolazione lineare.



- » Chiaramente *manca la simulazione della realtà* (es. muscoli che si gonfiano)

Definizione dei pesi

Si può fare a mano oppure usando dei template da adattare via algoritmi di ottimizzazione.

| Controllo dell'animazione

Ho due obiettivi:

- » Animazione realistica: creare un'animazione somigliante al mondo reale
 - > A mano o movimento catturato
- » Animazione artistica: avere anche deformazioni irrealistiche o improbabili
 - > A mano

Ultimamente è di tendenza usare personaggi autonomi che si muovano utilizzando AI.

“Rigs” di animazione

Le deformazioni di basso livello sono molto complesse da controllare; allora aggiungo dei manipolatori che controllano il modello in maniera intuitiva (es. controllo “sorriso”).

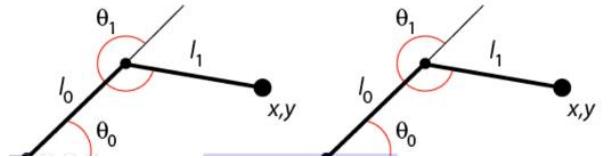
- » Cinematica diretta: consiste nell'impostare una gerarchia di trasformazioni e *animare le articolazioni manualmente, dalla superiore alla più in basso*.

Genero le pose controllando la posa dello scheletro

- | | |
|----------------------------------|--|
| » Controllo diretto agli artisti | » Difficile raggiungere vincoli, come “i piedi devono essere sul pavimento” o “mani sull’oggetto”; difficile raggiungere la fine |
| » Molto versatile | |

- » Cinematica inversa (IK) : fa il lavoro opposto, ovvero si *specificano dei vincoli e il sistema risolve* per la posa migliore, cercando di trovare la variazione più piccola (entro certi vincoli).

$$\begin{cases} p_x = l_0 \cos \theta_0 + l_1 \cos(\theta_0 + \theta_1) \\ p_y = l_0 \sin \theta_0 + l_1 \sin(\theta_0 + \theta_1) \end{cases} \text{ Solve equations for } \theta_0 \text{ and } \theta_1$$



Motion capture

Anche con le IK è comunque molto complesso e dispendioso produrre animazioni. Alternativamente si

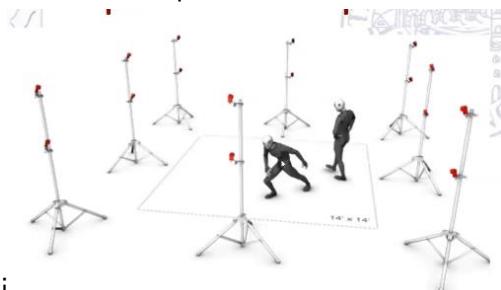
può registrare l'animazione di attori reali e riprodurla su un personaggio virtuale.

Problemi: registrare posizioni precise senza essere invadenti.

- » Acquisizione del movimento meccanico: si faceva all'inizio..



- » Motion capture ottico: uso più telecamere per tracciare i marcatori



> Marker molto invadenti

- » Motion capture senza marker: uso machine learning e telecamere normali.
- » Cattura con dispositivi inerziali: è più costoso ma evita i problemi dei sistemi di acquisizioni già descritti che possono dare problemi in certe situazioni (es. all'aperto). E' meno preciso (catturerebbe solo l'orientazione) ma è possibile avere catture di buona qualità

Autonomous characters

Si usa l'AI per far imparare al personaggio a muoversi. Questo avviene con reti neurali e machine learning, che date certe condizioni riesce a muoversi simulando stili di movimento. E' già molto usato nei videogiochi.

11 - Visualizzazione scientifica

Premessa: Princìpi di visualizzazione

Con visualizzazione si intende l'uso di rappresentazioni supportate da pc, visuali e possibilmente interattive dei dati al fine di amplificarne la comprensione.

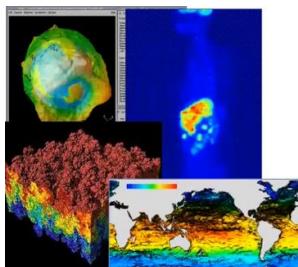
Visualizzare significa formare una visiome mentale, immagine; rendere visibile alla mente.

La visualizzazione grafica di concetti e misure non nasce con il calcolatore: infografiche, mappe, isosuperfici e vettori nel meteo esistevano già in precedenza.

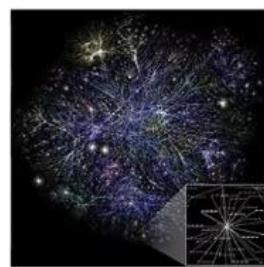
La principale motivazione della visualizzazione grafica è quella di far capire qualcosa all'end-user. Guardando invece quello che succede all'interno del calcolatore, infatti, è spesso quasi impossibile cogliere il significato dei dati grezzi! Risulta utile poiché la comunicazione visuale è molto più efficace, perché non richiede necessariamente l'uso della memoria.

Quindi, occorre generare immagini a partire dai dati utili a rivelare significati, idee e renderle intepretabili.

Una prima distinzione che si può fare è la seguente:



Visualizzazione
scientifica
Visualizzazione di
dati fisici



Visualizzazione
dell'informazione
Visualizzazione di dati astratti
(es. grafico di una funzione)

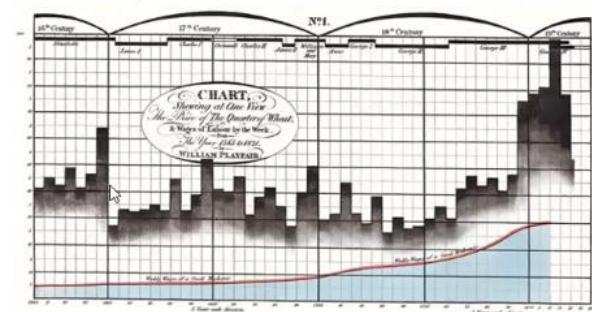
Le applicazioni tipiche sono:

- > Esplorazione dei dati: i dati puri sono difficili da analizzare perché possono essere moltissimi ed è difficile intuirne le relazioni. Posso, per esempio, confermare un mio modello
- > Misurare dei dati per quantificare degli esperimenti: dovrò capire come rendere chiare e immediate le misure
- > Comunicazione: rendere i risultati papabili al pubblico

Siamo già abituati a visualizzare l'informazione: grafici, istogrammi, reti, mappe...

