

Appunti di computer grafica

Lezioni precedenti al 30/09/2021 – Il sistema grafico (non chiede)

Cos'è il sistema grafico?

Il **sistema grafico** è un insieme di dispositivi hardware e software che interagiscono fra loro per produrre immagini grafiche. I *dispositivi hardware* rappresentano le risorse che rendono possibili la produzione delle immagini, mentre il *software* permette l'utilizzo dell'hardware per creare e manipolare immagini grafiche.

Il programma applicativo si interfaccia all'hardware del sistema grafico con funzioni di una libreria grafica, che prendono il nomi di API. I drivers della scheda video sono il software che permettono al sistema operativo di comunicare con la scheda grafica.

Device Raster

Un **device raster** è composto da una matrice rettangolare di campioni o pixel, e la risoluzione di un dispositivo raster misura la densità dei pixel. Possiamo classificare schermi e stampanti come device raster.

Immagini vettoriali vs immagini raster

Nella **grafica raster**, l'immagine è una griglia di pixel colorati.

I vantaggi della grafica raster stanno nel fatto che sono facili da modificare e da creare (siccome si lavora con i singoli pixel), tuttavia ingrandendo l'immagine è facile notare i pixel creando effetti sgradevoli e le modifiche possono sovrascrivere le informazioni di altri pixel, perdendo per sempre le informazioni della foto di partenza. Inoltre, per rotare queste immagini, la rotazione comporta una approssimazione dell'immagine originale, senza farla in modo preciso (almeno che ovviamente non si ruoti di 90°).

Nella **grafica vettoriale** (o ad oggetti) un'immagine è un insieme di costrutti geometrici (punti, linee, rettangoli, curve) ognuno dei quali è definito da un'equazione matematica. Per riprodurre l'immagine vettoriale su un dispositivo raster questa va però trasformata in pixel, e per farlo si esegue una operazione detta **rasterizzazione** (o **scan conversion**). Per le immagini bitmap invece non abbiamo un concetto di rasterizzazione, siccome sono già per definizione una griglia di pixel colorati.

Le formule matematica di una grafica vettoriale sono scritte in qualche linguaggio (es. PostScript), e ogni oggetto viene memorizzato in un database interno di grafici descritti matematicamente.

Con questo metodo, gli oggetti possono essere ingranditi, colorati e ruotati senza alcuna perdita di qualità. Inoltre, gli oggetti si possono trattare in modo indipendente e si possono mettere una sopra l'altro, senza perdere alcuna informazioni di partenza.

Per stampare una immagine vettoriale a schermo o su stampante, l'output sarà emesso alla risoluzione migliore possibile del device su cui verrà proiettata (è dunque **totalmente indipendente dalla risoluzione** della macchina su cui viene creata). Inoltre occupano meno spazio.

SVG (Scalable Vector Graphics) è uno standard per la rappresentazione delle immagini vettoriale, che vengono definite attraverso un file XML.

Raster Scan Display System

I sistemi grafici a display raster consistono solitamente di 3 componenti:

- **Monitor** (o display raster), connesso alla scheda grafica.
- **Scheda Grafica**, che contiene la **GPU** e una memoria RAM (**VRAM**) ad essa associata. Possiede inoltre una connessione alla scheda madre e, ovviamente, al monitor.
- Un **device driver**, mediante il quale il sistema operativo si interfaccia alla scheda video.

Come è fatta una scheda grafica?



Nella Scheda Grafica, la GPU è nascosta sotto una grande ventola, siccome dissipata molto calore..

Il compito della **GPU** è quello di elaborare le immagini eseguendo rapidamente gran parte dei calcoli geometrici e matematici richiesti, coordinando allo stesso tempo l'invio dell'immagine e facendo quindi in modo che arrivi al momento giusto.

Il compito della **VRAM** è quello di memorizzare le immagini in entrata prima di inviarle al monitor. In particolare, ciascuna locazione della VRAM memorizza i dati relativi a ciascun pixel, compreso il colore e la posizione su schermo. È dual-ported (permette lettura e scrittura in contemporanea). Possiede poi una porzione dedicata al **framebuffer**. [maggiori info qui]

La scheda grafica fa uso anche di un **BIOS** proprio, tramite un chip che memorizza le impostazioni della scheda video. Serve soprattutto per eseguire i test di diagnostica sulla memoria.

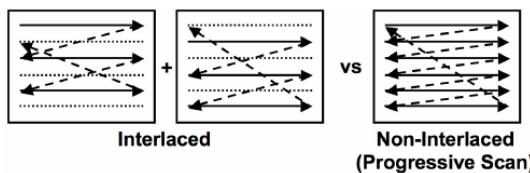
Il **DAC** è invece la componente che converte il segnale digitale generato dalla GPU in un segnale analogico utilizzabile dai monitor VGA, mentre nei monitor HDMI viene usato solo per la sincronizzazione verticale siccome i monitor HDMI sono già in grado di leggere il segnale digitale.

Caratteristiche dei monitor

Per quanto possano sembrare simili, c'è una differenza sostanziale fra framerate e refresh rate. Il **refresh rate**, misurato in hertz, rappresenta il numero massimo di frame (ovvero immagini) che possono essere visualizzati sullo schermo al secondo. Il **framerate** (misurato in FPS) misura quante immagini differenti al secondo vengono visualizzate. Per capire la differenza, un monitor può avere un refresh rate di 60 Hz, tuttavia il framerate della simulazione che viene visualizzata su quel monitor può essere anche 144 fps (anche se in questo modo la simulazione ci apparirà all'occhio come se stesse girando a 60 fps).

Possiamo avere due tipi di scansioni, ovvero modi in cui un'immagine viene caricata su schermo:

- la **scansione progressiva**, in cui l'immagine viene mostrata una riga per volta dall'alto verso il basso e da sinistra verso destra (ed è nativo nei display LCD).
- la **scansione interlacciata**, in cui prima si mostrano tutte le righe dispari della matrice e poi tutte quelle pari.



L'interlacciato è molto più prone allo **screen-tearing**, in cui singolo fotogramma viene visualizzato sullo schermo contenente informazioni da un altro fotogramma. Tuttavia, il progressive scan non è comunque esente da screen-tearing.

Framebuffer

Il **frame buffer** è una porzione della RAM presente nella scheda video, dove vengono memorizzate le immagini prima di essere visualizzate sul display. La componente principale del frame buffer è il **color framebuffer**, che contiene le componenti del colore per ogni pixel con tonalità diverse a seconda del numero di bit per locazione che contiene (es. 1bit se è monocromatico, 8bit per 256

tonalità etc..). Il color buffer può essere gestito in modalità *pseudocolor* (fa uso di una Look-Up Table con 2^N colori, e i valori memorizzati sono trattati come indirizzi alla Look-Up Table) oppure *truecolor* (i valori memorizzati sono i componenti effettivi dei colori, ovvero valori RGBA a 32 bit). La modalità pseudocolor è usata soprattutto quando si ha poca memoria.

Un altro buffer disponibile è lo **Z-buffer** (o **depth buffer**), che gestisce la visibilità degli oggetti su schermo. Per rendere le animazioni più fluide, molte volte si fa uso di un **doppio buffer** (anche triplo recentemente) che è composto da un **front buffer**, che contiene le informazioni sulla scena che sta per essere trasmesse, e un **back buffer** che contiene l'immagine subito successiva a quella nel front buffer. I buffer poi vengono scambiati dal processore grafico, e si aggiorna il back buffer.

Lezione 30/09/2021 – Prima lezione di laboratorio

Cos'è OpenGL?

OpenGL è una **specific**a (ovvero un elenco di funzioni che dicono cosa dare in input e cosa deve essere ritornato in output, ovvero se ne descrive il comportamento) che definisce un'API per differenti linguaggi e sistemi operativi, e che fornisce un ampio set di funzioni che possiamo usare per manipolare grafica e immagini.

Da questa specifica, i produttori hardware creano delle **implementazioni**, ovvero delle librerie di funzioni create rispettando quanto riportato sulla specifica OpenGL, facendo uso dell'accelerazione hardware (GPU) dove possibile. Ovviamente i produttori devono superare dei test specifici per poter fare in modo che questi siano qualificati per le implementazioni di OpenGL.

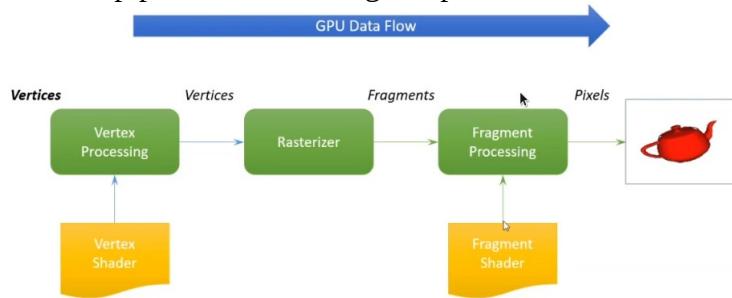
I produttori delle schede grafiche sviluppano le librerie di OpenGL, e ogni scheda grafica supporta delle versioni specifiche di OpenGL che sono le versioni di OpenGL sviluppate appositamente per quella scheda.

Sotto Linux esiste una combinazione di versioni di fornitori grafici e adattamenti di hobbisti (e secondo la prof se OpenGL mostra comportamenti strani è molto probabile che sia colpa dei produttori di schede grafiche).

Vulkan doveva essere la versione successiva di OpenGL (dopo la 4.6), e per questo fu inizialmente chiamata glNext, poi però è stata ribattezzata in Vulkan. È un API più di basso livello, e contiene dei driver meno pesanti, siccome in OpenGL viene allocato tutto l'hardware disponibile della macchina, mentre con Vulkan bisogna allocare le proprie risorse manualmente una ad una. Inoltre, mentre le applicazioni OpenGL girano, a lato CPU, con un solo core e senza multithreading, in Vulkan esiste il supporto al multithreading, ottimizzando anche la parte dell'applicazione eseguita dalla CPU. Altra cosa che permette Vulkan è la precompilazione degli shader, mentre con OpenGL vengono compilati a tempo reale [in realtà anche OpenGL permette di usare shader precompilati ma vabb].

OpenGL ES è il branch di OpenGL dedicato alle piattaforme mobili, e non permettono rendering immediato tramite pipeline fissa. WebGL è essenzialmente un sott'insieme di OpenGL ES che viene eseguito in una finestra del browser senza richiedere alcun plugin, ed è supportato da alcuni principali browser web. Attraverso l'elemento HTML5 canvas si ha accesso diretto al driver OpenGL.

Nel nostro corso si vedrà una pipeline di rendering semplificata,



Questo vuol dire che scriveremo shader solo per il Vertex Processing (attraverso i **Vertex Shader**) e nel Fragment Processing (attraverso i **Fragment Shader**), anche se con le versioni di OpenGL più moderne si può intervenire anche con il **Tesselation Shader** e il **Geometry Shader**, per arricchire di geometria i triangoli.

Cosa serve per sviluppare applicazioni OpenGL?

Consideriamo le applicazioni OpenGL sviluppano un output grafico su una finestra su uno schermo, dunque sarà necessario una libreria esterna che comunicherà col sistema di finestre del nostro sistema operativo (in caso di Linux, X11 o Wayland).

Useremo una libreria **GLUT** detta FreeGLUT, che ci permette di gestire tutte le operazioni di interfacciamento sulle finestre (un alternativa a GLUT è GLFW).

I sistemi operativi lavorano con le funzioni di libreria in modo diverso. Inoltre, OpenGL ha molte versioni e profili che presentano differenti funzioni, complicate da gestire. Dunque, usiamo una libreria per ridurre la complessità di accesso alle funzioni OpenGL e che lavora con le estensioni OpenGL. Una di queste librerie è **GLEW**, che fornisce meccanismi di run-time efficienti per determinare quali estensioni OpenGL sono supportate sulla piattaforma di destinazione.

Un'alternativa a questa libreria è GLAD.

Dunque dovremmo includere GL, GLU, GLUT e GLEW nella nostra applicazione.

Programmazione window-based

Tutti i programmi window-based sono controllati da **eventi**. Il programma dunque risponde a vari eventi, come il click del mouse, la ridimensione della finestra etc... e il sistema gestisce una **coda (FIFO)** di eventi, che riceve messaggi che affermano che certi eventi sono effettivamente avvenuti. Inoltre, ad ogni evento possono essere associate delle **Callback Functions**, ovvero funzioni di risposta. Quando il sistema rimuove un evento dalla coda, esegue la funzione di risposta associata all'evento. (es. la funzione `glutKeyboardFunc(myKeyboard)` chiama la funzione `myKeyboard()` quando si preme o si rilascia un tasto della tastiera, mentre la funzione `glutReshapeFunc(myReshape)` chiama la funzione `myReshape()` quando si ridimensiona la finestra).

Core-profile e modalità immediata

Inizialmente si usa la modalità immediata (ovvero la modalità che usava la *fixed pipeline*) per OpenGL, che era un metodo facile per disegnare grafica, e in questo modo la maggior parte delle funzionalità OpenGL era nascosta all'interno della libreria. Era però molto inefficiente, e per questo fu deprecata a partire dalla versione 3.2, motivando gli sviluppatori ad usare invece la modalità **core-profile**, che ha rimosso tutte le funzionalità obsolete, e che interrompe il disegno quando si usano le funzioni deprecate di OpenGL. Possiamo specificare il contesto all'interno di OpenGL attraverso funzioni apposite.

Fixed-pipeline e non-fixed pipeline

Prima del 3.1, OpenGL faceva uso di una “**fixed-pipeline**”, ovvero tutte le operazioni che OpenGL

supportava erano completamente definite, e un'applicazione dunque poteva modificare il loro funzionamento solo cambiando un insieme di valori di ingresso (come i colori o posizioni). L'ordine delle operazioni è stato sempre lo stesso. Il processo cambiò con OpenGL 2.0, in cui fu introdotto il linguaggio di shading (GLSL) che permette la programmazione di trasformazioni sui vertici (vertex shader) e lo shading di frammenti (fragment shader), rendendo così la fixed-pipeline obsoleta.

Shader: cosa sono

Gli **shader** sono dei microprogrammi che vengono eseguiti sulla GPU, e che consentono la programmabilità di diverse funzioni fasi della pipeline grafica moderna GPU. OpenGL prima del 3.0 supportava solo due tipi di shader, ovvero il vertex e il fragment, mentre OpenGL moderno (ovvero dopo la 3.0) che introduce il supporto a geometry shader, e OpenGL 4.0 aggiunge anche i tessellation control shader.

Siccome sempre più GPU venivano usate senza il fine di grafica 3D (es. calcolo AI, elaborazione immagini), le GPU moderne hanno due modalità di funzionamento: **modalità compute** o **modalità shader**. La prima permette alla GPU di funzionare come processori general purpose, mentre la seconda permette di lavorare come processori dedicati alla elaborazione grafica. Quando si utilizzano le funzionalità della versione più recente di OpenGL, solo le più moderne schede grafiche saranno in grado di eseguire l'applicazione. Questo è spesso il motivo per cui la maggior parte degli sviluppatori di solito prende come target versioni inferiori di OpenGL e abilita facoltativamente funzionalità delle versioni superiori.

Estensioni di OpenGL

Una grande caratteristica di OpenGL è il **supporto delle estensioni**. Ogni volta che un'azienda grafica presenta una nuova tecnica o una nuova grande ottimizzazione per il rendering, questa viene poi aggiunta come estensione implementata nei driver. Se l'hardware su cui viene eseguita un'applicazione supporta tale estensione, lo sviluppatore può utilizzare la funzionalità fornita dall'estensione per una grafica più avanzata o efficiente. In questo modo, uno sviluppatore grafico può ancora utilizzare queste nuove tecniche di rendering, senza dover aspettare che OpenGL includa la funzionalità nelle sue versioni future, semplicemente controllando se l'estensione è supportata dalla scheda grafica. Spesso, quando un'estensione è popolare o molto utile, alla fine diventa parte delle future versioni di OpenGL.

Macchina a stati

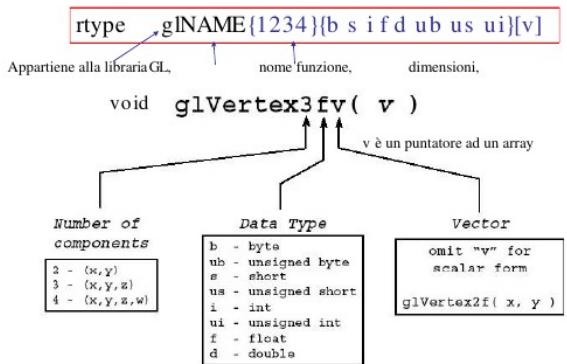
OpenGL è una grande macchina a stati: una raccolta di variabili che definiscono come OpenGL deve funzionare in ogni momento [questo sarà più chiaro quando vedremo i buffer OpenGL]. Lo stato a un determinato istante di OpenGL viene comunemente definito **OpenGL context**.

Ogni volta che diciamo a OpenGL che ora vogliamo disegnare linee anziché triangoli, cambiamo lo stato di OpenGL cambiando alcune *variabili di context* che impostano come OpenGL dovrebbe disegnare. Non appena cambiamo il contesto dicendo a OpenGL che dovrebbe tracciare linee, i comandi di disegno successivi disegneranno linee invece di triangoli.

Diversi tipi di rendering pipeline

Prima di tutto abbiamo il metodo con **immediate mode** (o anche detto **fixed-pipeline**). Questo ha alcune caratteristiche principali:

- Ogni volta che un vertice viene specificato dall'applicazione, la sua posizione viene spedita alla GPU.
- Crea un collo di bottiglia tra CPU e GPU
- Per disegnare gli stessi dati, bisogna spedirli nuovamente alla GPU (ridondanza).
- Fa uso di glVertex, glBegin/glEnd



```

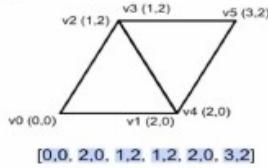
GLfloat red, greed, blue;
GLfloat coords[3];
glBegin (primType); % primType Tipo della primitiva geometrica
for ( i = 0; i < nVerts; ++i )
{
    gl Color3f( red, green,blue );
    glVertex3fv(coords);
}
glEnd();
  
```

Esempio dell'uso della fixed-pipeline

Un secondo tipo di rendering pipeline è l'uso dei **Vertex Buffer Object** (VBO), che memorizzano una determinata quantità di dati associati ai vertici, posizionali o descrittivi. Con un VBO è possibile calcolare tutti i vertici contemporaneamente, comprimerli in un VBO e passarli in massa a OpenGL per consentire alla GPU di elaborare tutti i vertici insieme. Possiamo usare i VBO con le

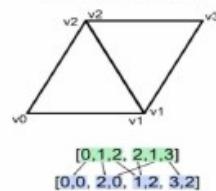
`glDrawArrays()`

DrawArrays (senza indexing)
%riduce il numero di chiamate a funzione



`glDrawElements()`

DrawElements (con indexing)
%riduce il numero di chiamate a funzione e l'uso ridondante di vertici condivisi.



funzioni associate `glDrawArrays()` e `glDrawElements()`, che riducono il numero di chiamate di funzione (inoltre il secondo elimina l'uso ridondante di vertici condivisi).

L'uso dei VBO permette di inviare grandi quantità di dati contemporaneamente alla scheda grafica e mantenerli nella memoria della scheda grafica, senza dover inviare dati un vertice alla volta. Ciò rende l'applicazione molto efficiente siccome la trasmissione dei dati alla scheda grafica dalla CPU è relativamente lento. Ogni VBO possiede un ID univoco corrispondente a quel buffer.

Ma cos'è un vertice?

Un **vertice** è una raccolta di attributi generici, tra cui coordinate posizionali, *colori*, *coordinate di texture*, normali e qualsiasi altro dato associato a quel punto nello spazio [metanota: colori e coordinate di texture saranno più chiare in avanti]. Come abbiamo visto, i dati dei vertici devono essere archiviati nei **Vertex Buffer Object (VBO)**. A loro volta, uno o più VBO devono essere archiviati in **Vertex Array Object (VAO)**. La posizione viene memorizzata in coordinate omogenee **a 4 dimensioni**.

Come usare VBO e VAO

I dati dei vertici devono essere archiviati in un VBO (questo deve essere fatto solo una volta) e quindi associati a un VAO. Per inizializzare un VBO e caricare il vettore di dati in esso:

In fase di dichiarazione

1. dichiarare una variabile dove mantenere l'id del VBO: `unsigned int VBO_1;`
2. generare un nuovo nome per il VBO e associarlo alla variabile:
 `glGenBuffers(1, &VBO_1);` in questo passaggio viene anche assegnato l'ID del VBO alla variabile `VBO_1`.
3. attivare il buffer: `glBindBuffer(GL_ARRAY_BUFFER, VBO_1);` [la funzione attiva il buffer, generandolo se necessario]. `GL_ARRAY_BUFFER` indica che stiamo usando un buffer object di tipo vertex (esistono infatti più tipi di buffer object).

- caricare il vettore dei dati (dichiarato come `vData`) nel VBO (sulla memoria della GPU) usando:
`glBufferData(GL_ARRAY_BUFFER, sizeof(vData), vData, GL_STATIC_DRAW);`
 Il secondo e il terzo parametro indicano la dimensione dei dati e dati effettivi da passare alla GPU, mentre il quarto parametro specifica come vogliamo che la scheda grafica gestisca i dati. Questo può assumere 3 forme che puoi vedere [qui](#).

In fase di rendering (ovvero, in `drawScene()`)

- bind VAO per usarlo in fase di rendering
`glBindVertexArray(vao)`
- E ora possiamo disegnare robaa sii!! con
`glDrawArrays(GL_TRIANGLES, 0, NumVertices);`

Per usare un VAO, dobbiamo eseguire una serie di passaggi:

- dichiarare il VAO: `unsigned int my_vao;`
- generare il nome del VAO chiamando la funzione `glGenVertexArrays(1, &my_vao);`
- impostare il VAO come attivo: `glBindVertexArray(my_vao);`

Durante il rendering:

- aggiornare i VBO associati a questo VAO.
- associare (bind) il VAO per usarlo nel rendering

Questi ultimi 2 passaggi vengono svolti nuovamente dalla chiamata di funzione

`glBindVertexArray(my_vao)`, che automaticamente fa anche il binding dei VBO (e EBO, che ancora non abbiamo visto) allocati in esso.

OpenGL: il lato pratico delle cose

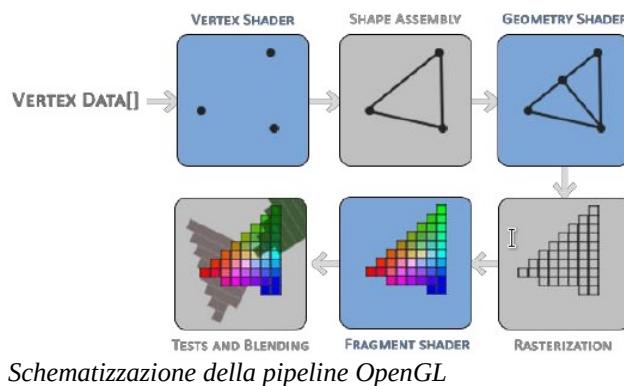
Possiamo dividere la pipeline grafica di OpenGL in due grandi fasi:

- la prima **trasforma le coordinate 3D in coordinate 2D (sottosistema geometrico)**
- la seconda parte **trasforma le coordinate 2D in pixel colorati reali (sottosistema raster)**

Dunque, potremmo dire che la pipeline grafica **prende come input un insieme di coordinate 3D e le trasforma in pixel 2D colorati sullo schermo**.

Ciascuno dei due grandi passaggi descritti sopra è diviso in tanti piccoli passaggi che appunto formano una pipeline, e che possono essere eseguiti in parallelo. Per questo motivo, le schede grafiche odierne sono divise in tanti piccoli core, ciascuno dei quali esegue piccoli programmi sulla per ogni fase della pipeline. Questi piccoli programmi sono, appunto, gli *shaders*. Alcuni di questi shader sono configurabili dallo sviluppatore che può scrivere i propri shader per sostituire gli shader predefiniti esistenti. Questo ci dà un controllo molto più preciso su parti specifiche della pipeline poiché funzionano sulla GPU, possono anche farci risparmiare tempo prezioso della CPU. Gli shader sono scritti nel Linguaggio di **Shading OpenGL (GLSL)**.

La pipeline OpenGL

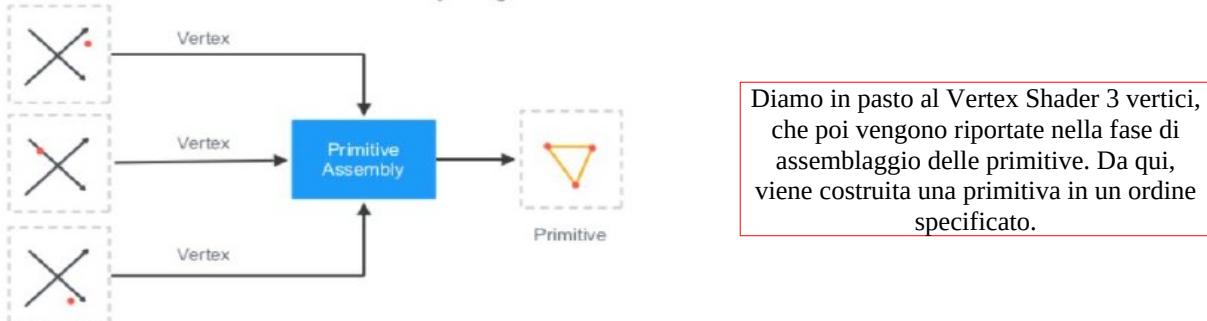


La prima parte della pipeline è il **Vertex Shader** che accetta come input un singolo vertice. Lo scopo principale del Vertex Shader è quello di trasformare le coordinate 3D in “altre” coordinate

3D (ovvero le cosiddette **coordinate di clipping**) e il Vertex Shader consente di eseguire alcune elaborazioni di base sugli attributi del vertice.

La fase di assemblaggio in primitive (**Shape Assembly**) accetta come input tutti i vertici (o il vertice, se si sceglie `GL_POINTS`) dal Vertex Shader che formano una primitiva ed assembra tutti i punti nella forma primitiva data (per esempio, un triangolo).

Per capire come funziona la fase di Shape Assembly, facciamo riferimento a questa figura:



Dopodiché, l'output della fase di assemblaggio delle primitive viene passato al **Geometry Shader**, che prende come input una raccolta di vertici che formano una primitiva e ha la capacità di **generare altre forme emettendo nuovi vertici per formare nuove (o altre) primitive**.

L'output del Geometry Shader viene quindi passato alla **fase di rasterizzazione** durante la quale le primitive risultanti **vengono mappate sui pixel** corrispondenti nella schermata finale, risultando in frammenti (**fragments**), e avviene il processo di interpolazione degli attributi sui frammenti a partire dagli attributi sui vertici (tra cui i colori di cui trattavamo all'inizio). Rappresentano l'input per il Fragment Shader.

I fragments rappresentano l'input per il **Fragment Shader**. Lo scopo principale del Fragment Shader è **calcolare il colore finale di un pixel** e di solito questo è lo stadio in cui si verificano tutti gli effetti OpenGL avanzati. Il Fragment Shader può contenere anche informazioni sulla scena 3D utili per calcolare il colore finale dei pixel (come luci, ombre, colore della luce e così via).

Dopo che tutti i valori di colore corrispondenti sono stati determinati, l'oggetto finale passerà attraverso un ulteriore stadio che chiamiamo **alfa-test e blending**. Questa fase controlla il corrispondente valore di profondità (z-buffer) (e stencil) del frammento e **li utilizza per verificare se il frammento risultante si trova davanti o dietro altri oggetti** e deve essere scartato di conseguenza.

Inoltre, in questa fase si controllano anche i **valori di alfa**, che definiscono l'opacità di un oggetto, e miscela gli oggetti di conseguenza [maggiori info su blending [qui](#)]

Il vertex e fragment shader sono SEMPRE da definire dal programmatore, in quanto non ci esistono shader di vertici o frammenti predefiniti sulla GPU.

Come scrivere un vertex shader

Prima di tutto, dobbiamo specificare 3 vertici ciascuno dei quali individuato da tre coordinate (x,y,z). OpenGL visualizza le coordinate 3D solo quando si trovano **in un intervallo specifico tra -1.0 e 1.0** (detto **range delle coordinate normalizzate o NDC**) su tutti e 3 gli assi (x, y, z).

Se per ciascun vertice la terza coordinata è uguale a zero, allora la profondità del triangolo rimane la stessa e si ha visualizzazione 2D. [disclaimer: Nelle applicazioni reali i dati di input NON sono generalmente già nelle coordinate del dispositivo normalizzate, quindi dobbiamo prima trasformare i dati di input in coordinate che rientrano nella regione visibile di OpenGL, ma per ora a noi questa cosa non importa].

Le coordinate dei tre vertici in devono essere definite in un array di float sottoforma di coordinate del dispositivo normalizzate, e verranno quindi trasformate in coordinate dello spazio dello schermo tramite la trasformazione della finestra passando i dati forniti dall'utente alla funzione `glViewport`. Ecco un esempio di Vertex Shader:

```
#version 430 core  
layout (location = 0) in vec3 aPos;
```

```

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}

```

Ogni shader deve cominciare con la dichiarazione della sua versione, specificando inoltre che stiamo usando il **core profile**. Successivamente dichiariamo tutti gli attributi del vertice in input nel vertex shader con la parola chiave `in`, in particolare creiamo una variabile `aPos` di tipo `vec3` che conterrà i dati in input. Specifichiamo la posizione della variabile in input tramite `layout (location = 0)`

Qualsiasi cosa che inseriamo come valore a `glPosition` viene usato come output del **vertex shader**. `glPosition` ha dimensione 4, dunque dovremo convertire l'input a 3 dimensioni in uno a 4 aggiungendo un parametro.

Compilare uno shader

Affinché OpenGL utilizzi lo shader, deve compilarlo **dinamicamente** in fase di esecuzione dal suo codice sorgente. La prima cosa che dobbiamo fare è creare **un oggetto shader**, a cui fa riferimento un ID. Per farlo possiamo usare la funzione `glCreateShader([TIPO_SHADER])`.

```

unsigned int vertexShader;
vertexShader = glCreateShader(GL_VERTEX_SHADER);

```

Quindi collegiamo il codice sorgente dello shader, all'identificativo dell'oggetto shader e compiliamo lo shader:

```

glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);

```

Il primo argomento indica l'ID dell'oggetto shader da compilare, il secondo argomento indica quante stringhe stiamo passando come codice sorgente, che è solo una, il terzo parametro è il codice sorgente effettivo dello shader.

Fragment Shader

Come sappiamo, il **Fragment Shader** consiste nel calcolare l'output del colore dei pixel. In questo primo esempio, per semplificare le cose, lo shader di frammenti produrrà sempre un colore arancione:

```

#version 420 core
out vec4 FragColor;
void main()
{
    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}

```

La parola chiave `out` specifica che l'output del fragment shader è il vettore di dimensione 4 "FragColor".

Il processo di compilazione del fragment shader è lo stesso del vertex shader, anche se dobbiamo specificare nella chiamata di `glCreateShader(GL_FRAGMENT_SHADER)`.

Non ci resta quindi che collegare i due programmi per renderizzare l'immagine.

Shader Program

Un **programma shader** è la versione finale collegata di più shader combinati, che dovrà essere attivato durante il rendering degli oggetti. Per farlo, usiamo un approccio simile a quello della normale creazione degli shader:

```

unsigned int shaderProgram;
shaderProgram = glCreateProgram(); // crea uno shader program

```

Non ci resta che collegare gli `shaderProgram` con gli altri due shader compilati:

```

glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);

```

Il risultato sarà un programma che possiamo attivare chiamando:
`glUseProgram(shaderProgram);`
prima di ogni draw call.

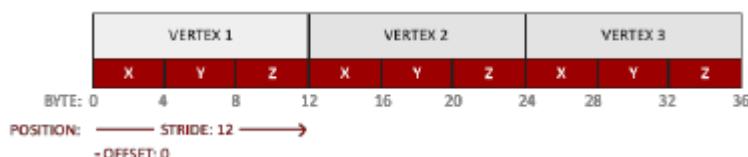
Pulizia e Vertex Attributes

È possibile eliminare gli oggetti shader dopo averli collegati all'oggetto programma.

```
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

In tutto questo, però, OpenGL non sa ancora come dovrebbe interpretare i dati dei vertici in memoria e come deve connettere i dati dei vertici agli attributi del Vertex Shader. Per fare esattamente questo, si fa uso dei **Vertex Attributes**.

È necessario specificare manualmente quale parte dei nostri dati di input va a quale attributo di vertice nel **Vertex Shader**. Ciò significa che dobbiamo specificare come OpenGL dovrebbe interpretare i dati del vertice prima del rendering. I nostri dati del buffer dei vertici sono formattati come segue:



Questo formato viene chiamato **stride** (trad. passo). Per ora, abbiamo inserito solo l'attributo della posizione. Ne vedremo delle belle quando avremo altri 3 tipi di attributi tipo (per colori, coordinate texture etc...).

Per dire a OpenGL come interpretare i dati del vertice, usiamo queste due funzioni:

```
glVertexAttribPointer (0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*) 0);
 glEnableVertexAttribArray (0);
```

Vediamo cosa vogliono dire i parametri nelle funzioni:

in `glVertexAttribPointer()`,

- il primo parametro specifica quale attributo del vertice vogliamo configurare (ovvero il primo, e l'unico) ed è 0 (corrisponde al layout nel vertex shader).
- il secondo specifica il numero dei parametri dell'attributo, il terzo il tipo di dato dei dati (ovvero float)
- il terzo indica lo stride, la dimensione di un dato attributo. Può anche essere visto come l'offset tra due attributi consecutivi del vertice.
- Il quarto rappresenta l'offset dell'inizio i dati di posizione nel buffer. Siccome i dati cominciano all'inizio dell'array di dati, il valore è 0.

La funzione `glEnableVertexAttribArray(0)` permette di abilitare l'attributo specificato nella funzione precedente, dandogli la posizione dell'attributo del vertice come argomento.

Appunti aggiuntivi

Riguardo alla sezione dei parametri dei VBO:

GL_STREAM_DRAW: i dati vengono impostati una sola volta e utilizzati dalla GPU al massimo alcune volte.

GL_STATIC_DRAW: i dati vengono impostati una sola volta e utilizzati più volte.

GL_DYNAMIC_DRAW : i dati vengono cambiati molto e utilizzati più volte.
Siccome i dati della posizione del triangolo nell'esempio di questa lezione non cambiano, possiamo usare **GL_STATIC_DRAW**.

Lezione 04/10/2021 – Matematica per la computer grafica

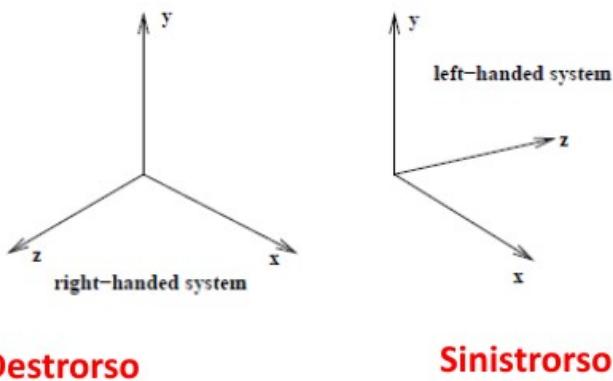
Introduzione

Come abbiamo già ripetuto un miliardo di volte, in computer grafica rappresentiamo gli oggetti nello spazio attraverso primitive lineari, come punti, linee, segmenti, piani e poligoni. Dunque, la geometria è un'aspetto importante nella computer grafica e avremo bisogno di sapere come trasformare gli oggetti, calcolare distanze, effettuare cambiamenti nei sistemi di coordinate...

Il sistema di coordinate

In computer grafica, per convenzione, si utilizza una rappresentazione che vede l'asse z come uscente dallo schermo e perpendicolare ad esso.

Esistono inoltre due modi di orientare gli assi nello spazio: **sinistrorso** e **destrorso**. Noi vedremo il **destrorso** nella modellazione e il **sinistrorso** nel rendering.



Destro

Sinistrorso

Durante il corso di CG, parleremo di **spazi vettoriali lineari** (in cui abbiamo due tipi diversi di oggetti, gli scalari e i vettori, a cui si aggiunge il concetto di prodotto interno) e **spazi affini** (che sono spazi vettoriali ma con l'aggiunta del concetto di punto). Lo spazio euclideo è uno spazio affine reale.

Definiamo i costrutti geometrici appena definiti in questo modo:

- **Scalare:** rappresenta una quantità specifica.
- **Punto:** entità il cui unico attributo è la posizione rispetto a un sistema di riferimento.
- **Vettore:** entità i cui unici attributi sono lunghezza e direzione, senza che esso abbia una posizione nello spazio (infatti posso traslare un vettore in modo indifferente).

Spazio vettoriale

Nello spazio vettoriale, come abbiamo detto, si hanno solo vettori e scalari. Definiamo con \mathbb{R}^n l'insieme dei vettori con n componenti reali. Per convenzione gli elementi di \mathbb{R}^n sono rappresentati come dei **vettori colonna**.

Le operazioni che possiamo fare con dei vettori sono:

- Somma di due vettori, che viene fatta con la regola del parallelogramma ed è commutativa.
- Il prodotto per uno scalare, che cambia la lunghezza del vettore.

Rappresentiamo i vettori come dei segmenti liberi, senza un punto di applicazione specifico.

Ripassino di algebra

Dati k vettori v_k e k scalari λ_k , la quantità $\lambda_1v_1 + \lambda_2v_2 + \dots + \lambda_kv_k$ si dice **combinazione lineare** dei vettori v_1, v_2, \dots, v_k con coefficienti $\lambda_1, \lambda_2, \dots, \lambda_k$.

I vettori v_1, v_2, \dots, v_k si dicono linearmente indipendenti se una loro combinazione lineare $\lambda_1v_1 + \lambda_2v_2 + \dots + \lambda_kv_k = 0$ solo per $\lambda_1 = 0, \lambda_2 = 0, \dots, \lambda_k = 0$, dunque se nessuno di essi può essere ottenuto come combinazione lineare degli altri.

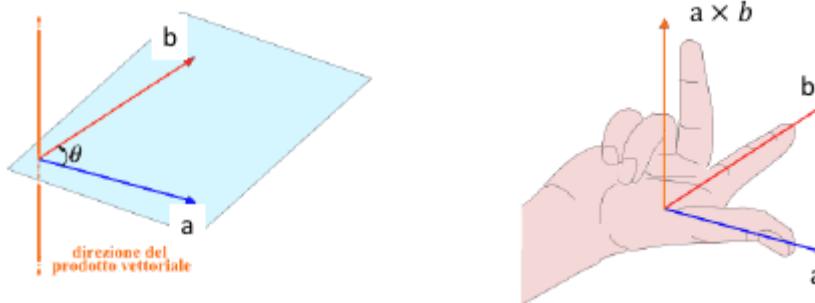
In uno spazio vettoriale V , la dimensione nello spazio è data dal numero massimo di vettori linearmente indipendenti. In uno spazio ad n dimensioni, un insieme di n vettori linearmente indipendenti forma **una base** per lo spazio. Data una base v_1, v_2, \dots, v_n , un qualunque vettore v in V può essere scritto come una combinazione lineare di tale base. Un esempio di base è la base canonica.

Il **prodotto scalare canonico** è definito come il prodotto “righe per colonne” che abbiamo visto al corso di algebra. Se il prodotto scalare di due vettori è nullo, allora due vettori si dicono **ortognali**. Per misurare la lunghezza di un vettore, usiamo la **norma 2** (o norma euclidea). Un vettore con lunghezza 1 è anche detto **vettore unitario**.

Per **normalizzare** un vettore, lo si moltiplica per il reciproco della sua norma euclidea. Un vettore normalizzato viene anche chiamato **versore**.

In geometria, possiamo anche definire il prodotto scalare (o **dot product**) tra due vettori come il prodotto fra la lunghezza dei due vettori e il coseno dell’angolo compreso tra i due vettori.

Il **prodotto vettoriale** (o cross product) tra due vettori è un vettore perpendicolare ad entrambi i vettori e ha direzione definita dalla regola della mano destra .



Il **cross product** è uguale a 0 se i due vettori sono paralleli.

Siano i, j, k la base canonica di R^3 , e siano a e b due vettori. Il prodotto vettoriale $a \times b$ è dato dalla matrice:

$$\begin{bmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix}$$

La norma a due del prodotto vettoriale rappresenta inoltre l'area del rettangolo individuato dai due vettori.

Spazi affini

I vettori però non rappresentano nulla dello spazio, ma solo degli spostamenti. Per poter introdurre il concetto di posizione, quindi, si deve passare agli **spazi affini**, che sono degli spazi vettoriali a cui aggiungiamo il concetto di punto. Oltre alle operazioni precedenti, quindi, vengono anche introdotti due nuove operazioni: [note su notazione: con le lettere maiuscole indichiamo dei punti]

- la **differenza punto-punto**, che definisce un vettore ($v = P - Q$). Possiamo vederlo come il vettore che congiunge due punti.
- la somma **punto+vettore**, che definisce un nuovo punto ($Q = P + v$).



L’operazione somma punto+punto non è definita.

Ovviamente, è molto importante non confondere punti e vettori.

Combinazioni affini

Una **combinazione affine** è una combinazione lineare di punti con coefficienti che hanno somma 1, ovvero:

$$P = \alpha_0 P_0 + \alpha_1 P_1 + \dots + \alpha_N P_N \text{ con } \alpha_0 + \alpha_1 + \dots + \alpha_N = 1.$$

I coefficienti $\alpha_0, \alpha_1, \dots, \alpha_N$ sono anche detti **coordinate baricentriche** (affini) di P nello spazio affine.

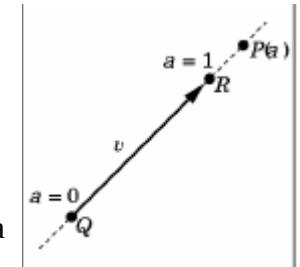
La combinazione affine di due punti distinti descrive la retta passante per i due punti. Siano Q e R due punti dello spazio affine reale e sia v il vettore da essi individuato. Possiamo dire che

$$v = R - Q$$

A questo punto consideriamo la loro combinazione affine, prendendo i coefficienti α, β tali che $\alpha + \beta = 1$. Dunque, possiamo scrivere P (che definisce un punto nella retta che va da Q a R) in questo modo:

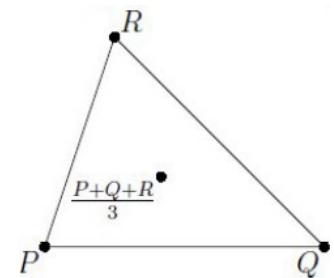
1. $P = \alpha R + \beta Q$
2. $P(\alpha) = \alpha R + (1 - \alpha)Q$, siccome $\beta = 1 - \alpha$
3. $P(\alpha) = Q + \alpha(R - Q)$
4. $P(\alpha) = Q + \alpha v$

In questo modo, $P(\alpha)$ rappresenta l'insieme dei punti definiti sulla retta che passa da Q a R .



La **combinazione convessa** è un tipo speciale di combinazione affine con **pesi esclusivamente positivi**. Nel caso della combinazione convessa di due punti, il punto risultante giace sul segmento che congiunge i due punti. Se i pesi sono entrambi pari a 0.5, il punto risultante si trova a metà tra i due, ovvero sarà coincidente al punto medio del segmento definito fra i due punti.

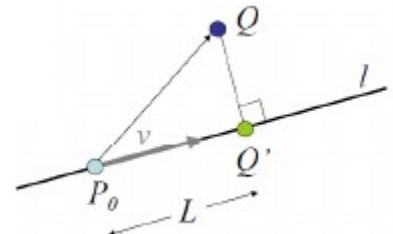
Nel caso di n punti che formano un poligono convesso, il punto risultante si trova **all'interno del poligono**. Se tutti i pesi sono uguali a $1/n$, il punto risultante si chiama **centroide** dell'insieme dei punti.



Calcolare la distanza fra un punto e una retta (ci serve per fare parallelismo coi piani)

In generale per calcolare la distanza fra un punto e una retta, possiamo usare facilmente il teorema di Pitagora.

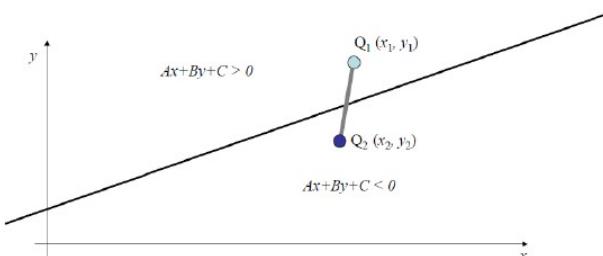
$$\begin{aligned} L^2 + \text{dist}(Q, Q')^2 &= \|Q - P_0\|^2 \\ L &= \frac{\langle Q - P_0, v \rangle}{\|v\|} \quad \text{Proiezione ortogonale di } Q - P_0 \text{ su } v \\ \Rightarrow \text{dist}(Q, Q')^2 &= \|Q - P_0\|^2 - L^2 = \|Q - P_0\|^2 - \frac{\langle Q - P_0, v \rangle^2}{\|v\|^2}. \end{aligned}$$



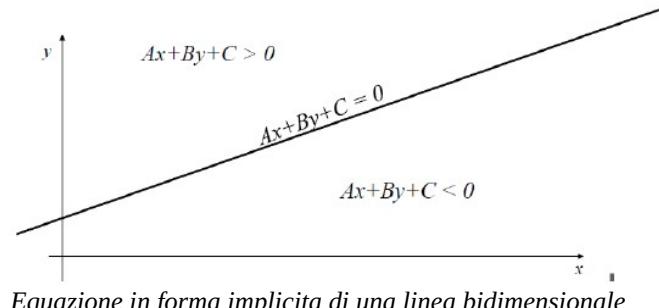
Altre proprietà della retta

Il segmento $Q_1 Q_2$ interseca la linea se e solo se
 $(Ax_1 + By_1 + C)(Ax_2 + By_2 + C) \leq 0$

$$Ax + By + C = 0, \quad A, B, C \in \mathbb{R}, \quad AB \neq 0$$



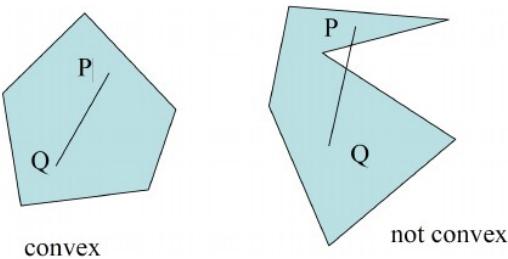
Modo semplice per vedere se un segmento interseca una retta



Equazione in forma implicita di una linea bidimensionale

Convessità

Un poligono si dice convesso se e solo se comunque presi due punti nell'oggetto tutti i punti sul segmento di linea tra questi punti sono anche nell'oggetto.



Dato un insieme di punti P_1, P_2, \dots, P_n , l'insieme di tutti i punti P che possono essere rappresentati come combinazioni convesse è detto **inviluppo convesso** (guscio convesso) dell'insieme. Il guscio convesso (convex hull) di un insieme di punti è la più piccola regione convessa che contiene tutti i punti dati.

Rappresentazione di piani nello spazio tridimensionale

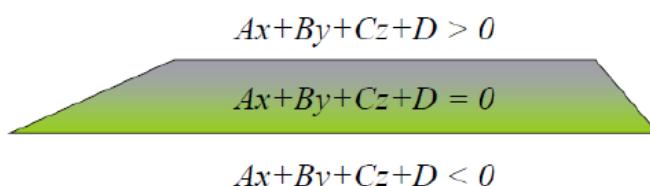
Un piano π è definito da una normale n ed un punto sul piano (P_0).

Un punto Q appartiene al piano se e solo se $\langle Q - P_0, n \rangle = 0$. La normale n è normale a tutti i vettori nel piano.

Per calcolare la distanza fra un punto Q e un piano π , è necessario proiettare Q sul piano nella direzione della normale al piano.

Possiamo anche rappresentare i piani usando la forma implicita, ovvero una formula con 3 incognite

$$Ax + By + Cz + D = 0, \quad A, B, C, D \in \mathbb{R}, \quad ABC \neq 0$$



Sistemi di riferimento e coordinate omogenee

Una base (tre vettori, linearmente indipendenti) non basta per definire la posizione di un punto: occorre anche un punto di riferimento, **l'origine** del sistema di riferimento. In questo modo, il concetto di base (ovvero dei 3 vettori linearmente indipendenti) si estende a quello di **riferimento (frame)** in uno spazio affine (o euclideo) specificando, oltre alla base, anche un punto P_0 detto **origine del riferimento**.

Dato un riferimento $F = (e_1, e_2, e_3, P_0)$, i punti ed i vettori dello spazio saranno esprimibili nel seguente modo:

$$\begin{aligned} P &= P_0 + v = P_0 + v_1 e_1 + v_2 e_2 + v_3 e_3 \\ v &= v_1 e_1 + v_2 e_2 + v_3 e_3 \end{aligned}$$

dove gli scalari (v_1, v_2, v_3) sono le coordinate del punto P nel sistema di riferimento (o frame) $F = (e_1, e_2, e_3, P_0)$. $\{e_1, e_2, e_3\}$ sono le basi canoniche dello spazio.

Siccome però rappresentare sia i vettori che i punti usando tre scalari è ambiguo, trovare un sistema di coordinate che porti ad una rappresentazione univoca per punti e vettori.

Ipotizziamo allora di poter definire il sistema di coordinate in questo modo:

$$\begin{aligned} P &= v_1 e_1 + v_2 e_2 + v_3 e_3 + 1 \times P_0 \\ v &= v_1 e_1 + v_2 e_2 + v_3 e_3 + 0 \times P_0 \end{aligned}$$

L'idea è quindi di aggiungere una **dimensione extra**. Ogni punto o vettore sarà così definito da 4 coordinate in questo modo (i seguenti sono dei vettori riga):

$$P: (v_1, v_2, v_3, 1)$$

$$v: (v_1, v_2, v_3, 0)$$

e questo rappresenta [come sono rappresentati i punti in uno spazio affine](#).

Lezione 07/10/2021 – OpenGL: precisazione su blending e curve parametriche (non chiede all'interrogaz.)

Introduzione

In questa parte di laboratorio impareremo come disegnare dei cerchi (e altre forme carine) con OpenGL, assieme a una introduzione delle interazioni con mouse e tastiera dell'utente con OpenGL.

Disegnare un cerchio

La formula che si usa per disegnare un cerchio, usando le formule di seno e coseno e conoscendo raggio e centro, è:

$$C(t) = \begin{cases} x(t) = r \cos(t) + c_x \\ y(t) = r \sin(t) + c_y \end{cases} \quad t \in [0, 2\pi]$$

In particolare, questa formula definisce le coordinate di ciascun punto della circonferenza.

Siccome normalmente usiamo dei triangoli per costruire forme in OpenGL, per adesso impareremo a disegnare un cerchio solo attraverso la costruzione di triangoli.

Dunque, la risoluzione del nostro cerchio dipenderà dal numero dei nostri triangoli.

Disegnare un cerchio

Un cerchio graficamente non può essere rappresentato con una funzione, siccome una funzione, dato un punto, ha una e una sola immagine. Dunque, per ovviare a questo inconveniente, si usano delle **equazioni parametriche** (ovvero quelle della foto qui sopra). In questo modo, ho una equazione parametrica $C(t)$ definita da due funzioni ($x(t)$ e $y(t)$) che al variare di t , mi ritornano le coordinate del punto che visita la circonferenza. In questo modo posso rappresentare situazioni in cui appunto ad un solo punto vengono associate due coordinate diverse.

$$C(t) = \begin{cases} x(t) = r \cos(t) \\ y(t) = r \sin(t) \end{cases}$$

Dunque, per disegnare una circonferenza, dovrò dividere l'intervallo in tante parti quanti punti voglio posizionare sulla curva, e il numero di punti corrisponde a un passo da un t_k a un t_{k+1} . Il numero dei vertici corrisponde al numero di vertici di [un poligono equilatero inscritto nella circonferenza](#). Per questo motivo, il numero di punti con cui definisco la curva corrisponde alla "risoluzione" della curva.

Alcune precisazioni su funzioni di OpenGL

- **glPolygonMode(face, mode)**: controlla la rasterizzazione dei poligoni.
Il parametro **face** descrive a quali facce del poligono deve essere applicata la rasterizzazione. Se specifichiamo come valore di face **GL_FRONT_AND_BACK**, allora la modalità di rasterizzazione viene applicata alle facce davanti e dietro di ogni poligono.
Il parametro **mode** ci permette di specificare quale modalità di rasterizzazione applicare.
Tra i valori che può assumere, abbiamo **GL_POINT**, **GL_LINE**, **GL_FILL**. [una specifica dettagliata di cosa fanno ciascuno di questi valori si può vedere [qui](#)].
- **glDrawArrays(GLenum mode, GLint first, GLsizei count)**, che abbiamo già visto in precedenza, prende in input **mode**, che ci dice il modo in cui vogliamo che le

primitive vengano disegnate, e poi **first** e **count**, che ci dicono l'indice del buffer da cui partire per completare il disegno e il numero di indici da renderizzare per poligono rispettivamente.

mode può assumere questi valori:

- **GL_POINTS**: permette la visualizzazione per punti.
- **GL_LINES**: permette di renderizzare il disegno come linee (collegando i vertici a due a due)
- **GL_LINE_STRIP**: collega i vertici fra loro in sequenza con delle linee.
- **GL_LINE_LOOP**: collega i vertici fra loro in sequenza, ma poi collega l'ultimo con il primo chiudendo il poligono.
- **GL_TRIANGLES**: prende i vertici nella lista tre alla volta e disegna triangoli separati per ogni terna di vertici riempiti del colore corrente.
- **GL_TRIANGLE_STRIP**: disegna una serie di triangoli basati su terne di vertici. Ogni vertice aggiuntivo determina un nuovo triangolo. I triangoli sono riempiti del colore corrente.
- **GL_TRIANGLE_FAN**: disegna una serie di triangoli connessi basati su terne di vertici, come se formassero un ventaglio (o un cerchio, dipende dai punti di vista).

Interazione con mouse e tastiera

GLUT fornisce alcune funzioni di risposta associata ad eventi input. In particolare, la posizione del mouse al momento del click o l'identificazione del tasto che è stato premuto vengono rese disponibili all'applicazione grafica e appropriatamente elaborate.

Tra le funzioni di elaborazione degli input che GLUT rende disponibile abbiamo:

- `glutMouseFunc(myMouse)` che registra `myMouse()` come la funzione di risposta che viene eseguita quando il bottone del mouse viene premuto o rilasciato.
- `glutMotionFunction(myMovedMouse)`, che registra `myMovedMouse()` come la funzione di risposta che viene eseguita quando il mouse viene mosso mentre uno dei bottoni è premuto.
- `glutKeyboardFunc(myKeyboard)` che registra `myKeyboard()` come la funzione di risposta che viene eseguita quando un tasto della tastiera viene premuto.

Ogni volta che usiamo una di queste funzioni, dobbiamo ricordarci che nella funzione che passiamo come input sarà necessario usare la callback function `glutPostRedisplay()`; che forza l'evento disegno, in questo modo la funzione `drawScene` (ovvero la funzione principale di rendering, che ovviamente può avere anche un nome diverso da questo) viene ridisegnata con il parametri aggiornati.

Abbiamo visto le funzioni che possiamo eseguire in caso di certi eventi in cui si preme un bottone del mouse oppure un bottone della tastiera, ma come facciamo ad inviare alla applicazione i dati veri e propri sulla posizione del mouse?

Per fare ciò, dobbiamo creare una funzione di risposta **void myMouse(int button, int state, int x, int y);** che viene chiamata ogni volta che accade un evento del mouse.

In questa funzione, x e y indicheranno le coordinate della posizione del mouse al momento dell'evento, button indica quale bottone del mouse è stato premuto (e assume i valori `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, `GLUT_RIGHT_BUTTON`). Il valore di state invece potrà essere `GLUT_UP` o `GLUT_DOWN`, e indica se il bottone è stato premuto (DOWN) oppure no (UP). C'è anche la possibilità di registrare il movimento della rotellina.

Per catturare la pressione di un tasto, dovremo specificare ancora un'altra funzione, ovvero **void myKeyboard(unsigned char key, int x, int y)** dove key indica il valore ASCII del tasto premuto e x ed y rappresentano la posizione del mouse al momento in cui è avvenuto l'evento.

Evento Idle

L'evento **idle** si verifica quando non avvengono altri eventi durante l'esecuzione del main loop. Un modo per gestire l'evento idle è l'uso della funzione **glutTimerFunc (unsigned int msecs , void (*func)(int value), value);** che esegue la funzione specificata nel secondo parametro dopo un almeno msecs millisecondi.

Funzione di riinizializzazione del framebuffer

Dopo che un evento è stato disegnato (o meglio, un frame è stato disegnato) si inserisce l'istruzione, alla fine del main loop, **glClearColor(GL_BUFFER_BIT);** che inizializza il framebuffer al colore specificato nel parametro della funzione.

Blending

Il **Blending** in OpenGL è una tecnica per implementare la trasparenza degli oggetti, e in generale il mescolarsi di più colori assieme.

La trasparenza riguarda gli oggetti (o parti di essi) che non hanno un colore solido, ma si presentano con un colore dato dalla combinazione di colori dell'oggetto stesso e di qualsiasi altro oggetto dietro di esso con intensità variabile. Ad esempio, una finestra di vetro colorato è un oggetto trasparente: il vetro ha un colore tutto suo, ma il colore risultante con cui si presenta mescola i colori di tutti gli oggetti dietro il vetro.

La **quantità di trasparenza** di un oggetto è definita dal **valore alfa** del suo colore, quarta componente del colore (se codificato in RGBA). Un valore alfa pari a 1 implica assenza di trasparenza, mentre un valore alfa pari a 0 implica la trasparenza totale dell'oggetto.

Nella pipeline grafica, dopo che è stato eseguito il fragment shader, l'oggetto finale passerà attraverso un ulteriore passo che si chiama **alfa test e blending** [ne abbiamo anche trattato [qui](#)]. Questa fase controlla i valori di alfa e ne miscela gli oggetti di conseguenza, oltre a fare il test dello z-buffer.

Il blending in OpenGL avviene con la semplice formula seguente:

$$C_{result} = C_{source} * F_{source} + C_{destination} * F_{destination}$$

dove:

C_{source} = colore di output del fragment shader.

$C_{destination}$ = colore attualmente memorizzato nel frame color buffer

F_{source} = valore alfa del colore di output del fragment shader

$F_{destination}$ = valore alfa del colore attualmente memorizzato nel frame color buffer.

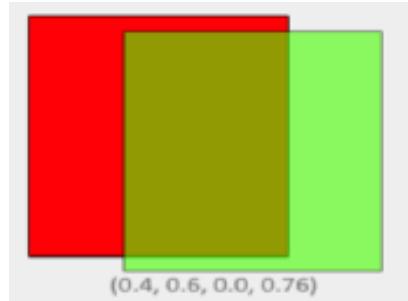
Dopo che il fragment shader è stato eseguito, l'equazione di blending viene eseguita tra l'output del colore del frammento generato dal fragment shader e con tutto ciò che è attualmente memorizzato nel buffer dei colori. F_{source} ed $F_{destination}$ possono essere impostati su un valore a scelta.

Facciamo un semplice esempio:

Abbiamo due quadrati: vogliamo disegnare il quadrato verde semitrasparente sopra il quadrato rosso. Il quadrato rosso sarà il colore di destinazione (e quindi già memorizzato nel frame color buffer) e ora disegneremo il quadrato verde (alfa=0.6) sopra il quadrato rosso (alfa=1.0). Dunque, possiamo moltiplicare il quadrato verde con il suo valore alfa (dunque $F_{source} = 0.6$) e il valore di $F_{destination}$ sarà quindi $(1 - 0.6 = 0.4)$, dunque il colore rosso contribuirà al 40% al colore risultante, mentre quello verde contribuirà al 60%.

Il colore risultante che si ottiene viene quindi memorizzato nel buffer colore, sostituendo il colore precedente.

In OpenGL, il blending non è abilitato di default, e quindi dovrà essere abilitato usando il costrutto



`glEnable(GL_BLEND);` che ci permette appunto di abilitare il blending.

La funzione `glBlendFunc(Glenum sfactor, Glenum dfactor)` specifica come vengono usati nell'equazione di blending i valori di F_{source} (`sfactor`) e di $F_{destination}$ (`dfactor`). In particolare, per ottenere lo stesso risultato dell'esempio dei quadrati, bisogna usare la funzione `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`.

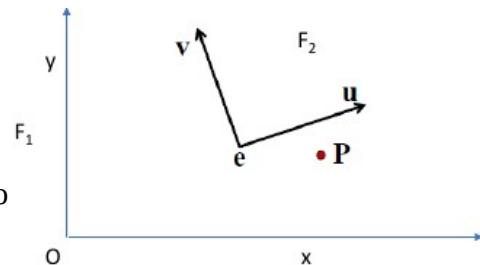
Lezione 11/10/2021 – Trasformazioni geometriche, curve parametriche, inizio curve Hermite

Cambiamento di sistemi di riferimento delle coordinate

Siano $F_1 = (x, y, O)$ ed $F_2 = (u, v, e)$ due frame di uno stesso spazio.

Supponiamo di conoscere le coordinate del punto P nel frame F_2 e vogliamo vedere quali sono le sue coordinate nel frame F_1 .

Per fare ciò, dobbiamo usare le **matrici di cambiamento del sistema di riferimento**. In poche parole, ci basta moltiplicare le coordinate del punto per la matrice del sistema di riferimento.



La matrice del sistema di riferimento M si ottiene esprimendo i vettori e origine di F_2 in termini di come dei vettori e un punto di F_1 .

$$\begin{aligned} u &= x_u x + y_u y + 0 \cdot O \\ v &= x_v x + y_v y + 0 \cdot O \\ e &= x_e x + y_e y + 1 \cdot O \end{aligned} \quad \longrightarrow \quad \begin{bmatrix} u \\ v \\ e \end{bmatrix} = \begin{bmatrix} x_u & y_u & 0 \\ x_v & y_v & 0 \\ x_e & y_e & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ O \end{bmatrix}$$

La matrice ottenuta è effettivamente la matrice M che ci serve.

La matrice ottenuta permette di cambiare il S.d.R. da F_1 a F_2 .

Facciamo un esempio per capire meglio:

Il punto P nel frame $F_1 = (x, y, O)$ avrà coordinate omogenee $a^T = (x_p, y_p, 1)^T$ e si esprimerà come:

$$P = x_p x + y_p y + 1 \cdot O$$

Il punto P nel frame $F_2 = (u, v, e)$ avrà coordinate omogenee $b^T = (u_p, v_p, 1)^T$ e si esprimerà come:

$$P = u_p u + v_p v + 1 \cdot e$$

P nel frame $F_1 = (x, y, O)$, in termini matriciali, si esprime come

$$P = a^T \begin{bmatrix} x \\ y \\ o \end{bmatrix}$$

Mentre P, nel frame F₂= (u, v, e), si esprime come

$$P = b^T \begin{bmatrix} u \\ v \\ e \end{bmatrix}$$

Siccome vogliamo che le due coordinate di P rappresentino effettivamente lo stesso punto, dobbiamo imporre questa uguaglianza:

$$a^T \begin{bmatrix} x \\ y \\ o \end{bmatrix} = b^T \begin{bmatrix} u \\ v \\ e \end{bmatrix}$$

Siccome sappiamo che $\begin{bmatrix} u \\ v \\ e \end{bmatrix} = M \begin{bmatrix} x \\ y \\ o \end{bmatrix}$, allora possiamo dire che: $a^T \begin{bmatrix} x \\ y \\ o \end{bmatrix} = b^T M \begin{bmatrix} x \\ y \\ o \end{bmatrix}$

Essenzialmente quindi:

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_e \\ y_u & y_v & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} \quad \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_e \\ y_u & y_v & y_e \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix}$$

Trasformazioni geometriche affini in 2D

Ogni trasformazione geometrica complessa può essere decomposta in una concatenazione di trasformazioni geometriche elementari quali **traslazione, scala e rotazione**. Queste trasformazioni modificano le coordinate dei punti di un oggetto per ottenerne un altro simile, ma differente per posizione, orientamento e dimensione. Bisogna tenere a mente che queste trasformazioni modificano **la geometria, ma non la topologia degli oggetti**, infatti la trasformazione di una primitiva geometrica si riduce alla trasformazione dei punti caratteristici (vertici) che la identificano nel rispetto della connettività originale (ovvero sono connessi sempre allo stesso modo, senza aggiungere nuovi punti e modificando così, appunto, la topologia dell'oggetto).

Le trasformazioni geometriche affini sono **trasformazioni lineari**, perciò vale la proprietà di linearità:

$$f(aP + bQ) = af(P) + bf(Q)$$

L'importanza della proprietà di linearità sta nel fatto che, note le trasformazioni dei vertici, **si possono ottenere le trasformazioni di combinazioni lineari dei vertici combinando linearmente le trasformazioni dei vertici**. Non dobbiamo ricalcolare le trasformazioni per ogni combinazione lineare (ovvero per ogni punto). Il sistema di coordinate noormalizzate di OpenGL (che appunto è quello che va da -1 a 1) ci vincola nella espressività della posizione degli oggetti.

Le trasformazioni lineari ci permettono inoltre il passaggio dal sistema di coordinate locali al sistema di coordinate del mondo. Infatti, ogni oggetto, a partire dal proprio sistema di riferimento (object space), viene trasformato opportunamente in un sistema di riferimento comune (world space) per andare a far parte dell'oggetto finale.

Traslazione, scalatura e rotazione

Traslare una primitiva geometrica nel piano significa muovere ogni suo punto P(x, y) di d_x unità lungo l'asse x e di d_y unità lungo l'asse y fino a raggiungere la nuova posizione P'(x', y') dove

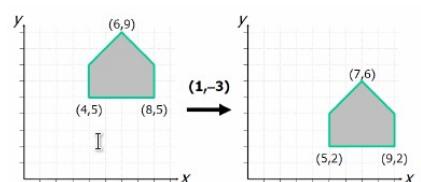
$$x' = x + d_x, \quad y' = y + d_y$$

In notazione matriciale:

$$P = \begin{bmatrix} x \\ y \end{bmatrix}; \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix}; \quad T = \begin{bmatrix} d_x \\ d_y \end{bmatrix};$$

$$\therefore P' = P + T$$

Dove T sarebbe il vettore traslazione.

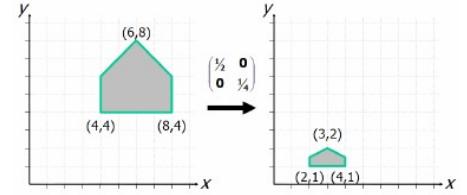


Scelto un punto C (punto fisso) di riferimento, **scalare** una primitiva geometrica significa riposizionare rispetto a C tutti i suoi punti in accordo ai fattori di scala s_x (lungo l'asse x) e s_y (lungo l'asse y) scelti. Se il punto fisso è l'origine O, la trasformazione da P in P' si ottiene in questo modo:

$$x' = s_x \cdot x, \quad y' = s_y \cdot y$$

In notazione matriciale $P' = S \cdot P$, dove

$$P = \begin{bmatrix} x \\ y \end{bmatrix}; \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix}; \quad S = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$



Se $s_x = s_y$, allora le proporzioni saranno mantenute e si parla di **scalatura uniforme**, altrimenti si parla di **scalatura non uniforme**.

Inoltre, con la scalatura, notiamo (nella figura) che se le dimensioni aumentano, allora l'oggetto si allontana dall'origine, altrimenti si avvicina: in poche parole, la scalatura in questo modo NON avviene rispetto al centro dell'oggetto!

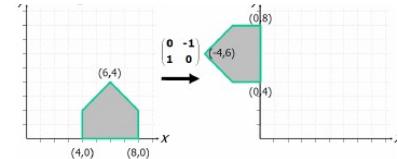
Una traslazione, per essere una trasformazione affine, dev'essere espressa come il prodotto di una matrice. Per come l'abbiamo definito per ora, dunque, la transazione non è una trasformazione affine (e quindi nemmeno lineare). Nel paragrafo successivo risolveremo questo.

Fissato un punto fisso C (detto pivot) di riferimento e un verso di rotazione (che può essere antiorario o orario), **ruotare** una primitiva geometrica attorno a C significa muovere tutti i suoi punti nel verso assegnato in maniera che si conservi, per ognuno di essi, la distanza da C. Una rotazione di angolo θ attorno all'origine O degli assi è definita come:

$$x' = x \cdot \cos \theta - y \cdot \sin \theta, \quad y' = x \cdot \sin \theta + y \cdot \cos \theta$$

In notazione matriciale abbiamo: $P' = R \cdot P$, dove

$$P = \begin{bmatrix} x \\ y \end{bmatrix}; \quad P' = \begin{bmatrix} x' \\ y' \end{bmatrix}; \quad R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$



Per convenzione, gli angoli sono considerati positivi quando misurati in senso antiorario, mentre invece sono negativi quando in senso orario.

Coordinate omogenee e traslazione

Come abbiamo detto prima, il fatto che la traslazione non sia una trasformazione affine (ovvero che non sia una moltiplicazione) rende più difficile le operazioni, in particolare nel caso di concatenazioni di rotazioni, scalature e traslazioni in un solo passo. In particolare, la rende incompatibile con il sistema di coordinate omogenee dei punti e vettori. Dunque, dobbiamo riscrivere le operazioni di trasformazione geometrica in questi altri modi:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{Traslazione}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{Scalatura}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad \text{Rotazione}$$

Traslazione: notare come sia cambiato
in modo radicale.

Altre operazioni: Riflessione e Deformazione (shear)

Altre operazioni sono la **riflessione** dell'immagini rispetto all'asse o all'origine

Riflessione rispetto all'asse y:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

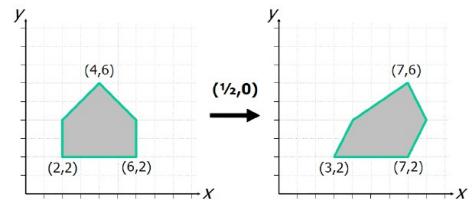
Riflessione rispetto all'origine degli assi:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

E la deformazione rispetto a un'asse (**shear**), che essenzialmente corrisponde alle relazioni
 $x' = x + ay$, $y' = y + bx$

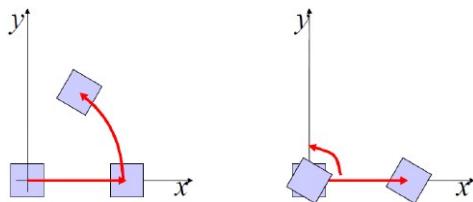
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Questo è una matrice
che permette la
deformazione rispetto
all'asse delle x



Composizione di Trasformazioni

L'ordine di concatenazione è importante siccome le trasformazioni sono associative ma **NON commutative**, (infatti ruotare e poi traslare è diverso da traslare e poi ruotare, come mostrato dalla figura qui sotto). La corretta sequenza delle trasformazioni T_1, T_2, T_3 si ottiene componendo T come $T = T_3 \cdot T_2 \cdot T_1$.



Ruotare e scalare attorno a un P generico (non l'origine)

Per ruotare attorno a un P generico, devo:

1. prima traslare P all'origine degli assi
2. poi eseguo una rotazione attorno all'origine
3. infine traslo dall'origine a P.

$$R_g = \begin{bmatrix} 1 & 0 & P_x \\ 0 & 1 & P_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -P_x \\ 0 & 1 & -P_y \\ 0 & 0 & 1 \end{bmatrix}$$

Per scalare attorno a un P generico, devo:

4. prima traslare P all'origine degli assi
5. poi eseguo una scalatura attorno all'origine
6. infine traslo dall'origine a P.

$$R_g = \begin{bmatrix} 1 & 0 & P_x \\ 0 & 1 & P_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -P_x \\ 0 & 1 & -P_y \\ 0 & 0 & 1 \end{bmatrix}$$

Invertibilità delle traslazioni (ripassino di algebra)

Sia M, una trasformazione tra quelle che abbiamo visto (traslazione, scalatura, rotazione,etc).

Poiché M è non singolare, allora esiste la matrice inversa M^{-1} tale che $MM^{-1} = I$, dove I rappresenta la matrice identità. Allora applicare ad un punto P' , trasformato di P mediante M, la matrice inversa della matrice di trasformazione M equivale a riottenere P. Se $P' = MP$, allora moltiplicando a destra e sinistra per M^{-1} otteniamo $P = M^{-1}P'$.

Fortunatamente per le trasformazioni viste le matrici inverse sono particolarmente semplici da calcolare.

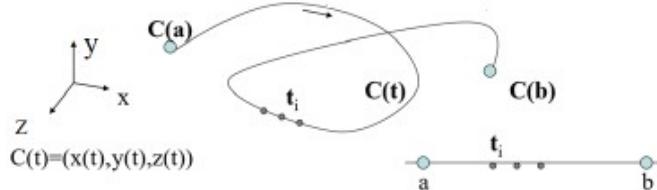
Infatti:

- L'inversa della matrice di trasformazione R di un angolo θ è data dalla matrice di rotazione di un angolo $-\theta$: $R(-\theta) = R^T(\theta) = R^{-1}(\theta)$ (R è infatti una matrice ortogonale).
- L'inversa della trasformazione di traslazione T di un vettore t: $T^{-1}(t) = T(-t)$.
- L'inversa della trasformazione di scala S: $S^{-1}(s) = S(1/s_x, 1/s_y)$.

Definizione di curva parametrica (essenzialmente una formalizzazione della lezione del 7/10)

Una **curva parametrizzata** in \mathbb{R}^3 , è un'applicazione $t \rightarrow (x(t), y(t), z(t))$ dove $x(t), y(t), z(t)$ sono funzioni continue del parametro $t \in [a, b] \subseteq \mathbb{R}$, dette **componenti parametriche della curva**.

Al variare di t , le coordinate $(x(t), y(t), z(t))$ individuano un punto che si sposta sulla curva.



Per fare un esempio:

$$C(t) = \begin{cases} x(t) = r \cos(t) \\ y(t) = r \sin(t) \end{cases}$$

Equazione parametrica del cerchio con centro all'origine e raggio r con $t \in [0, 2\pi]$

$$P(t) = \begin{cases} x = x_0 + t(x_1 - x_0) \\ y = y_0 + t(y_1 - y_0) \end{cases}$$

Equazione parametrica del segmento che congiunge due punti $P_0 = (x_0, y_0)$ e $P_1 = (x_1, y_1)$, con $t \in [0, 1]$

L'intervallo $I = [a, b]$ definisce un insieme di punti in cui si sceglie di visualizzare la curva anche se essa vive anche per valori al di fuori da questo intervallo: $(-\infty, +\infty)$. L'immagine in \mathbb{R}^3 tramite C dell'intervallo $I = [a, b] \subseteq \mathbb{R}$, $C(I)$, prende il nome di **supporto, sostegno o traiettoria** della curva.

Una curva si dice **regolare** se è **differenziabile** per ogni valore di $t \in I$ e se la **norma del vettore derivata** non è nulla in alcun punto di I .

Per capire meglio questo concetto, possiamo fare riferimento al modello fisico del moto della particella: ad ogni istante t_0 , le coordinate $(x(t_0), y(t_0))$ individuano un punto che si sposta sulla curva con velocità data dalla tangente alla curva parametrica in t_0 :

$$v(t_0) = \frac{dC(t)}{dt} = C'(t_0) = \begin{bmatrix} \frac{dx(t_0)}{dt} \\ \frac{dy(t_0)}{dt} \end{bmatrix} = \begin{bmatrix} x'(t_0) \\ y'(t_0) \end{bmatrix}$$

Calcolare la lunghezza di una curva

Per dare una approssimazione della lunghezza di una curva, possiamo considerare una serie di punti nella curva, definendo così una linea spezzata (o curva poligonale). La lunghezza di questa poligonale sarà data dalla somma delle lunghezze dei suoi lati.

Aggiungendo un punto, e ricalcolando la lunghezza della spezzata, ci avviciniamo sempre di più alla lunghezza della curva vera e propria.

Dunque, al tendere all'infinito, le lunghezze delle poligonalì associate formano una successione monotona non decrescente che converge all' integrale, che rappresenta quindi la lunghezza della traiettoria:

$$L(C) = \lim_{N \rightarrow \infty} L(P_N) = \sum_{i=1}^{N-1} \|C(t_{i+1}) - C(t_i)\| = \int_a^b \|C'(t)\| dt$$

Dunque, siano $x'(t), y'(t), z'(t)$ continue in $[a, b]$ le derivate prime delle componenti parametriche della curva $C(t) = (x(t), y(t), z(t))$, allora la lunghezza di C tra $C(a)$ e $C(b)$ è definita dalla seguente formula:

$$L = \int_a^b \|C'(t)\| dt = \int_a^b \sqrt{x'(t)^2 + y'(t)^2 + z'(t)^2} dt$$

Continuità geometrica e continuità parametrica di una curva

Se due segmenti di curva si uniscono ad un estremo, si dice che tra i due segmenti di curva c'è un **raccordo C⁰** che assicura l'assenza di salti.

Se due tratti di curva si uniscono in un punto P₀ e, inoltre, detti v₁ e v₂ i vettori velocità in P₀ del primo e secondo tratto di curva rispettivamente, si definisce

- **Continuità parametrica C¹:** le direzioni e i moduli dei vettori tangentici v₁, v₂ dei due segmenti curvi nel punto di contatto P₀ sono uguali.
- **Continuità geometrica G¹:** le direzioni dei vettori tangentici v₁, v₂ dei due segmenti curvi nel punto di contatto sono uguali, i moduli possono essere diversi.



Curve interpolanti di Hermite

Dati i punti del piano P_i, i = 0, ..., N, dove possiamo pensare che le coordinate del punto P_i possano essere considerate come valutazione di due funzioni parametriche x(t) ed y(t) nel valore del parametro t_i

$$p_i \equiv \begin{pmatrix} x_i = x(t_i) \\ y_i = y(t_i) \end{pmatrix}$$

Vogliamo costruire la curva C(t_i) interpolante i punti P_i, i = 0, ..., N:

Questo equivale a risolvere due problemi di interpolazione, uno per la funzione parametrica x e l'altro per la funzione parametrica y.

Scegliamo una base di funzioni φ per lo spazio in cui vogliamo costruire le funzioni interpolanti:

$$\text{Interpoliamo quindi le coppie } (t_i, x_i), i=0, \dots, N \quad \rightarrow \quad X_N(t) = \sum_{j=0}^N \alpha_j \varphi_j(t) \quad \text{tale che} \quad X_N(t_i) = x_i$$

$$\text{Interpoliamo quindi le coppie } (t_i, y_i), i=0, \dots, N \quad \rightarrow \quad Y_N(t) = \sum_{j=0}^N \beta_j \varphi_j(t) \quad \text{tale che} \quad Y_N(t_i) = y_i$$

$$C(t) = \begin{cases} X_N(t) \\ Y_N(t) \end{cases} \quad \rightarrow \quad C(t_i) = p_i \equiv \begin{pmatrix} x_i = x(t_i) \\ y_i = y(t_i) \end{pmatrix}$$

Dobbiamo quindi risolvere 3 problemi di interpolazione.

C(t_i) sarà la nostra curva interpolata finale.

Lezione 14/10/2021 – glm e GLSL (non chiede all'interrogaz.)

GLSL

Come abbiamo già accennato in precedenza, GLSL è un linguaggio di shading C-like. Non è l'unico linguaggio di shading al mondo, infatti per l'interfaccia DirectX si può usare HLSL. Essendo C-like, contiene anche molti tipi di strutture dati che possiamo utilizzare. Tra questi, abbiamo:

Tipi scalari: float, int, bool

Tipi vettori: vec2, vec3, vec4 (vettori di 2, 3 o 4 float)
ivec2, ivec3, ivec4 (vettori di 2, 3, o 4 interi)
bvec2, bvec3, bvec4 (vettori di 2, 3 o 4 booleani)

Tipi matrice: mat2, mat3, mat4

Texture sampling: sampler1D, sampler2D, sampler3D, samplerCube

Tipo di dato vettore

Per accedere alle varie componenti di un vettore, basta usare la sintassi con il punto come in C:

- È possibile accedere a un vettore per
 - .x, .y, .z, .w - posizione o direzione
 - .r, .g, .b, .a - colore
 - .s, .t, .p, .q - coordinate di texture
- color.rgb OK
- color.xgb ERRATO (non possiamo accedere ad x avendo gli accesso agli altri dettagli)

Abbiamo davvero tanti modi per dichiarare e costruire vettori in GLSL:

```
vec3 xyz = vec3(1.0, 2.0, 3.0);
vec3 xyz = vec3(1.0);           // [1.0, 1.0, 1.0]
vec3 xyz = vec3(vec2(1.0, 2.0), 3.0);
vec2 v = vec2(1.0, 2.0);        // composto da valori
v = vec2(3.0, 4.0);
vec4 u = vec4(0.0);            // inizializza tutti gli elementi a zero
vec2 t = vec2(u);              // prende le prime due componenti di u
vec3 vc = vec3(v, 1.0);        // composto da un vec2 ed un float
```

Possiamo anche accedere contemporaneamente a più elementi di un vettore:

```
vec4 a;
a.yz = vec2(1.0, 2.0);
a.xy = a.yx;
```

Tipi di dato matrici

La matrice viene memorizzata come una collezione di **colonne** (column-major) al contrario di C, in cui invece viene memorizzata come una collezione di righe. Tutti i tipi di matrice sono in virgola mobile.

Il tipo **matn** definisce una matrice quadrata con n righe, mentre **matnxm** definisce una matrice rettangolare con n colonne e m righe.

Ecco degli esempi di costruttori:

```
mat3 i = mat3(1.0); // matrice identità 3x3
mat2 m = mat2(1.0, 2.0, 3.0, 4.0);
```

E degli esempi di accesso agli elementi:

```
float f = m[column][row];
float x = m[0].x; // x è la componente della prima colonna
vec2 yz = m[1].yz; // yz sono componenti della seconda colonna
```

Operazioni e funzioni in GLSL

Le operazioni fra matrici e vettori sono molto semplici, infatti fare il prodotto fra matrici equivale a fare il prodotto fra due numeri:

```
vec3 xyz = // ...
vec3 v0 = 2.0 * xyz; // scale
vec3 v1 = v0 + xyz; // component-wise
vec3 v2 = v0 * xyz; // component-wise
mat3 m = // ...
mat3 v = // ...
mat3 mv = v * m; // matrice * matrice
mat3 xyz2 = mv * xyz; // matrice * vettore
mat3 xyz3 = xyz * mv; // vettore * matrice
```

GLSL permette di usare i comandi classici di C per il controllo di flusso (ovvero if else, while, do

while etc...).

Inoltre, ammette la definizione e l'uso di funzioni nel corpo dei suoi programmi. **Tuttavia, NON ammette l'uso della ricorsione.** Per le funzioni in GLSL bisogna anche aggiungere dei cosiddetti **qualifier**. In particolare, i qualificatori danno un significato speciale alla variabile.

- **in**: variabile di input che varia ad ogni chiamata dello shader (coordinata di vertici e loro attributi, colori, normali, etc)
- **out**: variabile di output (che sarà l'input per le fasi successive)
-

Ci sono poi delle funzioni builtin predefinite, tra cui funzioni trigonometriche:

```
float s = sin(theta);
float c = cos(theta); // gli angoli sono misurati in radianti
float t = tan(theta);
float theta = asin(s);
// ...
vec3 angles = vec3(/* ... */);
vec3 vs = sin(angles);
```

Abbiamo poi anche altre funzioni builtin, tra cui:

Come si può notare, contiene essenzialmente tutte le funzioni geometriche di cui abbiamo bisogno, infatti sono state studiate adhoc per la grafica.