I metodi di visualizzazione sono parte di tutti i pacchetti di elaborazione dati (Excel, matlab) ed esistono metodi specifici per la disciplina.

Come posso renderli più efficaci?

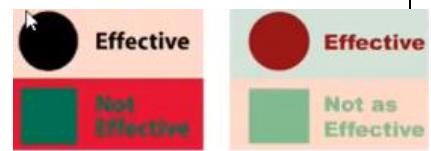


Scelte

» Colore

Non è una proprietà intrinsica delle cose posso modificarlo come voglio.

Di conseguenza, è bene effettuare delle scelte consapevoli:

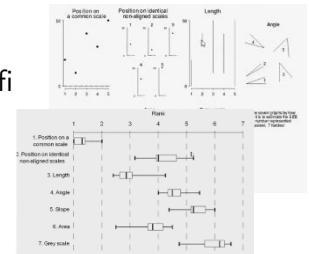


- > Colori brillanti e contrasto per *attirare l'attenzione*
- > Attenzione all'accessibilità e leggibilità: per esempio, è bene evitare problemi ai *daltonici*: cerco di rendere forti i contrasti
- > Fare *attenzione ai colori meno percepibili* (come il blu);
La distinguibilità dipende da fattori percettivi ma anche psicologici: se non ho tanti nomi per il blu, poi non riesco a distinguerli! xD
Distanzio i colori sia sullo spazio LUV che usando colori dal nome diverso.



» Plot

- > Su quale mappatura vedo meglio il rapporto fra le quantità? Non tutti i grafici risultano egualmente efficaci
 - > Es. con un grafico a torta ho pessima trasmissione della quantità assoluta ma buona trasmissione del rapporto. Non siamo vincolati a questi studi ma è una buona idea farlo.

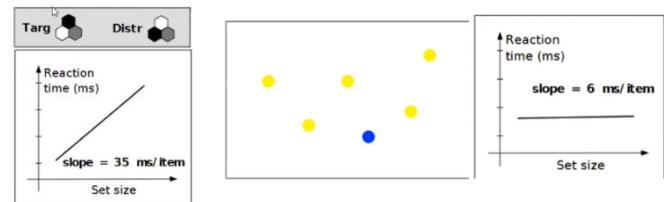


Principi di percezione

In realtà, è vero che il canale visivo trasmette molte informazioni ma le trasmette in modo variegato! Alcune sono catturate molto velocemente mentre altre non sono proprio calcolate se non vi ci si pone attenzione.

Percezione pre-attentiva:

- > Agisce velocemente
- > Parallelamente su tutto il campo vicino
- > Automatica
- > Processa colore, frequenza spaziale, orientazione
- > Per capirlo si fanno esperimenti!
Es. stimolo "colore emerge" o orientazione trova il punto blu è indifferente quanti siano i distrattori. Esistono pattern simili ma non preattentivi (es. esagoni)



Principi della Gestalt



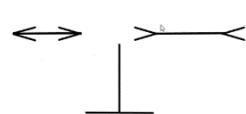
Sono altri principi derivanti dalla psicologia utili per visualizzare più informazioni correlate.

Essi dicono quali sono i principi visivi con cui possiamo raggruppare gli oggetti in una scena visiva; possiamo dunque raggruppare informazioni affini.

Percettivamente, raggruppiamo le forme per:

- > Prossimità
- > Continuità
- > Similarità
- > Periodicità
- > Simmetria

Illusioni ottiche



La vista spesso inganna: ci sono una serie di illusioni legate alla percezione visiva che vanno considerate. Per esempio, i grafici possono proiettare dati sulla lunghezza, ma la lunghezza può essere percepita in maniera errata

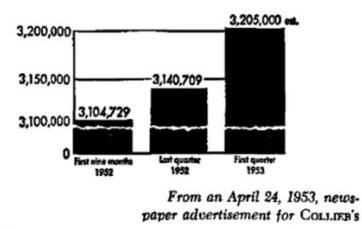
Criteri di scelta

Quello che ci interessa boostare è la capacità di convogliare il messaggio.

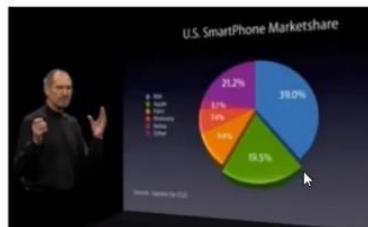
Metodologie di valutazione tipiche di Human Computer Interaction

Edward Tufte scrive diversi libri (*Visual Display of Quantitative Information*) che declinano i principi dovuti a fattori umani più che a percezione tecnica.

- > Massimizzare la percentuale di inchiostro dedicata alla rappresentazione dei dati, ovvero eliminare gli elementi superflui con fine esclusivo estetico
- > Massimizzare densità dei dati (uhhhhh nope ma che schifo)
- > Dire la verità sui dati!!
 - > Ci sono grafici che sono fuorvianti!



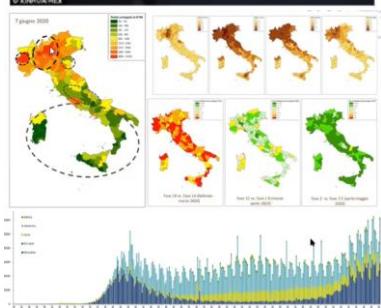
Il grafico non parte davvero da $y=0$ ma da $y=3100000$; se lo facessi partire da 0 mi accorgerei che le differenze sono minuscole



Essendo in 3D, 19 sembra maggiore di 21



E' cumulativo: quindi non è vero che le vendite sono sempre aumentate; anzi, dove l'inclinazione della retta diminuisce esse sono in realtà diminuite.



I numeri sono assoluti! Sto misurando gli abitanti, non l'effettiva densità di malattia

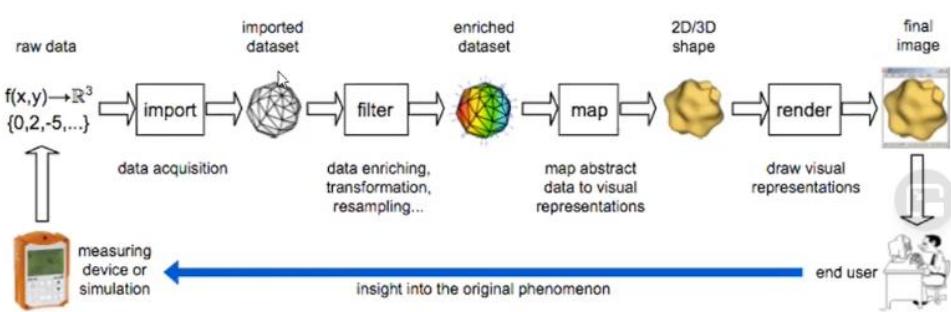
Il grafo giornaliero è del tutto dipendente dal numero di tamponi... e pieno di rumore

VISUALIZZAZIONE SCIENTIFICA

Con la visualizzazione scientifica, oltre alla visualizzazione di informazione ho la necessità di associare informazione alla geometria degli oggetti studiati.

Si rappresentano principalmente misure e simulazioni fatte nel mondo 3D:

- > Modelli industriali
- > Particelle microscopiche
- > Strutture biologiche
- > Organi del corpo umano
- > Strutture geologiche
- > Simulazioni fisiche (temperature, pressioni...)

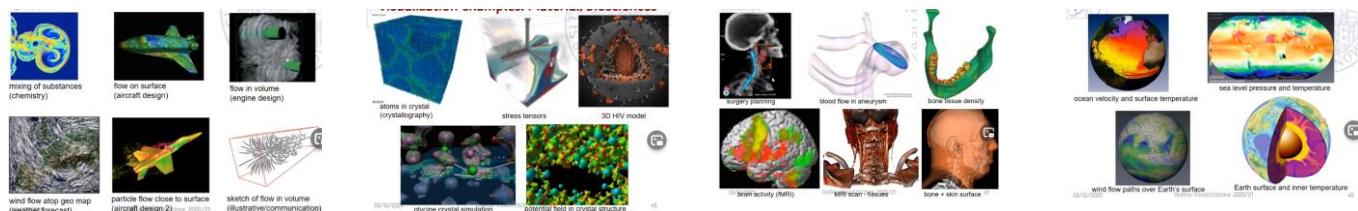


Servono fasi in più oltre al semplice rendering visto fino ad ora: vi si aggiunge anche la parte di interattività e una parte di pre-processing dei dati: i numeri non sono solo un modello geometrico, ma hanno anche altri *attributi particolari*. Queste misure

andranno oltre il visibile nell'immagine, quindi ci sarà uno *step di filtraggio* che sceglierà quali parti mappare nella visualizzazione per creare un rendering che rappresenti non solo la struttura geometrica am anche le altre informazioni utili.

Generalmente si usano altri *strumenti specifici* per gestire questa parte.

Esempi:

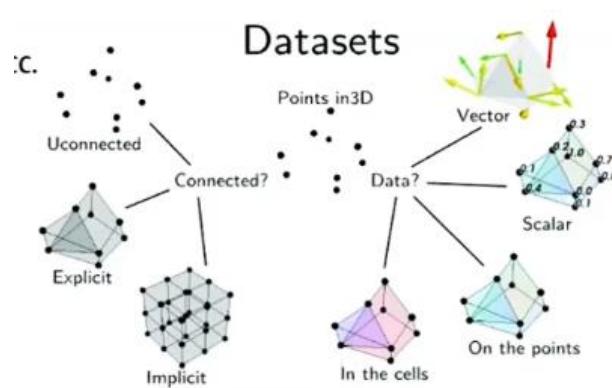


Fluid flow

Materiali e bioscienze

Medicina

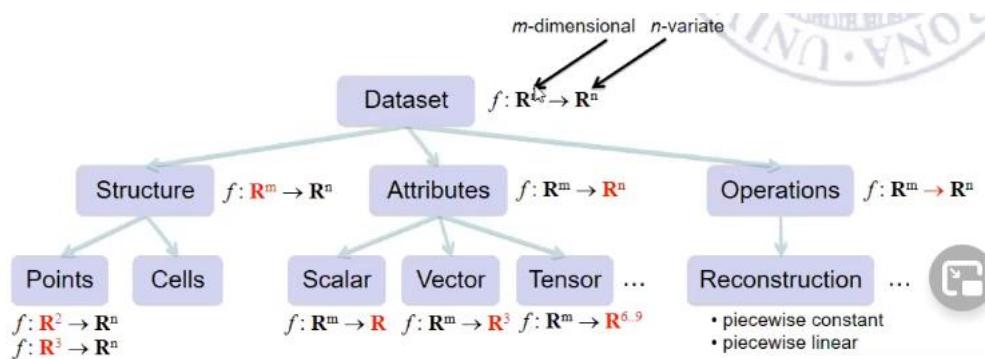
Geoscienze



- > Trasformare il dataset ottenuto in strutture dati visualizzabili: non partono mai come dati campionati su una mesh di superficie: magari faccio delle *misure puntuali sparse* che stanno su un volume
- > Trasformare questi campionamenti di misure nello spazio:
 - > In 2D posso mappare su immagini su immagini a schermo
 - > In 3D posso proiettare, ma può essere complesso rendere visibile ciò che vogliamo (prospettiva, occlusioni..)

Quello da cui partano sono dei dataset, ovvero il *campionamento di punti 2D o 3D di un dato*. Il tipo di dato può essere mappato su una griglia (e connettività/regioni di spazio) di punti o di celle. Gli attributi sono simili a quelli delle mesh, ma mappano delle misure numeriche di quantità fisiche.

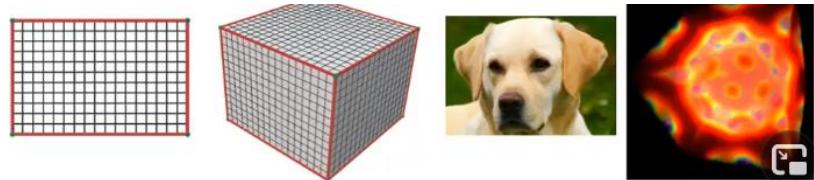
Dovrò anche preoccuparmi di *come ricavare la vera e propria stima delle variabili* e *come interpolare spazialmente le mie misure*



Griglie uniformi

Immagini raster (2D) e volumi voxelizzati (3D).

E' un campionamento discreto di qualunque variabile



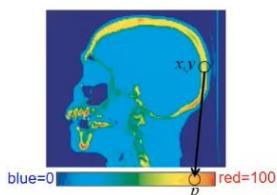
Anche in questo caso la pipeline grafica è (quella dell'altra volta,), e si implica che esista un *preprocessing* che consiste nel *selezionare l'informazione* che voglio far vedere e *mapparla* in qualche modo sul risultato. Di conseguenza, dallo stesso dato potrei avere mappature e campionamenti diversi, con risultati visivi differenti.