```
vec3 l = ... // ...
vec3 n = ... // ...
vec3 p = ... // ...
vec3 q = ... // ...

float f = length(l); // lunghezza di un vettore
float d = distance(p, q); // distanza tra punti

float d2 = dot(l, n); // prodotto scalare
vec3 v2 = cross(l, n); // prodotto vettoriale
vec3 v3 = normalize(l); // normalizzazione di un vettore

vec3 v3 = reflect(l, n); // reflect
```

Funzioni geometriche generali

E altre funzioni ancora....

```
float xToTheY = pow(x, y);
float eToTheX = exp(x);
float twoToTheX = exp2(x);

float l = log(x); // ln
float l2 = log2(x); // log2

float s = sqrt(x);
float is = inversesqrt(x);
```

Funzioni esponenziali

```
float ax = abs(x); // absolute value
float sx = sign(x); // -1.0, 0.0, 1.0
float m0 = min(x, y); // minimum value
float m1 = max(x, y); // maximum value
float c = clamp(x, 0.0, 1.0);

// altre: floor(), ceil(),
// step(), smoothstep(), ...
```

Funzioni matematiche generali

Esempi di shader in GLSL – vertexShader

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform mat4 Model;
uniform mat4 View;
uniform mat4 Projection;
void main()
{
    gl_Position = Projection * View * Model * vPosition;
    color = vColor;
}
```

in: input dello shader che varia in base all'attributo del vertice

out: output dello shader

uniform: input dello shader che è costante lungo la chiamata glDraw

Per ora possiamo vedere le uniform come delle variabili globali, a cui possiamo accedere tramite il codice in C per poi modificarle in tempo reale. Le vedremo meglio dopo qui.

Un banale vertex shader che trasforma ogni vertice secondo la matrice **Model**, **View** e **Projection**. **Model** è la matrice che contiene tutte le nostre trasformazioni che l'oggetto deve subire prima che esso venga visualizzato. **View** per ora non ci interessa, mentre **Projections** mappa il nostro mondo a coordinate normalizzate.

Ogni esecuzione di `glDrawArray()` invoca lo shader con nuovi valori di vertice, in particolare le variabili di `in` cambiano ad ogni chiamata dello shader.

Esempi di shader in GLSL – fragmentShader

```
in vec4 color;
out vec4 FragColor;

void main()
{
    FragColor = color;
}
```

`in`: generato nel rasterizer
interpolando i colori nei
vertici della primitiva

Eseguito dopo il rasterizzatore, e quindi opera su ogni frammento di ogni primitiva visualizzata. Ogni frammento è stato generato dal rasterizzatore, che è un built-in, non programmabile. Quando `color` arriva nel fragmentShader dal vertexShader, questo viene interpolato (ovvero i colori vengono sfumati).

Uniform

Le **uniform** sono delle variabili globali il cui valore rimane costante rispetto ad una chiamata di `glDrawArray`, (cioè non cambia durante il rendering di una primitiva), che vengono passate dall'applicazione OpenGL agli shaders. Vengono utilizzate per condividere i dati tra un programma applicativo, vertex shader e fragment shader. Le variabili di tipo uniform possono essere lette ma non modificate dal vertex o fragment shaders.

Per comunicare il valore delle variabili dal programma allo shader, devo prima ottenere i puntatori alle variabili uniform presenti all'interno dello ShaderProgram (che, ricordiamo, è la combinazione di fragment e vertex shader compilati e linkati fra loro):

```
GLint location = glGetUniformLocation(ShaderProgram, "[nome_var]");
```

Successivamente le variabili uniformi vengono inizializzate con i loro valori usando i comandi, che possono essere, in base al tipo di dato, `glUniformMatrix4fv`, `glUniform3f` etc.. Queste funzioni specificano il valore da assegnare ad una variabile uniforme per il program corrente. Ecco un esempio dell'uso:

```
GLuint loc_s;
void init()
{
    prog = createProgram("v.gls", "f.gls");
    /* Otteniamo i puntatori alle variabili uniform per poterle utilizzare in seguito */
    loc_s = glGetUniformLocation(prog, "s");
}

void drawScene(void)
{
    float valore_s = 2.0;
    /* Communicate the variable value to the shader */
    glUniform1f(loc_s, valore_s);
    ...
}
```

glm

glm è una libreria matematica C++ per la programmazione grafica. Contiene numerosissime strutture e funzioni che ci vengono in nostro aiuto nella computer grafica.

Ogni volta che usiamo glm dobbiamo includere le librerie in modo corretto, usando questi header C:

```
#include <glm/glm.hpp> //Tipi di matrici e vettori  
// Matrici di trasformazione  
#include <glm/gtc/matrix_transform.hpp>  
#include <glm/gtx/transform.hpp>  
#include <glm/gtc/type_ptr.hpp>
```

In glm possiamo definire davvero tanti tipi di dato, che siano vettori o matrici, che non sto a scrivere perché volendo si possono cercare su internet in modo facile e veloce. Tra questi comunque basta sapere che si hanno un nome diverso per formato dei dati (int, float...) e per numero di componenti. In generale però hanno la stessa notazione di GLSL (fatta apposta per uniformare).

Per fare le operazioni di moltiplicazione, possiamo usare le funzioni builtin fornite da glm (ovvero dotProduct e crossProduct, e anche altre), oppure usare direttamente l'operando * per fare il prodotto scalare.

Come GLSL, anche glm fa uso della definizione di matrici per colonna (ovvero che bisogna descrivere ogni vettore colonna alla dichiarazione).

```
mat4 M3(vec4(1,2,3,4), vec4(1,2,3,4), vec4(1,2,3,4), vec4(1,2,3,4));
```

Come costruire le matrici di trasformazione in glm?

LOL, e noi chi ci siamo imparati mille cose su come farlo. glm permette di creare matrici di traslazioni in modo facile e veloce. Basta solo fornire una matrice predichiarata, e le componenti del vettore di traslazione (T_x , T_y , T_z):

```
M4 = translate(M4, vec3(Tx, Ty, Tz));
```

Cosa simile per la matrice di scalatura, dove (S_x , S_y , S_z) sono le tre dimensioni della scalatura.

```
M4 = scale(M4, vec3(Sx, Sy, Sz));
```

...e anche per la matrice di rotazione, dove (R_x , R_y , R_z) sono le componenti dell'asse:

```
M4 = rotate(M4, radians(theta), vec3(Rx, Ry, Rz));  
// radians fa la conversion da gradi a radianti.
```

L'oggetto deve essere trasformato prima che i dati vengano passati alla uniform. Solo dopo che sono stati passati, allora si potrà chiamare la funzione glDrawArrays(GL_TRIANGLES, 0, nvertices), che disegna l'oggetto.

Momento di vero godimento.

Trasformazioni in OpenGL

Come sappiamo, le trasformazioni ci permettono di passare da uno spazio ad un altro, e vengono usate per realizzare una scena 3D e la pipeline del rendering. **Usiamo trasformazioni rappresentate da matrici 4x4.**

Con trasformazioni di modellazione intendiamo le trasformazioni che vengono fatte per modellare un oggetto sul modello geometrico, generando la **Model Matrix**.

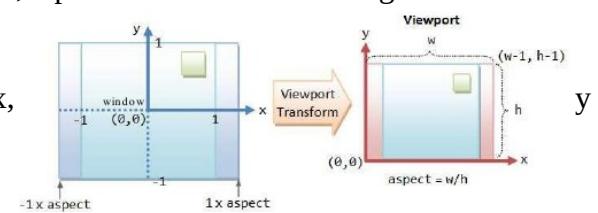
Con trasformazioni di proiezione intendiamo il processo della creazione della **Projection Matrix**. Viene creata in questo modo:

```
Projection = ortho(xmin, xmax, ymin, ymax);
```

Questa matrice di trasformazione trasforma tutte le coordinate dei vertici contenuti nel sistema di coordinate del mondo, in coordinate nel **sistema di riferimento normalizzato**.

Un ultima trasformazione molto importante è la **trasformazione di Viewport**. La **Viewport** è la regione rettangolare dello schermo dove l'immagine viene proiettata. Di default è la regione che coincide con la finestra aperta sullo schermo. Tuttavia, è possibile modificare la regione della finestra sullo schermo su cui andare a visualizzare il disegno con la funzione

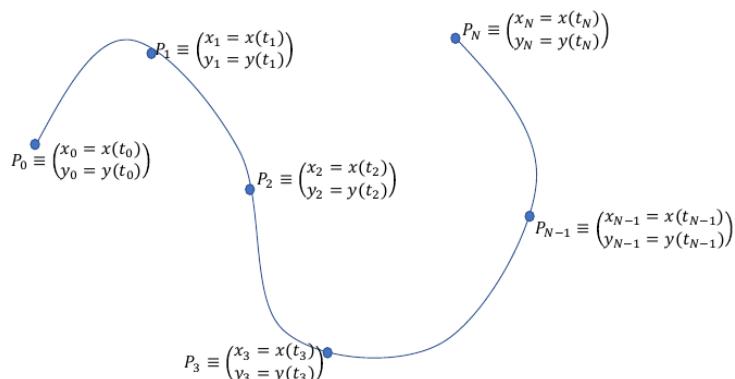
`glViewport(x, y, larghezza, altezza)`, dove x, specificano l'angolo inferiore sinistro del rettangolo della finestra, in pixel. Essenzialmente quindi possiamo definire il disegno su una piccola porzione della finestra.



Lezione 21/10/2021 – curve interpolanti di Hermite pt. 2

Continuazione sulle curve di Hermite

La curva che otteniamo sarà qualcosa simile a questo:



che passerà per i punti in cui vogliamo che la curva passi necessariamente.

Da $N+1$ punti da interpolare quindi vorremo costruire N segmenti di curve costruite **a partire da polinomi cubici**, tali che nei punti di giunzione abbiano lo stesso valore e la stessa derivata prima.

Affrontiamo questo problema nel caso più semplice di dati che descrivono una funzione, cioè date le coppie (x_i, y_i) , dove $y_i = f(x_i)$, $i=0 \dots N$, dovremo costruire N **segmenti di polinomi cubici** che nei punti di giunzione si *raccordino* in valore e derivata prima, e che quindi interpolano due punti successivi.

$$X_N(t_i) = x_i$$

Supponiamo che ogni punto P abbia delle coordinate (x_i, y_i) che supponiamo essere ottenute come la valutazione di un certo polinomio sconosciuto X_N (come direbbe Aspert, definito ma non calcolato) che valutato in un certo parametro t_i mi restituisca x_i .

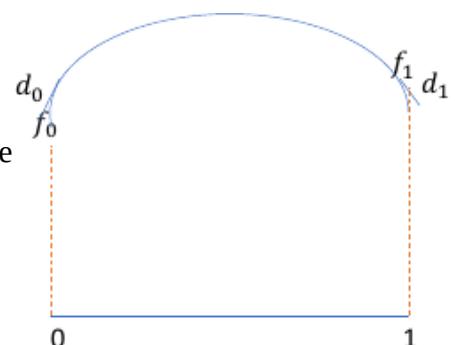
Questa stessa cosa la dobbiamo fare anche per la coordinata y_i , attraverso il polinomio Y_N .

Dunque, per trovare la curva, dovremo risolvere questi due problemi di interpolazione, imponendo i vincoli appena citati.

Lavoreremo prima su una componente e poi sull'altra quindi.

Come calcolare la curva di Hermite?

Consideriamo di assegnare all'intervallo $[0, 1]$ i valori f_0 e f_1 della funzione e i valori d_0 e d_1 della derivata prima (quindi l'andamento della funzione



agli espremi).

Il polinomio cubico $P_3(t) = a_0 + a_1t + a_2t^2 + a_3t^3$ che interpola questi dati, cioè tale che:

$$P_3(0) = f_0 \quad P_3(1) = f_1$$

$$P'_3(0) = d_0 \quad P'_3(1) = d_1$$

dovrebbe calcolarsi imponendo la seguente relazione di interpolazione (ovvero devo risolvere questo sistema linea re):

$$\begin{array}{l} P_3(0) \\ P_3(1) \\ \hline \end{array} \left\{ \begin{array}{l} a_0 = f_0 \\ a_0 + a_1 + a_2 + a_3 = 1 \\ a_1 = d_0 \\ a_1 + 2a_2 + 3a_3 = d_1 \end{array} \right. \begin{array}{l} P'_3(0) \\ P'_3(1) \\ \hline \end{array} \quad P'_3(t) = a_1 + 2a_2x + 3a_3x^2$$

Hermite dimostra che il polinomio $P_3(t) = a_0 + a_1t + a_2t^2 + a_3t^3$ che interpola questi dati e che rispetta i vincoli prima definiti si può esprimere come:

$$P_3(t) = f_0 \cdot \varphi_0(t) + d_0 \cdot \varphi_1(t) + f_1 \cdot \psi_0(t) + d_1 \cdot \psi_1(t)$$

dove:

$$\varphi_0(t) = 2 \cdot t^3 - 3 \cdot t^2 + 1$$

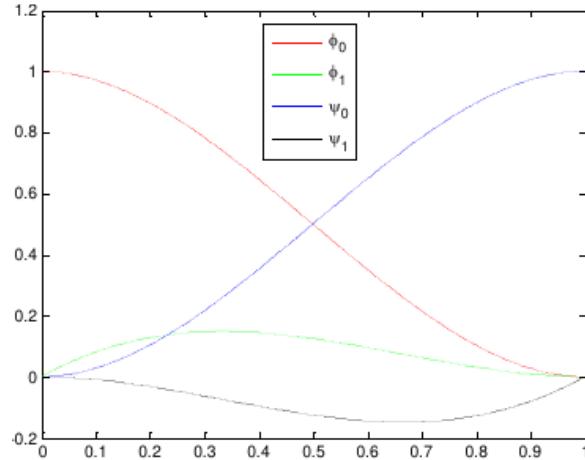
$$\varphi_1(t) = t^3 - 2 \cdot t^2 + t \quad 0 \leq t \leq 1$$

$$\psi_0(t) = -2 \cdot t^3 + 3 \cdot t^2$$

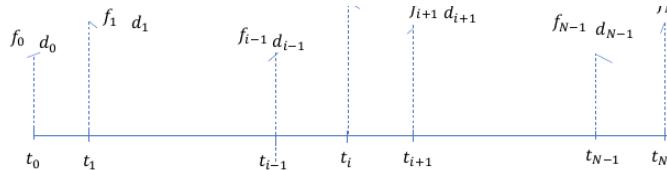
$$\psi_1(t) = t^3 - t^2$$

Possiamo essenzialmente usare queste equazioni quindi per definire la funzione della curva.

Qui a destra abbiamo la rappresentazione grafica di queste 4 formule. Notiamo come in φ_0 sia l'unica in cui $f_0 \neq 0$. Questo perché in $t=0$, le altre formule si annullano tranne φ_0 , come richiede il vincolo della nostra interpolazione!



Tuttavia, noi non dobbiamo considerare un singolo intervallo, bensì dobbiamo lavorare su N intervalli. Dunque, dobbiamo generalizzare questo concetto al caso in cui si abbiano 0 sotto intervalli.



Se però abbiamo i valori (t_i, f_i, d_i) con $i = 0, \dots, N$ (ovvero un valore del punto e uno della derivata per ogni estremo del sottointervallo), il polinomio interpolatore di Hermite $P_H(t)$ tale che $P_H(t_i) = f_i$ e $P'_H(t_i) = d_i$, con $i = 0, \dots, N$ si esprime come:

$$P_H(t) = \sum_{i=0}^N \Phi_i(t)$$

dove:

$$\Phi_i(t) = f_i \cdot \varphi_{0,i}(t) + d_i \cdot \varphi_{1,i}(t) + f_{i+1} \cdot \psi_{0,i}(t) + d_{i+1} \cdot \psi_{1,i}(t)$$

Essenzialmente quello che stiamo facendo è calcolare ogni polinomio per ogni intervallino nel

quale la curva viene divisa, sommando poi il risultato di ogni intervallo. Ovviamente ad ogni intervallo non sarà necessario ric算are l'intero polinomio P_H , **bensì basterà calcolare Φ_i per ogni intervallo** (infatti gli elementi di un intervallo non andranno ad influenzare la parte della curva di un altro intervallo, siccome vengono azzerati). Questa è una caratteristica molto importante delle curve di Hermite). Dunque, per calcolare la curva, ci basterà capire in che intervallo si trova t e calcolare il polinomio ad esso associato (**concetto di modellazione locale**).

Siccome però le 2 funzioni $\phi_{1,i}(t)$, $\psi_{1,i}(t)$ sono definite nell'intervallo $[0, 1]$ (e sono cambiano in base a questo intervallo, le altre due invece sono invarianti per trasformazioni affini), **sarà necessario applicare una trasformazione affine** in modo che il punto $t \in [t_i, t_{i+1}]$ venga mappato in $\hat{t} \in [0, 1]$.

Consideriamo quindi la trasformazione affine che a $t \in [t_i, t_{i+1}]$ faccia corrispondere

$$\hat{t} \in [0, 1]:$$

$$\hat{t} = \frac{t - t_i}{t_{i+1} - t_i}$$

Allora vale il seguente risultato:

$$\phi_{1,i}(t) = \phi_{1,i}(\hat{t})(t_{i+1} - t_i), \quad \psi_{1,i}(t) = \psi_{1,i}(\hat{t})(t_{i+1} - t_i)$$

Siccome i valori delle derivate, al contrario dei valori della funzione, non ci vengono dati subito, è necessario trovare un criterio per stimare i valori delle derivate.

Ne esistono tantissimi (molti di questi sono stati insegnati al corso di Metodi Numerici fatto dalla prof, ma che invece noi di Bologna non abbiamo fatto lololol), quello che useremo di più sarà il metodo del **rapporto incrementale**

$$d_i = \frac{p_{i+1} - p_i}{t_{i+1} - t_i} \quad i = 0, \dots, N-1$$

Lezione 25/10/2021 – Hermite pt. 3, curve Bezier, pipeline Geometrica

Continuazione curve di Hermite

La nostra curva finale $C(t)$ sarà così composta da questa equazione:

$$C(t) = \begin{cases} C_x(t) = \sum_{i=0}^{N-1} \phi_i^x(t) \\ C_y(t) = \sum_{i=0}^{N-1} \phi_i^y(t) \end{cases}$$

Come si può notare, dobbiamo applicare il calcolo della curva di Hermite sia per la componente x che per la componente y della curva.

Possiamo facilmente dimostrare che la curva che creiamo è di classe C^1 , siccome la funzione ottenuta, per come l'abbiamo costruita, nei punti di giunzione fra i vari intervalli coincide per valore e per derivata prima.

Altri metodi per il calcolo delle derivate

Abbiamo il metodo delle **differenze finite**, che permette di calcolare la derivata prima come la media di due rapporti incrementali consecutivi.

Abbiamo poi il **cardinal spline**, che consiste in un singolo rapporto incrementale dimezzato, e

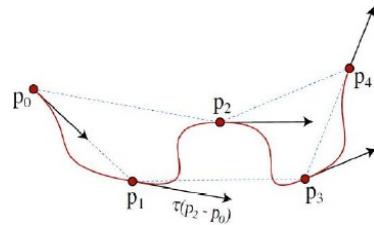
$$d_i = \frac{1}{2}(d_i + d_{i-1}) = \frac{p_{i+1} - p_i}{2(t_{i+1} - t_i)} + \frac{p_i - p_{i-1}}{2(t_i - t_{i-1})}$$

moltiplicato per un parametro di tensione c, che agisce sulla lunghezza della tangente.

$$d_i = (1 - c) \frac{p_{i+1} - p_{i-1}}{2(t_{i+1} - t_{i-1})}$$

Se poniamo il valore del parametro di tensione a 0, otteniamo un altro metodo ancora, ovvero il **Catmull Rom**, che essenzialmente calcola una tangente nel punto p_i in modo che sia parallela al segmento che congiunge il punto precedente con quello successivo.

$$d_i = \frac{p_{i+1} - p_{i-1}}{2(t_{i+1} - t_{i-1})}$$



Infine, abbiamo l'ultimo metodo, ovvero la **Spline di Kochanek-Bartles** (o **TBC Splines**), in cui sono definiti 3 parametri, ovvero la Tensione, il Bias e la Continuity.

Dati $n+1$ punti da interpolare mediante n segmenti di curva cubica di Hermite, per ogni curva abbiamo un punto iniziale p_i ed un punto finale p_{i+1} con tangenti d_i e d_{i+1} , le formule della derivata corrispondono a:

$$d_i = (1 - T)(1 + B)(1 + C) \frac{p_i - p_{i-1}}{2(t_i - t_{i-1})} + (1 - T)(1 - B)(1 - C) \frac{p_{i+1} - p_i}{2(t_{i+1} - t_i)}$$

$$d_{i+1} = (1 - T)(1 + B)(1 - C) \frac{p_i - p_{i-1}}{2(t_i - t_{i-1})} + (1 - T)(1 - B)(1 - C) \frac{p_{i+1} - p_i}{2(t_{i+1} - t_i)}$$

Dove il parametro tension (T) varia la lunghezza del vettore tangente (ovvero quanto è schiacciata la curva), il parametro bias (B) cambia la direzione del vettore di tangente e il valore continuity (C), varia la continuità.

In pratica corrisponde il metodo che abbiamo implementato nella esercitazione del giorno prima, anche se avevamo posto i parametrici TBC a 0, rendendolo quindi pari al metodo delle differenze finite.

Differenza fra spline e le altre interpolazioni/curve.

Le spline sono delle funzioni polinomiali a tratti, definiti su un certo numero di sottointervalli, e nel punto di contatto tra due intervalli successivi hanno una certa regolarità. La loro caratteristica è che sono definite da polinomi di grado 3, al contrario delle altre curve.

Come abbiamo visto nel corso di Calcolo Numerico, le curve di interpolazioni che costruivamo potevano avere un grado enorme, anzi spesso più c'erano punti, più aumentava il grado. Tuttavia, spesso questo comportava un aumento dell'oscillazione della curva. **Le spline, grazie alle loro caratteristiche costruttive, mitigano l'effetto oscillatorio di queste curve** (che per esempio c'era nella interpolazione di Rouge per esempio).

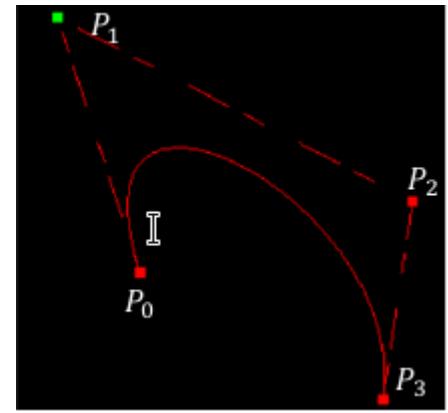
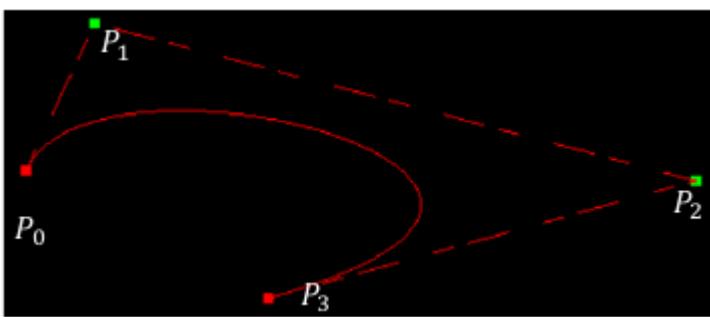
Le curve di Hermite si possono definire delle curve spline di grado C^1 .

Curve di Bezier

La caratteristica delle **curve di Bezier** è che non sono delle curve interpolanti: questo vuol dire che al contrario delle curve di Hermite, le curve di Bezier non passano necessariamente per dei punti che abbiamo definito noi. Piuttosto, approssimano la forma del poligono che si ottiene collegando i punti sul piano.

Abbiamo visto che nelle curve interpolanti di Hermite, abbiamo a che fare con segmenti di curva che sono dati dalla combinazione lineare di quattro valori P_1, P_2, D_1, D_2 mediante quattro funzioni base. P_1, P_2 due assegnano il valore della curva, D_1, D_2 ne modellano la forma. L'idea che è alla base delle curve di Bezier è di sostituire i valori D_1 e D_2 (che nella curva di Hermite vengono interpolati) con altri due valori puntuali, che non vengono interpolati ma solo approssimati e la cui posizione influenza la forma della curva.

I dati di un segmento di curva cubica di Bezier sono quindi quattro valori puntuali che chiameremo P_0, P_1, P_2, P_3 , di cui P_0 e P_3 vengono interpolati, mentre gli altri due vengono solo approssimati. **La loro posizione determina il vettore tangente all'inizio ed alla fine del segmento di curva.**



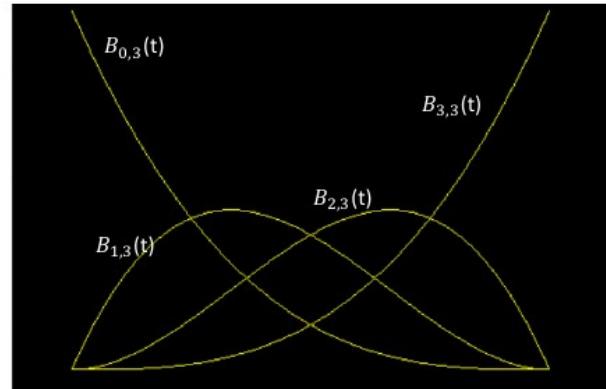
Come per le curve di Hermite, abbiamo delle funzioni base che permettono di ottenere il segmento di curva cubica di Bèzier. Queste sono i **polinomi di Bernstein** (di grado 3), e sono:

$$B_{0,3}(t) = (1-t)^3$$

$$B_{1,3}(t) = 3t(1-t)^2$$

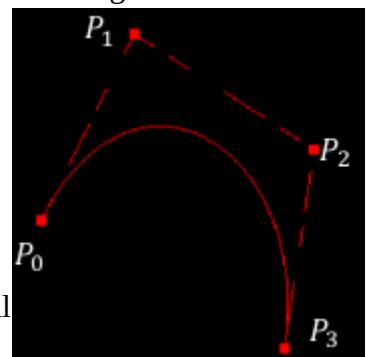
$$B_{2,3}(t) = 3t^2(1-t)$$

$$B_{3,3}(t) = t^3$$



Il segmento di curva cubica di Bèzier che interpola P_0 e P_3 e che in P_0 e P_3 è tangente ai segmenti P_1-P_0 e P_3-P_2 rispettivamente è dato da:

$$C(t) = \sum_{i=0}^3 P_i B_{i,3}(t) \quad t \in [0,1]$$

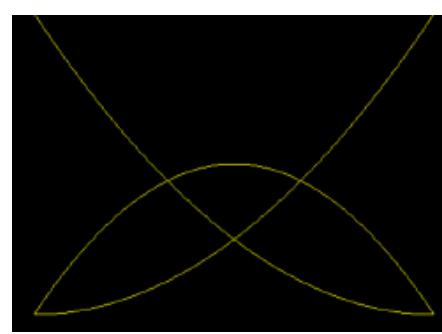
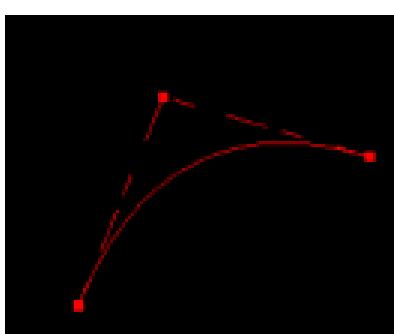


I punti P_i sono detti **punti di controllo della curva** (o maniglie) ed individuano il poligono di controllo della curva. [N.B. ovviamente nella formula bisogna mettere un n al posto del 3 per renderla generale per n punti].

Nel caso più generale di curve di Bezier di grado n abbiamo a che fare con $n+1$ punti, di cui il primo e l'ultimo sono interpolati, il secondo e il penultimo danno la direzione della derivata negli estremi, mentre gli altri punti vengono “approssimati” e servono a modellare la forma della curva.
Dunque, se abbiamo curve di Bezier di grado n, allora le funzioni di base di Bernstein saranno di grado n, e saranno definiti nell’intervallo $[0, 1]$ dalla formula:

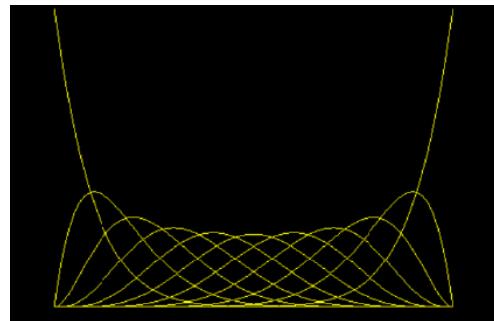
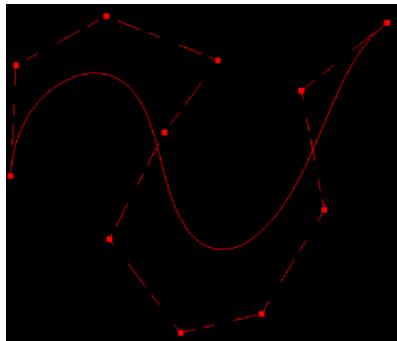
Coeffiente binomiale $\rightarrow B_{i,n}(t) = \binom{n}{i} (1-t)^{n-i} t^i \quad i = 0, \dots, n$

Vediamo un esempio della curva di Bernstein di grado 2:



Sulla sx abbiamo la curva vera e propria, a dx invece abbiamo il grafico delle curve che compongono la curva.

Esempio di grado 10:



Siccome le funzione base hanno un **supporto globale**, ovvero sono diverse da 0 su tutto l'intervallo di studio, tutte le funzione di base andranno ad influenzare un valore C(t). Questo è contrariamente a quanto succede nelle curve di Hermite, in cui invece ogni polinomio Φ_i andava ad influenzare solamente l'intervallo i in cui esso era definito. **Non abbiamo dunque il concetto di modellazione/controllo locale**, e valutare la curva ad ogni passo sarà dunque più laborioso. Inoltre, questo vuol dire che se per esempio voglio aggiungere della geometria ad un modello (es. una sfera a cui voglio aggiugere un naso modellandolo) influerò l'intero modello, e non avrò il controllo locale della curva. Per questo motivo non sono idonee alla modellazione, ma sono comunque carine. Inoltre spesso si può ovviare a questo problema (per esempio, in Blender si possono usare delle curve di Bezier ad intervalli).

Le curve di Bezier presentano moltissime proprietà interessanti, che derivano dai polinomi di base di Bernstein.

Finora noi abbiamo visto **il polinomio di Bernstein** definito unicamente sugli interavalli $[0, 1]$, ma possiamo comunque anche associare questi polinomi ad un intervallo generico $[a, b]$:

$$P_n(x) = \sum_{i=0}^n c_i B_{i,n}(x) \quad x \in [a, b]$$

dove i $B_{i,n}(x)$ [con $i=0, \dots, n$] sono **le funzioni di base di Bernstein di grado n** definiti su $[a, b]$ dalla relazione:

$$B_{i,n}(x) = \binom{n}{i} \frac{(b-x)^{n-i} (x-a)^i}{(b-a)^n} \quad i = 0, \dots, n$$

e c_0, c_1, \dots, c_n sono i coefficienti che identificano il polinomio espresso nella base di Bernstein.

[N.B. non so perché la prof ha detto queste cose subito sotto o a che servano, riporto per completezza]

Confrontiamo questo con la funzione di base di Bernstein con la rappresentazione mediante di un polinomio mediante la base monomiale (ovvero la classica somma di monomi per costruire un polinomio $1, x, x^2, \dots, x^n$):

$$P_n(x) = \sum_{i=0}^n a_i x^i$$

La prima cosa che notiamo è che mentre le basi monomiale sono tutte di grado diverso crescente, le basi di Bernstein sono tutte le di base n.

[fine N.B.]

Nelle applicazioni si suole usare l'intervallo $[0, 1]$, quindi diciamo che la visione di questo metodo del calcolo nell'intervallo $[a, b]$ ha un valore più teorico che altro.

Proprietà dei polinomi di Bernstein di grado n

1) Un polinomio di Bernstein di grado n è definito in $[0, 1]$ è legato al polinomio di Bernstein di grado n-1 definito anch'esso su $[0, 1]$, dalla seguente formula:

$$B_{i,n}(t) = t \cdot B_{i-1,n-1}(t) + (1-t) \cdot B_{i,n-1}(t) \quad t \in [0,1]$$

Ovvero, ciascun polinomio di grado n è la **combinazione convessa** di polinomi di grado $n-1$ (la combinazione convessa consiste nella combinazione di quantità positive mediante coefficienti positivi, la cui somma vale 1, come avevamo descritto anche [qui](#)).

La parte interessante è anche il fatto che $B_{i,n}(x) = 0 \forall i \notin [0, n]$, ovvero che la curva è uguale a 0 per ogni i che non appartiene all'intervallo $[0, n]$ dove n è il grado del polinomio di Bernstein.

2) Possiamo anche notare come il codominio di ogni curva **sia sempre ≥ 0** , e ciò è interessante dal punto di vista della stabilità numerica, siccome è convegno sempre somme di quantità positive (abbiamo la definizione di stabilità numerica a calcolo numerico, dove abbiamo visto che le somme fra numeri con segno opposto sono invece instabili).

3) I polinomi di Bernstein sono una **partizione dell'unità**. Questo indica che, fissato un valore di t , la somma di tutte le funzioni base su quel punto è pari a 1.

$$\sum_{i=0}^n B_{i,n}(t) = 1$$

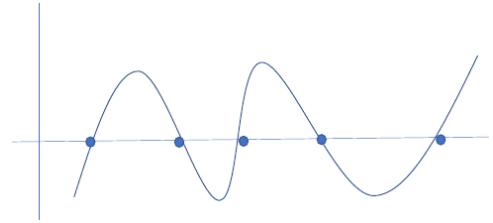
4) Il polinomio $p(x) = \sum_{i=0}^n c_i B_{i,n}(x)$ è una combinazione lineare convessa dei valori dei suoi coefficienti (siccome la loro somma è 1, come abbiamo potuto vedere dalla *prima e terza proprietà*), e quindi:

$$\min_{i=0,\dots,n} \{c_i\} \leq p(x) \leq \max_{i=0,\dots,n} \{c_i\}$$

Questa proprietà ci permette di dare una prima rappresentazione intuitiva del piano del nostro disegno, siccome ci dice che un p. d. B. è compreso fra il suo coefficiente più piccolo e quello più grande. Quindi, se io so quali sono i suoi coefficienti, **so già approssimativamente quale sarà la sua posizione nel piano**.

-

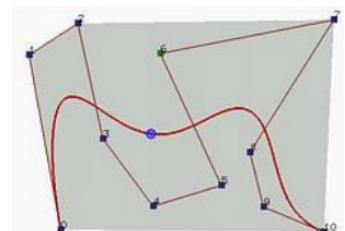
5) Un'altra proprietà graficamente molto figa è il fatto che **il numero di variazioni di segno di un polinomio espresso nella base di Bernstein è minore o uguale al numero di variazioni di segno dei suoi coefficienti (variation diminishing)**. Questo ci consente di avere per il polinomio un limite superiore per il numero dei suoi zeri semplici in $[a,b]$.



Proprietà dell'inviluppo convesso

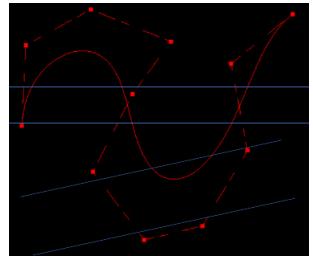
Assegnati i punti c_0, c_1, \dots, c_n , si definisce **inviluppo convesso** dell'insieme dei punti $\{c_i\}$ la più piccola regione convessa che contiene i punti dati.

Il grafico del polinomio espresso nella base di Bernstein è contenuto nell'inviluppo convesso dell'insieme dei vertici del suo poligono di controllo.



Proprietà dell'approssimazione di forma

Il numero di intersezioni di una qualsiasi retta con il poligono di controllo è maggiore o uguale del numero di intersezioni della stessa retta con il polinomio.



Pipeline di rendering – fase geometrica

Definita una camera virtuale (o telecamera virtuale) e una scena tridimensionale, la pipeline di

rendering costruisce una serie di trasformazioni che proiettano la scena tridimensionale in un'immagine in una finestra contenuta nello schermo bidimensionale.

In questo stadio, un modello geometrico è infatti trasformato mediante trasformazioni di modellazione, vista, proiezione e viewport, ovvero trasformazioni tra vari sistemi di riferimento. Inizialmente il modello geometrico viene creato nello **spazio locale del modello**.

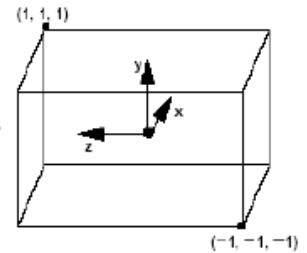
Viene quindi posizionato ed orientato in scena nel **World Coordinate System (WCS)** mediante **trasformazioni di modellazione** (affini) sui vertici del modello.

Il WCS è unico e dopo che tutti i modelli geometrici necessari per creare una scena sono stati posizionati tutti i modelli sono rappresentati in questo spazio.

Poichè solo i modelli 'visti' dalla camera virtuale sono resi, tutti i modelli sono quindi trasformati mediante una **trasformazione di vista** nel sistema di riferimento della camera o camera frame (**View Coordinate System – VCS**).

Dal centro del sistema della camera solo una porzione di volume, detto **volume di vista**, contiene la scena che è visibile all'osservatore.

Quindi il sistema di rendering elabora la trasformazione di proiezione per trasformare il volume di vista contenente la scena visibile dall'osservatore, in un cubo di lato 2 e diagonale di estremi $(-1, -1, -1)$ e $(1, 1, 1)$. Dunque, queste proiezioni trasformano un volume (volume di vista) in un altro volume (cubo), in coordinate normalizzate NDC che vanno da -1 a 1.



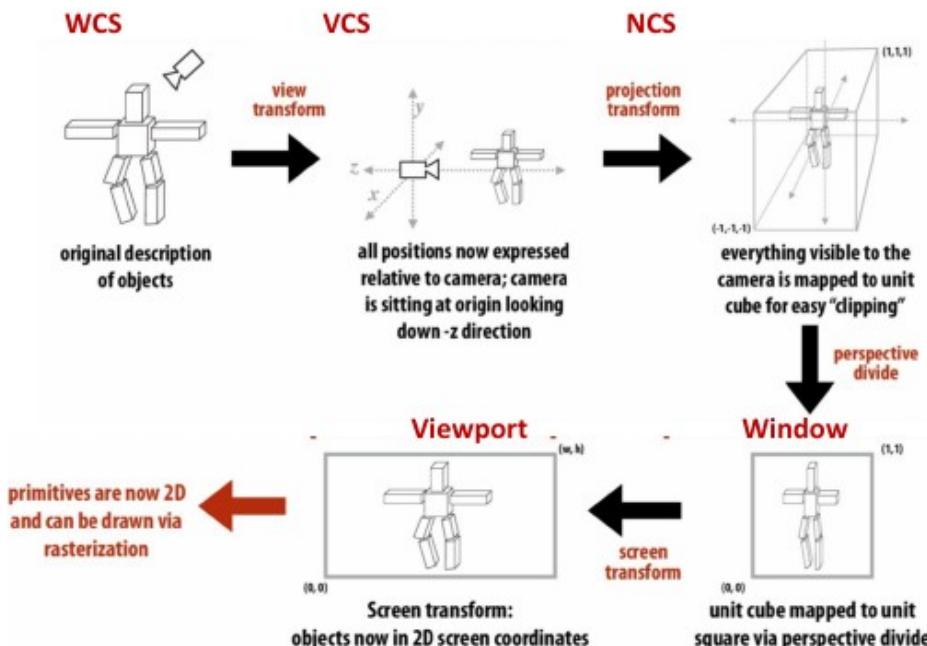
Per poi passare dalla rappresentazione ad un immagine 2D (view plane, o viewport), si fa la proiezione della scena dentro il cubo ad una faccia del cubo, attraverso una proiezione ortogonale, nella quale la x e la y si conservano, e la z (che rappresenta la profondità dei vertici) viene memorizzata nel **Z-buffer**, che è una speciale memoria del framebuffer.

Perché è necessario il passaggio dal volume di vista al cubo normalizzato? Perché facilita molto le operazioni di clipping, ovvero le primitive che risiedono nel volume di vista verranno disegnate su schermo, le altre invece verranno scartate o parzialmente mostrate.

Infatti, solo le primitive interne al volume di vista sono passate all'ultima trasformazione del geometry stage, la **trasformazione di viewport** (o window-viewport o screen mapping).

Quest'ultima converte le coordinate (x, y) reali di ogni vertice, in coordinate schermo espresse in pixel (device coordinate system). Quest'ultime, insieme con la coordinata $-1 \leq z \leq 1$ forniranno l'input per lo stadio successivo della pipeline, la fase di **rasterizzazione**.

Ecco una rappresentazione schematica di ciò che abbiamo appena detto:



Vediamo ora le fasi un po' più in dettaglio.

1 - Trasformazioni di Modellazione (da OCS a WCS)

Sappiamo che ogni oggetto è definito in un suo sistema di coordinate, detto “**Object Coordinate System**”. Le trasformazioni di modellazione permettono di muovere, orientare, e trasformare modelli geometrici all'interno di un sistema di riferimento comune, **sistema di coordinate del Mondo (WCS)**. Ciò viene eseguito moltiplicando le coordinate di ciascun vertice per una **matrice di trasformazione affine 4x4** (T_M).

2 - Trasformazioni di Vista (da WCS a VCS)

Questa fase dello studio geometrico prende in input i vertici in 3D definiti in WCS, e da in output le coordinate in **VCS (View Coordinate System)**, ovvero un sistema di coordinate della telecamera sintetica della nostra scena. La matrice di vista (View Matrix) sarà quindi utilizzata per trasformare ogni vertice degli oggetti in scena dal WCS al VCS.