GRIGLIE 2D

Colormap

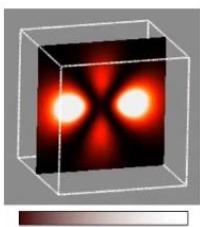
Si definisce colormap una lookup table che specifica i colori da associare a ciascun valore. [[integration](#)]

Percezione e scelta della colormap

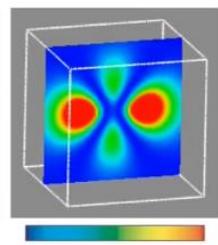


Vorremmo che consenta di *invertire intuitivamente la mappa*, cioè vedendo il colore dovrei capire facilmente il valore.

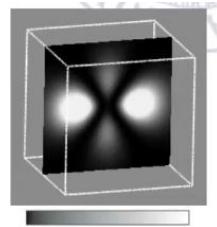
L'informazione è percepita diversamente in base anche al tipo di colore scelto; pur avendo la barra di riferimento, le *proprietà della vista umana* fanno sì che potrebbe non essere banale distinguere alcune sfumature.



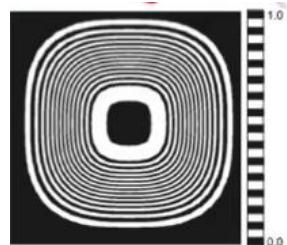
Heat colormap



Rainbow colormap



Greyscale colormap



Colormap a zebra

- » Ottima evidenziazione dei massimi
- » Valori bassi rappresentati meglio della mappa grigia

- » Non è chiaro subito quale tinta sia il massimo
- » Valori bassi separabili meglio
- » Problemi per daltonismo

Si usa spesso perché, magari, non si vuole effettivamente far capire quale sia la relazione fra le temperature ma evidenziare le regioni con un valore molto alto (= non capisco bene la differenza fra giallo e verde ma mi interessa il rosso).

- » Ottima evidenziazione dei massimi
- » Valori bassi poco chiari

Posso avere problemi se c'è una gamma color correction o meno: potrei avere che i livelli bassi sono molto molto poco distinguibili.

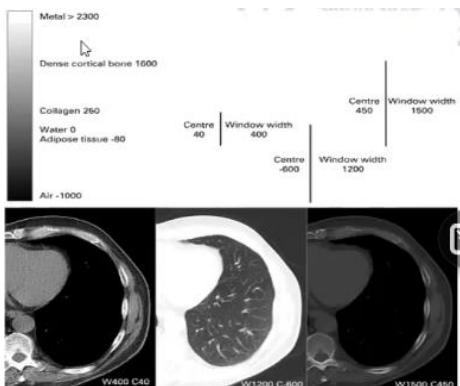
- » Evidenzio le variazioni locali e non i valori assoluti

(Bande sottili = cambiamento veloce)

Quindi, quando scelgo la colormap, devo farlo in modo da massimizzare:

- > Usabilità: grado in cui un prodotto può essere *usato da particolari utenti* per raggiungere certi obiettivi con efficacia, efficienza e soddisfazione in uno specifico campo d'uso.
 - » Tener conto dei "comportamenti-tipo" durante l'utilizzo di internet da parte dell'utente appartenente al *target* del progetto.
- > Accessibilità: possibilità da parte dei sistemi informatici di fornire i servizi anche a coloro che sono affetti da disabilità temporanee e non.
 - » Va in relazione all'aspetto tecnologico del sito e alla compatibilità dei codici dei vari linguaggi di programmazione con i programmi assistivi e di supporto per *disabili* e con hardware/software non aggiornati. [[integration](#)]

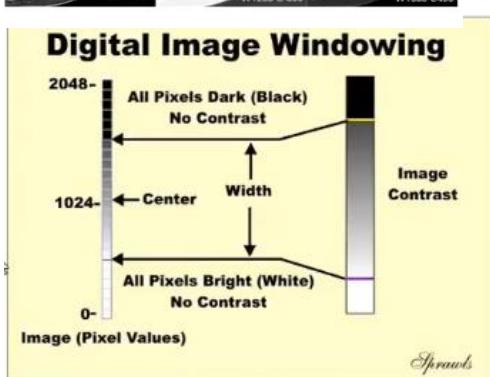
WINDOWING



È il mapping del range: consiste nel mappare linearmente un sottoinsieme del range misurato nella colormap.

le differenze fra i modelli della misura magari non sono mappabili 1:1 sulle differenze mostrabili dal display. Quindi, nel preprocessing prima di visualizzare il dato, può essere utile mappare i valori numerici cercando di far vedere solo il range di valori che ci interessa far vedere.

Esempio tipico in medicina: il valore numerico misurato nelle TAC è un indice della densità dei tessuti.

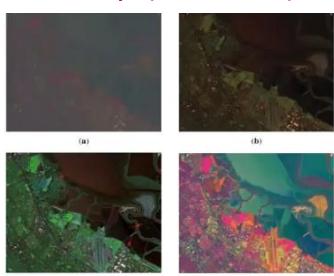


A seconda del task potrò scegliere una mappatura diversa.

Questo succede anche in fotografia: le immagini HDR permettono di acquisire un'immagine con un range dinamico superiore a quello da noi visualizzabile. Non hanno senso mappare linearmente i range: mi conviene farlo su pezzi per rendere più o meno visibili parti separate dell'immagine.

DATI MULTIDIMENSIONALI

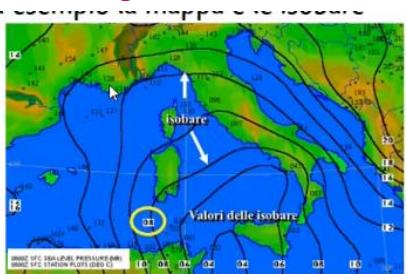
Colormap (di nuovo)



L'idea è mappare un segnale in N dimensioni in 3 dimensioni RGB. Perdo però completamente le direzioni (eventuali vettori)...

Più spesso si applica overlay, magari sovrapponendo linee a una colormap. Questo significa che i campi ulteriori non saranno visualizzati in maniera densa, ma verrà data un'idea dell'andamento della variabile.

Contouring



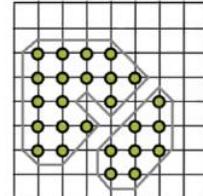
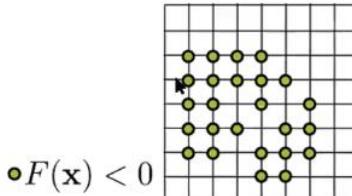
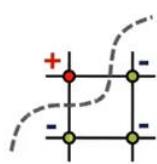
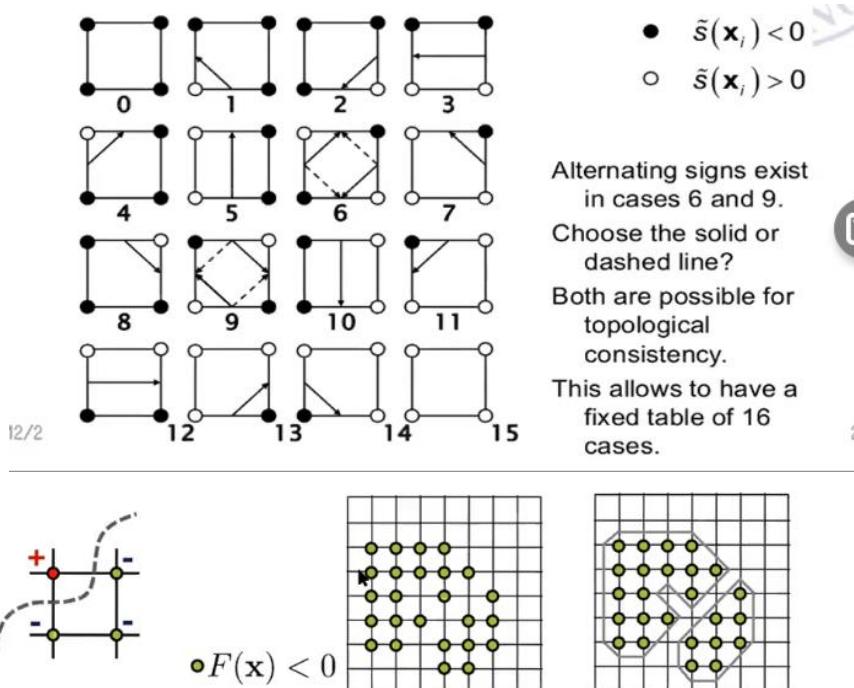
Ricavo un secondo campo sopra il primo mostrandone solo i contorni.

Il campo sovrapposto avrà spesso un campionamento misurato su una griglia diversa; è un problema risolto poiché data una seconda griglia posso ricavare il colore ricavato nella griglia dell'immagine base per interpolazione. L'approssimazione mediante interpolazione sarà accuratissima ma posso dare il problema per risolto.

Mi serve un algoritmo che mi permetta di calcolare le isoline da un campionamento!

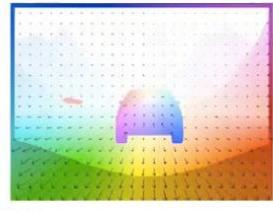
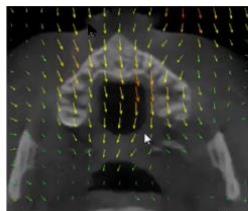
» MARCHING SQUARES

(ne esistono versioni anche per griglie non regolari)



Posso ricondurre ciascuna cella a una tabella di possibili pezzi di linea. Ci sono ambiguità (codice 6 e 9), che verranno risolte dopo in base alla continuità della curva.

Glifi (per griglie 2D)



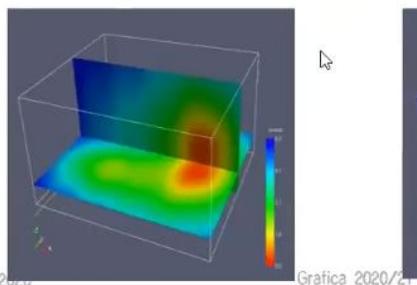
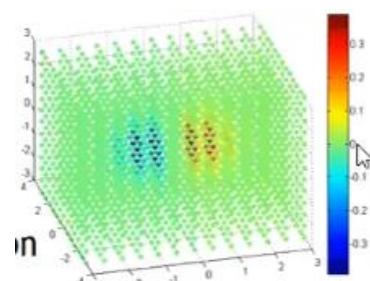
Consentono di vedere anche quantità vettoriali o tensoriali!
Prendo la mia immagine sottostante in 2D e vi sovrappongo un campo di vettori come frecce, misurata in ciascuna posizione

Ovviamente non si può rappresentare a densità piena, ma possiamo campionarle. Ovviamente a seconda di dove metto le frecce l'andamento può essere più o meno chiaro.

Griglie 3D

In 3D è un pasticcio: è difficile capire cosa ci sia “all'interno” a causa di occlusioni ed effetti prospettici.

Quello che si fa, tipicamente, è prendere i campionamenti volumetrici e li trasformo in superfici con texture, per esempio attraverso clip-plane che ci permettono di mappare i valori dei nodi sui punti di un piano (o altre superfici)



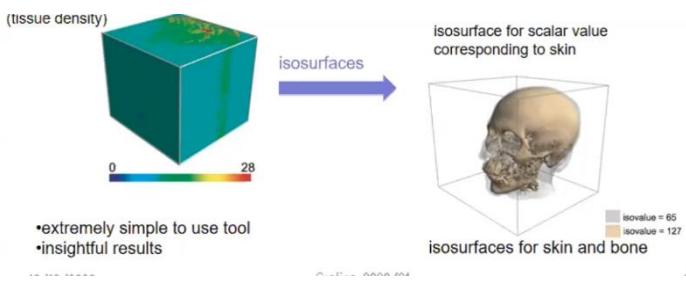
La mappatura non è del tutto ovvia!
Dovremo interpolare in una griglia definita su piano/superficie i valori definiti nei punti nello spazio.

Qui interpolare significa ricavare un valore nel piano in funzione di combinazione del vecchio campionamento nello spazio. Per

interpolare fra i punti posso fare intepolazione trilineare, identica alla bilineare nel texture mapping!
 Se i valori sono campionati con punti non regolari si usano tecniche diverse, che però non vediamo. (ad esempio faccio la triangolazione dei punti arrivando a creare tetraedri, e poi interpolazione con le coordinate baricentriche; oppure media pesata dei punti più vicini, simili alle radial basis function(?))