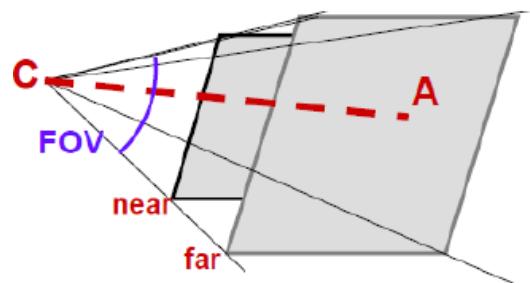
Per fare in modo che l'oggetto venga posizionato in coordinate di mondo, quindi, dovremo fare una serie di passi:

1. **Specificare la vista 3D:** dunque, dovremo impostare (ovvero posizionare) la telecamera sintetica (ovvero la camera/videocamera di cui parlavamo prima), che sarà orientata e posizionata in coordinate di mondo (WCS).
2. Dopotiché, dovremo costruire la trasformazione di vista a partire dal posizionamento e orientamento della fotocamera. Ciò vuol dire che dovremo creare una **matrice di trasformazione** T_M , che permette il cambio di sistema di riferimento [questo passo viene già svolto da OpenGL con una relativa funzione].
3. Applichiamo questa trasformazione ad ogni vertice dell'oggetto.

Definire la vista – camera

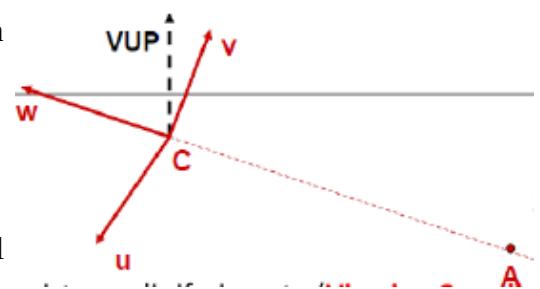
Abbiamo bisogno di sapere quattro cose sul nostro modello di fotocamera sintetica per costruire la trasformazione della vista

- **Punto C:** posizione della telecamera in WCS (da dove si sta guardando).
- **Punto A:** il punto A permette il calcolo del vettore C-A, che specifica in quale direzione sta puntando la telecamera (dove A è il centro della scena).
- **field of view o FOV (grandangolo, normale...)**
- posizionamento dei piani di clipping (che sono due piani che delimitano il frustum, e solo ciò che sta fra essi viene renderizzato).



Abbiamo poi altri 3 assi/vettori molti importanti, calcolati a partire dai parametri descritti prima:

- **L'asse w o direzione di vista:** che stabilisce la direzione la direzione unitaria verso cui punta la fotocamera. Per convenzione, la telecamera guarda in direzione -w, dove w è calcolato come: $w = \frac{C - A}{\|C - A\|}$. [Notare come quindi sia un vettore normalizzato, un versore].
- **View Up Vector (VUP):** determina il modo in cui la fotocamera viene ruotata attorno alla direzione di vista.
- **Asse u:** è un asse unitario che punta alla destra dell'osservatore, ed è perpendicolare sia all'asse w che al vettore up VUP, e dunque si calcola con formula $u = \frac{VUP \times w}{\|VUP \times w\|}$.



Questo modo che abbiamo presentato per specificare la vista prende il nome di **camera “look at”**. Ci sono anche altri modi per specificare il tipo di vista: come simulazione di volo, telecamera rotante etc.. ma per ora non ci interessano.



Costruzione della matrice di Trasformazione di Vista

Ora che sappiamo come impostare la telecamera, abbiamo bisogno di una matrice di trasformazione di vista.

Questa, dati i frame VCS e WCS, calcolano la matrice di trasformazione di Vista T_v , che converte ogni punto P_w da coordinate WCS a coordinate VCS, con la formula $P_v = T_v P_w$.

Per ottenere la matrice, bisogna prima di tutto esprimere **il sistema di riferimento della camera in VCS (C, u, v, w) in termini del sistema WCS (O, x, y, z)**:

$$\begin{aligned} u &= u_x x + u_y y + u_z z + 0 \cdot O \\ v &= v_x x + v_y y + v_z z + 0 \cdot O \\ w &= w_x x + w_y y + w_z z + 0 \cdot O \\ C &= C_x x + C_y y + C_z z + 1 \cdot O \end{aligned}$$

Da cui otteniamo la matrice M

$$M = \begin{bmatrix} u_x & v_x & w_x & C_x \\ u_y & v_y & w_y & C_y \\ u_z & v_z & w_z & C_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La matrice M mappa le coordinate di un punto nel sistema VCS nelle coordinate del sistema WCS. Siccome noi però vogliamo eseguire l'operazione opposta, dobbiamo invertire la matrice, ottenendo così:

$$P_v = M^{-1} P_w = T_v P_w$$

In questo modo, otteniamo le coordinate del punto da VCS a WCS.

Per capire meglio, facciamo un passo indietro e ricapitoliamo le cose che abbiamo detto fin'ora: [così tante matrici, così poco spazio nel nostro cervellino...].

Supponiamo di voler rendere una scena con un oggetto da un certo punto di vista (camera).

Dovremo fare due trasformazioni:

- L'oggetto viene posizionato in **coordinate mondo (WCS)** con matrice T_M (ovvero la matrice di modellazione), la camera è posizionata in coordinate mondo con matrice T_v (ovvero la matrice di vista).
- La seguente trasformazione prende ogni vertice dell'oggetto dal sistema di coord. locali (che chiamiamo P), e le trasforma in coord. mondo, e poi a coordinate mondo WCS a coord. della camera VCS:

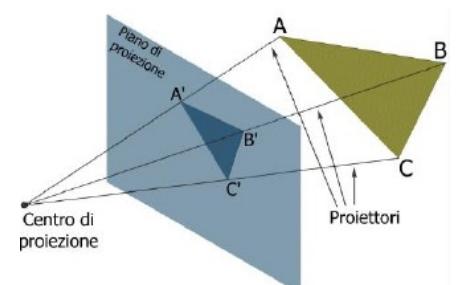
$$P_v = T_v T_M P$$

Tuttavia, per visualizzare effettivamente gli oggetti, i punti dovranno poi essere proiettati nello spazio 2D. Questo è il compito della trasformazione di proiezione.

3 - Projection Transform (da VCS a coordinate schermo)

Questa fase dello stadio geometrico prende in input i vertici in 3D definiti in VCS, e da in output le coordinate.

Proiezioni geometriche – alcune definizioni

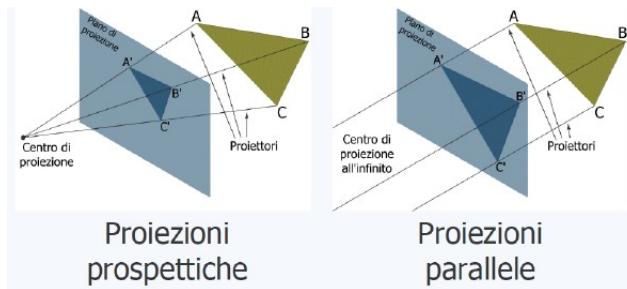


Si dice **proiezione** una trasformazione geometrica con il dominio in uno spazio di dimensione n ed il codominio in uno spazio di dimensione n-1 (o minore). A noi interessano solo le proiezioni da 3 a 2 dimensioni. Per definire la proiezione di un oggetto 3D, abbiamo bisogno di un insieme di **rette di proiezione** (dette *proiettori*) aventi origine comune da un centro di proiezione (che in genere coincide con la posizione della telecamera), che passano per tutti i punti dell'oggetto e intersecano un piano di proiezione per formare la proiezione vera e propria.

Questo tipo di proiezioni si chiamano **proiezioni geometriche piane**, e sono caratterizzate dal fatto che:

- I proiettori sono rette (potrebbero essere curve generiche)
- La proiezione è su di un piano (potrebbe essere su una superficie generica)

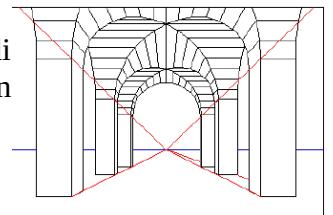
Le proiezioni geometriche piane sono suddivisibili in due classi base: **prospettive** e **parallele**. La differenza fra le due classi è data dalla distanza tra il centro di proiezione ed il piano di proiezione: se tale distanza è finita, allora la proiezione è prospettica, altrimenti se è infinita è parallela. Nel caso di proiezioni parallele si parla di una *direzione di proiezione*.



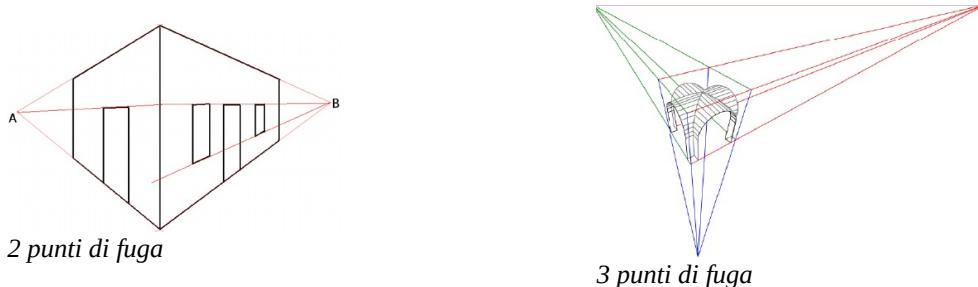
Normalmente la proiezione prospettica è la più realistica, in quanto riesce a riprodurre il modo con cui nella realtà vediamo gli oggetti: oggetti più grandi sono più vicini all'osservatore, oggetti più piccoli sono più lontani. Inoltre le distanze uguali lungo una linea non vengono proiettate su distanze uguali sul piano dell'immagine (la proiezione prospettica non è una trasformazione affine). Gli angoli sono conservati solo su piani paralleli al piano di proiezione.

Proiezioni prospettive

Le proiezioni prospettive sono caratterizzate dai **vanishing point**: la proiezione di ogni insieme di linee parallele (non parallele al piano di proiezione) converge in un punto detto vanishing point (punto di convergenza). Il numero di questi punti è infinito, come il numero delle possibili direzioni di fasci di rette parallele.



Se l'insieme di linee parallele è a sua volta parallelo ad uno degli assi coordinati il punto di convergenza si chiama **axis vanishing point**. Di questi punti ce ne possono essere al più tre.

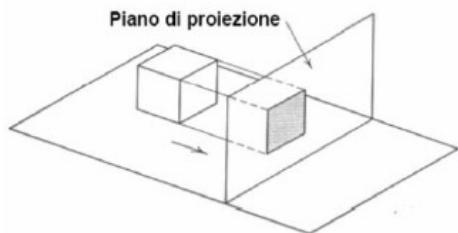


Proiezioni parallele

Le proiezioni parallele si classificano in base alla relazione che c'è tra la direzione di proiezione e la

normale al piano di proiezione.

Si parla di **proiezione ortografica** se la direzione di proiezione coincide con la normale al piano di proiezione. Altrimenti, si parla di **proiezione obliqua**.



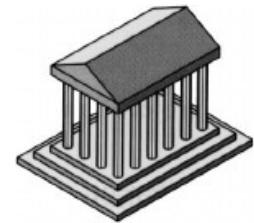
Proiezione ortografica



Proiezione obliqua

Abbiamo poi le proiezioni ortografiche assonometriche, che sono ortografiche, ma siccome lla direzione di proiezione non è allineata con un asse principale, vengono mostrano facce diverse di un oggetto, assomigliando in questo alle proiezioni prospettive.

Un tipo speciale di proiezione assonometrica è la *proiezione isometrica*, dove la direzione di proiezione è identificata da una delle bisettrici degli ottanti dello spazio cartesiano.



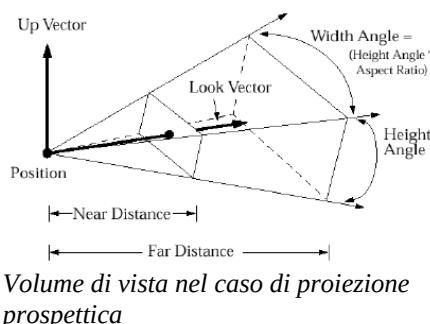
Esempio di proiezione isometrica

La proiezione obliqua è il tipo più generale di proiezioni parallele, e è caratterizzata dal fatto che i proiettori possono formare un angolo qualsiasi con il piano di proiezione (un esempio è la cavaliera).

Come proiettare da 3D a 2D?

Prima di poter proiettare a un'immagine 2D, dobbiamo definire il **volume di vista**. Questo consiste nel volume di spazio fra il **piano di clipping anteriore e posteriore**, che definisce lo spazio che la telecamera può "vedere". Il tipo di proiezione definisce la forma del volume di vista.

Infine, sarà necessario proiettare il volume di vista nel sistema di coordinate normalizzate o di clipping (**Normalizzazione**), ovvero il cuboide con il centro nell'origine definito in $[-1,1] \times [-1,1] \times [-1,1]$; questo siccome il clipping rispetto ad un cubo con assi paralleli agli assi coordinate è più semplice da realizzare.



Volume di vista nel caso di proiezione prospettica

Normalizzazione

La normalizzazione consiste nel proiettare il volume di vista nella sistema di coordinate normalizzato, e quindi trasformare il volume di vista in un volume di vista parallelo (cubo) (detto **Spazio immagine**).

La normalizzazione viene svolta siccome consente una singola pipeline sia per la visualizzazione prospettica che per la visualizzazione ortografiche.

Inoltre, rimaniamo in quattro coordinate omogenee dimensionali il più a lungo possibile per conservare le informazioni tridimensionali necessarie per la rimozione delle superfici nascoste.

Infine, come avevamo accennato, si semplifichiamo sia il clipping che la proiezione.

Lezione 28/10/2021 – Laboratorio: editor spline

In questa lezione di laboratorio, abbiamo creato un editor di curve di Hermite.

Lezione 4/11/2021 – Laboratorio: funghetto e testo su schermo, EBO

Lezione 8/11/2021 – Bezier-De Casteljau e continuazione su 3D

Algoritmo di De Casteljau

Il metodo più banale per calcolare una curva di Bezier sarebbe l'uso della formula che abbiamo già visto anche [qui](#). Infatti, abbiamo tutte le componenti, siccome i vertici le segniamo tutti noi "a mano", le funzioni base di Bernstein le conosciamo siccome conosciamo la loro formula [sappiamo che la funzione base relativa all'indice i di grado n è questa formula [qui](#), sostituendo $[a,b]$ con $[0,1]$].

Usare questo metodo però non sarebbe affatto efficiente, quindi De Casteljau ha proposto un algoritmo fatto apposta per questo, che permette di calcolare ciascuna delle due componenti x,y della curva senza calcolare la funzione base, ma elaborando i coefficienti della combinazione lineare.

Per arrivare a ciò, dobbiamo fare una serie di passaggi:

$$P(t) = \sum_{i=0}^n c_i [t B_{i-1,n-1}(t) + (1-t) B_{i,n-1}(t)] = \sum_{i=0}^n t c_i B_{i-1,n-1}(t) + \sum_{i=0}^n c_i (1-t) B_{i,n-1}(t) =$$

Per la proprietà 1, vista [qui](#).

$$\text{poichè } B_{-1,n-1}(t) = B_{n,n-1}(t) = 0 \\ = \sum_{i=1}^n t c_i (t) B_{i-1,n-1}(t) + \sum_{i=0}^{n-1} c_i (1-t) B_{i,n-1}(t) = \\ = \sum_{i=0}^{n-1} t c_{i+1} (t) B_{i,n-1}(t) + \sum_{i=0}^{n-1} c_i (1-t) B_{i,n-1}(t) =$$

Uso un piccolo trucco, per fare un calcolo in meno. Sostituisco n con $n-1$, ma poi per far tornare i conti nella sommatoria devo portare la sommatoria a $[0, n-1]$.

Ho raccolto le funzioni base.

Pongo poi $c_i^{[1]} = t \cdot c_{i+1}^{[0]} + (1-t) \cdot c_i^{[0]}$, con $i=0, \dots, n-1$ si ha: $P(t) = \sum_{i=0}^{n-1} c_i^{[1]} B_{i,n-1}(t)$

A questo punto, continuiamo a sostituire a $P(t)$, e otteniamo:

$$P(t) = \sum_{i=0}^{n-1} c_i^{[1]} B_{i,n-1}(t) = \sum_{i=0}^{n-1} c_i^{[1]} (t B_{i-1,n-2}(t) + (1-t) B_{i,n-2}(t)) = \\ = t \sum_{i=0}^{n-1} c_i^{[1]} B_{i-1,n-2}(t) + (1-t) \sum_{i=0}^{n-1} c_i^{[1]} B_{i,n-2}(t) =$$

Sostituiamo con la formula della funzione base di Bernstein.

A questo punto sappiamo nuovamente che ci sono funzioni di base di Bernstein = 0 per $i = -1$ e $i=n$ (quindi sempre per la proprietà 1).

$$= t \sum_{i=0}^{n-2} c_{i+1}^{[1]} B_{i,n-2}(t) + (1-t) \sum_{i=0}^{n-2} c_i^{[1]} B_{i,n-2}(t) =$$

A sto punto quindi posso applicare la stessa situazione di prima, con $c_i^{[2]} = t \cdot c_{i+1}^{[1]} + (1-t) \cdot c_i^{[1]}$, e otterremo così nuovamente una formula del tipo: $P(t) = \sum_{i=0}^{n-2} c_i^{[2]} B_{i,n-2}(t)$

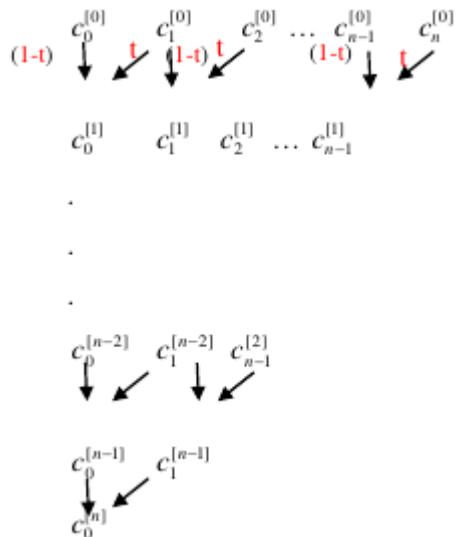
Continuiamo allora ad iterare nello stesso modo, ed al passo j-esimo otteniamo:

$$P(t) = \sum_{i=0}^{n-j} c_i^{[j]} B_{i,n-j}(t) \quad \text{con } \boxed{c_i^{[j]} = t \cdot c_{i+1}^{[j-1]} + (1-t) \cdot c_i^{[j-1]}} \quad \text{e } i = 0, \dots, n-j \text{ e } j = 1, \dots, n$$

Algoritmo di De Casteljau

L'**algoritmo di De Casteljau** non fa quindi uso delle funzioni base, ma ad ogni passo fa una combinazione convessa dei coefficienti al passo precedente.

L'algoritmo può anche essere schematizzato in questo modo:



Ad ogni passo j, calcolo n-j volte $c^{[j]}$. Ad ogni passo, però, perdo un coefficiente da calcolare.

Per ottenere $c_i^{[j]}$, devo fare:
 $c_i^{[j]} = (1-t)c_i^{[j-1]} + tc_{i-1}^{[j-1]}$

In questo modo, la valutazione di un polinomio nella base di Bernstein ha costo computazionale $2 * O(n^2/2)$, siccome ogni coefficiente si calcola con due moltiplicazioni, e calcoliamo $n^2/2$ coefficienti.

Possiamo capire ancora meglio De Casteljau vedendo questo codice usato nel costruttore di linee qua sotto:

```
int i;
float step = 1.0 / (float)(Curva.CP.size() - 1);

float passotg = 1.0 / (float)(pval - 1);

for (tg = 0; tg <= 1; tg += passotg)
{
    c = Curva.CP;

    for (j = 1; j < Curva.CP.size(); j++)
        for (i = 0; i < Curva.CP.size() - j; i++)
            c[i] = vec3((1 - tg) * c[i].x + tg * c[i + 1].x, (1 - tg) * c[i].y + tg * c[i + 1].y, 0.0);

    forma->vertici.push_back(vec3(c[0].x, c[0].y, 0.0));
    forma->colors.push_back(vec4(1.0, 0.0, 0.0, 1.0));
}

forma->nv = forma->vertici.size();
```

Come si può notare, non abbiamo nemmeno guardato il polinomio di Bernstein, ma ci siamo limitati a calcolare i coefficienti e basta.

Interpolazione Lineare e interpretazione geometrica di de Casteljau

L'interpolazione lineare calcola un valore compreso tra due valori. L'interpolazione lineare tra due punti a e b con un parametro t è data da:

$$\text{Lerp}(t, a, b) = (1-t)a + t(b)$$

Perché ci serve l'interpolazione lineare? Come abbiamo detto prima, l'interpolazione lineare gioca un ruolo fondamentale nel calcolo delle curve di Bezier.

Per capire come funziona effettivamente l'algoritmo di de Casteljau, vediamo l'interpretazione geometrica del procedimento in sé.

Consideriamo una curva di Bezier cubica (quindi definita da 4 punti $P_0 \dots P_3$).

Consideriamo i CP (ovvero i punti di controllo della curva, che potevamo anche trovare, per esempio, nelle curve di Hermite) che per una curva di Bezier cubica sono 4, e un valore del parametro t (ad esempio $t = 0.4$, dove t è il valore della traiettoria della curva al tempo t).

Passo 1

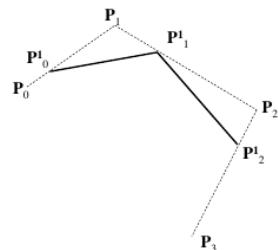
Calcoliamo 3 punti P_0^1, P_1^1, P_2^1 , ottenuti rispettivamente come l'interpolazione lineare fra i punti P_0 e P_1 (ovvero $\text{Lerp}(t, P_0, P_1)$), P_1 e P_2 , P_2 e P_3 .

Passo 2

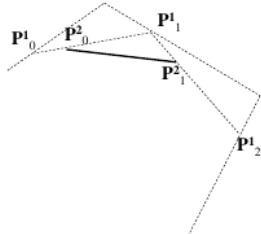
Calcoliamo 2 punti P_0^2, P_1^2 , ottenuti rispettivamente come l'interpolazione lineare fra i punti P_0^1 e P_1^1 (ovvero $\text{Lerp}(t, P_0^1, P_1^1)$), P_1^1 e P_2^1 .

Passo 3, ovvero il passo finale

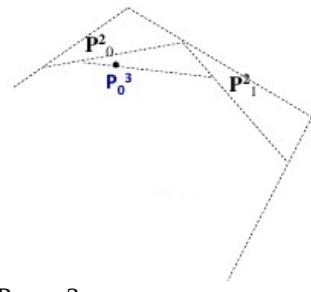
A questo punto, ci basta calcolare l'ultimo punto P_0^3 , che rappresenta il valore della curva di Bezier per $t=0.4$



Passo 1



Passo 2



Passo 3

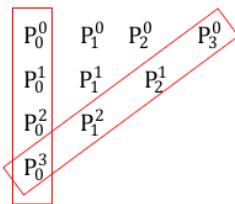
Ripetendo questo procedimento per ogni $t \in [0, 1]$, otteniamo la sequenza di punti che disegnerà la curva.

Splittare (suddividere) una curva

L'algoritmo di de Casteljau permette di suddividere una curva di Bezier di grado n in due sottocurve di grado n, senza troppi calcoli in più. In particolare, questo metodo è usato nel raffinamento delle curve.

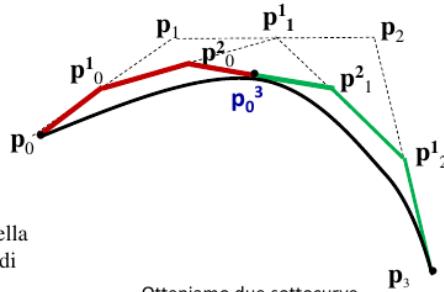
Prendiamo una curva $C(t)$ definita in $[0, 1]$, e la suddividiamo in $t=t_0$. Applichiamo l'algoritmo di De Casteljau per $t = t_0$ e otterremo questi risultati:

Applichiamo l'algoritmo di de Casteljau per $t=t_0$



Punti di controllo della prima sottocurva di curva di Bezier

Punti di controllo della seconda sottocurva di Bezier



Otteniamo due sottocurve
 $C_1(t) = \sum_{i=0}^n P_0^i B_{i,n}(t)$ $t \in [0, t_0]$
 $C_2(t) = \sum_{i=0}^n P_{n-i}^i B_{i,n}(t)$ $t \in [t_0, 1]$

$$C_1(t) = \sum_{i=0}^n P_0^i B_{i,n}(t) \quad t \in [0, t_0]$$

$$C_2(t) = \sum_{i=0}^n P_{n-i}^i B_{i,n}(t) \quad t \in [t_0, 1]$$

I coefficienti della prima colonna rappresentano i punti di controllo della prima curva, mentre la diagonale a destra dei coefficienti rappresentano i punti di controllo della seconda curva.
 P_0^3 , che è il vertice di controllo condiviso dai due poligoni, è anche il punto dove la curva è stata divisa.

La suddivisione rappresenta anche un modo grossolano per disegnare/approssimare la nostra curva (per esempio in fase di caricamento). Inoltre, se iteriamo il procedimento per più punti, la composizione dei poligoni ottenuti converge alla curva originale (ovvero, più aumentiamo i punti, più la spezzata generata dai punti di controllo si avvicina alla nostra curva).

Proprietà dell'algoritmo di de Casteljau

L'algoritmo è **numericamente stabile**, infatti si eseguono solo delle moltiplicazioni e somme per coefficienti positivi [N.B. abbiamo visto le definizioni di numericamente stabile in calcolo numerico].

È **invariante per le trasformazioni affini**, quindi possiamo applicare le trasformazioni ai punti di controllo e poi applicare de Casteljau e ottenere la stessa curva trasformata.

Inoltre, come abbiamo visto, fornisce un metodo semplice per suddividere le curve di Bezier.

Elevamento di grado di un polinomio nella base di Bernstein (Degree Elevation)

Ci sono alcune situazioni in cui risulta necessario esprimere un polinomio di grado n come un polinomio di grado $n+1$, per esempio nel caso in cui bisogna sommare due polinomi di grado diverso.

Mentre nei polinomi monomiali questo normalmente è semplice, la cosa risulta più complessa per i polinomi di Bernstein. Ad esempio:

$$p_1(t) = a_0 B_{0,3}(t) + a_1 B_{1,3}(t) + a_2 B_{2,3}(t) + a_3 B_{3,3}(t)$$

Ci potrebbe venire in mente di calcolare le componenti, ma sarebbe lungoooooo...

$$p_2(t) = c_0 B_{0,2}(t) + c_1 B_{1,2}(t) + c_2 B_{2,2}(t)$$

Dunque si fa un **degree elevation**. In pratica, si esprime lo stesso polinomio con una nuova base, come se fosse un polinomio nella base di Bernstein di grado $n+1$.

$$P_n(t) = \sum_{i=0}^n c_i B_{i,n}(t) \longrightarrow P_{n+1}(t) = \sum_{i=0}^{n+1} d_i B_{i,n+1}(t)$$

Vogliamo quindi trovare i coefficienti d_i tali per cui $P_n(t) = P_{n+1}(t)$

Per fare ciò, moltiplichiamo $P_n(t)$ per $[(1-t)+t]$:

$$\begin{aligned} P_n(t) [(1-t)+t] &= \sum_{i=0}^n c_i \binom{n}{i} [(1-t)^{n+1-i} t^i + (1-t)^{n-i} t^{i+1}] = \\ &= \sum_{i=0}^n c_i \binom{n}{i} [(1-t)^{n+1-i} t^i] + \sum_{i=0}^n c_i \binom{n}{i} (1-t)^{n-i} t^{i+1} \end{aligned}$$

A questo punto, estendiamo la prima sommatoria ottuneta per $i=0, \dots, n+1$, ponendo $c_{n+1}=0$ e $c_{-1}=0$, ed esprimiamo la seconda sommatoria per $i=1, \dots, n+1$. In poche parole, **estendo le mie sommatorie ad $n+1$** . Otteniamo così:

$$=\sum_{i=0}^{n+1} c_i \binom{n}{i} [(1-t)^{n+1-i} t^i] + \sum_{i=1}^{n+1} c_{i-1} \binom{n}{i-1} (1-t)^{n-i+1} t^i =$$

$$P_n(t) = \sum_{i=0}^{n+1} \left(c_i \binom{n}{i} + c_{i-1} \binom{n}{i-1} \right) (1-t)^{n-i+1} t^i$$

A questo punto, consideriamo il polinomio di Bernstein di grado $n+1$:

$$P_{n+1}(t) = \sum_{i=0}^{n+1} d_i \binom{n+1}{i} (1-t)^{n+1-i} t^i$$

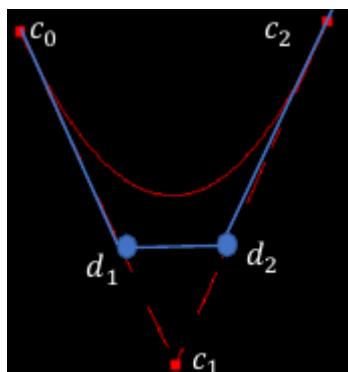
Per fare in modo che rappresentino lo stesso polinomio, devono avere gli stessi coefficienti, cioè deve accadere che:

$$d_i \binom{n+1}{i} = c_i \binom{n}{i} + c_{i-1} \binom{n}{i-1} \quad \xrightarrow{\text{da cui}} \quad d_i = \frac{i}{n+1} c_{i-1} + \left(1 - \frac{i}{n+1} \right) c_i$$

Dal risultato ottenuto, si intuisce che i nuovi coefficienti sono ottenuti **interpolando lineramente per $i/n+1$** i vecchi coefficienti.

Osserviamo inoltre che la formula che fornisce i nuovi coefficienti nella base di grado $n+1$ non è altro che una combinazione convessa dei vecchi coefficienti. Si ha quindi che il nuovo poligono di controllo è interno all'inviluppo convesso dei nuovi coefficienti.

Ecco un esempio grafico:



$$P_2(t) = \sum_{i=0}^2 c_i B_{i,2}(t)$$

$$P_3(t) = \sum_{i=0}^3 d_i B_{i,3}(t)$$

In questo modo, come prima, il polinomio acquisisce un punto di controllo in più, che lo raffina.

OpenGL3D – proiettare il volume di vista nel sistema di coordinate normalizzato

Come abbiamo visto nella lezione precedente, proiettare il volume di vista nel sistema di coordinate normalizzato (Image Space) ci permette di usare **una singola pipeline sia per la visualizzazione prospettica che per la visualizzazione ortografiche**. In questo modo, semplifichiamo sia il clipping (siccome tagliare rispetto a dei piani semplici è più facile rispetto a tagliare il cubo rispetto a dei piani con delle equazioni particolari) che la proiezione.

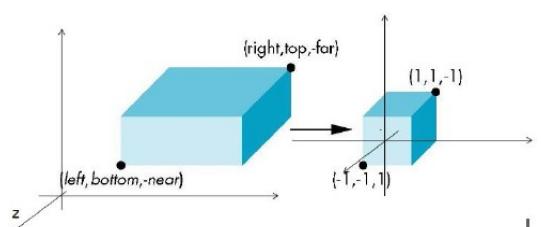
La scena è delimitata da un cuboide centrato nell'origine con coordinate x, y, z in $[-1, 1]$ (dove z è la profondità, 1 più vicino e -1 più lontano).

La visualizzazione 2D finale della scena 3D (l'immagine finale) verrà infine calcolata proiettando la porzione di scena contenuta nel volume della vista canonica in una finestra nel piano dell'immagine.

La normalizzazione nel caso della proiezione ortogonale

consiste in due passi:

1. Spostare il centro del parallelepipedo nell'origine
2. Scalare per avere una vista con lati di lunghezza 2 (siccome le coordinate vanno da -1 a 1)



$$\mathbf{v}' = \mathbf{P} \cdot T_v \cdot T_m \cdot \mathbf{v}$$

$\mathbf{P}(left, right, top, bottom, near, far) =$

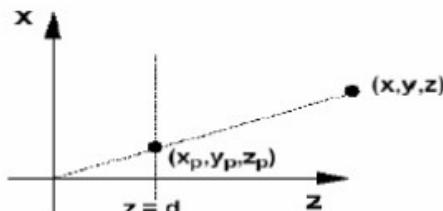
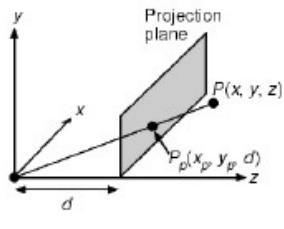
$$\begin{bmatrix} \frac{2}{right-left} & p & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{near-far} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ecco un esempio di matrice di proiezione, che come si può notare esegue sia le operazioni di traslazione che di scalatura della vista.

Essenzialmente, è la matrice che ci ritorna, per esempio, ortho.

Matematica delle proiezioni

Consideriamo un punto con coordinate xyz, e vogliamo trovare le coordinate della sua proiezione sul piano con equazione z=d (In poche parole, abbiamo quindi un piano parallelo al piano xy e posto a distanza d). Il nostro punto di vista è l'origine.



Consideriamo il punto (x, y, z), e vogliamo sapere le coordinate (x_p, y_p, z_p) della sua proiezione su z=d.

Per trovare le coordinate dei punti, dobbiamo eseguire alcune proporzioni.

In particolare, per trovare la coordinata x del punto dobbiamo considerare il piano xz (quello visto nella figura sopra) e usare la proporzione $x : x_p = z : d$, da cui posso ricavare x_p con la formula

$$x_p = \frac{d \cdot x}{z} = \frac{x}{z/d}$$

Analogamente, dobbiamo fare la stessa cosa con il piano yz per trovare la coordinata del punto y_p , usando la proporzione $y : y_p = z : d$.

La coordinata z del punto sarà semplicemente d, dunque le coordinate del punto finale saranno (x_p, y_p, d) .

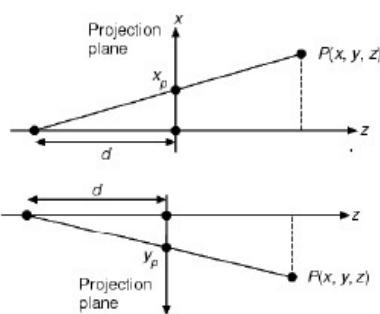
A noi però interessa avere una matrice per esprimere questa relazione, siccome tutte le operazioni in OpenGL vengono svolte attraverso l'uso di moltiplicazioni per matrici.

$$\begin{bmatrix} x * d/z \\ y * d/z \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Il secondo vettore è quello che si ottiene attraverso la moltiplicazione per matrici a destra, mentre il vettore a sinistra è dato dividendo il vettore ottenuto per la sua quarta coordinata.

Notare come il quarto vettore sia effettivamente quello che rappresenta le coordinate del nostro punto.

Per chiarire, facciamo un altro esempio, con punto di vista $(0,0,-d)$ e piano di proiezione $z=0$.



$$x \cdot x_p = d + z \cdot d$$

$$x_p = \frac{d \cdot x}{z+d} = \frac{x}{(z/d)+1}$$

$$y \cdot y_p = d + z \cdot d$$

$$y_p = \frac{d \cdot y}{z+d} = \frac{y}{(z/d)+1}$$

$$\begin{bmatrix} x \\ y \\ 0 \\ (z+d)/d \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ (z+d)/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x * d/(z+d) \\ y * d/(z+d) \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ (z+d)/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Essenzialmente, in questo caso dobbiamo immaginare di traslare il piano e il punto di vista di +d, in modo che ritorni all'origine. A questo punto possiamo rifare le proporzioni viste in precedenza.

Notare come nella matrice z sia uguale a 0, siccome appunto il piano che andiamo a prendere in considerazione è appunto $z=0$, e per fare in modo che $z+d/d$ compaia nella formula, dovremo mettere $1/d$ e 1 nell'ultima riga della matrice.

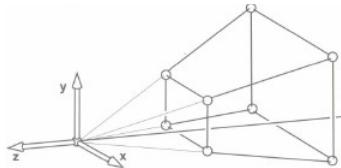
$$\mathbf{M}'_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix}$$

In entrambi i casi (ovvero sia questo che quello precedente) la matrice ottenuta è la **matrice di proiezione**.

Se in entrambi i casi d tendesse a ∞ , allora avremo avuto una proiezione parallela.

Caso della proiezione prospettica

Nel caso della proiezione prospettica, avremo a che fare con un volume di vista con forma simile a un cubo distorto (detto anche, appunto, **frostum**) che dovrà essere trasformato in un cubo dopo l'operazione di normalizzazione.



In geometria, il frostum è la figura che si ottiene tagliando con due piani paralleli un solido.

Per farlo, bisognerà eseguire anche sta volta tre operazioni (dove le ultime due sono le stesse già viste nella proiezione ortogonale...):

- si modificano le coordinate dei punti del far plane in modo da ottenere un parallelepipedo.
- si trasla il parallelepipedo in modo che abbia centro nell'origine
- lo si scala in modo da ottenere un cubo con lato pari a 2.

Dobbiamo individuare la matrice di proiezione prospettica M_{persp} che faccia la prima trasformazione (per le altre due possiamo usare benissimo la matrice della proiezione ortogonale).

Otterremo così che la matrice di proiezione sarà $P = P_{ortho}M_{persp}$.

La matrice ottenuta da questa composizione sarà questa [N.B. la prof ha detto che non è importante da sapere, ma io lo metto comunque per completezza]:

$$P = \begin{bmatrix} \frac{2*near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2*near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & \frac{-2*far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

OpenGL, per convenzione, durante questa fase di trasformazione da coordinate di vista a NDC, inverte l'asse z, usando così a tutti gli effetti un left-handed coordinate system. Per questo motivo, $z=-1$ è la posizione più vicina allo schermo, mentre 1 è la più lontana.

Window Transformations e trasformazione window-viewport

Una volta che il volume di vista è stato mappato in un cubo con centro l'origine e lato 2, l'immagine 2D finale è ottenuta semplicemente annullando la coordinata dopo la proiezione ortogonale nel piano z = 0.

La **Window Transformation** è una proiezione ortogonale che mappa punti in NCS (x, y, z) espressi in $[-1,1] \times [-1,1] \times [-1,1]$ (spazio immagine 3D) in una regione rettangolare (x, y) in $[-1,1] \times [1,1]$ (finestra 2D) [in poche parole, **elimina la terza dimensione**].

$$P_{ortho} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \boxed{0} \\ 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow P_{ortho} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ 0 \\ 1 \end{pmatrix}$$

La matrice moltiplicata per il vettore da un vettore senza coordinata z.

Abbiamo poi un'ultima [FINALMENTE] trasformazione rimanente da eseguire: la **trasformazione window-viewport**.

In poche parole, abbiamo due aree rettangolari, che sono la window e la viewport. La **window** è la finestra 2D attraverso la quale si guarda la scena 3D e che verrà visualizzata sullo schermo in un'area rettangolare, una finestra fisica, detta **viewport** (sistema di coordinate schermo).

La viewport quindi è un array di $(nx \times ny)$ pixel. Alle coordinate reali di ogni vertice nella window dovranno corrispondere **copie di indici interi che individuano il corrispondente pixel nella viewport**.

Bisognerà quindi fare opportune operazioni di traslazione (per poter portare il rettangolo all'origine) e scalatura per fare in modo che la scena della window venga mostrata nella viewport in coordinate di device.

Abbiamo così concluso la parte del sottosistema geometrico. Finalmente. Ora vedremo il sottosistema di rasterizzazione nella lezione dopo.

Lezione 11/11/2021 – La nostra prima scena 3D: primitive!

Lorem Ipsum

Lorem Ipsum

Lezione 15/11/2021 – Fine Bezier, Inizio Spline, Inizio Sottosistema Raster, Algoritmo di Cohen-Sutherland (Clipping), Algoritmo di Bresenham (Rasterizzazione)

Derivata di una curva di Bezier

Data una curva di Bezier (che come sappiamo si esprime come una combinazione lineare di $n+1$ vertici di controllo con $n+1$ funzioni di base di Bernstein di grado n), la derivata prima di questa curva di Bezier si esprime come:

$$C'(t) = \sum_{i=0}^n P_i B'_{i,n}(t)$$

dove $t \in [0, 1]$.

Per la derivata delle funzioni base vale la formula:

$$B_{i,n}(t) = \binom{n}{i} (1-t)^{n-i} t^i \quad \longrightarrow \quad B'_{i,n}(t) = n(B_{i-1,n-1}(t) - B_{i,n-1}(t))$$

che è facilmente dimostrabile calcolando la derivata della formula.

Questa espressione mette in evidenza il fatto che la curva di Bezier è tangente negli estremi al segmento che unisce i primi due punti e gli ultimi due punti di controllo rispettivamente. Infatti:

Se $i=0$ nella prima sommatoria $B_{-1,n-1}(t) = 0$

Se $i=n$ nella seconda sommatoria $B_{n,n-1}(t) = 0$.

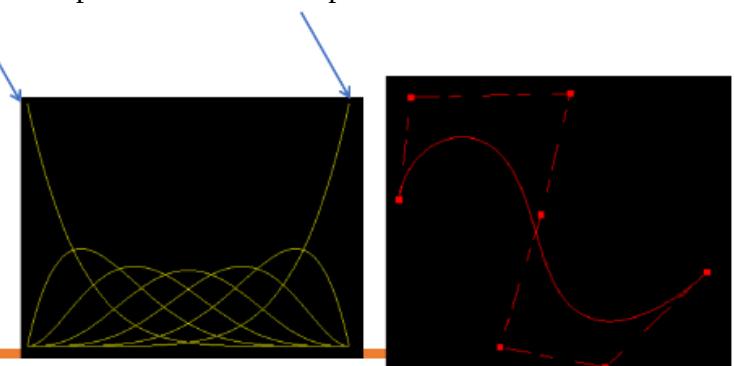
Possiamo così arrivare alla conclusione che il vertice di controllo P_0 corrisponde al valore del parametro $t=0$, mentre il vertice di controllo P_n corrisponde al valore del parametro $t=1$.

$$C(0) = P_0 B_{0,n}(0) + P_1 B_{1,n}(0) + \dots + P_n B_{n,n}(0)$$

$$C(0) = P_0 B_{0,n}(0) = P_0$$

$$C(1) = P_0 B_{0,n}(1) + P_1 B_{1,n}(1) + \dots + P_n B_{n,n}(1)$$

$$C(1) = P_n B_{n,n}(1) = P_n$$



La curva di Bezier interpola il primo e l'ultimo vertice di controllo, siccome in $t=0$ il valore della curva è pari a P_0 , mentre in $t=1$ è pari a P_n .