10/12/2020

Isosuperfici



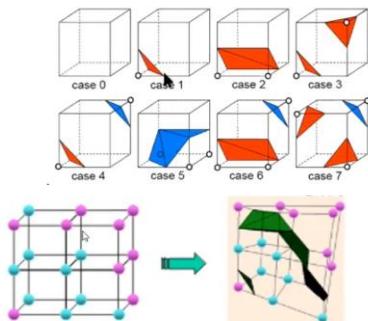
Un'altra possibilità è visualizzare con la grafica di superficie le superfici di livello, come visto in 2D per le linee di livello.

Si calcolano come in 2D! L'algoritmo più comune è il marching cubes, anche se in questo caso ci sono molte più combinazioni possibili e situazioni ambigue.

MARCHING CUBES

Marching Cubes (Lorensen and Cline '87):

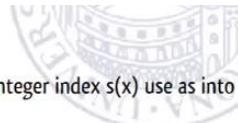
- Idea semplice: considero le celle cubiche formate da 8 vertici
- Avranno valore maggiore o minore della soglia (1/0)
- 256 possibili configurazioni
 - Ma molte sono le stesse, modulo rotazione o riflessione
 - A ognuna corrispondono ben determinati elementi di superficie



Create the lookup table

Loop over cells:

- find sign of for the 8 corner nodes, giving 8-bit integer index $s(x)$ use as into (256 case) table
- find intersection points on edges listed in table, using linear interpolation
- generate triangles according to table



Post-processing steps:

- connect triangles (share vertices)
- compute normal vectors
 - by averaging triangle normals (problem: thin triangles!)
 - by estimating the gradient of the field $s(x)$ (better)

Ovviamente, possono esserci ambiguità.

Dopo aver inserito gli elementi, sposto i triangoli per “unirli”.

Problemi:

- > Ho difficoltà in alcuni casi ambigui, come quando ho valori 1/0 ai vertici opposti diagonali della cella: se scelgo una delle due possibilità posso ottenere topologie diverse della mesh e in particolare rinire con buchi o errori topologici. Alcune ambiguità derivano, nell'algoritmo originale, dal fatto di usare soli 14 casi: già prendendone 23 (considero in modo diverso le configurazioni che variano per una riflessione)
- > Non posso informare su tutto lo spazio -> posso sfruttare la trasparenza ma chiaramente è limitato.
- > Luci?? -> me le invento tanto non mi interessa il realismo

Rendering volumetrico diretto

Nasce per cercare di risolvere il problema delle occlusioni. E' molto utilizzata in ambito biomedico: è una tecnica di computer graphics che evita di fare il rendering sulla superficie.

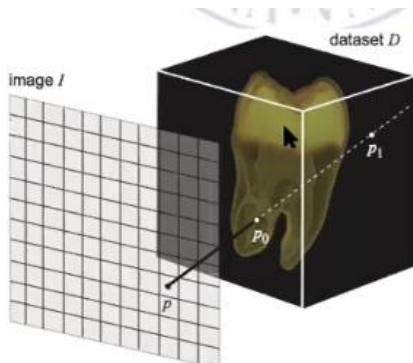
Cerca di risolvere i problemi delle tecniche viste precedentemente

<ul style="list-style-type: none"> » Se faccio solo superfici (slicing o anche solo superfici esterne) perdo informazioni sull'insieme » Potrei applicare un alfa a tutti i piani tagliati 	<ul style="list-style-type: none"> » Se faccio solo il volume perdo info sulla topologia interna <ul style="list-style-type: none"> > Con le tecniche già viste possiamo vedere più superfici in trasparenza (es. ossa + muscoli trasparenti), ma accumulando più contorni diventa pesante... Può andare bene se allineo le fette al piano di immagine (allineandomi sugli assi rischio di avere le fette perpendicolari al piano immagine e non vedere niente) > E' comunque un pasticcio: se muovessi la telecamera ricomincerei da capo.

La soluzione è di applicare il paradigma del raycasting non alle superfici ma a tutto il volume

→ faccio il raycasting come quando faccio il rendering di superficie; in raycasting si tracciavano i raggi e si assegnava al pixel dell'immagine il valore incrociato da un raggio.

In *volume rendering* traccio il raggio, ma anziché fermarmi al primo volume il mio modello discretizza una proprietà del volume e decido di fare uno shading che campionando un certo numero di punti numero il raggio riprende tutti i valori incontrati. Le immagini create non sono una vista prospettica, ma **un'approssimazione bidimensionale di tutta l'informazione che si incontra lungo il raggio**. Mappare questa info permette di rappresentare parecchie informazioni utili.



Consideriamo volume $s : D \rightarrow \mathbf{R}$
da disegnare sull'immagine I

Per ogni pixel $p \in I$

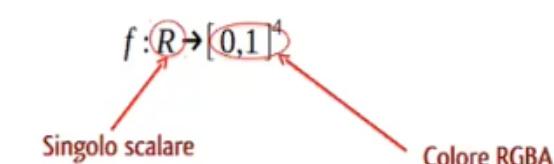
- Costruire un raggio \mathbf{r} ortogonale a I passante per p
- Calcolare intersezioni p_0 e p_1 di \mathbf{r} con D
- esprimere $I(p)$ in funzione di s lungo \mathbf{r} tra p_0 e p_1

Seguirò la procedura di raycasting, ovvero *lancio un raggio per ogni pixel* dell'immagine. Mi servirò di:



Funzione raggio:

Definisce come aggregare le misure prese lungo il raggio, ovvero *per tutti i campioni che incontro associa un colore da mandare all'immagine*



Funzione di trasferimento:

Mappa lo scalare ottenuto in un colore.

Fra un punto e l'altro (griglia immagine \neq griglia dei dati) farò interpolazione.

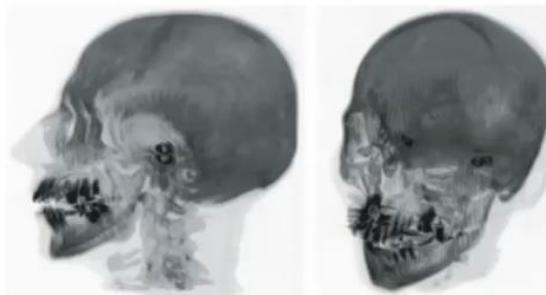
Esempi di funzione raggio

Il modo in cui vengono composti i vari campionamenti dipende da quello che voglio ottenere:

- > Massimo o minimo:

$$I(p) = f(\max_{t \in [0, T]} s(t))$$

Per esempio rendere visibile qualcosa di molo denso. Utile perché vedo "attraverso", ma perdo del tutto la visione di tridimensionalità – posso risolvere muovendo la vita (quindi vedo la profondità per parallassi).