Cambiamento di Base – da base Monomiale a base di Bernstein (aggiunto per completezza)

A volte è necessario passare dalla base di Bernstein alla base monomiale e viceversa.

$$f(t) = \sum_{i=0}^n c_i B_{i,n}(t) = \sum_{i=0}^n a_i t^i$$

Questa è la trasformazione che deve avvenire.
La prima parte è il polinomio di Bezier, mentre la seconda è il monomio.

Bernstein to Monomial

La matrice di cambiamento di base che fa passare dalla *base dei polinomi di Bernstein di grado n* alla *base monomiale di grado n* è la **matrice Λ (lambda)** di ordine $n+1$ i cui elementi sono dati da

$$\Lambda = \lambda_{jk} = \begin{cases} \binom{k}{j} \binom{n}{j}^{-1} & k \geq j \\ 0 & \text{altrimenti} \end{cases}$$

Ogni monomio t^j si può esprimere nella forma:

$$t^j = \sum_{k=0}^n \lambda_{jk} B_{n,k}(t)$$

Monomial to Bernstein

La matrice di cambiamento di base che fa passare dalla *base monomiale di grado n* alla *base dei polinomi di Bernstein di grado n* è la **matrice Λ^{-1}** di ordine $n+1$ i cui elementi sono dati da

$$\Lambda^{-1} = \lambda_{jk}^{-1} = \begin{cases} (-1)^{k-j} \binom{n}{k} \binom{k}{j} & k \geq j \\ 0 & \text{altrimenti} \end{cases}$$

Ogni funzione base $B_{j,n}(t)$ si può esprimere nella forma:

$$B_{j,n}(t) = \sum_{k=0}^n \lambda_{jk}^{-1} t^k$$

Per quanto riguarda i coefficienti avremo che:

$$p(t) = \sum_{i=0}^n a_i t^i = [a_0 \ a_1 \ \dots \ a_n] \begin{bmatrix} 1 \\ t \\ t^2 \\ \vdots \\ t^n \end{bmatrix} = [c_0 \ c_1 \ \dots \ c_n] \begin{bmatrix} B_{0,n}(t) \\ B_{1,n}(t) \\ B_{2,n}(t) \\ \vdots \\ B_{n,n}(t) \end{bmatrix}$$

Ma,

$$p(t) = [a_0 \ a_1 \ \dots \ a_n] \Lambda \begin{bmatrix} B_{0,n}(t) \\ B_{1,n}(t) \\ B_{2,n}(t) \\ \vdots \\ B_{n,n}(t) \end{bmatrix}$$

Segue quindi: $[c_0 \ c_1 \ \dots \ c_n] = [a_0 \ a_1 \ \dots \ a_n] \Lambda$

$$[a_0 \ a_1 \ \dots \ a_n] = [c_0 \ c_1 \ \dots \ c_n] \Lambda^{-1}$$

Quando mai potrebbe essere utile qualcosa del genere? Beh potrebbe essere utile in fase di valutazione di un polinomio. Infatti, de Casteljau ha complessità $O(n^2)$, mentre lo schema di Horner (che è un altro metodo per la valutazione dei polinomi) ha complessità $O(n)$. Quindi per esempio possiamo usare lo schema di Horner che ci mette di meno. Consideriamo che comunque ci sarebbe il cambiamento di base che ha una certa complessità, quindi insomma dipende dalla situazione. Inoltre de Casteljau è stabile, Horner no.

Proprietà delle curve di Bezier

Come abbiamo visto, il **grado dei polinomi di Bernstein** che parametrizzano la curva è sempre

dato dal numero dei punti di controllo meno 1. Ad esempio, se abbiamo $n+1$ punti di controllo, il grado è n.

Questo è uno strumento che ci dà flessibilità, infatti è possibile aumentare i gradi di libertà della curva aumentando il numero dei punti di controllo. Allo stesso tempo, però, può avere anche delle conseguenze negative, perché all'aumentare del numero dei vertici di controllo corrisponde un aumento del grado del polinomio e la **complessità computazionale cresce**, in quanto ogni valutazione costa $O(n^2)$.

La curva “segue” (o approssima) la forma del poligono di controllo e si trova sempre nell’inviluppo convesso dei punti di controllo. Come abbiamo visto, il primo e l’ultimo punto di controllo sono l’inizio e la fine della curva e vengono interpolati. I vettori tangentи alla curva nel primo e nell’ultimo punto coincidono con il primo e l’ultimo lato del poligono di controllo.

Le intersezioni della curva con una retta sono sempre in numero minore o al massimo uguale a quelle che la retta ha con il poligono di controllo (variation diminishing).

Le curve di Bézier sono invarianti per trasformazioni affini. La curva è trasformata applicando una qualunque trasformazione affine ai suoi punti di controllo [come abbiamo visto [qui](#)].

Svantaggi dell’uso delle curve di Bezier

I vertici di controllo **non permettono di effettuare un controllo locale della curva**. Muovendo, infatti, uno qualunque dei suoi vertici di controllo si ottiene un effetto su tutta la curva, anche se esso è meno evidente nelle zone lontane dal punto che si è spostato. Questo perché i polinomi di base di Bernstein sono diversi da zero su tutto l’intervallo (**supporto globale**).

Per ovviare a questo problema, un approccio che spesso si usa è quello di spezzare a 4 a 4 le curve in modo da ottenere tante curve cubiche, che poi vengono raccordate con opportune condizioni di regolarità.

Curve spline

I difetti delle curve di Bezier consistono nella mancanza di controllo locale della curva e nel fatto che il grado dei polinomi di base di Bernstein è strettamente legato al numero di vertici controllo (se i vertici di controllo sono $n+1$ il grado dei polinomi di base di Bernstein è n). **Le curve spline permettono di superare entrambi i limiti.**

Dati i vertici di controllo P_i , $i=1,\dots,N$, una curva spline si definisce come:

$$C(t) = \sum_{i=1}^N P_i N_{i,m}(t)$$

con $t \in [a, b]$, e le funzioni base $N_{i,m}(t)$:

- hanno ordine m, molto inferiore rispetto al numero dei vertici di controllo
- ed inoltre è possibile controllare localmente la forma delle curve, in quanto le funzioni base B-spline sono a **supporto compatto** (contrariamente al supporto globale), non sono diverse da zero su tutto l’intervallo $[a,b]$, ma hanno un supporto locale che dipende dal loro ordine.

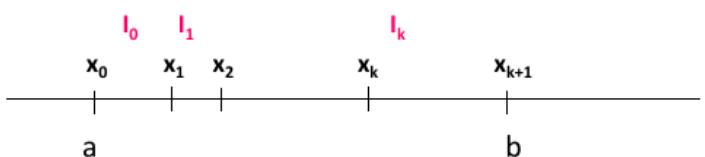
Spline polinomiali a nodi multipli

Sia $[a,b]$ un intervallo chiuso e limitato, sia Δ una partizione di $[a,b]$ così definita:

$$\Delta = \{a=x_0 < x_1 < \dots < x_k < x_{k+1}=b\}$$

Ovvero dividiamo il nostro intervallo in $k+1$ sottointervalli, che sono chiusi a sinistra e aperti a destra (a parte per l’intervallo $[k, k+1]$):

$$\begin{aligned} I_i &= [x_i, x_{i+1}) & \text{per } i=0, \dots, k-1. \\ I_k &= [x_k, x_{k+1}] \end{aligned}$$

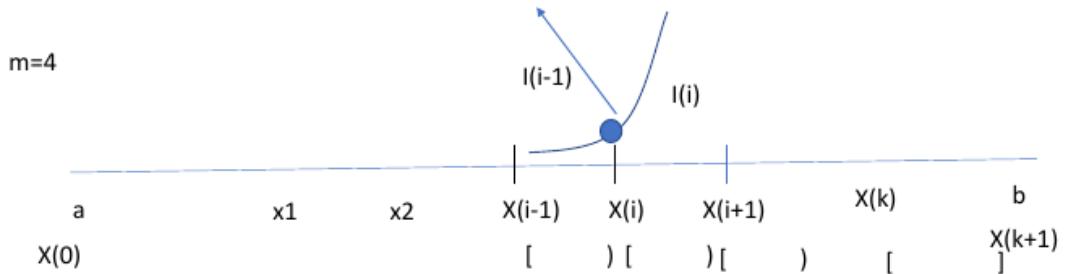


Sia poi m un intero positivo tale che $m < k$ e sia $M = (m_1, m_2, \dots, m_k)$ un vettore di interi positivi tali che $1 \leq m_i \leq m \quad \forall i = 1, \dots, k$

Si definisce funzione **spline polinomiale di ordine m** con nodi x_1, \dots, x_k di molteplicità m_1, \dots, m_k , una funzione $s(x)$ tale che in ciascun sotto intervallo I_i (con $i = 0, \dots, k$) coincide con un polinomio $s_i(x)$ di ordine m e che nei nodi x_i ($i = 1, \dots, k$) soddisfi le condizioni di continuità:

$$\frac{d^j s_{i-1}(x_i)}{dx^j} = \frac{d^j s_i(x_i)}{dx^j} \quad i = 1, \dots, k \quad j = 0, \dots, m - m_i - 1$$

Ecco un esempio, con la mia solita stanca spiegazione:



Nei nodi interni, il polinomio a dx e a sx coincidono in valore e derivate dalle derivata prima alla derivata di ordine $(m - m_i - 1)$. È questo ciò che ci dice la formula sopra.

Possiamo **decidere noi** fino a che grado di continuità si può arrivare in ciascun x_i , giocando con il valore di m_i .

Maggiore sarà la molteplicità m , minori saranno le condizioni di raccordo sui nodi.

Se poniamo $m_i=1$ ($i=1, \dots, k$) otteniamo la **spline a nodi semplici**, che ha la massima regolarità per essere una funzione spline. In questo modo, in ogni nodo si ha il raccordo dalla derivata 0-esima fino alla derivata $m - 2$.

Se $m_i=m$ non abbiamo alcun raccordo nel nodo i -esimo.

Facciamo un esempio, con $m=4$. In questo caso, se, per un certo i , $1 \leq i \leq k$, $m_i = 1$, allora il nodo x_i è *semplice*. Quindi dovranno coincidere per:

$$s_i''(x_i) = s_{i-1}''(x_i)$$

$$s_i'(x_i) = s_{i-1}'(x_i)$$

$$s_i(x_i) = s_{i-1}(x_i)$$

Altrimenti, se $m_i = 2$, il nodo x_i è *doppio*. Dovrà coincidere solo per:

$$s_i'(x_i) = s_{i-1}'(x_i) \quad s_i(x_i) = s_{i-1}(x_i)$$

Quindi, nel primo caso, chiedo che i polinomi della funzione della spline siano di classe C^2 .

Se avessi considerato $m_i = 3$, avrei ridotto ancora di più la richiesta di regolarità della curva (chiedo solo che ci sia il raccordo il valore, e avrò quindi una cuspidè!).

In base alla molteplicità (ovvero al valore di m_i che associo a ciasun x_i) posso far convivere nella stessa curva situazioni di irregolarità diverse.

Le curve di Hermite sono anch'esse degli esempi di curve spline (come abbiamo visto [qui](#)), in particolare in ciascuno dei sotto intervalli erano definite da un polinomio cubico, che nel punto di controllo avevano stesso valore e stessa derivata prima (C^1).

Normalmente invece, le curve spline sono definite da delle funzioni di classe C^2 oppure di classe C^{m-2} .

Normalmente, arrivare alla derivata terza non è una decisione saggia nella definizione di queste curve. Infatti in questo modo ci avviciniamo sempre di più a un polinomio, siccome in un polinomio è di classe C^∞ è continuo insieme a tutte le sue derivate. Qui invece trattiamo di funzioni polinomiale a tratti, e semplicemente stiamo ridimensionando la richiesta di regolarità per avere maggiore flessibilità.

Per definizione, quindi, una funzione spline **non può avere una regolarità superiore a m-2**.

Lo **spazio delle spline polinomiali** di ordine m a nodi multipli con nodi x_1, \dots, x_k di molteplicità $M = (m_1, \dots, m_k)$, che verrà indicato con $S_m(\Delta, M)$, ha dimensioni pari a $m + K$, dove $K = \sum_{i=1}^k m_i$.

Dimostrazione intuitiva: per ciascuno dei $k+1$ sottointervalli il polinomio di ordine m ha m gradi di libertà, quindi si hanno **$(k+1)*m$ gradi di libertà** (che sarebbe stato il numero minimo di gradi di libertà se i polinomi non avessero avuto un comportamento “regolato”, senza le condizioni di raccordo quindi);

A queste si devono sottrarre le $m - m_i$ condizioni di raccordo (siccome impongo che sia continua per $j = 0, \dots, m - m_i - 1$ che fanno in totale $m - m_i$ condizioni) imposte su ogni nodo x_i , $i = 1, \dots, k$. Si ha, quindi:

$$m(k+1) - \sum_{i=1}^k (m - m_i) = mk + m - \sum_{i=1}^k m + \sum_{i=1}^k m_i = mk + m - mk + K = m + K$$

Base dello spazio delle Spline B-spline normalizzate

Una buona base per lo spazio delle spline $S_m(\Delta, M)$ è la base delle B-spline normalizzate.

Funzioni base B-spline – caso a nodi semplici

Assegnata una successione di nodi $\dots < x_1 < x_2 < \dots$ semplici, si definisce B-spline normalizzata di ordine m relativa al nodo x_i (con $x \in [a, b]$), una funzione $N_{i,m}(x)$ che gode delle seguenti proprietà:

$N_{i,m}(x) = 0$ per $x < x_i$ $x > x_{i+m}$, (supporto compatto);

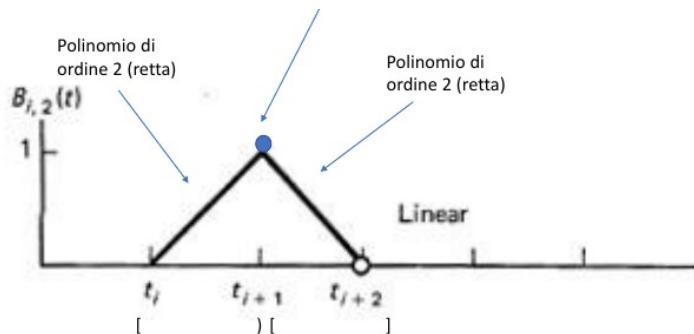
1) il supporto è compatto, come abbiamo detto prima. Ci sono quindi due valori tali per cui, se è minore del primo o maggiore del secondo il valore della funzione è pari a 0.

$\int_{x_i}^{x_{i+m}} N_{i,m}(x) dx = \left(\frac{x_{i+m} - x_i}{m} \right)$ (Condizione di normalizzazione);

2) la condizione di normalizzazione, che indica che l'area della funzione dev'essere pari a tale funzione.

Facciamo degli esempi:

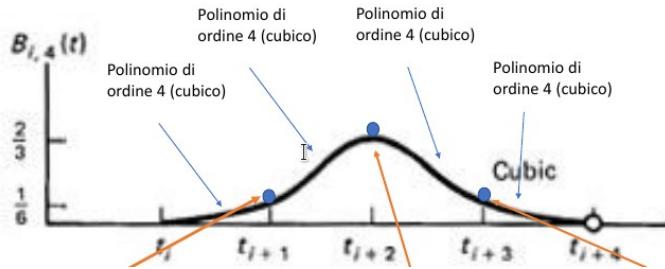
consideriamo $m=2$, ciò vuol dire che le funzioni spline saranno di grado 1 (grado = ordine - 1, siccome vuol dire che lo posso derivare 2 volte e poi otterrei $f(x) = 0$). La curva che potremmo ottenere sarà:



I due polinomi coincidono in valore, siccome la loro continuità è di classe $C^{m-2} = C^0$

Come possiamo notare dall'esempio sopra appena fatto, la B-spline è definita su un numero di sottointervalli limitato e pari al suo ordine.

Vediamo il caso con $m=4$ (il valore più usato, ed è quello usato anche in Blender):



Siccome i nodi sono semplici, si coincide per derivata prima e derivata seconda (oltre che al valore) nei punti di raccordo. Inoltre, siccome $m=4$, la funzione è definita in 4 intervalli.

Le B-spline che di solito si usano sono quelle cubiche, ovvero con $m=4$.

Dunque, in ciascun intervallo I_j $j = i, \dots, i+m-1$, la funzione B-spline coincide con un polinomio di ordine m che si raccorda opportunamente con i vicini, in modo tale che $N_{i,m}(x) \in C^{m-2}$ in $[a, b]$.

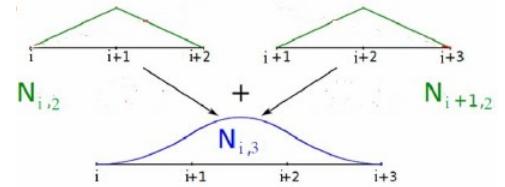
Formule di Cox per la valutazione di una B-spline

Per calcolare le funzioni base, si utilizzano le **formule di Cox**, che consentono di calcolare $N_{i,m}(x)$ mediante combinazione di due funzioni base di ordine $m-1$ (quindi di ordine minore), con degli opportuni coefficienti. Più precisamente, si ha:

$$N_{i,1}(x) = \begin{cases} 1 & x_i \leq x \leq x_{i+1} \\ 0 & \text{altrimenti} \end{cases}$$

per $h=2, \dots, m$ si ha

$$N_{i,h}(x) = \left(\frac{x - x_i}{x_{i+h-1} - x_i} N_{i,h-1}(x) + \frac{x_{i+h} - x}{x_{i+h} - x_{i+1}} N_{i+1,h-1}(x) \right)$$

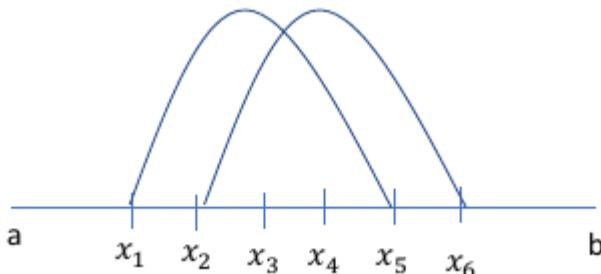


Formule di Cox per la valutazione di una B-spline

Vediamo ora come utilizzare le funzioni B-spline di ordine m per costruire una base per lo spazio delle spline $S_m(\Delta, M)$. Poiché $S_m(\Delta, M)$ ha dimensione $m+K$, bisogna essere in grado di costruire $m+K$ funzioni base B-spline [ovvero questo sarà il numero necessario di basi per lo spazio].

Quindi, dire che lo spazio delle spline ha dimensione $m+K$, vuol dire che la sua base deve essere costituita da $m+K$ funzioni base.

Consideriamo il caso in cui $m = 4$ e $k = 6$ nodi ognuno con molteplicità 1. Dobbiamo essere in grado di costruire $m + k = 10$ funzioni base. E non possiamo fare ciò solo con i k nodi che abbiamo ($k < k + m$). Ogni curva, infatti, sarà definita in m intervalli.

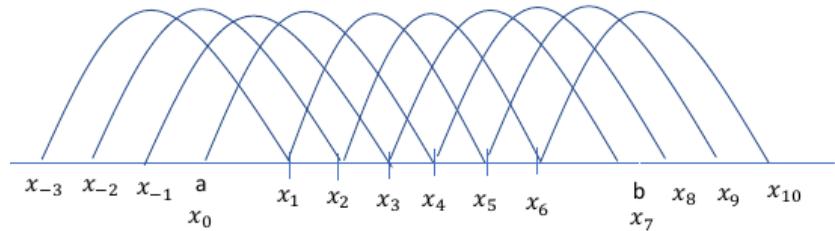


Se $m=4$, allora ciascuna curva avrà bisogno di 4 intervalli: i 6 intervalli non bastano, siccome alcune curve “sfociano”.

Abbiamo bisogno quindi di più intervalli.

Aggiungo quindi 2m nodi finti. Di questi, m vengono inseriti prima di x_1 ed altrettanti dopo x_k . La nuova partizione nodale creatasi prende il nome di **partizione nodale estesa Δ^*** .

Nel nostro esempio $m=4$ e $k=6$, inseriamo $m=4$ nodi fittizi a sinistra prima di x_1 e $m=4$ nodi fittizi a destra dopo di x_6



Riusciamo così a costruire tutte le B-spline che formano la base dello spazio delle Spline.

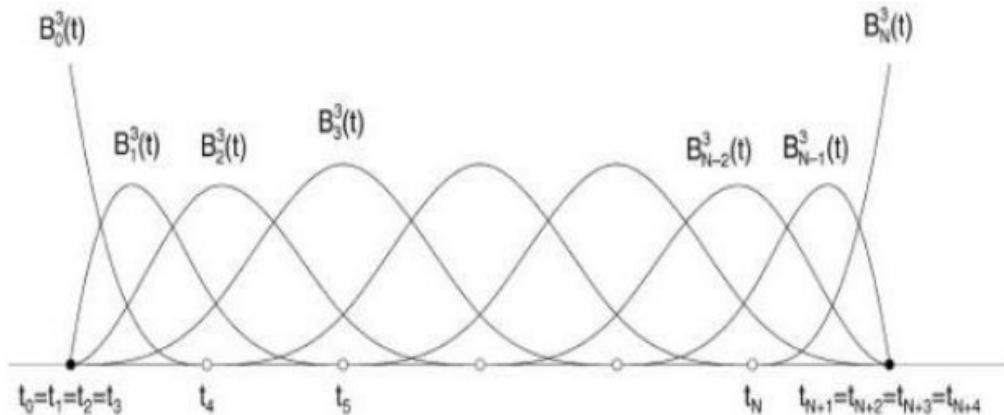
Funzioni base B-spline – caso a nodi non semplici, quindi con $m_i > 1$ - partizione nodale estesa

Data la partizione nodale $\Delta = \{a = x_0 < x_1 < \dots < x_k < x_{k+1} = b\}$ ed il vettore di molteplicità $M = (m_1, \dots, m_k)$, si costruisce la partizione nodale estesa $\Delta^* = \{t_i\}_{i=1}^{2m+K}$ con $K = \sum_{i=1}^k m_i$ nel seguente modo:

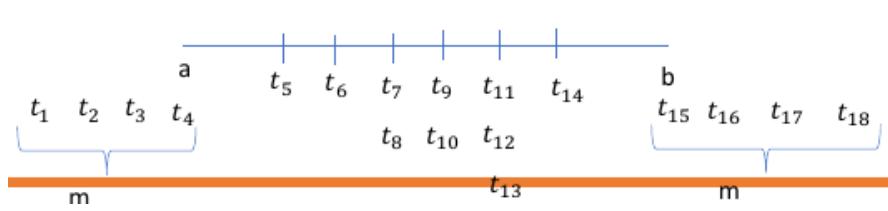
- a) $t_1 \leq t_2 \leq t_3 \leq \dots \leq t_{2m+K}$ i nuovi nodi sono semplici, ma possono coincidere.
- b) $t_m \equiv a, t_{m+K+1} \equiv b$
- c) $t_{m+1} \leq t_{m+2} \leq t_{m+3} \leq \dots \leq t_{m+K}$ sono scelti in modo tale che siano coincidenti con:

$$(x_1 \equiv x_1 \equiv \dots \equiv x_1) < (x_2 \equiv x_2 \equiv \dots \equiv x_2) < \dots < (x_k \equiv x_k \equiv \dots \equiv x_k)$$
Per m_1 volte, m_2 volte ... m_k volte rispettivamente.
- d) $t_i \leq a$ con $i = 1, \dots, m-1$ (possono coincidere tutti con a)
 $t_i \geq b$ con $i = m+K+2, \dots, 2m+K$ (possono coincidere tutti con b)

Cosa vuol dire tutto ciò? Per capirlo per bene, facciamo un esempio. Prendiamo il caso in cui $M = (m_1, m_2, m_3, m_4, m_5, m_6), M = (1, 1, 1, 1, 1, 1)$ [dunque è il caso con nodi semplici, siccome la molteplicità di ciascun nodo è 1]. Avremo $m = 4$ nodi fittizi coincidenti con l'estremo a , e $m = 4$ nodi fittizi a destra coincidenti con l'estremo b .



Vediamo un caso più curioso. Prendiamo in considerazione una curva spline con $m = 4$, $k = 6$ e molteplicità $M = (1, 1, 2, 2, 3, 1)$. La dimensione della base è $m + K$, in questo caso è costituita da 14 funzioni base. Tuttavia, la partizione nodale estesa è costituita da $2m + K$ nodi, quindi nel nostro caso è costituita da 18 nodi. Questo vuol dire che avremo una situazione del genere:



La somma delle molteplicità (ovvero K) è pari a 10.
In ciascun nodo, possiamo notare che il numero dei nodi è pari alla molteplicità in ciascun nodo.

Agli estremi invece, come prima, avremo i nodi fittizi.

Se m_i in un punto fosse stato uguale a 4, avrei avuto un breakpoint, in cui la linea si spezza, ovvero non c'è raccordo nemmeno per valore, siccome $(m - m_i - 1) = -1$ in questo caso.

[METANOTA: Il discorso delle curve spline verrà ripreso alla lezione della settimana dopo.]

Sottosistema raster

Abbiamo precedentemente visto il [sottosistema geometrico](#), adesso vedremo il sottosistema raster, ma prima dobbiamo fare una puntualizzazione sul sottosistema geometrico.

Piccola aggiunta nel sistema geometrico – clipping

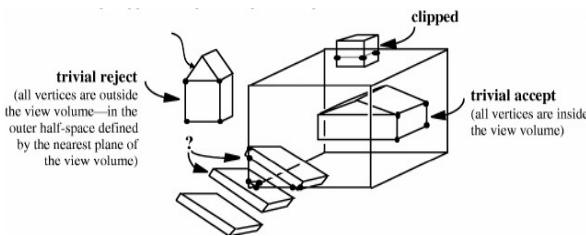
Subito dopo la fase di trasformazione della proiezione ([Projection-Transform](#)) che trasforma il volume di vista in NDC, abbiamo la **fase di clipping**, ovvero la fase in cui gli oggetti che si trovano al di fuori del volume di vista vengono eliminati.

Il clipping inoltre avviene subito prima della proiezione dello spazio a schermo.

I poliedri/modelli geometrici della nostra scena vengono trasformati anch'essi in coordinate normalizzate durante il processo di Projection-Transform, e vengono ritagliati rispetto ai limiti del volume di vista canonico, un poligono alla volta. A loro volta, i poligoni vengono tagliati uno spigolo alla volta. Quando gli oggetti vengono tagliati (quindi quando sono solo parzialmente dentro al volume di vista), vengono creati dei nuovi vertici.

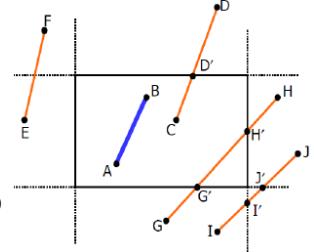
Se tutti i vertici di un poliedro sono esterni al volume di vista, allora vengono semplicemente scartati.

Come abbiamo visto, i calcoli di intersezione sono banali grazie ai piani normalizzati del volume di vista trasformato.



Per capire meglio, facciamo un esempio considerando un volume di vista 2D.

L'algoritmo che si usa per il clipping consiste nel determinare le intersezioni fra la retta su cui giace il segmento (estraendo la sua equazione parametrica) e le 4 rette su cui giacciono i lati del rettangolo di clipping. Dopo che si sono individuati i punti di intersezione, occorre verificare l'appartenenza al rettangolo di clipping (come G' e H') oppure no (come I' e J').



Il metodo che normalmente si usa, però, non si basa su questo algoritmo, bensì sull'**algoritmo di Cohen-Sutherland**.

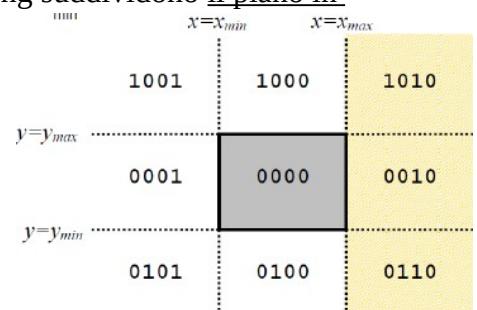
Algoritmo di Cohen-Sutherland per il clipping

Con l'**algoritmo di Cohen-Sutherland** [ovviamente consideriamo l'algoritmo nello spazio 2D], dobbiamo immaginarcici che le rette che delimitano il rettangolo di clipping suddividono il piano in nove regioni.

Ad ogni regione, viene associato un codice numerico a 4 cifre binarie.

Di questo codice:

- il bit 1 rappresenta l'area $y > y_{\max}$ (edge alto)
- il bit 2 rappresenta l'area $y < y_{\min}$ (edge basso)
- il bit 3 rappresenta l'area $x > x_{\max}$ (edge destro)
- il bit 4 rappresenta l'area $x < x_{\min}$ (edge sinistro)



Il clipping di un segmento viene fatto attraverso il confronto dei suoi estremi sulla base delle regioni di appartenenza. Se il codice di entrambi gli estremi è 0000 (e quindi l'OR ritorna 0), allora si può

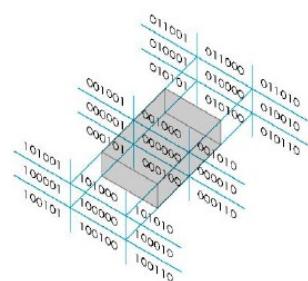
decidere in modo banale che il segmento rientra nell'area di clipping. Se, invece, l'**AND** logico tra i codici degli estremi restituisce un risultato non nullo, allora il segmento è esterno all'area di clipping. In questo caso, infatti, gli estremi giacciono in uno stesso semipiano, e quindi non c'è intersezione con il rettangolo di clipping.

Se il risultato dell'AND è nullo, la situazione si fa più complessa. In tal caso, infatti:

- bisogna individuare l'intersezione tra il segmento e la retta relativa al primo bit discordante tra i codici (il vertice con bit posto a 1, $y = y_{\max}$)
- l'estremo del segmento che aveva il bit a 1 viene sostituito dal nuovo vertice (ovvero dall'intersezione che abbiamo trovato prima).
- si itera il procedimento per il secondo bit, e poi il secondo estremo viene sostituito con l'intersezione e la retta relativa al secondo bit (ovvero, $y = y_{\min}$).

Per il **3D**, la versione di Cohen-Sutherland è molto simile, e consiste nel dividere il volume di clipping in 27 regioni, con un bitcode di 6 cifre, dove:

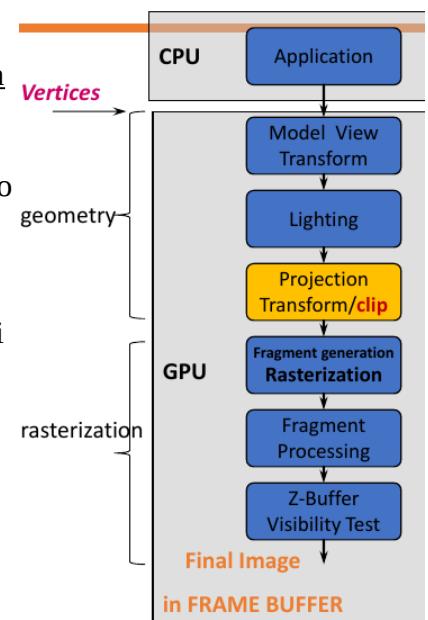
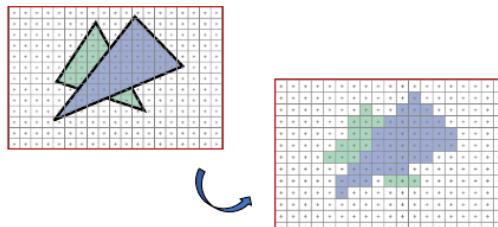
- la prima rappresenta la regione dietro al piano posteriore
- la seconda davanti al piano frontale
- i restanti 4 bit uguali a quelli precedenti



Scan Conversion – Rasterization

La fase di rasterizzatore, come sappiamo, determina i pixel interni alle primitive che hanno subito l'insieme delle trasformazioni del sottosistema geometrico. Questa fase produce così dei **frammenti**. Noi NON possiamo intervenire nella fase di rasterizzazione, della quale si occupa esclusivamente la GPU: non possiamo scrivere shader per esso [possiamo invece intervenire nella fase di fragment processing, per la quale scriviamo i fragment shader].

I **frammenti** hanno una posizione (detta **pixel location**) e altri attributi, come colore e coordinate texture che vengono determinate interpolando i valori sui vertici.



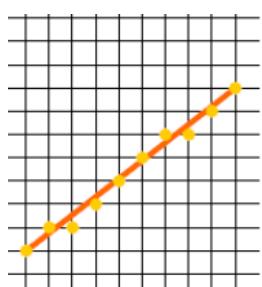
Nella **fase di fragment processing**, si elaborano i frammenti aggiungendo colori e texture, e si disegna l'oggetto dal più lontano al più vicino.

Ma perché nasce la rasterizzazione? Come sappiamo, questa fase nasce siccome le primitive geometriche che andiamo a disegnare sono continue, mentre lo schermo è un dispositivo "discreto". La fase di rasterizzazione è composta da algoritmi efficienti che generano, appunto, i frammenti a partire dalle primitive geometriche. Inoltre, enumera i frammenti occupati dalla primitiva e interpola i valori, detti attributi, lungo la primitiva.

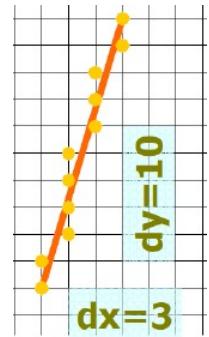
Rasterizzazione di un segmento – algoritmi di rasterizzazione

Immaginiamo di disegnare una segmento. L'algoritmo di rasterizzazione dovrà quindi individuare le coordinate dei pixel che giacciono sul segmento ideale o che sono il più vicino possibile ad essa.

Lo spessore minimo del segmento rasterizzato risulterà di un pixel. Per i coefficienti angolari $|m| \leq 1$, la rasterizzazione presenta un pixel per colonna.



Per i coefficienti angolari $|m| > 1$, allora la rasterizzazione presenta un pixel per riga.



Questo tipo di algoritmo viene chiamato **Algoritmo DDA (Digital Differential Analyzer)**,

$$y = mx + q, \quad |m| \leq 1 \rightarrow \text{il prossimo pixel si accende per } x+1$$

$$y_{\text{new}} = m(x + 1) + q = mx + q + m \equiv y + m$$

$$|m| \geq 1 \rightarrow \text{il prossimo pixel si accende per } y+1$$

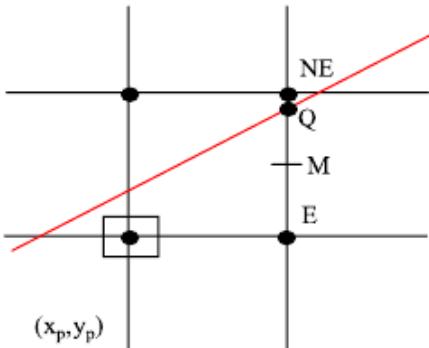
$$x_{\text{new}} = \frac{1}{m}(y + 1 - q) = \frac{1}{m}(y - q) + \frac{1}{m} \equiv x + \frac{1}{m}$$

y_{new} e x_{new} sono le coordinate dei punti sullo schermo a partire dalle coordinate dei punti della primitiva.

Il problema di questo algoritmo è che viene in aritmetica floating point, per cui bisogna arrondire i valori ottenuti ad un valore intero. Inoltre, è un algoritmo incrementale (che ci interessa e vedremo poi perché).

Algoritmo di Bresenham o Mid-point

Algoritmo di Bresenham risolve il problema dell'errore introdotto con l'algoritmo DDA dovuto all'aritmetica floating point, facendo uso solo di operazioni dell'aritmetica intera. Anch'esso è un **algoritmo differenziale**: ovvero fa uso delle informazioni calcolate per individuare il pixel al passo i per individuare il pixel al passo $i+1$.



La linea rossa rappresenta la linea dei pixel. Supponiamo di accendere il pixel (x_p, y_p) . Allora potremo accendere solo o il pixel NE oppure il pixel E. L'idea dell'algoritmo di Bresenham ci spiega proprio quale scegliere fra questi due pixel.

Riprendendo quanto detto nel riquadro rosso, l'algoritmo di Bresenham procede in questo modo: Calcoliamo il punto medio M fra i punti NE ed E. Se la retta che vado a disegnare si trova sotto il punto medio, vuol dire che devo accendere NE. Altrimenti, se il punto della retta che devo andare a disegnare si trova sotto M, accendo E. Si tratta quindi di vedere esclusivamente se il punto della retta si trova sopra o sotto il punto E.

[Metanota]: questa lezione è stata una gran rottura. Direi di bannare globalmente in tutto il mondo le lezioni da 3 ore, siccome non fanno altro che spappolarti il cervello stile Davoli. Mi ci sono volute due settimane per recuperare sta lezione. DUE. Normalmente faccio anche due lezioni al giorno, ma per questa, ce ne sono voluti TROPPI.]

Lezione 18/11/2021 – Lezione persa! T_T

Questa lezione è stata un po' persa... fortunatamente non era nulla di ché, solo la costruzione di una scena 3D carina. Vabbé... alla fine basta vedere il codice onestamente.

Lezione 22/11/2021 – Continuazione Spline, De Boor, Knot-insertion e Modello di Phong

Ripassino sulle spline (non seguire se si è compresa la parte precedente)

Puoi trovare il resto delle informazioni sulle spline [qui](#).

Ricordiamo che la spline coincide con una funzione $s(x)$, che è composta tante $s_i(x)$, una per sottointervallo. Tra questi polinomi, si richiede che ci sia un raccordo negli estremi dell'intervallo (in cui ciascun $s_i(x)$ è definito) con una regolarità m , che indica che le funzioni devono coincidere per valore per derivata $j = m - m_i - 1$. Se $j = 0$ sono in raccordo solo per valore, se $j = 1$ anche per derivata prima etc.. [tutte cose che abbiamo già visto anche nelle lezioni precedenti].

m rappresenta l'ordine del polinomio (quindi $m-1$ equivale al grado del polinomio) per ciascun intervallo.

Le curve spline non permettono che il valore j sia superiore a $m - 2$, infatti in questo modo la regolarità sarebbe troppo alta e si ricadrebbe nei polinomi.

Un'altra cosa che ci interessa calcolare è lo spazio dello spline. Noi sappiamo che lo **spazio delle spline polinomiali** di ordine m a nodi multipli con nodi x_1, \dots, x_k di molteplicità $M = (m_1, \dots, m_k)$, che verrà indicato con $S_m(\Delta, M)$, ha dimensioni pari a $m + K$, dove $K = \sum_{i=1}^k m_i$, ovvero la somma delle molteplicità dei nostri nodi.

Dunque, con $S_m(\Delta, M)$ indichiamo lo spazio delle spline di ordine m , dove M è il vettore delle molteplicità associate a ciascun nodo e Δ è la partizione di $[a, b]$ divisa in intervalli.

Siccome ho $k+1$ intervalli, e ad ogni intervallo dovrò associare un polinomio, avrò $k+1$ polinomi $s_i(x)$.

Questi polinomi, inoltre, non sono liberi (ovvero, “ognuno per conto suo”), siccome dobbiamo porre delle condizioni di regolarità nei k nodi interni. Andiamo ad imporre $m - m_i - 1$ condizioni di regolarità. I coefficienti del polinomi saranno quindi determinati da queste condizioni di regolarità che devo imporre.

Abbiamo anche visto come (e perché) lo spazio delle spline polinomiali di ordine m a nodi multipli ha dimensione pari a $m + K$.

Le funzioni base di una curva sono le **funzioni di Cox**, che scriviamo come $N_{i,m}(t)$ (dove il primo indice indica il nodo dell'intervallo $[a,b]$ a cui essa è riferita, mentre m specifica l'ordine del polinomio), e la cosa interessante di queste funzioni hanno delle proprietà che ci permettono di superare dei limiti che hanno invece le curve di Bezier, ovvero il fatto che sono $\neq 0$ solo in alcuni sottointervalli definiti dell'intervallo.

Una proprietà figa delle formule di Cox, è il fatto che possiamo calcolare la base di ordine m attraverso una combinazione lineare di una base di ordine $m-1$ (ad esempio una spline di ordine 3 si può ottenere due spline di ordine 2 con gli opportuni coefficienti).

Abbiamo poi detto che con k nodi veri, dobbiamo essere in grado di costruire $m + K$ funzioni base. Supponiamo di avere una spline di ordine $m=4$, e siano i nostri nodi x_1, x_2, \dots, x_6 ognuno due quali ha molteplicità 1. Dovremo allora essere in grado $m + K$ funzioni base, ovvero $4 + 6 = 10$ funzioni base (con solo 6 nodi!).

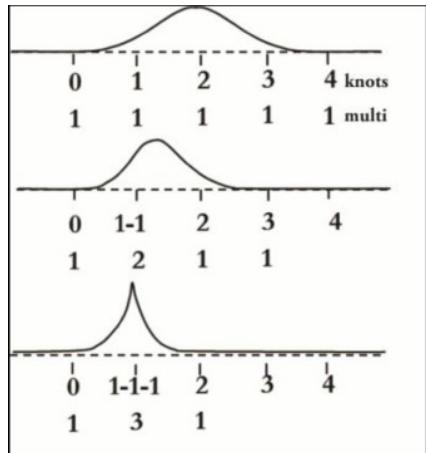
La prima funzione base va da x_1 a x_5 , siccome è definita in 4 sotto-intervalli (siccome $m=4$), la seconda curva va da x_2 a x_6 , ma dalla terza in poi abbiamo dei problemi, siccome la nostra curva va oltre l'intervallo. Dunque, quello che facciamo è costruire dei nodi immaginari che vanno prima e

dopo gli estremi dell'intervallo.

Se i nodi hanno molteplicità > 1, allora dovrà aggiungere tanti nodi quanto è la sua molteplicità per ciascun nodo.

Effetto della molteplicità sui nodi spline

Possiamo vedere un esempio grafico degli effetti della molteplicità sulla curva:



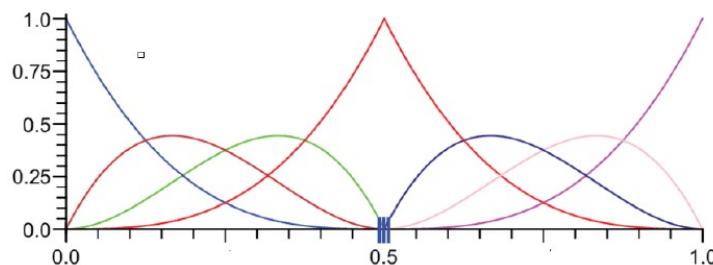
Più la molteplicità aumenta, meno la curva diventa regolare. La curva collassa sul nodo di molteplicità massima.

Come possiamo vedere qua sotto, se la molteplicità è pari all'ordine della spline, si ha addirittura una discontinuità.



Esempio di una spline curiosa

Abbiamo una spline definita solo da un nodo (quindi $k=1$) di ordine $m=4$, con molteplicità $M=(3)$. Il numero totale di nodi fittizi è $2m + K$. Il numero di **funzioni base necessarie**, invece, sarà $m+K=7$.



Proprietà delle funzioni di Cox e di b-spline

1. Le B-spline formano una base per lo spazio $S_m(\Delta, M)$: ciascuna $s(x) \in S_m(\Delta, M)$ può essere espressa come $s(x) = \sum_{i=1}^{m+K} c_i N_{i,m}(x)$.
2. Hanno un **supporto compatto**, ovvero $N_{i,m}(x) = 0$ per $x < x_i$ e $x > x_{i+m}$. Non sono definite quindi oltre il loro intervallo. Ciò vuol dire che possiamo modificare un punto senza modificare l'intera curva. Inoltre, sono **non negative** nel loro supporto:
 $N_{i,m}(x) \geq 0 \quad x \in [x_i, x_{i+m}]$
3. Costruiscono una **partizione dell'unità**, cioè $\sum_{i=1}^{m+K} N_{i,m}(x) = 1$.

Riguardo alla 2 e 3, consideriamo questa proprietà che è presente anche nelle curve di Bezier: sappiamo che al funzione di una curva spline si esprime come una combinazione lineare di $m+K$ funzione base, ovvero $s(x) = \sum_{i=1}^{m+K} c_i N_{i,m}(x)$. Tutte queste funzioni base hanno valore che è 0 o > 0, ma minore di 1. Quindi, vale:

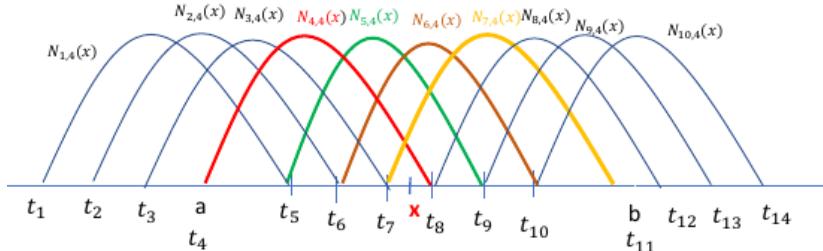
$$\min_{i=1, \dots, m+K} \{c_i\} \leq s(x) \leq \max_{i=1, \dots, m+K} \{c_i\}$$

La proprietà più importante, però, è quella del **controllo locale**. Per la proprietà di supporto

compatto, la valutazione del polinomio in un punto $x \in [t_i, t_{i+1})$ si riduce a:

$$s(x) = \sum_{i=l-m+1}^l c_i N_{i,m}(x)$$

siccome le uniche B-spline diverse da 0 sul punto x sono $N_{l-m+1,m}(x), \dots, N_{l,m}(x)$,



Quando valuto x , le funzioni colorate sono le uniche che devo considerare, le altre hanno valore pari a 0 su x . Quindi posso anche non calcolarle.

Possiamo trovare una spiegazione più approfondita di questa proprietà delle spline nelle lezioni dopo [più precisamente, [qui](#)]

Anche per le spline, vale la proprietà di **variation diminishing** che abbiamo visto per le curve di Bezier.

Valutazione di una funzione spline

Abbiamo visto che, data la partizione nodale estesa Δ^* , una $s(x) \in S_m(\Delta^*, M)$ può esprimersi o con la formula “semplice” generale, che valuta ogni funzione base per tutto l’intervallo ovvero $s(x) = \sum_{i=1}^{m+K} c_i N_{i,m}(x)$, oppure possiamo usare la formula che valuta la funzione solo nel punto in cui è diversa da 0:

$$s(x) = \sum_{i=l-m+1}^l c_i N_{i,m}(x) \quad \text{Con } x \in [t_i, t_{i+1})$$

Noi ovviamente useremo la seconda, siccome è quella più ottimizzata.

Algoritmo di De Boor per valutare una spline

L’**algoritmo di De Boor** è molto simile a quello di [De Casteljau](#), per valutare una spline in un punto. Anche questo, infatti, non si basa sul costruire le funzioni base, ma parto dalla definizione di spline e ad essa sostituisco la funzione in termini di due funzioni base di ordine inferiore.

Sappiamo infatti che:

$$N_{i,h}(x) = \left(\frac{x - t_i}{t_{i+h-1} - t_i} N_{i,h-1}(x) + \frac{t_{i+h} - x}{t_{i+h} - t_{i+1}} N_{i+1,h-1}(x) \right)$$

Dunque, dalla formula iniziale vista nel paragrafo prima, sostituiamo:

$$s(x) = \sum_{i=l-m+1}^l c_i \left(\frac{x - t_i}{t_{i+m-1} - t_i} N_{i,m-1}(x) + \frac{t_{i+m} - x}{t_{i+m} - t_{i+1}} N_{i+1,m-1}(x) \right) =$$

$$= \sum_{i=l-m+1}^l c_i \left(\frac{x - t_i}{t_{i+m-1} - t_i} N_{i,m-1}(x) \right) + \sum_{i=l-m+1}^l c_i \left(\frac{t_{i+m} - x}{t_{i+m} - t_{i+1}} N_{i+1,m-1}(x) \right) =$$

A questo punto estendiamo la prima sommatoria facendola partire da $i = l - m + 2$, siccome le funzioni base di ordine $m-1$ diverse da 0 su di esso sono quelle che hanno indice che va da $l-m+2$ fino ad l . Aggiustiamo poi anche la seconda sommatoria, da $i=l-m+2$ fino ad $l+1$.

$$= \sum_{i=l-m+2}^l c_i \left(\frac{x - t_i}{t_{i+m-1} - t_i} N_{i,m-1}(x) \right) + \sum_{i=l-m+2}^{l+1} c_{i-1} \left(\frac{t_{i+m-1} - x}{t_{i+m-1} - t_i} N_{i,m-1}(x) \right) =$$

Tuttavia, siccome $N_{l+1,m-1}(x)$ è nulla su $x \in [t_i, t_{i+1})$, possiamo scrivere la seconda sommatoria così:

$$= \sum_{i=l-m+2}^l c_i \left(\frac{x - t_i}{t_{i+m-1} - t_i} N_{i,m-1}(x) \right) + \sum_{i=l-m+2}^l c_{i-1} \left(\frac{t_{i+m-1} - x}{t_{i+m-1} - t_i} N_{i,m-1}(x) \right) =$$

A questo punto raccolgo:

$$= \sum_{i=l-m+2}^l \left(\frac{c_i(x-t_i) + c_{i-1}(t_{i-1+m}-x)}{t_{i-1+m}-t_i} \right) N_{i,m-1}(x) =$$

E pongo $c_i^{[1]} = \left(\frac{c_i^{[0]}(x-t_i) + c_{i-1}^{[0]}(t_{i+m-1}-x)}{t_{i+m-1}-t_i} \right)$. Da questo ottengo:

$$\sum_{i=l-m+2}^l c_i^{[1]} N_{i,m-1}(x).$$

Alla quale, come sappiamo dall'algoritmo di De Casteljau, possiamo riapplicare il procedimento. Al passo j -esimo avremo così:

$$s(x) = \sum_{i=l-m+j+1}^l c_i^{[j]} N_{i,m-j}(x). \quad \text{dove } c_i^{[j]} = \left(\frac{c_i^{[j-1]}(x-t_i) + c_{i-1}^{[j-1]}(t_{i+m-j}-x)}{t_{i+m-j}-t_i} \right).$$

e al passo $j = m-1$, avremo finalmente il valore della spline calcolato in x , che sarà l'ultimo coefficiente dello schema ad albero.

L'albero ottenuto da questo algoritmo sarà:

$$\begin{array}{c} c_{l-m+1}^{[0]} \ c_{l-m+2}^{[0]} \dots \ c_l^{[0]} \\ \hline c_{l-m+2}^{[1]} \dots \ c_l^{[1]} \\ \dots \quad \dots \\ c_l^{[m-1]} \end{array}$$

La complessità computazionale è di $3m(m-1)$ moltiplicazioni e divisioni e di $2m(m-1)$ addizioni e sottrazioni.

Come nell'algoritmo di de Casteljau, riusciamo a valutare la curva in un punto mediante combinazione dei coefficienti.

Al contrario dell'algoritmo di De Casteljau, però, nel quale si perdeva il coefficiente a destra, noi ad ogni passo perdiamo il coefficiente a sinistra.

Nell'algoritmo di de Boor, io lavoro solo su **m coefficienti**, in particolare quelli che vanno da $l-m+1$ fino ad 1 (al contrario di De Casteljau, in cui lavoravamo su TUTTI i coefficienti).

Dunque, l'algoritmo ha una complessità $O(m^2)$. Il nostro algoritmo esegue **$m-1$ passi**.

Caso speciale (per spiegare knot-insertion)

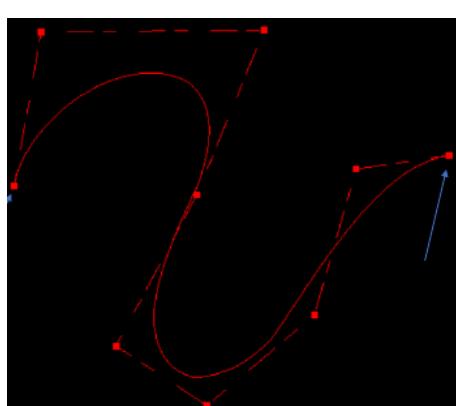
Abbiamo $k=5$ nodi semplici (quindi $M=(1,1,1,1,1)$) e ordine $m=4$, la dimensione dello spazio sarà quindi $4+5=9$, dunque avremo bisogno di 9 funzioni base.

Dati i nostri vertici di controllo P_i , la nostra curva spline sarà:

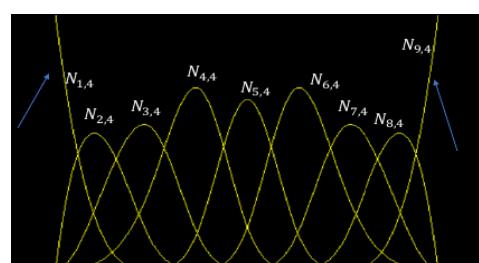
$$C(t) = \sum_{i=1}^N P_i N_{i,m}(t)$$

Dobbiamo costruire una partizione nodale estesa in maniera tale che la dimensione dello spazio sia pari al numero dei vertici di controllo, altrimenti i torni non contano (P_i , $i = 1, \dots, m+K$).

Nel nostro esempio, quindi, il numero dei vertici di controllo dovrà essere 9.



Essenzialmente dunque la nostra curva interpola comunque i due punti agli estremi (ovvero ci passa), facendo in modo che i nodi fintizi coincidano con gli estremi dell'intervallo. Così, abbiamo una sola curva delle tante che compongono la spline che ha valori > 0 per gli estremi.



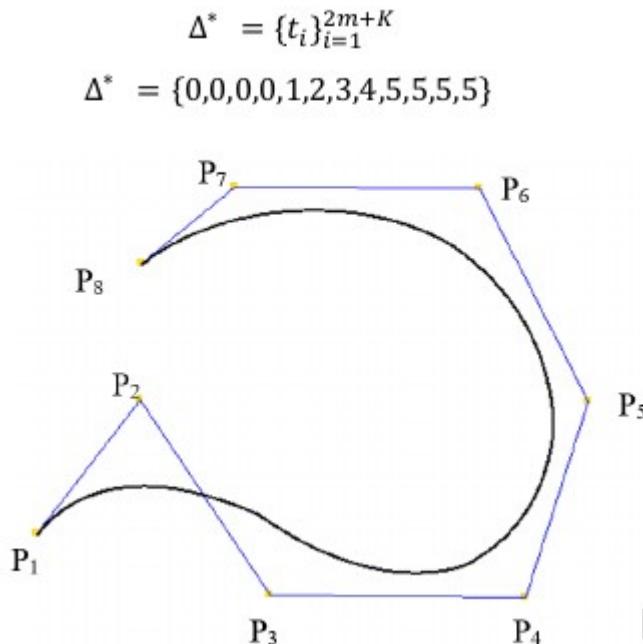
La spline non sempre interpola, infatti in Blender ci saranno dei casi in cui dovremo imporre delle condizioni. Queste condizioni sono quelle di imporre tutti i nodi fittizzi a destra e a sinistra coincidenti con gli estremi dell'intervallo, e solo in questo modo ci sarà interpolazione con gli estremi, altrimenti semplicemente la nostra curva non passerà per gli estremi.

Algoritmo Knot Insertion

L'**algoritmo di Knot Insertion** è una generalizzazione dell'algoritmo di degree elevation che abbiamo già visto per le curve di Bezier. Per capire bene l'algoritmo, è bene tenere a mente che i nodi dell'intervallo NON coincidono con i vertici di controllo.

In questo caso, al posto di aumentare il grado del polinomio, si vuole inserire un nodo in più in uno degli intervalli della curva, in modo tale che abbia lo stesso rappresentazione ma con un nodo in più.

Dunque, la nostra partizione nodale estesa avrà un nodo in più, e vogliamo scoprire quali sono i coefficienti di rappresentazione della nostra spline in questo spazio che ha un nodo in più. Facciamo un esempio per capire meglio.



La nostra curva ha parametri $m=4$, $K=4$, $N=8$ (dove N è il numero dei punti) e $t' = 5/2$. Abbiamo 4 nodi veri, e i primi 4 nodi fittizzi coincidono con lo 0, mentre i 4 nodi a destra coincidono con l'estremo a sinistra (l'intervallo in questo caso è $[0, 5]$). Il fatto che i nodi fittizzi a dx e sx coincidano con l'inizio e la fine dell'intervallo mi permette di interpolare i punti a dx e sx.

Con t' indichiamo il nuovo nodo che andiamo ad inserire, ovvero $t'=5/2$, che si troverà fra il nodo 2 e il nodo 3.

Dalla formula che abbiamo visto in precedenza, in pratica vogliamo esprimere questa curva partendo da una partizione nodale di partenza, e ci chiediamo come variano i coefficienti di rappresentazione della spline, che appunto deve mantenere lo stesso grafico.

Andando ad inserire un nodo in più, la partizione nodale da Δ^* diventerà $\hat{\Delta}^* = \{\hat{t}_i\}_{i=1}^{2m+K+1}$, e non cambierà in sé la struttura della curva, bensì cambierà il suo poligono di controllo. Andiamo a vedere come cambiano i suoi vertici di controllo.

Possiamo notare dalla formula subito qui sotto che la modifica è solo locale, infatti il poligono cambierà solo parzialmente.

$$\hat{P}_i = \begin{cases} P_i & i \leq l - m + 1 \\ \lambda_i P_i + (1 - \lambda_i) P_{i-1} & l - m + 2 \leq i \leq l \\ P_{i-1} & i \geq l + 1 \end{cases} \quad \text{con} \quad \lambda_i = \frac{t - t_i}{\hat{t}_{i+m} - \hat{t}_i}$$

Tornando all'esempio di prima, il risultato che otteremo sarà:

Notiamo come gli unici vertici che sono cambiati sono i vertici che vanno da $l - m + 2$ a l (dove $[l, l+1]$ è l'intervallo in cui sono andati ad aggiungere un nodo). I lambda, come possiamo notare, dipendono solo dall'ampiezza della partizione nodale e dal nodo che vogliamo inserire, e non dal sotto intervallo $[tl, tl+1]$

$$\hat{P}_1 = P_1, \quad \hat{P}_2 = P_2, \quad \hat{P}_3 = P_3$$

$$\hat{P}_7 = P_6, \quad \hat{P}_8 = P_7, \quad \hat{P}_9 = P_8$$

$$\hat{P}_4 = \lambda_4 \cdot P_4 + (1 - \lambda_4) \cdot P_3$$

$$\lambda_4 = \frac{\hat{t} - \hat{t}_4}{\hat{t}_8 - \hat{t}_4} = \frac{5}{6}$$

$$(1 - \lambda_4) = \frac{1}{6}$$

$$\hat{P}_5 = \lambda_5 \cdot P_5 + (1 - \lambda_5) \cdot P_4$$

$$\lambda_5 = \frac{\hat{t} - \hat{t}_4}{\hat{t}_9 - \hat{t}_5} = \frac{1}{2}$$

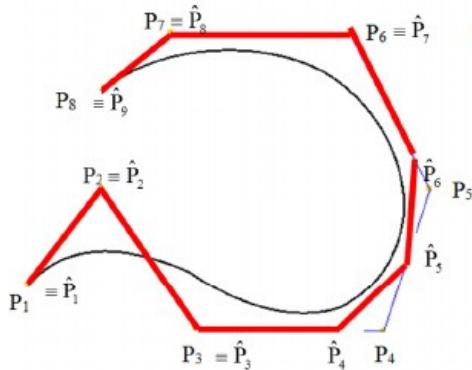
$$(1 - \lambda_5) = \frac{1}{2}$$

$$\hat{P}_6 = \lambda_6 \cdot P_6 + (1 - \lambda_6) \cdot P_5$$

$$\lambda_6 = \frac{\hat{t} - \hat{t}_6}{\hat{t}_{10} - \hat{t}_6} = \frac{1}{6}$$

$$(1 - \lambda_6) = \frac{5}{6}$$

Il poligono della nostra curva quindi cambierà in questo modo:



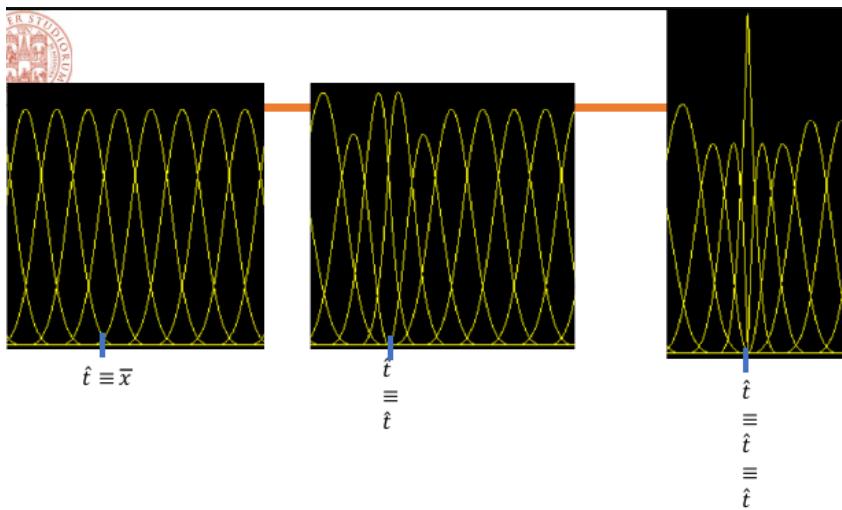
Notiamo come si è aggiunto anche un vertice di controllo in questo modo. In particolare, gli unici vertici che abbiamo modificato sono quelli in prossimità di dove è stato aggiunto il nuovo vertice di controllo. Dunque, cambiano in dipendenza dell' \hat{t} , ovvero dell'estremo sinistro dell'intervallo in cui ho introdotto il vertice di controllo.

Inoltre, notiamo come i vertici che cambiano sono un'interpolazione lineare dei vertici precedenti.
I vertici che sono stati modificati/aggiunti appartengono ad una combinazione lineare tra due coefficienti del poligono di controllo di partenza.

Anche qui, come nelle curve di Bezier, più nodi aggiungo, più il poligono si avvicina alla curva.
Un'altra cosa interessante: se aggiungo un nodo (o più nodi) in corrispondenza di uno stesso nodo (aumentandogli, quindi, la sua molteplicità), **ad un certo punto la curva passerà per quel punto!**

Utilizzo dell'algoritmo di Knot-insertion per la valutazione di una spline in un punto

Vediamo perché vale ciò che abbiamo detto subito prima.



Per capire meglio: se mettiamo un nodo nella stessa posizione, avremo non una, ma due funzioni che arrivano (si annullano) e partono da quello stesso nodo.

Se abbiamo 3 nodi, vuol dire che l'unica funzione base che rimane deve valere uno.

Infatti, se un nodo ha molteplicità $m-1$, allora ci sarà una sola funzione base con valore pari a 1 (siccome, come sappiamo, le funzioni base per un certo m sono una partizione dell'unità)

In poche parole, se aggiungo molti nodi sempre nello stesso punto, avremo che in tale punto c'è solo una funzione base, che è moltiplicata per quel punto.

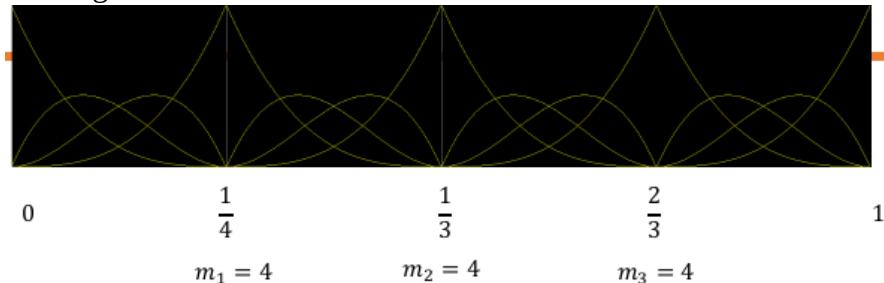
Dunque, possiamo usare l'algoritmo di Knot Insertion per valutare la curva in quel punto. Per fare ciò, basta aggiungere un nodo con molteplicità pari ad $m-1$.

Knot-insertion per spezzare la curva

Possiamo usare sempre lo stesso algoritmo per spezzare la curva, aggiungendo in corrispondenza del valore di t per cui voglio spezzare la curva un nodo con molteplicità m , siccome in questo modo il raccordo sarebbe per $j=-1$.

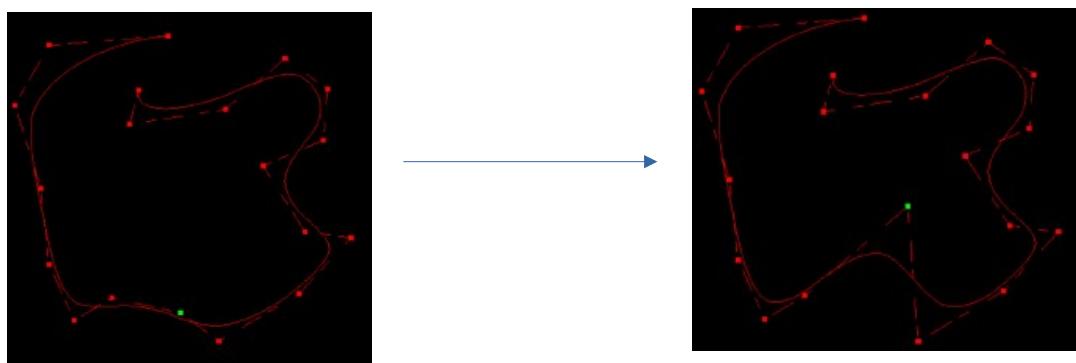
Spezzare una curva spline in delle curve di Bezier

Consideriamo questa situazione, in cui la partizione nodale è costituita da $\Delta = \{0, 1/4, 1/3, 2/3, 1\}$. Se a ciascuno di questi nodi assegno una molteplicità pari a 4 (quindi da ciascun nodo devono partire 4 funzioni base), possiamo notare come in ciascuno di questi sottointervalli si siano formati delle basi di Bezier di grado 3.



Dunque, per spezzare la curva, basta inserire un nodo con molteplicità m nell'intervallo in cui vogliamo spezzare la curva [piccolo esempio [qui](#)].

Esempio di controllo locale nella curva spline



Spostando il vertice di controllo (in verde nella figura), la modifica riguarda solo gli intervalli internodali per i quali passa la funzione base che è moltiplicata per il punto che ho spostato.

Inoltre, aumentando la molteplicità in un punto, si può notare come la curva diminuisca la propria classe (per esempio, da classe C^2 diventa classe C^1). In questo modo, la curva diventa anche sempre più tangente al suo poligono.

[**METANOTA:** in tutta onestà non so perché la prof ha fatto vedere questo esempio qui, siccome si ricollega invece alla parte fatta più in avanti. Lol almeno ho messo i link, ringraziami bro].

Modelli di illuminazione e shading

I **modelli di illuminazione** sono delle funzioni che ci descrivono il colore di una superficie in un determinato punto, in funzione di alcuni parametri quali:

- le interazioni tra le luci e le superfici
- le proprietà che possono avere le superfici
- la natura della radiazione luminosa che li colpisce

L'uso di un modello di illuminazione è necessario per ottenere una rappresentazione realistica delle superfici tridimensionali.

I **modelli di shading** determinano come viene applicato il modello di illuminazione e quali argomenti prende in input per determinare il colore di un punto sulla superficie.

Alcuni modelli di shading richiamano il modello di illuminazione per ogni pixel nell'immagine, altri invece richiamano il modello di illuminazione solo per alcuni pixel dell'immagine e colorano i rimanenti pixel per interpolazione.

Come si differenziano lighting e shading

- Il **Lighting** consiste nel processo di calcolo dell'intensità luminosa (cioè della luce in uscita) in un particolare punto 3D, **soltamente su una superficie**, in funzione delle caratteristiche della luce e del materiale. Questa operazione viene fatta in coordinate di vista, prima di applicare la trasformazione di Proiezione che porta nelle coordinate di clip e potrebbe alterare le normali, quindi interessa il vertex shader.
- Lo **Shading** consiste nell'**assegnare il colore al pixel**, (specifica come la luce viene usata per colorare il pixel). Questa operazione viene fatta durante il rasterization stage, e interessa il fragment shader.

Modelli di illuminazione

Distinguiamo fra modelli di illuminazione globale e modelli di illuminazione locale.

I **modelli di illuminazione locale** interessano un punto singolo sulla superficie, e dalla sorgente luminosa che lo illumina direttamente. Il resto della scena è illuminato da una luce ambientale. Questo modello di illuminazione non tiene conto delle riflessioni all'interno dell'ambiente (non si tiene conto dei rimbalzi della nostra luce e delle altre luci quindi).

Nei **modelli di illuminazione globale**, invece, il colore del punto viene calcolato in termini:

- della luce emessa direttamente dalle sorgenti luminose
- della luce che raggiunge il punto dopo la riflessione e la trasmissione della luce dalla propria e dalle altre superfici che si trovano nell'ambiente circostante.

Modello fisico reale (utile per sapere alcuni termini che useremo)

A livello di modello fisico esatto, il modo in cui la radiazione luminosa viene riflessa da una superficie dipende da:

- lunghezza d'onda della radiazione luminosa;
- angolo di incidenza tra superficie e radiazione luminosa;
- natura e microstruttura superficiale del materiale irradiato;
- altre proprietà fisiche del materiale irradiato, quali permeabilità e conduttività.

Sappiamo inoltre che quando la luce colpisce la superficie, una parte è assorbita, e una parte è riflessa. L'**opacità** di una superficie è una misura di quanta luce penetra attraverso la superficie. Un valore di opacità pari a 1 ($\alpha = 1$) corrisponde ad una superficie completamente opaca. Una superficie con opacità 0 ($\alpha = 0$) è trasparente: tutta la luce passa attraverso ad essa.

A seconda della sua opacità, la superficie di un materiale può essere:

- Trasparente: se trasmette la luce e attraverso di essa è possibile osservare un oggetto. Il quarzo e la calcite normalmente sono trasparenti.
- Traslucido: se trasmette la luce diffondendola ma non è trasparente. Sebbene una superficie traslucida permetta la trasmissione della luce, non consentirà l'osservazione nitida di un oggetto osservato attraverso di essa.
- Opaco: se è impenetrabile alla luce visibile, anche sui bordi esterni più sottili. La maggior parte dei minerali metallici è opaca.

Superfici speculari e diffuse

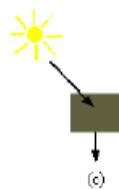
Le superfici si differenziano in **speculari** e **diffuse**.

Le superfici sono speculari se la luce che lo colpisce viene riflessa lungo la direzione di riflessione formando un angolo ristretto di luce. Se un oggetto fosse **perfettamente speculare** (come uno specchio per esempio), la luce sarebbe riflessa in unica direzione di riflessione, altrimenti verrebbe riflessa lungo un cono di direzioni lungo la direzione di riflessione.





(b)



(c)

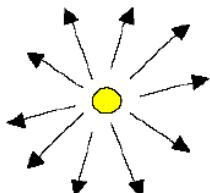
Le **superfici diffuse** sono caratterizzate dal fatto che la luce riflessa è diffusa in tutte le direzioni. In particolare, le superfici perfettamente diffuse emettono luce ugualmente in tutte le direzioni, (terreno, muro dipinto, etc).

Le **superfici traslucide** lasciano penetrare parte della luce, che poi riemerge da un altro punto dell'oggetto. Questo processo, chiamato rifrazione, caratterizza i materiali quali vetro e acqua. Una parte della luce incidente può anche essere riflessa dalla superficie.

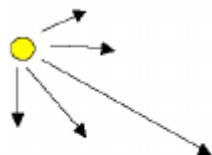
Tipi di sorgenti luminose

Abbiamo vari di tipi di sorgenti luminose, tra cui:

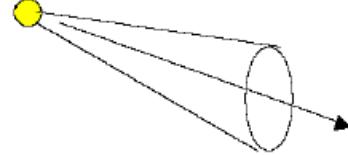
- Una sorgente **point light** ha una posizione nello spazio, ma non ha né un volume né un'area. Emette luce ugualmente in ogni direzione.
- Una sorgente luminosa di tipo direzionale, **directional light**, non ha una posizione nello spazio, può essere pensata come posizionata all'infinito. La direzione della luce emessa è costante.
- Una **spotlight** è una sorgente luminosa puntuale, che emette luce differentemente in ogni differente direzione. È definita mediante una direzione ed un angolo: il volume illuminato forma un cono infinito.
- Un'**area light** è una sorgente puntuale caratterizzata da un'area



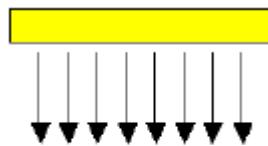
Point light



Directional light



Spot light



Area light

Modello di illuminazione di Phong

Il **modello di illuminazione di Phong** viene usato per determinare il colore di un oggetto in funzione del suo materiale e della luce che lo colpisce direttamente (è quindi un modello di illuminazione locale). L'unico modello fisico modellato a partire dal modello di illuminazione di Phong è la riflessione diretta: le equazioni che vedremo riescono solo a simulare il comportamento di materiali opachi, e non di materiali trasparenti o semitransparenti.

Ci sono tre differenti classi di modelli di illuminazione:

- Luce ambientale
- Riflessione diffusa
- Riflessione speculare

Questi 3 modelli danno luogo al modello di illuminazione di Phong. Qui sotto li illustriemo uno ad uno.

Altri modelli di illuminazione

Il modello di illuminazione più semplice è quello della **luce emissiva**, ma è anche il meno realistico. Si basa sul fatto che ogni oggetto è dotato di una propria intensità luminosa, senza che vi siano fonti esterne di illuminazione.

In termini matematici un modello di illuminazione può essere espresso mediante una equazione di illuminazione, che descrive come ogni punto dell'oggetto sia illuminato in funzione della sua posizione nello spazio.

Per la luce emissiva, il modello di illuminazione ha formula: $I = k_i$, dove I è l'intensità risultante del colore, mentre k_i è la luminosità intrinseca dell'oggetto.

Non essendoci termini dipendenti dalla posizione del punto, si può calcolare una sola volta per tutto l'oggetto. Questo metodo non è per nulla realistico, non dipende dalla posizione del punto sull'oggetto etc.., ma è solo il valore k_i che viene assegnato ad ogni punto dell'oggetto.

Luce ambientale

Con il metodo della luce emissiva, oggetti che non sono illuminati dalla luce appaiono neri. Nella vita reale invece non è così: queste parti sono illuminate dalla illuminazione globale, cioè dalla luce riflessa dall'ambiente circostante. Questa luce viene approssimata da una luce costante detta **luce ambientale**.

Questo tipo di componente viene simulata supponendo che l'oggetto sia illuminato da una sorgente di luce diffusa (e non direzionale) dovuta al riflesso della luce sulle molteplici superfici presenti nell'ambiente.

Se supponiamo che la luce dell'ambiente colpisce ugualmente tutte le superfici da tutte le direzioni, allora l'equazione dell'illuminazione diventa:

$$I = I_a k_a$$

dove I_a è **l'intensità della luce dell'ambiente**, supposta costante per tutti gli oggetti.

La quantità di luce dell'ambiente riflessa dalla superficie dell'oggetto è determinata da k_a , coefficiente di riflessione ambientale, che varia tra 0 ed 1 (se è 1 tutto il contributo della luce ambientale viene riflesso dalla superficie).

Rappresenta una frazione della luce ambientale che viene riflesso dalla superficie.

Il coefficiente di riflessione ambientale è una proprietà che caratterizza il materiale di cui la superficie è fatta.



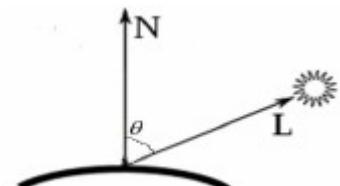
Comunque, con questo metodo, gli oggetti illuminati da sola luce ambientale sono ancora uniformemente illuminati su tutta la loro superficie.



Riflessione diffusa

Supponiamo di posizionare nella scena una sorgente luminosa puntiforme (point light source) i cui raggi sono emessi uniformemente in tutte le direzioni. In questo caso la luminosità di ogni singolo punto dipenderà dalla sua distanza dalla sorgente luminosa e dalla direzione in cui i raggi incidono rispetto alla superficie. La **riflessione diffusa**, detta anche riflessione Lambertiana, **è caratteristica dei materiali opachi**, tipo il gesso.

Queste superfici appaiono ugualmente luminose da qualunque punto di vista vengano osservate: non modificano la loro apparenza al variare del punto di vista poiché riflettono la luce uniformemente in tutte le direzioni. La luminosità dipende solo dall'angolo θ formato dalla direzione del raggio luminoso (L) (nella figura dovrebbe essere al contrario) e la normale alla superficie nel punto di incidenza (N).



Per le superfici lambertiane la quantità di luce vista dall'osservatore è indipendente dalla posizione dell'osservatore, ed è proporzionale al $\cos\theta$, angolo di incidenza della luce.

L'equazione dell'illuminazione diffusa è:

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

I_p è l'**intensità della sorgente luminosa puntiforme**, k_d è il **coefficiente di riflessione diffusiva** ed è una costante che varia tra 0 ed 1 e varia da un materiale all'altro.

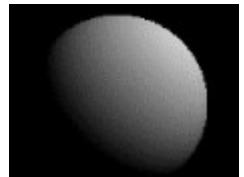
L'angolo θ deve avere un valore compreso fra 0° e 90° per contribuire all'illuminazione del punto, in altre parole un punto della superficie non è illuminato da sorgenti luminose che stanno "dietro" di esso.

Se i vettori N ed L sono stati normalizzati, allora possiamo sostituire il $\cos(\theta)$ con il prodotto scalare fra i due vettori: $I = I_p k_d \bar{N} \cdot \bar{L}$

Se una sorgente di luce puntiforme è sufficientemente distante dagli oggetti, essa forma lo stesso angolo con tutte le superfici che condividono la stessa normale.

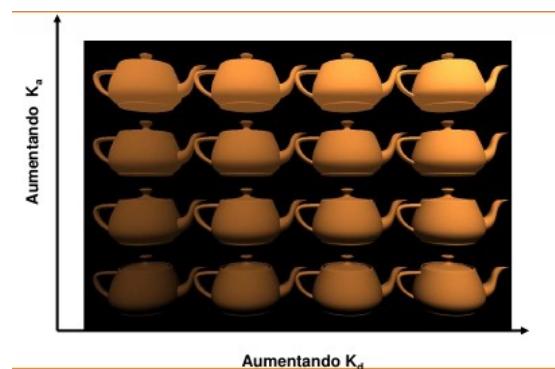
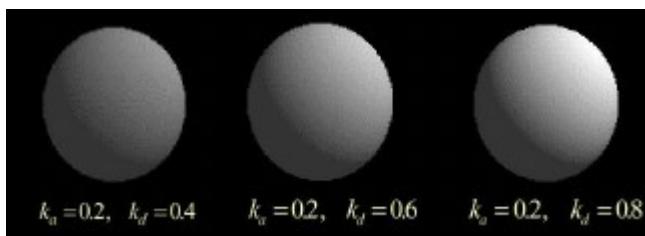
In questo caso la luce è detta **sorgente luminosa direzionale** ed L è costante per la sorgente luminosa. **Con la riflessione diffusa, la parte della superficie non illuminata dalla luce è nera.** È come se l'oggetto fosse illuminato da una torcia.

Aggiungendo il termine dell'illuminazione ambientale all'equazione dell'illuminazione della riflessione diffusa abbiamo:



Oggetto che usa solo la riflessione diffusa, senza quella globale.

$$I = I_a k_a + I_p k_d (\bar{N} \cdot \bar{L})$$



Possiamo anche introdurre nella formula un **fattore di attenuazione della sorgente luminosa**, che tiene conto dell'attenuazione dell'intensità dell'illuminazione all'aumentare della distanza. Questo fattore, che indichiamo con f_{att} , è inversamente proporzionale alla distanza della sorgente di luce dalla superficie (più si è lontani, meno la luce sarà intensa).

$$I = I_a k_a + f_{att} I_p k_d (\bar{N} \cdot \bar{L}) \quad \text{dove} \quad f_{att} = 1/(d_L)^2$$

Per come abbiamo scelto f_{att} , se la luce è molto lontana f_{att} non varia molto, mentre se è molto vicina varia vistosamente, assegnando colori considerevolmente differenti a superfici caratterizzate da uno stesso angolo tra N ed L . Inoltre, in questo modo, quando la sorgente è troppo vicina si "brucia la scena" e quindi non si riesce più a vedere gli oggetti per bene.

Sebbene questo comportamento sia corretto per una sorgente luminosa puntiforme, nella realtà gli oggetti non sono illuminati da una sorgente puntiforme, dunque normalmente si usa questa formula per calcolare f_{att}

$$f_{att} = \min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right)$$

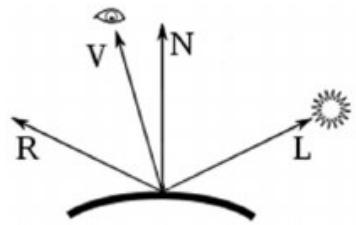
In questo modo, quando distanza è 0, non va all'infinito, bensì ad 1.

Riflessione speculare

Se la superficie di un oggetto non è completamente opaca, la luce non viene riflessa ugualmente in

tutte le direzioni. Data una superficie totalmente lucida, come uno specchio, la luce viene riflessa nella direzione di riflessione R che, geometricamente, non è altro che L (direzione di incidenza della luce) riflessa rispetto ad N (normale alla superficie).

L'osservatore quindi potrà vedere la riflessione R solo se la direzione di vista è allineata con la riflessione, quindi solo se l'angolo formato dalla direzione di vista e la **direzione di riflessione** α è 0.

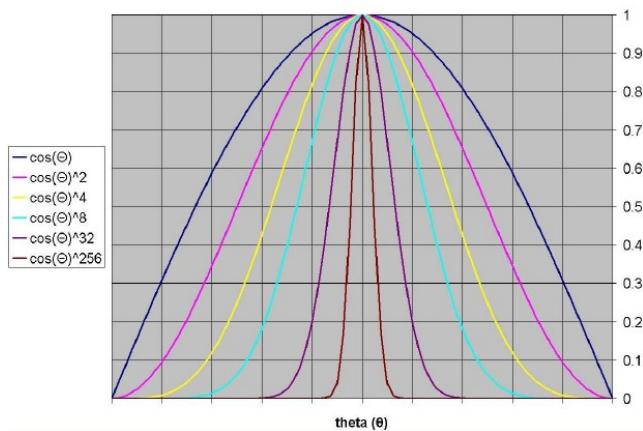


Illuminazione per riflettori imperfetti

Phong ha anche sviluppato un modello di illuminazione per riflettori non perfetti, come per esempio un oggetto di plastica o di cera. Il modello assume che si abbia riflessione massima per $\alpha=0$ e che essa decada rapidamente all'aumentare di α . Questo decadimento viene approssimato dalla formula $\cos^n \alpha$, dove n viene detto **coefficiente di riflessione speculare del materiale**.

Il valore di n può variare tra 1 e valori molto alti (anche superiori al 100), secondo il tipo di materiale che si vuole simulare.