Esempio
MIP di CT della testa

white = low density (air)
black = high density (bone)

Utile, ma non si percepisce
profondità
(se si muove sì)

- > Media lungo il raggio:

$$I(p) = f\left(\frac{\int_{t=0}^T s(t) dt}{T}\right)$$

Faccio la media di tutti i valori. E' come se facessi una radiografia bidimensionale:



Human torso CT
black = low density (air)
white = high density (bone)
Average intensity projection
è equivalente a singola
acquisizione X-ray

- > Distanza dal primo valore superiore a una certa soglia

- > Attribuire un valore (fisso) se c'è un valore maggiore di una soglia (es. solo se densità maggiore di un valore)

> Possiamo poi pensare di aggiungere uno shading per rendere migliore la visualizzazione tridimensionale -> volumetric shading: in ciascun punto del volume definisco una normale



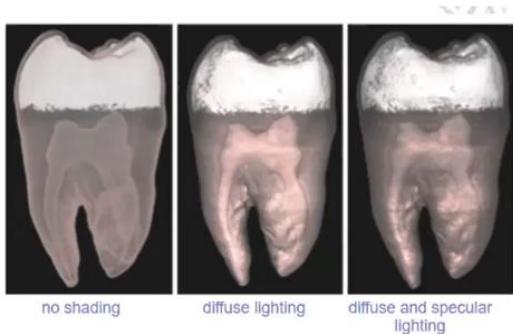
isosurface
) (hardware ray casting)

$$\vec{n} = \nabla_s(t) = \left(\frac{\partial_s(t)}{\partial_x}, \frac{\partial_s(t)}{\partial_y}, \frac{\partial_s(t)}{\partial_z} \right)$$

La direzione della massima variazione è data dal gradiente della funzione scalare!

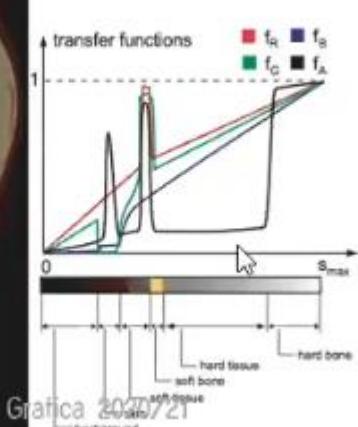
Da qui posso ricavare un algoritmo di shading. Si può verificare che l'isosuperficie risultante dal ray casting volumetrico funziona meglio.

Posso inoltre aggiungere particolari effetti facendo il compositing dei vari elementi della visualizzazione, cambiando la ray functione transfer function.



Esempi di funzione trasferimento

Prendo i valori campionati e li compongo con una serie di funzioni che mappino, per esempio, i diversi valori in RGB-alfa.



**Esempio
Human head CT**

**Enfatizza ossa
e visualizza muscoli**

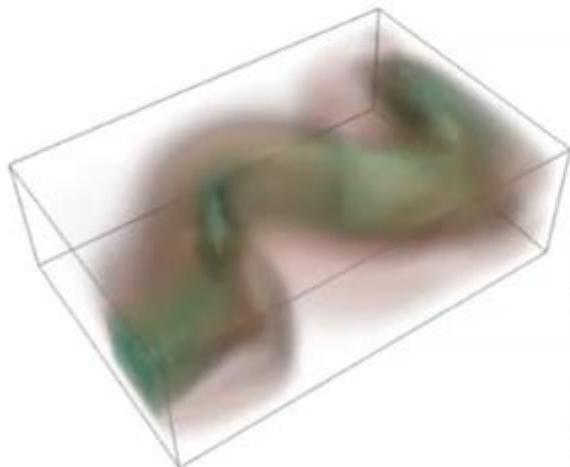
(Es. metto alfa=0 su tutto tranne che sui punti che hanno valore/densità come quello delle ossa)

Potrei mappare anche il modulo del gradiente (=evidenzio le discontinuità).

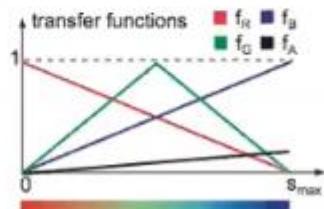
Inoltre posso aggiungere l'illuminazione stile phong (volumetric shading).

Posso applicare le stesse tecniche anche a dati non biometrici, come il flusso:

Volume rendering of fluid flow vector field magnitude



red = slow flow
green = more rapid flow
blue = fastest flow
Question: why is the blue hard to see?



Problemi di implementazione

In passato questo tipo di rendering era più raro, in quanto molto complesso (e non semplificabile via pipeline di renderizzazione). Ora ci sono anche tecniche in tempo reale.

La difficoltà dipende da:

- > Campionamento: se vado a densità troppo bassa = passo troppo grande rischio di avere aliasing! Ma una densità troppo alta mi porterebbe a un peso di calcolo eccessivo.



[#####]

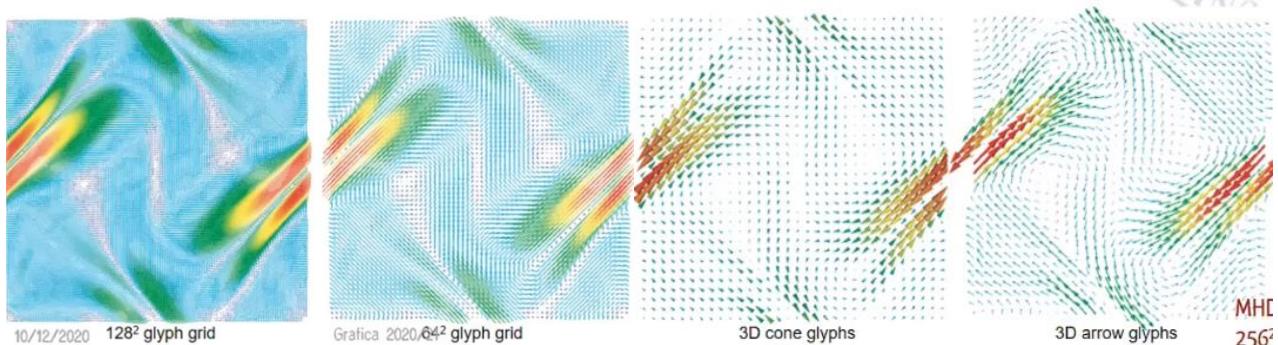
Vettori su colore

Esiste un solo caso in cui si utilizza la mappatura direzione->colore, ed è la codifica delle normali – già vista in Blender.