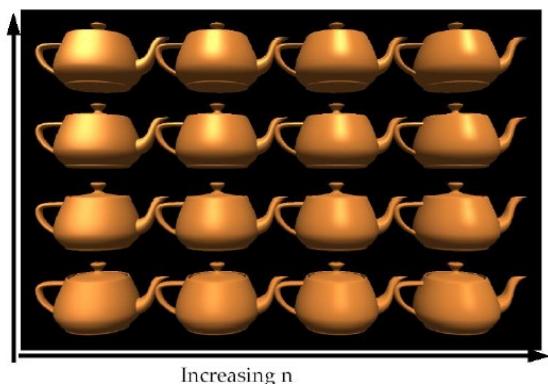
Una superficie a specchio sarebbe teoricamente rappresentata da $n=\infty$.



Notiamo come all'aumentare dell'esponente, la nostra funzione tende ad un impulso. θ in questo caso corrisponde al nostro α .

Aumentando l'esponente del coseno si possono ottenere delle macchie (la macchia bianca di riflessione attorno ad una mela per esempio) attorno alla direzione di riflessione: per esponenti più alti la macchia si riduce ad una riduce ad un impulso, per esponenti più piccoli invece la macchia aumenta di dimensione.

Ridimensionando la macchia, si ridimensiona il cono di direzione lungo cui viene riflessa la luce.



All'aumentare di n la luce riflessa si concentra in una regione sempre più stretta centrata sull'angolo di riflessione. Al limite, quando n tende all'infinito, il comportamento simulato è esattamente quello di uno specchio.

I valori di n compresi nell'intervallo [100, 500] corrispondono approssimativamente alle superfici metalliche. I valori inferiori a 100 corrispondono ai materiali che mostrano una ampia zona di massima lucentezza.

Modello di Phong: equazione finale

$$I = I_a k_a + f_{att} I_p (k_d \cos \theta + k_s (\cos \alpha)^n)$$

oppure

$$I = I_a k_a + I_p k_d (\bar{N} \bullet \bar{L}) + I_p k_s (\bar{R} \bullet \bar{V})^n$$

dove k_s coefficiente di riflessione speculare, che varia tra 0 ed 1 e dipende dal particolare materiale.

FINE

Questa lezione è stata pure più lunga dell'altra, ma almeno l'ho finita in poco tempo.

Lezione 25/11/2021 – Esercitazione sull’illuminazione!

Lezione 29/11/2021 – Continuazione curve spline, Bling-Phong, Eliminazione delle superfici non visibili

Continuazione spline (ripassino)

Abbiamo visto come le spline possano essere calcolate attraverso l’algoritmo di De Boor, attraverso cui valutiamo la curva andando avanti finché il nostro polinomio non assume un valore pari ad una funzione base di ordine 1 (ovvero una costante). Il coefficiente che andiamo a eliminare, al contrario di eliminarlo da destra come per de Casteljau, eliminiamo il coefficiente da sinistra.

Abbiamo poi l’algoritmo di Knot-Insertion, che è la controparte dell’algoritmo di degree-elevation che possiamo trovare nelle curve di Bezier. Con esso andiamo ad aggiungere un nodo (che è diverso da un punto di controllo) all’interno della curva, mantenendo però sempre lo stesso grafico della curva. Quello che abbiamo visto è che con questo algoritmo aumenta il numero di coefficienti e di funzioni base nella sua formula di rappresentazione, assieme anche al numero dei vertici di controllo.

Si dimostra poi che alcuni dei coefficienti della nuova partizione nodale (con il nodo in più) si ottengono da quelli precedenti, però quelli fra $l-m+2$ e l (dove l è l’estremo sinistro dell’intervallo in cui andiamo ad inserire questo nuovo nodo, left) si ottengono attraverso una interpolazione lineare fra i questi coefficienti.

Possiamo usare prima l’algoritmo di knot-insertion per valutare la spline in un punto, oppure per suddividere la spline in due curve separate. Abbiamo anche visto come, se prendiamo i nodi fittizi a sinistra e a destra come coincidenti all’inizio e alla fine dell’intervallo c’è interpolazione, altrimenti no.

Continuazione con esempio di molteplicità estesa

Data una curva spline, si vuole spezzare la curva in 4 curve spline. Per fare ciò, diamo a molteplicità 4 a ciascuno dei nodi interni.

$$\Delta^* = \left\{ 0, 0, 0, 0, \frac{1}{4}, \frac{1}{3}, \frac{2}{3}, 1, 1, 1, 1 \right\} \quad \text{→} \quad \widehat{\Delta}^* = \left\{ 0, 0, 0, 0, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, \frac{2}{3}, \frac{2}{3}, \frac{2}{3}, \frac{2}{3}, 1, 1, 1, 1 \right\}$$

In questo modo, si spezza la curva in 4 curve spline. Non cambia però la curva in sé, ma solo la poligonale.

Derivata di una curva spline (poco importante)

La derivata prima della curva B-spline si esprime nella seguente forma:

$$C'(t) = \sum_{i=l-m+1}^l P_i N'_{i,m}(t)$$

dove

$$N'_{i,m}(t) = \frac{m-1}{t_{i+m-1}-t_i} N_{i,m-1}(t) - \frac{m-1}{t_{i+m}-t_{i+1}} N_{i+1,m-1}(t)$$

Proprietà delle curve spline

Se ho una partizione nodale, in generale la nostra spline si esprime come una combinazione lineare dei punti nei vertici di controllo per le funzioni base, ovvero:

$$C(t) = \sum_{i=1}^N P_i N_{i,m}(t)$$

Ma le funzione base sono tante quanto la dimensione dello spazio delle spline, ovvero $S_m(\Delta, M)$. La dimensione dello spazio è $N = m + K$.

Quindi, affinché io possa fare la combinazione lineare, le funzioni base devono essere tante quante i vertici di controllo. Quindi, **il numero dei vertici di controllo è pari a $N = m + K$.**

Possiamo notare quindi l'ordine m delle nostre funzioni base è fisso, anche se andiamo ad aumentare il numero dei vertici di controllo (contrariamente alle curve di Bezier, in cui invece l'ordine era strettamente legato al numero dei vertici di controllo).

Altre proprietà: controllo locale

Abbiamo visto come se spostiamo un vertice di controllo, la nostra spline non sarà influenzata su tutto l'intervallo della curva, bensì solo in un numero limitato di parte su cui vivono le funzioni base che sono legate a quel vertice di controllo.

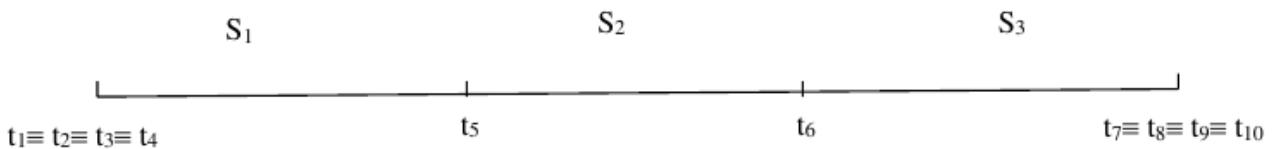
Prima di spiegare in modo più dettagliato questo concetto, dobbiamo introdurre il concetto di **spans**. **Le spans sono una parte di curva che corrispondono a ciascun intervallo internodale.** Infatti, poiché si ha che per $\bar{t} \in [t_i, t_{i+1})$

$$C(t) = \sum_{i=l-m+1}^l P_i N_{i,m}(t)$$

solo i punti di controllo P_i (con $i = l - m + 1, \dots, l$) influenzano la curva nel valore del parametro \bar{t} .

Facciamo un esempio per capire meglio.

Consideriamo una spline con $m = 4$, $k = 2$, $K = 2$ (quindi ognuno dei nodi ha molteplicità 1) e con formula $C(t) = \sum_{i=1}^6 P_i N_{i,4}(t)$. I vertici di controllo, come abbiamo detto prima, devono essere $m+K$, quindi saranno 6.



Indichiamo con S_1 la span relativa all'intervallo internodale $[t_4, t_5]$, S_2 la span relativa all'intervallo internodale $[t_5, t_6]$, S_3 la span relativa all'intervallo internodale $[t_6, t_7]$.

Vediamo cosa succede alla valutazione della nostra curva:

$$\text{Se } t \in [t_4, t_5] \quad C(t) = P_1 N_{1,4}(t) + P_2 N_{2,4}(t) + P_3 N_{3,4}(t) + P_4 N_{4,4}(t)$$

$$\text{Se } t \in [t_5, t_6] \quad C(t) = P_2 N_{2,4}(t) + P_3 N_{3,4}(t) + P_4 N_{4,4}(t) + P_5 N_{5,4}(t)$$

$$\text{Se } t \in [t_6, t_7] \quad C(t) = P_3 N_{3,4}(t) + P_4 N_{4,4}(t) + P_5 N_{5,4}(t) + P_6 N_{6,4}(t)$$

Con ciò che abbiamo appena scritto con queste formule, intendiamo, per esempio, che se voglio valutare la mia spline nella span S_1 , per ogni valore di t che varia in questo intervallo io noto che le uniche funzioni base che vanno a determinare il valore della spline sono quelle che vanno dall'indice 1 a 4.

La regola che dobbiamo applicare per determinare ciò è questa:

Se $x \in [t_i, t_{i+1})$ le uniche funzioni base che sono $\neq 0$ su questo punto sono quelle che vanno da $N_{i-m+1,m}(x), \dots, N_{i,m}(x)$ [come abbiamo visto [qui](#)].

Dunque, se localizzo x in una certa span (quindi in un certo intervallo internodale), allora la spline in quel punto si ottiene combinando soltanto le funzioni base nel modo che abbiamo detto prima. Facendo riferimento all'esempio di prima, possiamo concludere che la span S_1 dipende solo dai

punti di controllo P_1, \dots, P_4 e quindi se io sposto uno di questi vertici di controllo viene modificato il valore della spline solamente in quello span.

Un esempio visivo di questa proprietà [qui](#).

[curiosità] In Blender non possiamo scegliere la molteplicità dei nodi. RIP. [fine curiosità].

Curve di Bezier come caso speciale delle curve spline (interessante ma poco importante)

Possiamo vedere le curve di Bezier come un caso speciale delle curve spline. Supponiamo di avere $m=4$ e di non avere nessun nodo interno (quindi $k = 0$ e conseguentemente $K = 0$), quindi avremo solo i nodi fittizi a sinistra e solo i nodi fittizi a destra (quindi avremo 4 curve che partono dall'estremo a sinistra e vanno all'estremo a destra).

Queste 4 curve corrisponderanno ai polinomi di base di Bernstein di grado 3.

Possiamo quindi dire che le curve di Bezier di grado $m-1$ possono essere considerate come un caso speciale delle curve spline di ordine m con 0 nodi interni.

Invarianza per le trasformazioni affini

La curva B-spline è invariante per trasformazioni affini. Ciò implica che se bisogna traslare, ruotare o deformare una curva, **è sufficiente applicare la trasformazione solo ai punti di controllo** e poi costruire la curva sui punti trasformati. La curva quindi subirà la trasformazione desiderata come se la trasformazione fosse applicata a tutti i punti della curva.

Proprietà del guscio convesso (strong convex hull)

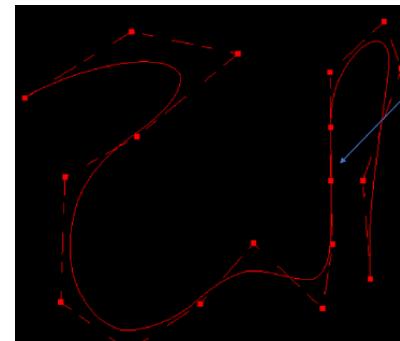
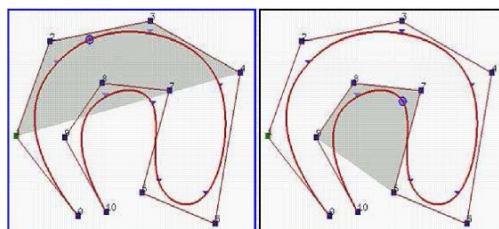
Questa proprietà è interessante, siccome si ricollega alle curve NURBS [vedremo perché qui] e le curve spline su Blender.

Sappiamo che la curva spline si esprime nella pratica si esprime così:

$$C(t) = \sum_{i=l-m+1}^l P_i N_{i,m}(t) \quad \text{Con } t \in [t_l, t_{l+1})$$

La proprietà ci dice che la curva è contenuta nell'inviluppo convesso dei punti di controllo

P_i , con $i = l-m+1, \dots, l$.



Questa proprietà implica che, se questi m punti di controllo sono allineati, allora curva è “costretta” a coincidere con la retta che passa per questi punti, siccome l’inviluppo convesso degenera in una retta.

Inoltre, se inseriamo $m-1$ (quindi 3 se prendiamo la nostra classica curva con $m=4$) vertici di controllo coincidenti, allora la curva passa per il punto multiplo, siccome l’inviluppo convesso degenera in un punto.

Vantaggi delle curve spline

- Permetto il controllo locale
- Il numero dei vertici di controllo non è strettamente legato all’ordine
- Una curva di Bezier può essere considerata come un caso particolare della curva spline
- La molteplicità ci permette di darci un grado di flessibilità in più rispetto alle curve di Bezier
- Grazie al convex hull, hanno un maggiore controllo della formalizzazione

Svantaggi delle curve spline

- Essendo curve polinomiali (come tutte quelle che abbiamo visto fin'ora), va notato come queste non permettano di rappresentare molte curve semplici come cerchi ed elissi.
- Inoltre, NON godono dell'invarianza per trasformazioni proiettive. Per esempio, se dobbiamo fare una trasformazione da 2D a 3D, non possiamo applicare ai vertici della curva la trasformazione di proiezione (ovvero moltiplicare per la matrice di proiezione) e disegnare la curva a partire dai vertici proiettati. Potremo solo applicare la trasformazione proiettiva a tutti i punti della curva già valutata nello spazio 2D, e ciò incide molto sulla complessità computazionale. **Questo vale anche per le curve di Bezier.**
Le curve che non godono di questa proprietà sono anche dette **curve intere**.

Se solo avessimo un modo per superare questo problema...

Curve razionali

...per questo esistono le **curve razionali**. Le curve razionali sono nate proprio per risolvere il problema delle curve spline dell'invarianza per trasformazioni proiettive [anche su Blender, infatti, abbiamo le curve NURBS, che corrispondono alle curve spline razionali].

Le curve razionali inoltre risolvono il problema delle spline di non fornire una rappresentazione analitica vera delle coniche (ovvero le elissi, circonferenze, parabole etc), ma solo una loro approssimazione.

Curve parametriche razionali

Le curve razionali devono il loro nome al fatto che le coordinate dei punti appartenenti ad esse si esprimono come delle funzioni razionali di parametro t, ovvero **come rapporto tra due polinomi nella variabili t**.

$$C(t) = \begin{cases} X(t) = \frac{x(t)}{w(t)} \\ Y(t) = \frac{y(t)}{w(t)} \end{cases}$$

dove w(t) è un polinomio non nullo.

Dunque, possiamo facilmente rappresentare una circonferenza usando questo tipo di formula:

$$C(t) = \begin{cases} X(t) = \frac{1-t^2}{1+t^2} \\ Y(t) = \frac{2t}{1+t^2} \end{cases}$$

Possiamo dimostrare che le curve parametriche razionali sono invarianti anche per trasformazioni affini (e non solo trasformazioni proiettive come abbiamo detto prima).

Curve di Bezier razionali

Sia $w_i > 0$, una curva di Bezier razionale si definisce in questo modo:

$$C(t) = \frac{\sum_{i=0}^n w_i P_i B_{i,n}(t)}{\sum_{i=0}^n w_i B_{i,n}(t)} = \begin{cases} \frac{\sum_{i=0}^n w_i x_i B_{i,n}(t)}{\sum_{i=0}^n w_i B_{i,n}(t)} & 0 \leq t \leq 1 \\ \frac{\sum_{i=0}^n w_i y_i B_{i,n}(t)}{\sum_{i=0}^n w_i B_{i,n}(t)} \end{cases}$$

che possiamo anche scrivere, in formato più simile a quello “classico” delle curve di Bezier, con delle nuove funzioni base:

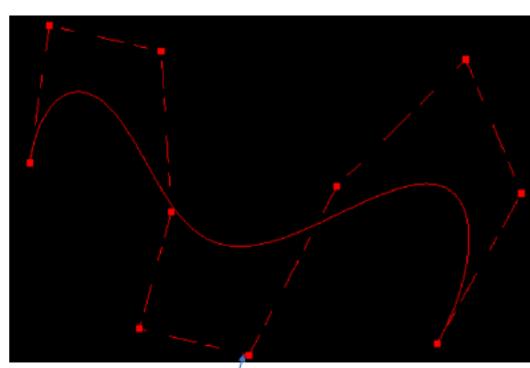
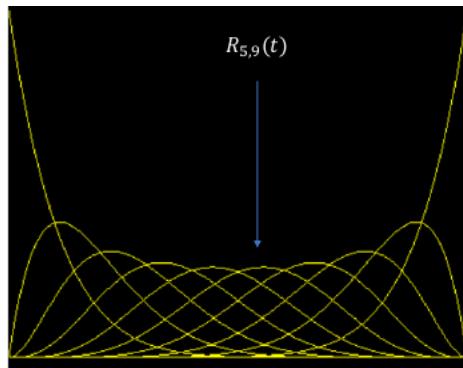
$$C(t) = \sum_{i=0}^n P_i R_{i,n}(t) \quad \text{dove} \quad R_{i,n} = \frac{w_i B_{i,n}(t)}{\sum_{i=0}^n w_i B_{i,n}(t)}$$

$R_{i,n}$ sono le funzioni base delle curve razionali di Bezier. Possiamo studiare le proprietà delle curve razionali studiando le proprietà di queste funzioni base:

- Non sono negative sul loro supporto $R_{i,n} \geq 0$
- Anche queste costituiscono una partizione dell’unità, $\sum_{i=0}^n R_{i,n}(t) = 1$
- $R_{i,n}(0) = 1, R_{i,n}(1) = 1$
- Inoltre, se $w_i = 1$, con $i = 0, \dots, n \Rightarrow R_{i,n} = B_{i,n}$

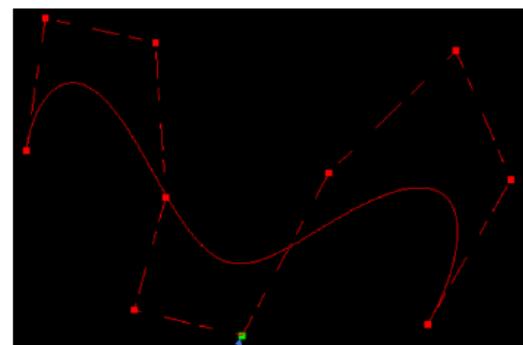
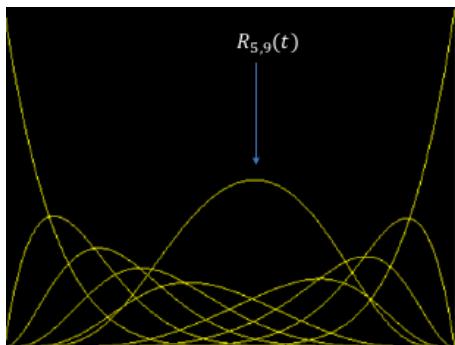
Aumentare il peso w_i di un punto, corrispondente ad un punto P_i , equivale a tirare la curva verso quel punto. I pesi rappresentano un ulteriore strumento di modellazione che influisce notevolmente sulle curve di Bezier. I pesi modificano la nostra curva facendola “attrarre” sempre di più ai nostri vertici di controllo. Nel caso tutti i pesi siano pari ad 1, le funzioni base coincidono con le normali funzioni di base di Bernstein.

Vediamo un esempio per capire meglio:



Nella prima riga, la nostra curva presenta un peso pari a 1 per tutti i punti.

Nella seconda, invece, il punto verde w_5 ha peso pari a 2.8, e come si può notare questo si avvicina al nostro punto di controllo.



Siccome vale la partizione dell’unità, possiamo anche notare come tutte le altre curve della formula vengano modificate (nel caso della seconda immagine, infatti, si abbassano).

Possiamo facilmente ricavare queste curve a partire dalle normali curve di Bezier, che già sappiamo valutare grazie all’algoritmo di de Casteljau.

Siano $P_i = (x_i \ y_i)^T$ i nostri punti 2D della curva. Possiamo facilmente definire i punti 3D della curva in questo modo:

$$P_i^w \equiv \begin{pmatrix} w_i \cdot x_i \\ w_i \cdot y_i \\ w_i \end{pmatrix}, i=0,..n$$

che rappresentano il punto nello spazio 3D e con $z=w_i$.

In questo modo, posso ottenere i miei punti di partenza dello spazio 2D dividendo il punto per la coordinata $z=w_i$.

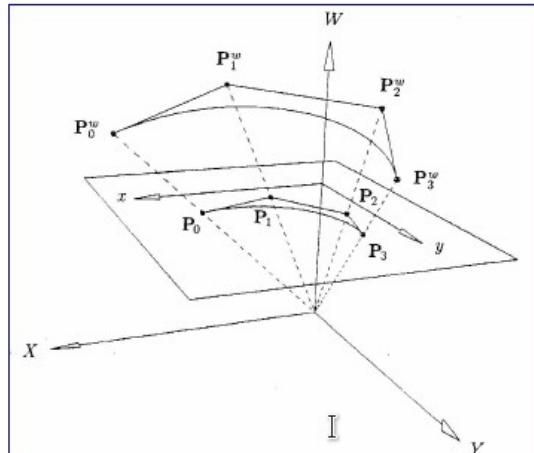
Fatta questa considerazione, possiamo dire che una curva di Bezier razionale 2D può essere vista come la proiezione sul piano 2D di una curva di Bezier in 3D. Infatti, se consideriamo i punti 3D della curva P_i^w definiti come nella pagina precedente, avremo che la curva di Bezier 3D avrà questi vertici di controllo:

$$C_3(t) = \sum_{i=0}^n P_i^w B_{i,n}(t) = \begin{cases} \sum_{i=0}^n w_i x_i B_{i,n}(t) \\ \sum_{i=0}^n w_i y_i B_{i,n}(t) \\ \sum_{i=0}^n w_i B_{i,n}(t) \end{cases}$$

Operativamente, per proiettare la curva 3D sul piano proiettivo 2D ($w=1$) basta dividere le prime due componenti di $C_3(t)$, per la terza. Per questo motivo, diciamo che la curva 2D è una proiezione della curva 3D. Infatti se divido per la terza coordinate ottengo:

$$C_2(t) = \begin{cases} \frac{\sum_{i=0}^n w_i x_i B_{i,n}(t)}{\sum_{i=0}^n w_i B_{i,n}(t)} \\ \frac{\sum_{i=0}^n w_i y_i B_{i,n}(t)}{\sum_{i=0}^n w_i B_{i,n}(t)} \end{cases}$$

che è esattamente l'equazione della curva di Bezier razionale nello spazio 2D



Sapendo ciò, possiamo facilmente valutare una curva in 3D applicando de Casteljau alle 3 componenti della curva (quelle della x, y e w), e poi da essa arrivare alla curva 2D eseguendo una serie di semplici divisioni.

Rappresentare una circonferenza con le curve di Bezier razionali

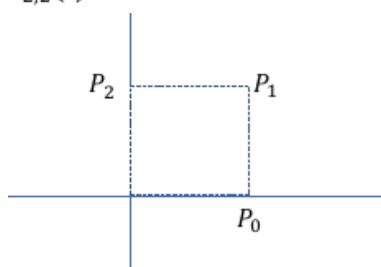
Supponiamo di avere una situazione del genere:

$$P_0 \equiv (1,0), P_1 \equiv (1,1), P_2 \equiv (0,1),$$

$$B_{0,2}(t) = (1-t)^2$$

$$B_{1,2}(t) = 2t(1-t)$$

$$B_{2,2}(t) = t^2$$



Vogliamo riuscire a rappresentare, dal punto di vista analitico, queste due espressioni:

$$C(t) = \frac{\sum_{i=0}^2 w_i P_i B_{i,2}(t)}{\sum_{i=0}^2 w_i B_{i,2}(t)} = \begin{cases} X(t) = \frac{1-t^2}{1+t^2} \\ Y(t) = \frac{2t}{1+t^2} \end{cases}$$

Come determinare, quindi, i pesi per fare in modo di ottenere tale espressione analitica? Per farlo, possiamo facilmente dimostrarlo calcolando un sistema di equazioni:

Siccome i pesi devono essere tali che $\sum_{i=0}^2 w_i B_{i,n}(t) = 1 + t^2$:

$$w_0 B_{0,2}(t) + w_1 B_{1,2}(t) + w_2 B_{2,2}(t) = 1 + t^2$$

Valutiamo in $t = 0$, e poi in $t = 1$ e otteniamo:

$$w_0(1-t)^2 + w_1 2t(1-t) + w_2 t^2 = 1 + t^2 \quad \text{per } t=0 \quad \rightarrow w_0 = 1$$

$$w_0(1-t)^2 + w_1 2t(1-t) + w_2 t^2 = 1 + t^2 \quad \text{per } t=1 \quad \rightarrow w_2 = 2$$

e infine, usando i pesi calcolati in precedenza, valutiamo in $t=1/2$, e otteniamo che $w_1=1$.

Dunque, con questi 3 pesi, riusciamo a scrivere l'equazione di una circonferenza.

Proprietà delle curve di Bezier razionali

Le proprietà delle curve di Bezier razionali sono le stesse identiche delle curve di Beszier normali, con l'aggiunta dell'**invarianza per trasformazioni proiettive**.

Tutti gli algoritmi che abbiamo visto per le curve di Bezier normali sono utilizzabili anche per le curve di Bezier razionali, aggiungendo opportune modifiche per fare in modo che essi lavorino su 3 componenti. **[La prof non ha spiegato le curve Nurbs, ma le chiede. Puoi trovarle qui]**

Punti cardine del modello di illuminazione di Phong

Le caratteristiche fondamentali del modello di Illuminazione di Phong sono:

- **Le sorgenti luminose sono puntiformi;**
- Le componenti speculari e diffusiva sono modellate solo in modo locale;
- La componente dell'ambiente viene modellata come costante, senza tener conto delle interriflessioni tra gli oggetti della scena della radiazione luminosa.

Nel caso nella scena vi sia più di una luce, basta **sommare i termini per ogni sorgente luminosa**. Così se abbiamo m sorgenti luminose l'equazione dell'illuminazione diventa:

$$I = I_a k_a + \sum_{1 \leq i \leq m} I_{pi} [k_d (\bar{N} \bullet \bar{L}_i) + k_s (\bar{R}_i \bullet \bar{V})^n]$$

Il colore degli oggetti viene definito definendo opportunamente i coefficienti di riflessione diffusa e ambientale. Bisogna considerare tre equazioni dell'illuminazione, una per ogni componente del modello del colore considerato:

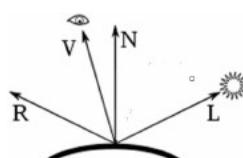
$$I_r = I_a k_{ar} + I_p [k_{dr} (\bar{N} \bullet \bar{L}) + k_s (\bar{R} \bullet \bar{V})^n]$$

$$I_g = I_a k_{ag} + I_p [k_{dg} (\bar{N} \bullet \bar{L}) + k_s (\bar{R} \bullet \bar{V})^n]$$

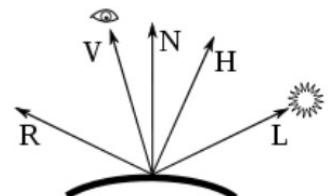
$$I_b = I_a k_{ab} + I_p [k_{db} (\bar{N} \bullet \bar{L}) + k_s (\bar{R} \bullet \bar{V})^n]$$

Modello di illuminazione di Blinn-Phong (diversa da quella base di Phong)

Nel modello di illuminazione di Phong, è necessario ricalcolare continuamente il prodotto scalare tra la direzione di vista (V) e il raggio di una sorgente luminosa (L) riflessa (R) su una superficie, e il calcolo della direzione di riflessione R è oneroso dal punto di vista computazionale.



Blinn ha proposto una variante del modello di illuminazione di Phong, che interessa l'aspetto della **riflessione speculare** della luce. Introduciamo così l'**halfway vector** (H), un vettore unitario esattamente a metà strada tra la **direzione della vista e la direzione della luce**.



Quando la direzione di vista V è perfettamente allineata con il vettore direzione di riflessione, il vettore H (halfway vector) si allinea perfettamente con il vettore normale. Più la direzione della vista è vicina alla direzione di riflessione, maggiore è il contributo speculare. Possiamo calcolare il vettore H con questa formula:

$$H = \frac{L + V}{\|L + V\|}$$

Il contributo della componente speculare della luce viene espresso in termini del prodotto scalare tra la direzione del vettore normale N (normalizzato) alla superficie nel punto di incidenza della luce e il vettore Halfway normalizzato. Se indichiamo con β l'angolo formato tra N ed H, il contributo della luce speculare diventa:

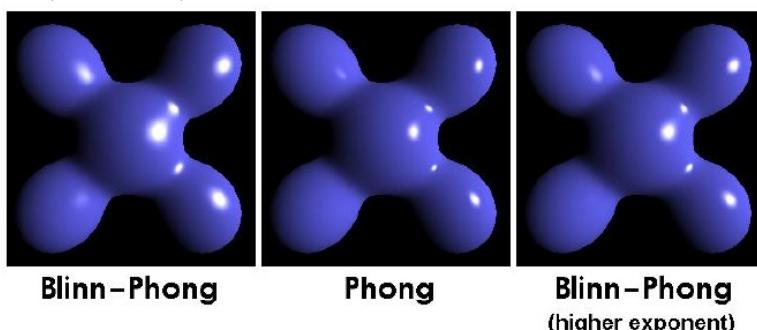
$$I_p k_s (N \cdot H)^n = I_p k_s \cos(\beta)^n$$

In questo modo, l'effetto della specularità viene approssimato, e la formula completa dell'illuminazione di Bling-Phong diventa:

$$I = I_a k_a + f_{att} I_p (k_d (L \cdot N) + k_s (N \cdot H)^n)$$

Mentre i riflessi di Phong sono sempre rotondi per una superficie piana, i riflessi di Blinn-Phong diventano ellittici quando la superficie viene vista da un angolo ripido.

L'esponente del coseno, ovvero n, viene anche detto **coefficiente di shininess**.



ATTENZIONE: anche se è già stato detto prima, ci tengo a precisare che questo tipo di illuminazione è adatto solo per le luci puntiformi! Per le spotlight, per esempio, il calcolo diventa molto più complesso.

Calcolo di R nel metodo di Phong (per chi è interessato)

La prof non sembra dare molta importanza a questi 5 minuti che ha spiegato... in ogni caso, per completezza, sappi che queste parti sono le slides 41-44 sull'illuminazione/shading, e che la formula finale per il calcolo di R è:

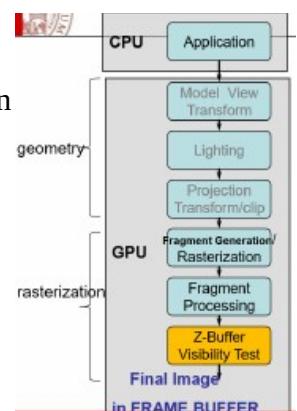
$$R = 2(L \cdot N)N - L$$

[La prof non ha spiegato i metodi di Shading, ma li chiede. Puoi trovarle [qui](#)]

Eliminazione delle superfici nascoste

Nella fase di rasterizzazione, una delle ultime operazioni consiste proprio nella determinazione dei frammenti visibili e non; infatti, sebbene ogni frammento generato dalla rasterizzazione corrisponda a una posizione in un buffer di colore, non vogliamo visualizzare il frammento colorando il pixel corrispondente se il frammento proviene da un oggetto dietro un altro oggetto opaco. Per fare ciò si fa uso di algoritmi per la rimozione della superficie nascosta (o per la determinazione della superficie visibile), che si basano sulle relazioni spaziali tridimensionali tra gli oggetti.

Normalmente, noi non ci dobbiamo preoccupare del sistema di eliminazione delle



superfici nascoste, questo perché abbiamo attivato il **depth buffer**, e che è implementato sulla scheda video a livello hardware, quindi non c'è necessità di un intervento nostro.

Algoritmi per l'eliminazione delle superfici nascoste

L'idea dell'eliminazione delle superfici nascoste sembra semplice, ma richiede una gran potenza di calcolo. Dunque, sono stati ideati numerosi di numerosi ed accurati algoritmi per la determinazione delle superfici visibili e sono state progettate architetture special-purpose per risolvere il problema. Possiamo dividere gli algoritmi per l'eliminazione delle superfici in:

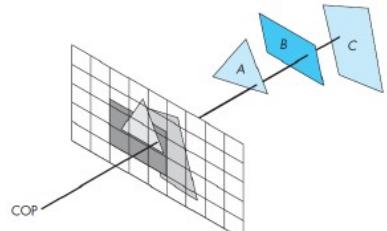
- **Image-space** (o image-precision), sono parte integrante del processo di proiezione siccome tengono conto in maniera implicita della z, che avviene nel processo di proiezione.
- **Object-space** (o object-precision), lavorano in coordinate dell'oggetto in maniera tale che vengano renderizzati gli oggetti nell'ordine in cui questi vengono disegnati (per esempio, come abbiamo fatto nel progetto 2D: prima sfondo, poi personaggio etc...). Sono molto pesanti computazionalmente.

Algoritmi Image-space

Un algoritmo è Image-space quando, dati k oggetti, determina per ognuno degli $m \times n$ pixel nell'immagine quale dei k oggetti è visibile.

Quindi, per ogni pixel:

- si considera un raggio che parte dal centro di proiezione e passa per quel pixel.
- il raggio viene intersecato con ciascuno dei piani determinati dai k poligoni per determinare per quali piani il raggio attraversa un poligono.
- Infine, si determina quale intersezione è più vicina al centro di proiezione e si colora il pixel in esame usando la gradazione di colore del poligono nel punto di intersezione considerato.



Considerando i k poligoni della scena e un display $n \times m$, questa operazione deve essere eseguita **nmk volte**. La complessità risulta di ordine $O(nmk)$. Naturalmente, per aumentare l'accuratezza delle immagini visualizzate, si può considerare anche più di un raggio per pixel.

Per fare il test di collisione del raggio, dovremo confrontare ciascun raggio con tutti i poligoni della scena, proprio per capire quali possono essere una collisione e quali no.

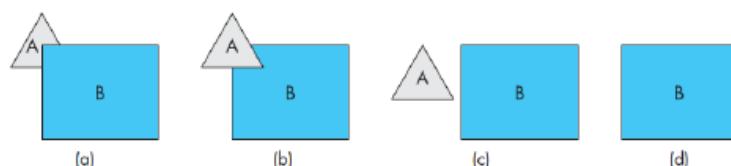
Questo metodo viene anche chiamato **raycasting**.

Algoritmi Object-space

Un algoritmo Object-space confronta ciascun oggetto della scena direttamente con ciascun altro, eliminando oggetti interi o porzioni di oggetti che non sono visibili.

Dunque, data una scena tridimensionale composta da k poligoni piatti ed opachi, si può derivare un generico algoritmo di tipo object-precision considerando gli oggetti a coppie. Data una coppia di poligoni, ad esempio A e B, ci sono quattro casi da considerare:

- A oscura completamente B dalla fotocamera: visualizzeremo solo A;
- B oscura A: visualizzeremo solo B;
- A e B sono completamente visibili: visualizzeremo sia A che B;
- A e B si oscurano parzialmente l'un l'altro: dobbiamo calcolare le parti visibili di ciascun poligono.



Siccome dobbiamo confrontare ciascun poligono con il resto dei poligoni, la complessità di questo

calcolo è O(k²). L'approccio object-precision funziona meglio per scene che contengono relativamente pochi poligoni.

Le operazioni da svolgere per il confronto fra poligoni e per verificare in che relazione stanno sono abbastanza complesse.

Sebbene questo approccio sembra migliore dell'approccio image precision per k<mn (siccome di norma il numero dei pixel è sicuramente maggiore del numero dei poligoni), i suoi passi sono più complessi e richiedono più tempo di elaborazione, ed è spesso più lento e più difficoltoso da implementare.

[Disclaimer: la complessità quindi sarebbe più alta di k²]

Siccome questi due algoritmi sono troppo complessi dal lato computazionale, abbiamo delle tecniche di eliminazione dei poligoni nascosti.

Tecniche di depth testing

Le tecniche che permettono di snellire le superfici nascoste sono:

- Extents box e bounding volume
- Back Face Culling
- Partizionamento spaziale

Extents e Bounding Volume

Possiamo vedere le **Extent** come un'estensione di un oggetto, in una dimensione: infatti, con extent indichiamo l'intervallo tra il valore massimo ed i valore minimo dell'oggetto in quella dimensione.

Per un oggetto sullo schermo, che quindi è a due dimensioni, l'extent è il più piccolo rettangolo allineato con gli assi che contiene l'oggetto.

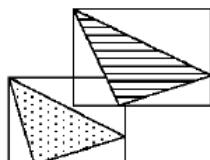
Consideriamo due oggetti 3D e gli extent rettangolari delle loro proiezioni.

Supponiamo di aver considerato la proiezione ortografica, per cui la proiezione sul piano (x,y) (z=0) si ottiene semplicemente ignorando la componente z nelle coordinate.

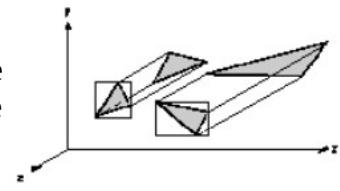
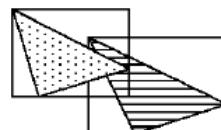
Se gli extent non si sovrappongono, le proiezioni non dovranno essere testate.

Altrimenti, possiamo avere due casi:

Le loro proiezioni non si sovrappongono



Le loro proiezioni si sovrappongono



In entrambi i casi, bisogna realizzare i confronti per determinare se le proiezioni di sovrappongono.

In un caso, i confronti stabiliranno che le due proiezioni non si intersecano. La sovrapposizione degli extent è un falso allarme.

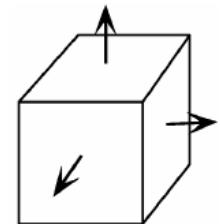
Il test dell'extent rettangolare è detto anche test bounding box. Gli extent possono essere anche utilizzati per circoscrivere gli oggetti, in questo caso sono detti bounding volume.

Backface Culling

Questo metodo si può usare solo per oggetti solidi poliedrici. Definiamo le **normali alle superfici** per ogni faccia del poliedro (che hanno direzione che punta all'esterno del poliedro).

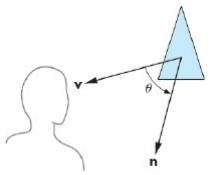
A questo punto, sappiamo che se un oggetto è rappresentato da un poliedro solido chiuso, le facce poligonali del poliedro delimitano completamente il solido.

Supponendo di aver definito i poligoni in maniera tale che le normali alle loro superfici siano tutte dirette verso l'esterno, le facce che hanno una normale che punta verso l'osservatore possono essere visibili, quelle con la normale che punta dall'altra parte rispetto



all'osservatore sicuramente non lo sono.

Si eliminano, quindi, tutti i poligoni il cui vettore normale è orientato verso il semipiano opposto all'osservatore.

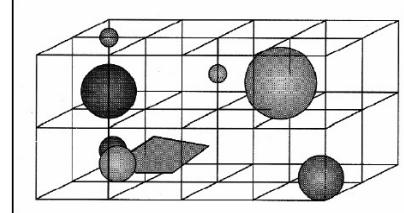


Per determinare se la normale di un poligono è diretta verso l'osservatore, basta considerare l'angolo tra la normale e l'osservatore (che indichiamo con θ): il poligono in esame definisce la parte anteriore di un oggetto se e solo se $-90^\circ \leq \theta \leq 90^\circ$ (quindi se $\cos(\theta) \geq 0$).

Di solito, l'abbattimento **della facce posteriori viene eseguito prima del clipping**, poiché è un'operazione molto rapida e influenzera una percentuale di triangoli molto maggiore rispetto al ritaglio.

Partizionamento Spaziale

L'idea di questa tecnica consiste nel **dividere lo spazio ed assegnare ad ogni sottospazio una parte degli oggetti**. In questo modo i confronti tra gli oggetti sono molto ridotti, tuttavia non è sempre possibile dividere bene lo spazio. Per ovviare a questo problema si usano tecniche di partizionamento adattivo. Vedremo poi degli esempi e in generale il tema in maggiore dettaglio [qui](#).



Lezione 6/12/2021 – Superfici parametriche, continuazione sottosistema raster

Superfici Parametriche

Una superficie è il luogo dei punti visitati da una curva (detta profilo) **che si muove attraverso lo spazio secondo la direzione indicata da un'altra curva**. Ad esempio, se abbiamo un profilo nel piano $z=0$ e facciamo che ogni suo punto si muova secondo la circonferenza, avremo un solido di rotazione.

Una superficie parametrica di Bezier si ottiene spostando i punti di controllo di una curva di Bézier lungo altre curve di Bézier.

Per capire meglio, consideriamo la curva di Bezier di grado n , nel parametro u che varia in $[0,1]$:

$$s(u) = \sum_{i=0}^n P_i B_{i,n}(u)$$

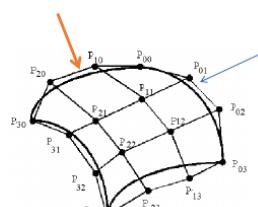
Ogni vertice di controllo P_i , **muovendosi**, genera una curva di Bezier di grado m , nel parametro v che varia in $[0,1]$.

$$P_i = P_i(v) = \sum_{j=0}^m P_{i,j} B_{j,m}(v)$$

Combinando queste due equazioni, otteniamo le coordinate del punto sulla superficie:

$$s(u, v) = \sum_{i=0}^n \sum_{j=0}^m P_{i,j} B_{i,n}(u) B_{j,m}(v)$$

Parliamo così non più di poligono di controllo, bensì di **mesh di controllo**, di dimensione $(n+1)(m+1)$ e definita dai punti $P_{i,j}$.