Glifi (3D)

In generale si tratta di una primitiva grafica in cui i parametri che caratterizzano sono mappati sulle componenti di un vettore (o tensore).

Di fatto sono le rappresentazioni dove disegno frecce, segmentini o coni orientati, eventualmente su cui mappo l'intensità con il colore.

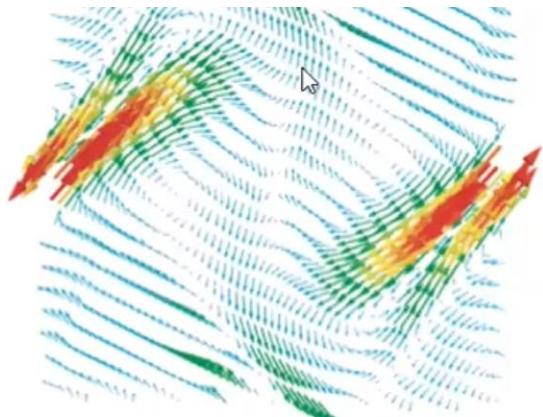


Problemi e relative soluzioni

» Scelta del campionamento:

- > Regolare
 - Rischio che mi si *sovrappongano* molto le frecce, oppure che per *aliasing* si generino dei pattern e arrivo ad avere
- > Random
 - Meglio :)
 - E' bene applicare un algoritmo pseudorandom che mi assicuri una *densità abbastanza*

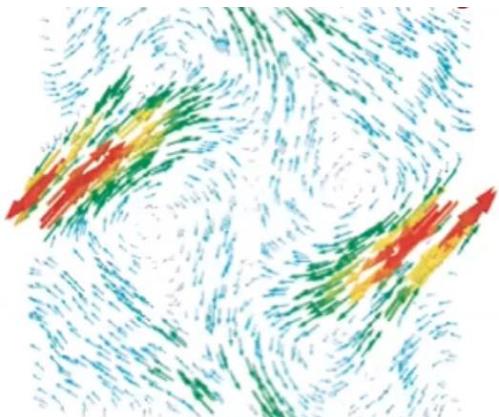
artefatti



2/2020

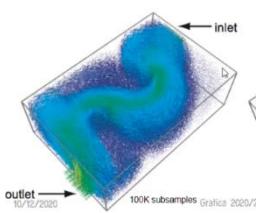
samples on a rotated grid Grafica 2c

costante.



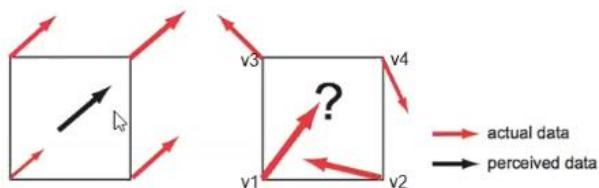
random samples, quasi-uniform density

» Occlusion



- > Scelgo appropriatamente il punto di vista
- > Trasparenza: riesco a togliere l'effetto di occlusione. Magari posso rendere gli elementi meno veloci più trasparenti.

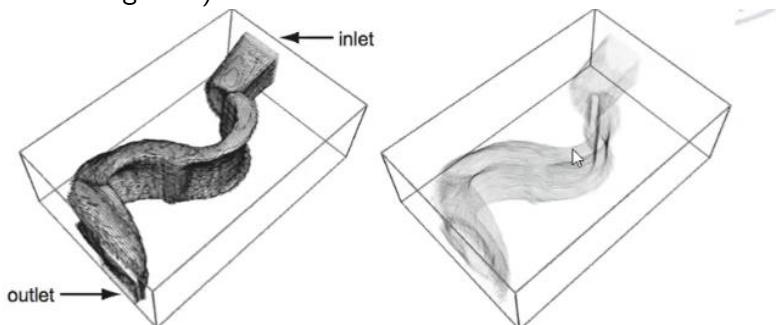
» Percezione dell'interpolazione dei vettori:



A differenza dell'interpolare "a occhio" il colore, farlo su vettori è estremamente meno intuitivo per l'uomo. E' dunque sostanziale posizionare i campionamento in modo che l'osservatore debba fare solo interpolazioni semplici.

»

- > Una soluzione è calcolare le frecce solo su superfici (tendenzialmente le frecce saranno uniformi I guess?)



10/12/2020

Grafica 2020/21

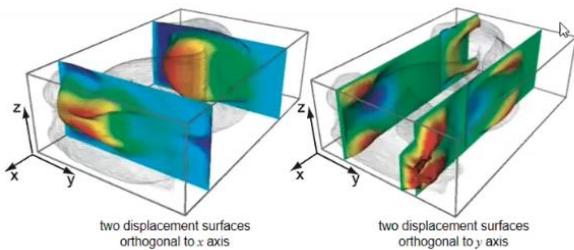
68

- > Altra soluzione è optare per visualizzazioni del tutto differenti, come il displacement plot

Displacement plot

Rappresento il campo velocità mostrando una superficie e come si sposterebbero i punti in essa

$$S_{\text{displ}} = \{x + \mathbf{v}(x)\Delta t, \forall x \in S\}$$

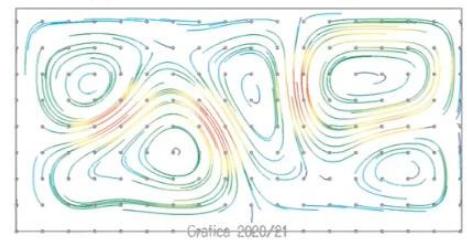


Problema: autointersezioni, spostamento limitato...

Stream lines

Valuto lo spostamento dei singoli punti seguendo i punti in maniera continua, facendo un integrale del loro spostamento nel campo (= suppongo che il campo sia stazionario).

Le traiettorie danno una buona idea di come è fatto il canale; inoltre possiamo anche appiccicare anche delle strutture tubolari che rendano percettibili le misure, o applicare colori per mostrare il modulo.



- > Non si sovrapporranno mai (campo statico)
- > Se faccio linee troppo lunghe perdo un po' di chiarezza.
- > Non vedo l'intensità (a meno di colori)

Se il campo non è stazionario posso fare lo stesso via particle traces.

- > La densità deve essere quasi uniforme
- > Perdo comprensione.. al solitò dovrò fare scelte appropriate per la visibilità.