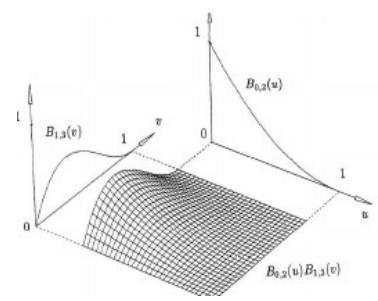


Mesh di controllo

Definiamo $\varphi_{i,j,n,m}(u, v) = B_{i,n}(u) B_{j,m}(v)$. Questa sarà la nostra unica funzione base, che si ottiene facendo il prodotto tensoriale (ovvero cartesiano) fra le funzioni base di Bernstein. [se quest'ultima frase ti confonde, non preoccuparti, non è importante]. L'effetto del prodotto tensoriale si può vedere nell'immagine a fianco a destra.

La formula quindi diventa:

$$s(u, v) = \sum_{i=0}^n \sum_{j=0}^m P_{i,j} \varphi_{i,j,n,m}(u, v)$$



Dunque, ad ogni coppia di valori (u, v) corrisponde un valore z .

Il fatto che ogni funzione base si ottenga a partire da funzioni base monovariante (ovvero con un solo parametro) di Bernstein implica che le funzioni base ottengono molte proprietà delle funzioni base di Bernstein.

Siccome siamo nello spazio 3D, la nostra superficie verrà definita da 3 componenti:

$$s(u, v) = \sum_{i=0}^n \sum_{j=0}^m P_{i,j} B_{i,n}(u) B_{j,m}(v) = \begin{cases} \sum_{i=0}^n \sum_{j=0}^m x_{i,j} B_{i,n}(u) B_{j,m}(v) \\ \sum_{i=0}^n \sum_{j=0}^m y_{i,j} B_{i,n}(u) B_{j,m}(v) \\ \sum_{i=0}^n \sum_{j=0}^m z_{i,j} B_{i,n}(u) B_{j,m}(v) \end{cases}$$

I nostri punti $P_{i,j}$ hanno 3 coordinate adesso.
 $P_{i,j} = (x_i, y_i, z_i)$

Possiamo anche esprimere la superficie di Bezier in forma matriciale.

$$s(u, v) = [B_{0,n}(u) \quad B_{1,n}(u) \quad \dots \quad B_{1,n}(u)] \begin{bmatrix} P_{0,0} & P_{0,1} & \dots & P_{0,m} \\ P_{1,0} & P_{1,1} & \dots & P_{1,m} \\ \vdots & & & \vdots \\ P_{n,0} & P_{n,1} & & P_{n,m} \end{bmatrix} \begin{bmatrix} B_{0,m}(v) \\ B_{1,m}(v) \\ \vdots \\ B_{m,m}(v) \end{bmatrix}$$

Forma matriciale per le superfici parametriche di Bernstein (poco importante)

Ricordiamo la relazione che lega le funzioni base di Bernstein di grado n e le funzioni di base monomiale di grado n :

$$\begin{bmatrix} B_{0,n}(u) \\ B_{1,n}(u) \\ \vdots \\ B_{n,n}(u) \end{bmatrix} = M \begin{bmatrix} 1 \\ u \\ \vdots \\ u^n \end{bmatrix} \quad \begin{bmatrix} B_{0,m}(v) \\ B_{1,m}(v) \\ \vdots \\ B_{m,m}(v) \end{bmatrix} = N \begin{bmatrix} 1 \\ v \\ \vdots \\ v^m \end{bmatrix}$$

M è la matrice del cambiamento di base che ci fa passare dalla base monomiale di grado n alla base di Bernstein di grado n . Stessa cosa per N, ma relativa alla seconda base di Bernstein.

e quindi possiamo scrivere la formulazione matriciale di una superficie di Bezier

$$s(u, v) = [1 \quad u \quad \dots \quad u^n] M^T \begin{bmatrix} P_{0,0} & P_{0,1} & \dots & P_{0,m} \\ P_{1,0} & P_{1,1} & \dots & P_{1,m} \\ \vdots & & & \vdots \\ P_{n,0} & P_{n,1} & & P_{n,m} \end{bmatrix} N \begin{bmatrix} 1 \\ v \\ \vdots \\ v^m \end{bmatrix}$$

Nel caso di una superficie di Bezier bicubica si ha che:

$$M = N = \begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

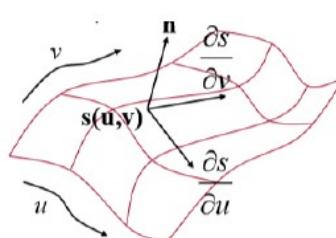
Normali nelle superfici parametriche

Abbiamo visto come nell'illuminazione sia fondamentale il calcolo delle normali alla superficie, in modo da poter applicare il metodo di illuminazione.

Nel caso delle superfici, sappiamo che per calcolare l'equazione della normale alla superficie in un punto, devo calcolare il piano tangente alla superficie in quel punto, e il segmento perpendicolare al piano tangente rappresenta la normale. Sappiamo poi che per calcolare il piano tangente alla superficie in un punto basta fare il prodotto tensoriale tra le derivate parziali della superficie rispetto a un parametro e poi rispetto all'altro.

$$\mathbf{n}^* = \frac{\partial s(u, v)}{\partial u} \times \frac{\partial s(u, v)}{\partial v}$$

$$\mathbf{n} = \frac{\mathbf{n}^*}{\|\mathbf{n}^*\|}$$



Il vettore normale \mathbf{n} ad una superficie parametrica è un vettore normalizzato normale (quindi perpendicolare) alla superficie in un determinato punto (u, v) . È calcolato dal prodotto vettoriale di due vettori tangenti alla superficie in quel punto.

Le derivate parziali alla superficie possiamo ricavarle con la nostra conoscenza della derivata della curva di Bezier.

$$\begin{aligned}\frac{\partial s(u, v)}{\partial u} &= \frac{\partial}{\partial u} \sum_{i=0}^n \sum_{j=0}^m P_{ij} B_{i,n}(u) B_{j,m}(v) = \\ &= n \sum_{i=0}^n \sum_{j=0}^m (P_{i+1,j} - P_{ij}) B_{i,n-1}(u) B_{j,m}(v) \\ \frac{\partial s(u, v)}{\partial v} &= \frac{\partial}{\partial v} \sum_{i=0}^n \sum_{j=0}^m P_{ij} B_{i,n}(u) B_{j,m}(v) = \\ &= m \sum_{i=0}^n \sum_{j=0}^m (P_{i,j+1} - P_{ij}) B_{i,n}(u) B_{j,m-1}(v)\end{aligned}$$

Curve di Bezier isoparametriche

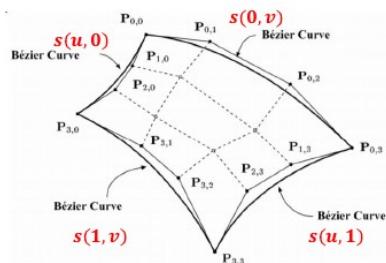
Fissato un valore del parametro v o u , per esempio $v = v'$, allora la superficie $s(u, v)$ si riduce ad una curva.

Se invece fisso $v = 0$, solo $B_{0,m}$ è diversa da zero è pari ad 1 [anche se le formule sono in rosso, non sono poi così importanti.].

$$s(u, 0) = \sum_{i=0}^n P_{i,0} B_{i,n}(u) B_{0,m}(0) = \sum_{i=0}^n P_{i,0} B_{i,n}(u)$$

Se invece fisso $v = 1$, ottengo che solo la funzione base di Bernstein m -esima, $B_{m,m}$ è diversa da zero e vale 1.

$$s(u, 1) = \sum_{i=0}^n P_{i,m} B_{i,n}(u) B_{m,m}(0) = \sum_{i=0}^n P_{i,m} B_{i,n}(u)$$



In base a quanto detto, ciò che otteniamo nei vari casi di $u = 0, v = 0$ etc.. sono le curve di confine della nostra superficie parametrica.

Proprietà delle funzioni base $B_{i,n}(u)B_{j,m}(v)$

- Partizione dell'unità
- Non-negatività
- Convex-hull (ovvero la superficie è definita da una combinazione lineare convessa ed è quindi contenuta nell'inviluppo convesso).
- Invariazia per trasformazioni affini.
- La superficie di Bezier interpola i 4 vertici di controllo $P_{0,0}, P_{0,m}, P_{n,0}$ e $P_{n,m}$ e approssima la forma del mesh di controllo.

Valutazione delle superfici di Bezier

Possiamo usare una specie di trucco per la valutazione delle superfici di Bezier.

Definiamo $Q_j(u) = \sum_{i=0}^n P_{i,j} B_{i,n}(u)$

La superficie di Bezier si può riscrivere nel seguente modo:

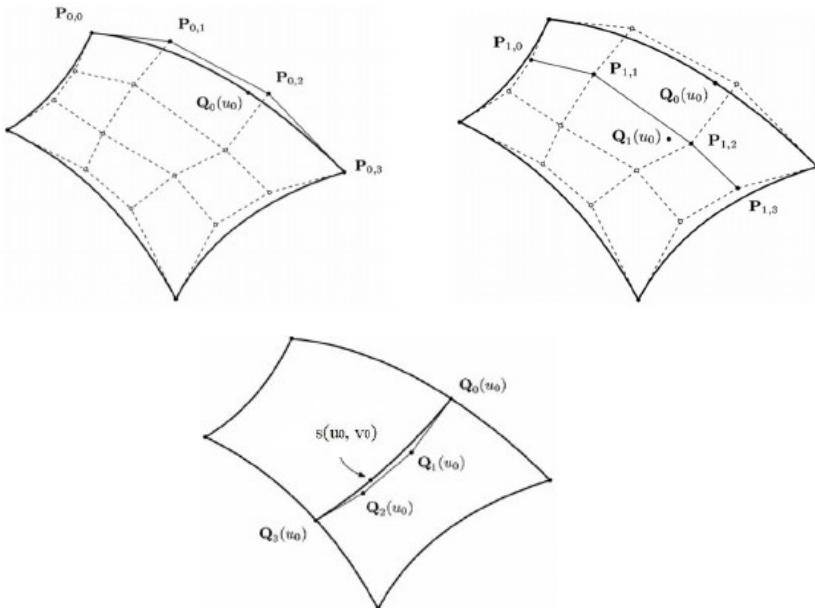
$$s(u, v) = \sum_{j=0}^m Q_j(u) B_{j,m}(v)$$

A questo punto quello che abbiamo ottenuto è una curva di Bezier di grado m i cui vertici di controllo sono i punti $Q_j(u)$.

A questo punto fissiamo un valore $u = u_0$, e valutiamo $Q_j(u) = \sum_{i=0}^n P_{i,j} B_{i,n}(u)$ con $j = 0 \dots m$, utilizzando $m+1$ volte l'algoritmo di De Casteljau.

Dopodiché, riapplichiamo l'algoritmo di De Casteljau alla curva che otteniamo dopo, con vertici di controllo $Q_j(u)$.

Vediamo un'immagine per capire meglio.



Q_0 rappresenta la curva che va da sinistra verso destra (il confine in alto di questa curva) e che ha punti di controllo $P_{0,0} \dots P_{0,3}$.

Al passo $j = 1$, calcoliamo la curva Q_1 definita dai vertici di controllo $P_{1,0} \dots P_{1,3}$.

Dopo aver fatto questo fino a $j = m$, il punto $s(u_0, v_0)$ sulla superficie viene calcolato come un punto sulla curva di Bezier definita dai vertici di controllo $Q_0(u_0), Q_1(u_0), Q_2(u_0), Q_3(u_0)$.

In poche parole, prima si procede in un verso, e poi nell'altro.

Superfici Spline

Come per le superfici di Bezier, le superfici spline sono date da una curva spline i cui vertici di controllo non sono fermi, ma si muovono descrivendo un'altra curva spline.

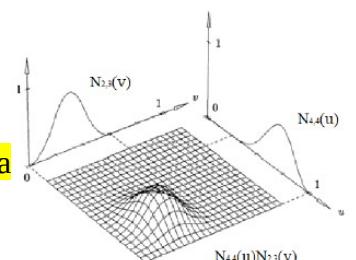
Dunque l'equazione di una superficie spline è data da:

$$s(u, v) = \sum_{i=1}^{n+H} \sum_{j=1}^{m+K} P_{i,j} N_{i,n}(u) N_{j,m}(v) = \begin{cases} \sum_{i=1}^{n+H} \sum_{j=1}^{m+K} x_{i,j} N_{i,n}(u) N_{j,m}(v) \\ \sum_{i=1}^{n+H} \sum_{j=1}^{m+K} y_{i,j} N_{i,n}(u) N_{j,m}(v) \\ \sum_{i=1}^{n+H} \sum_{j=1}^{m+K} z_{i,j} N_{i,n}(u) N_{j,m}(v) \end{cases}$$

che è essenzialmente la stessa formula che abbiamo usato per le superfici di Bezier, ma utilizzando le funzioni spline normalizzate (di Cox) al posto delle funzioni di Bernstein.

Anche qui, le proprietà delle funzioni base dervano dalle proprietà delle funzioni base univariate.

La caratteristica di questa superficie è che, siccome noi sappiamo del supporto locale delle curve spline [che possiamo rivedere [qui](#)] possiamo dire che la nostra superficie sarà diversa da 0 solo nel prodotto cartesiano dell'intervallo in cui le due funzioni di base sono diverse da 0.



Proprietà delle B-spline

- Invarianza affine
- Proprietà locale dell'inviluppo
- Modifica locale: se spostiamo un vertice di controllo, non cambia l'intera superficie, bensì solo nel rettangolo $(x_i, x_{i+n}) \times (y_j, y_{j+m})$
- NON vale la proprietà del variation diminishing.

Valutazione delle superfici spline

Ci comportiamo allo stesso modo della valutazione delle superfici di Bezier, ma usando l'algoritmo di De Boor.

Superfici NURBS (ovvero spline razionali)

Sono definite dalla formula:

$$s(u, v) = \frac{\sum_{i=1}^{n+H} \sum_{j=1}^{m+K} w_{i,j} P_{i,j} N_{i,n}(u) N_{j,m}(v)}{\sum_{i=1}^{n+H} \sum_{j=1}^{m+K} w_{i,j} N_{i,n}(u) N_{j,m}(v)}$$

che implica che ad ogni vertice di controllo viene associato un peso $w_{i,j} > 0$,

Continuazione della fase di rasterizzazione

Dalla volta scorsa (in realtà da molto tempo prima) abbiamo visto la rasterizzazione di una linea retta, attraverso l'uso di vari algoritmi [maggiori info [qui](#)].

Ripassino algoritmo di Bresenham (o Midpoint)

L'**algoritmo di Bresenham** (o **Midpoint**) risolve il problema dei calcoli floating point esposto da algoritmi come il DDA. Come l'algoritmo precedente (ovvero quello DDA) è un **algoritmo differenziale**, ovvero in base alle informazioni al passo i riesce ad individuare le informazioni per il pixel al passo $i+1$.

L'idea di base di questo approccio è quella di individuare, passo dopo passo, il pixel più vicino all'effettivo punto di intersezione Q tra la retta e l'ascissa $x = x_p + 1$ corrispondente al successivo punto da accendere. I candidati, ad ogni passo sono solo i due pixel E ($x_p + 1, y_p$) ovvero NE ($x_p + 1, y_p + 1$).

L'implementazione è, anche in questo caso, **incrementale**.

Si consideri l'equazione della retta in forma esplicita:

$$F(x, y) = ax + by + c = 0$$

Si definisce una variabile di decisione d definita come:

$$d = F(x_M, y_M) = F\left(x_p + 1, y_p + \frac{1}{2}\right) = a(x_p + 1) + b\left(y_p + \frac{1}{2}\right) + c$$

Per $d > 0$ si accenderà NE, mentre per $d \leq 0$ si accenderà E.

L'algoritmo consiste nel **calcolare incrementalmente la sola variabile di decisione**.

Calcolare il pixel successivo diventa poi semplice, infatti: se ad un certo passo $p+1$ si accende il pixel E, allora il nuovo valore della variabile d_{new} , noto d_{old} al passo precedente p , si ottiene come:

$$d_{new} = a(x_p + 2) + b\left(y_p + \frac{1}{2}\right) + c = a(x_p + 1) + b\left(y_p + \frac{1}{2}\right) + c + a = d_{old} + a = d_{old} + a = d_{old} + dy$$

Questa è
l'unica parte
della formula
che ci
interessa.

Alternativamente, se al passo $p+1$ si accende il pixel NE, allora:

$$d_{new} = a(x_p + 2) + b\left(y_p + \frac{3}{2}\right) + c = a(x_p + 1) + b\left(y_p + \frac{1}{2}\right) + c + a + b = d_{old} + a + b = d_{old} + (dy - dx)$$

Bisogna fare attenzione alla notazione, infatti dy e dx sono i delta (quindi la variazione) fra la coordinata x e y del nuovo punto.

d rappresenta inoltre un valore che può essere maggiore o minore di 0, siccome stiamo calcolando "l'intersezione" fra i punti M e la retta. Infatti, stiamo così stabilendo se il punto M appartiene alla retta o no, e sappiamo dalla geometria che se il valore è 0 allora M appartiene alla retta, se è > 0 allora la retta è maggiore rispetto a M e quindi passa sopra al punto, mentre se è < 0 allora è minore, quindi passa sotto al punto.

Possiamo fare un test mentale con una semplice retta applicando lo stesso algoritmo a $x+y+4=0$ e l'origine, per vedere appunto come funziona questo ragionamento.

Per eseguire questo algoritmo, però, dobbiamo partire da una d di partenza. Dunque, il processo si inizializza calcolando il valore di partenza. Partendo da (x_0, y_0) calcoliamo d_{start} :

$$d_{start} = F\left(x_0 + 1, y_0 + \frac{1}{2}\right) = a(x_0 + 1) + b\left(y_0 + \frac{1}{2}\right) + c = ax_0 + by_0 + c + a + \frac{b}{2} = \\ = F(x_0, y_0) + a + \frac{b}{2} \equiv a + \frac{b}{2} = dy - \frac{dx}{2}$$

Per eliminare la divisione ed utilizzare solo l'aritmetica intera si può far riferimento alla nuova variabile di decisione 2d che deve solamente essere confrontata con il valore 0 ad ogni passo.

Se usiamo questa variabile, dobbiamo però moltiplicare i nostri valori che abbiamo stabilito prima per 2.

Quindi, avremo:

$$d_{start} = 2dy - dx$$

$$\text{deltaE} = 2dy$$

$$\text{deltaNE} = 2(dy - dx)$$

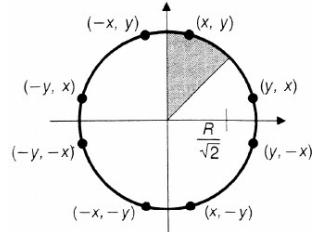
```
Calcola dx=x1-x0;
Calcola dy=y1-y0;
Calcola dstart=2dy-dx;
Calcola deltaNE=2(dy-dx);
Calcola deltaE=2dy;
x=x0; y=y0; d=dstart;
Ripeti finché x<x1
  Se d<=0 allora
    d+=deltaE;
    x++;
  Altrimenti
    d+=deltaNE;
    x++;
    y++;
  Accendi il pixel (x,y)
```

Algoritmo per il calcolo del midpoint

Midpoint per le circonferenze

Esiste una variazione dell'algoritmo del Mid-point apposita per le circonferenze.

Questa variazione prende in considerazione la formula implicita della circonferenza, $F(x, y) = x^2 + y^2 - R^2 = 0$, e calcola il midpoint per il primo ottante $x \in [0, R/\sqrt{2}]$ e poi si replica per gli altri.

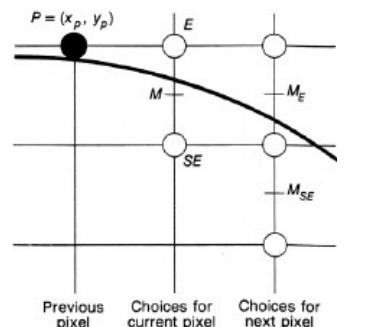


L'idea è molto simile all'algoritmo che abbiamo visto prima, ma cambia l'equazione della funzione implicita.

Nel caso del primo ottante, infatti, per calcolare d la formula sarà:

$$d = F(x_M, y_M) = \left(x_p + 1\right)^2 + \left(y_p - \frac{1}{2}\right)^2 - R^2$$

In questo modo, se $d \geq 0$ giace all'interno della circonferenza (e quindi in questo caso accenderemo SE), sltrimenti sarà all'esterno (e accenderemo E). Possiamo anche procedere come per il metodo precedente, e calcolare i 2 d_{new} e d_{start} usando lo stesso metodo del Midpoint per le rette.



$$d_{new} = F(x_p + 2, y_p - 1/2) = d_{old} + (2x_p + 3)$$

$$d_{start} = F\left(0 + 1, R - \frac{1}{2}\right) = F\left(1, R - \frac{1}{2}\right) =$$

$$d_{new} = F(x_p + 2, y_p - 3/2) = d_{old} + 2(x_p - y_p) + 5$$

$$= 1 + \left(R - \frac{1}{2}\right)^2 - R^2 = 1 + R^2 - R + \frac{1}{4} - R^2 = \frac{5}{4} - R$$

Anche qui, per mantenere l'uso dell'aritmetica intera, si considera $h = d - 1/4 \rightarrow$ da cui $h_{start} = 1 - R$. h si confronta sempre con 0, come per le rette.

Rasterizzazione di un triangolo

Dopo la fase di proiezione, i lati del triangolo risultano proiettati su un piano. Dobbiamo ora capire quali dei punti sono interni al triangolo. L'interno del triangolo è l'insieme di punti che si trova all'interno di tutti e tre i semispazi definiti da queste linee.

Per capire se un punto è interno al triangolo (**Inside Triangle Test**), usiamo queste ipotesi:

- Un punto è all'interno di un triangolo se si trova nel semispazio negativo di tutte e tre gli edge di confine
- I vertici del triangolo sono ordinati in senso antiorario
- Il punto deve trovarsi sul lato sinistro di ogni edge di confine (questo funziona solo se i

vertici sono ordinati in senso antiorario)

Con quello che abbiamo appena detto si intende che, dati due vertici del triangolo $P_0 \equiv (x_0, y_0)$ e $P_1 \equiv (x_1, y_1)$, calcoliamo la retta che passa per questi due punti in forma implicita:

$$(x - x_0)(y_1 - y_0) - (x_1 - x_0)(y - y_0) = 0$$

$$x(y_1 - y_0) + y(x_0 - x_1) - x_0(y_1 - y_0) + y_0(x_1 - x_0) = 0$$

a b c

In questo modo, abbiamo calcolato l'edge E_0 . Dunque, a questo punto, possiamo dire che:

$$(x, y) \text{ interno al triangolo} \Leftrightarrow E_i(x, y) \leq 0, \forall i = 0, 1, 2$$

Questo funziona solo se consideriamo i vertici in senso anti orario e in generale questo metodo funziona solo per poligoni convessi.

Il punto siu

Algoritmo per la rasterizzazione

Per ogni triangolo

Calcola la proiezione dei vertici, calcola gli E_i (a_i, b_i, c_i)

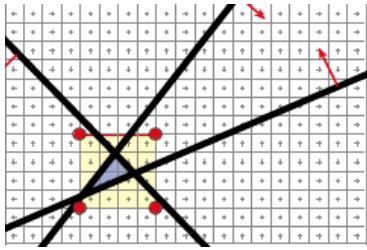
Calcola il bounding box (bbox) sullo schermo, taglia il bbox rispetto ai limiti dello schermo

Per ogni pixel (x, y) nel bbox

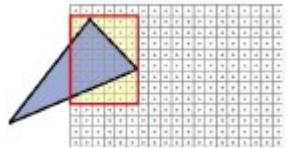
Valuta le funzioni edge E_i , $i=1, 2, 3$ nel centro del pixel

If all $E_i < 0$ % il pixel è interno

Framebuffer[x, y] = triangleColor



Supponiamo di avere un triangolo che esce fuori dallo schermo. Si calcola quindi il bounding box del triangolo sullo schermo, e lo si taglia rispetto ai limiti (del bounding box). In questo modo noi analizziamo il triangolo solo nei vertici che ricadono dentro lo schermo.



Come facciamo se abbiamo poligoni concavi?

In questi casi, quello che si fa è applicare una **tesselazione**, che converte ogni poligono in triangoli. Dopodiché, possiamo eseguire la scan conversion esattamente come abbiamo fatto per ogni altro elemento.

Questa fase viene svolta dopo la rasterizzazione (ovviamente), e ci permette di fare una scan conversion di qualsiasi tipo di figura.

Ma che vuol dire $\text{Framebuffer}[x, y] = \text{Color}$?

In pratica, di ogni triangolo noi memorizziamo i vertici, e poi, oltre alle informazioni sulla geometria, per ogni vertice memorizziamo anche informazioni quali i colori, coordinate di texture, normali dei vertici etc. In particolare, per i colori, si utilizza l'interpolazione per calcolare i valori di questi dati all'interno del triangolo. Per esempio:

- Specifichiamo un colore (R,G,B) su ognuno dei vertici di un triangolo
- Per ogni pixel interno al triangolo, calcoliamo le coordinate di interpolazione per quel pixel.
- Usiamo queste coordinate interpolate, per calcolare un colore interpolato (R,G,B) per quel pixel.

Possiamo avere più metodi per interpolare i colori. Quella che si usa normalmente è l'interpolazione di tipo baricentrico.

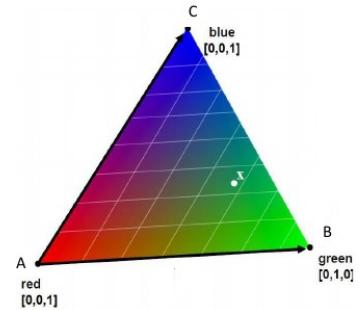
Per-pixel color: barycentric interpolation

Il colore di ogni pixel del triangolo si ottiene combinando il colore agli estremi per dei valori α, β, γ (che sono le coordinate baricentriche) definiti in questo modo:

$$\alpha = \frac{\text{area}_{xBC}}{\text{area}_{ABC}} = \frac{\frac{1}{2}E_{BC}(x)}{\frac{1}{2}E_{BC}(A)} = \frac{E_{BC}(x)}{E_{BC}(A)}$$

$$\beta = \frac{E_{CA}(x)}{E_{CA}(B)}$$

$$\gamma = \frac{E_{AB}(x)}{E_{AB}(C)}$$



Il colore in un punto x è la combinazione affine dei colori sui tre vertici dei triangoli:

$$x_{color} = \alpha A_{color} + \beta B_{color} + \gamma C_{color}$$

In pratica quindi si calcola **x sulla base delle coordinate baricentriche**.

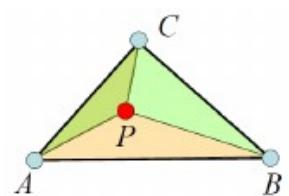
Coordinate baricentriche di un punto appartenente ad un triangolo (per capire meglio)

Le coordinate baricentriche di un punto interno al triangolo si calcolano in questo modo:

$$P = \alpha_0 A + \alpha_1 B + \alpha_2 C$$

$$\alpha_0 + \alpha_1 + \alpha_2 = 1, \alpha_i > 0, i = 0, 1, 2$$

$$\alpha_0 = \frac{\text{area}(PBC)}{\text{area}(ABC)} \quad \alpha_1 = \frac{\text{area}(PCA)}{\text{area}(ABC)} \quad \alpha_2 = \frac{\text{area}(PAB)}{\text{area}(ABC)}$$



Clipping: ripassino

Il clipping comporta l'eliminazione di oggetti che si trovano al di fuori del volume di visualizzazione e quindi non possono essere visibili nell'immagine. Dopo di esso, si passa alla rasterizzazione, che produce i frammenti che oggetti finali rimanenti. Questi frammenti saranno quelli candidati a contribuire all'immagine finale.

Nella fase di rimozione delle superfici nascoste, che avviene dopo la rasterizzazione, ci chiediamo: tutte queste superfici che abbiamo generato sono appartententi ad oggetti visibili oppure no? Dunque, se io avrò due triangoli sovrapposti (che potrò appunto esaminare con il raggio proiettore che abbiamo visto [qui](#)), dovrò andare ad accendere quello che avrà la z più vicina all'osservatore.

Questa fase avviene nel sottosistema raster, a proiezione finita, quindi teoricamente non abbiamo più il valore z. Per questo motivo, nella fase di proiezione, la z (dei vertici) per ogni frammento non viene scartata, ma viene conservata in un buffer detto z-buffer [disclaimer: in realtà, si prende la z dei singoli vertici, che viene interpolata per creare uno z singolo associato al frammento].

Abbiamo poi visto moltissimi algoritmi per l'eliminazione delle superfici nascoste, e onestamente non ho voglia di riscriverli, quindi andate a vedere se volete, ma io vado avanti.

Algoritmo Z-Buffer o depth-buffer

È uno degli algoritmi per la determinazione delle superfici visibili più usati, e che sono implementati in OpenGL (sia lato software che hardware).

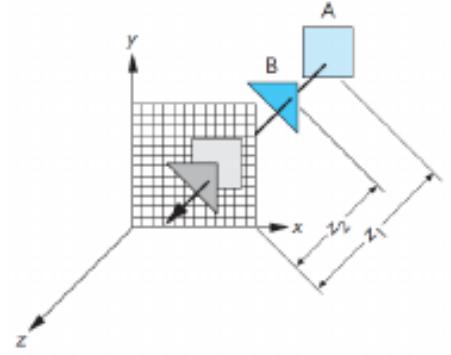
È un algoritmo di tipo image-precision. Tra l'altro è anche molto semplice.

[In OpenGL, attiviamo questo algoritmo con `glEnable(DEPTH_BUFFER)`]

Supponiamo di dover rasterizzare uno dei due poligoni mostrati in figura.
Tracciamo il raggio dal centro di proiezione e un pixel.

Se stiamo rasterizzando B, il suo colore apparirà sullo schermo se la distanza z_2 è inferiore alla distanza z_1 del poligono A.

Al contrario, se stiamo rasterizzando A, il pixel che corrisponde al punto di intersezione non apparirà su il display.



Supponiamo di avere un buffer, detto **z-buffer o depth buffer**, con la stessa risoluzione del frame buffer, dove sono memorizzate le profondità di ogni frammento che viene rappresentato nel framebuffer (quindi per ogni frammento che corrisponde ad una posizione x,y del framebuffer, nel framebuffer ce la sua corrispondente z). Ad esempio, se abbiamo un display 1024×1280 e utilizziamo numeri interi standard per il calcolo della profondità, possiamo usare un buffer z 1024×1280 con elementi a 32 bit.

Pseduocodice dell'algoritmo

► For (ogni poligono)

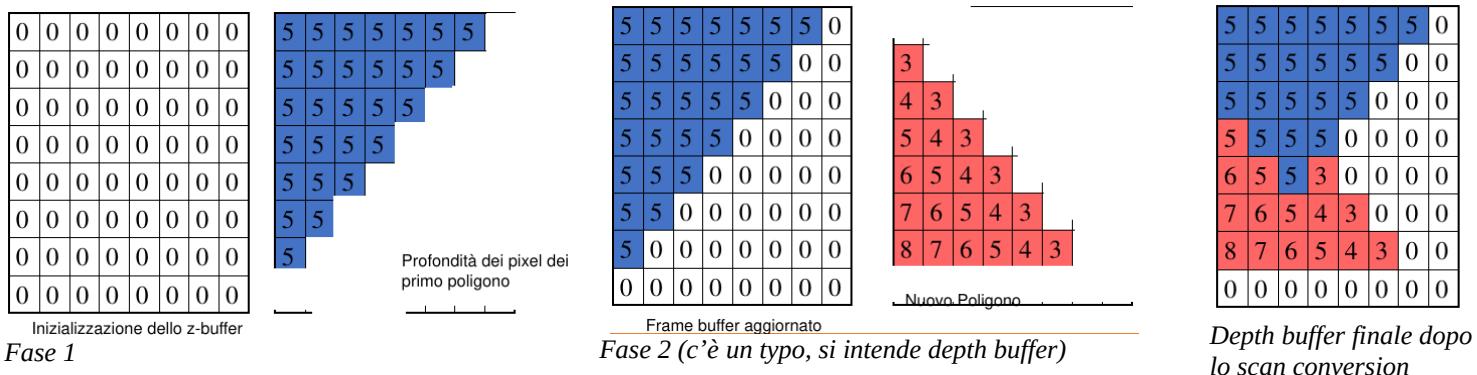
► For (ogni pixel del poligono proiettato)

- $z =$ distanza dall'osservatore del poligono nel punto (x,y) del frame buffer.
- If ($z >=$ valore di z memorizzato nel dept-buffer in posizione (x,y))
 - Disegna il colore del pixel del poligono nella posizione (x,y) del frame buffer.
 - Scrivi il valore di z nella posizione (x,y) del depth-buffer.

► End for

► End for

Esempio del funzionamento:



Lezione 7/12/2021 – Continuazione Z-buffer, depth-sort, raycasting, raytracing

Continuazione algoritmo z-buffer

L'algoritmo non richiede che gli oggetti siano poligoni. Può esser utilizzato per qualsiasi oggetto purchè possa essere determinato il valore di z per ogni suo punto nella sua proiezione. Altri vantaggi sono:

- Non richiede che gli oggetti siano ordinati.
- Non richiede confronti tra gli oggetti.

Fissato un pixel, l'ordinamento delle z avviene solo tra quei poligoni che contengono quel pixel.

Il tempo speso per i calcoli per la determinazione delle superfici visibili tende ad essere indipendente dal numero dei poligoni, perché in media, il numero dei pixel coperti da ogni poligono decresce quando il numero dei poligoni nel volume di vista cresce.

[RICORDA, l'ordinamento si fa rispetto alla z del centroide del frammento!]

Algoritmi che si basano sulla lista di priorità: Depth Sort

Questo tipo di algoritmi sono di tipo Object-space (in realtà vedremo qui che sono più un tipo ibrido). Definiscono un ordine di visibilità per gli oggetti ed assicurano che se gli oggetti verranno renderizzati in quell'ordine, la visualizzazione sarà corretta.

Si basano su una **lista di priorità**, dove:

- gli elementi con priorità più bassa oggetti più lontani dall'osservatore
- gli elementi priorità più alta oggetti più vicini all'osservatore

Dopo che è stata determinata la priorità, i poligoni vengono sottoposti a scan-conversion uno alla volta all'interno del frame buffer, iniziando dal poligono a priorità più bassa.

[**RICORDA**, anche qui, l'ordinamento si fa rispetto alla z del centroide del frammento!]

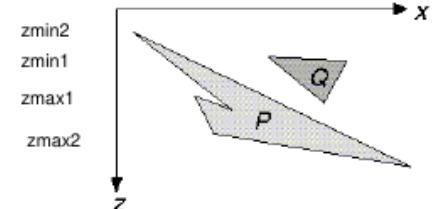
In questo modo, gli oggetti più lontani saranno oscurati da quelli più vicini come i pixel dei poligoni più vicini sovrascrivono quelli dei pixel più lontani.

Se dopo l'ordinamento nella lista non compaiono neppure due poligoni adiacenti che si sovrappongono nel senso della profondità, la lista si trova ordinata correttamente. Ovviamente, l'algoritmo funziona solo se la lista è stata ordinata correttamente.

Risoluzione dell'ambiguità in depth sort

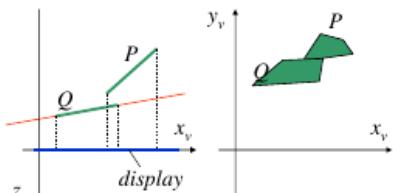
Una cosa interessante sta nel fatto che, con questo algoritmo, anche se gli oggetti si sovrappongono in z, è possibile ancora determinare un ordine corretto. Ecco come.

Sia P il poligono corrente della lista ordinata di poligoni di cui dobbiamo effettuare la scan conversion. Prima di effettuare la scan-conversion di questo poligono nel frame buffer, è necessario testarlo con ogni altro poligono Q avente la stessa z del poligono P, per provare che P non può oscurare Q e che P può essere scritto prima di Q.

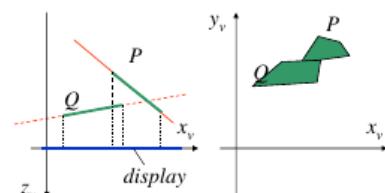


In generale, vanno fatti 5 test per confrontare Q con P. Se almeno uno dei cinque test viene superato, allora P viene disegnato nel frame buffer ed il poligono successivo nella lista diventa il nuovo P (ovvero il prossimo poligono con la stessa z andrà testato con gli altri).

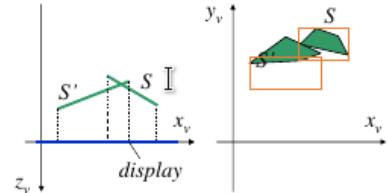
1. Le coordinate in x dei due poligoni non si sovrappongono (vuol dire che le due figure non si sovrappongono per la dimensione x). Queste due condizioni implicano che le due figure hanno stessa z, ma diversa x o y.
2. Le coordinate in y dei due poligoni non si sovrappongono.
3. P è completamente dietro Q? Si consideri il piano contenente Q. P è contenuto completamente nel semipiano opposto a quello dove è l'osservatore?
4. Si consideri il piano contenente P. Q è contenuto completamente nello stesso semipiano a quello dove è l'osservatore?
5. Le proiezioni dei poligoni sul piano (x,y) non si sovrappongono (ciò può essere determinato confrontando gli spigoli di ogni poligono con gli spigoli dell'altro).



Rappresentazione del test 3



Rappresentazione del test 4



Rappresentazione del test 5

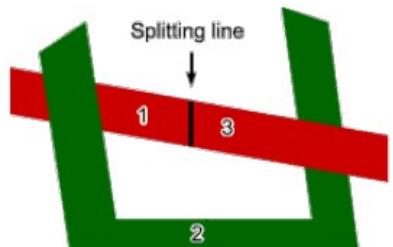
Se tutti i cinque test falliscono, per il momento supponiamo che P oscura Q e testiamo se Q può essere disegnato prima di P. **Non sarà necessario ripetere i test 1,2,5**, ma basterà ripetere i test 3 e 4, che verranno riformulati con i poligoni scambiati (i.e. Q è dietro P (3'), P è davanti Q (4')).

L'algoritmo di depth-sorting è di tipo **ibrido**:

- **Object precision** nella fase relativa al confronto delle profondità.
- **Image precision** nella fase di scan-conversion dei poligoni nel frame buffer.

Come si gestisce la sovrapposizione ciclica?

Se gli oggetti si sovrappongono ciclicamente a qualche altro o penetrano qualche altro, non è possibile stabilire un ordine corretto con i metodi standard. In questo caso è necessario **splittare** uno o più oggetti per rendere possibile un ordine lineare.

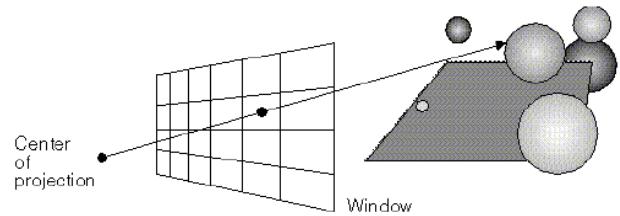


Lo splitting viene fatto in base alla figura, attraverso algoritmi ad-hoc. Questo viene fatto in coordinate dell'oggetto.

Raycasting per l'eliminazione delle superfici invisibili

Il **raycasting** è un meccanismo image-precision che viene a cavallo tra il processo di proiezione e di colorazione.

L'intero processo fa riferimento ad un **centro di proiezione** e ad una **window** posizionata su un view plane arbitrario e pensata come una griglia regolare, i cui elementi corrispondono hanno dimensioni pari alla dimensione dei pixel della risoluzione desiderata.



Essenzialmente, con questo metodo noi tracciamo dei raggi immaginari di luce, dal centro di proiezione (**occhio dell'osservatore**) verso gli oggetti presenti nella scena, uno per il centro di ogni pixel della scena.

Il colore di ogni pixel è settato a quello dell'oggetto più vicino in corrispondenza del punto di intersezione tra il raggio e gli oggetti trovati lungo il percorso dello stesso raggio.

Il test di intersezione va fatto con TUTTI gli oggetti della scena (anche con quelli che non intersecano il raggio, proprio perché dobbiamo scoprire se il raggio lo interseca o no lol. Tuttavia, esistono dei metodi che possiamo utilizzare per ottimizzare).

Algoritmo di raycasting – pseudocodice

- Seleziona il centro di proiezione e la finestra sul piano di vista.
- For (ogni scan line nell'immagine)
 - For (ogni pixel nella scan-line)
 - Determina il raggio dal centro di proiezione che passa attraverso il pixel;
 - For (ogni oggetto nella scena)
 - Se l'oggetto è intersecato dal raggio ed è più vicino all'osservatore registra l'intersezione e il nome dell'oggetto

Definisce il colore del pixel come quello dell'intersezione con l'oggetto più vicino

Il cuore di un algoritmo di ray-casting è la determinazione dell'intersezione di un raggio con un oggetto. Per gli oggetti della scena è possibile scrivere delle **routine di intersezioni** con il raggio. (siccome gli oggetti della scena possono essere costituiti da un insieme di poligoni planari, volumi poliedrici o volumi definiti da superfici parametriche quadratiche o polinomiali).

Dal momento che un algoritmo di ray-casting impiega il 75-95% dei suoi calcoli per la individuazione delle intersezioni, l'efficienza della routine di intersezione pesa sull'efficienza dell'algoritmo (individuare l'intersezione di una linea arbitraria nello spazio con un particolare oggetto può essere dispendioso).

Intersezione raggio-sfera

Normalmente, per esaminare se avviene un intersezione fra il raggio e un oggetto, ci basta calcolare il sistema di equazioni contenenti le equazioni che definiscono l'oggetto e l'equazioni con le equazioni che definiscono la retta.

Tuttavia, esistono alcuni casi in cui tali calcoli sono facilitati grazie alla forma particolare dell'oggetto;

un esempio tipico è quello della sfera di centro (a, b, c) e di raggio r :

$$(x - a)^2 + (y - b)^2 + (z - c)^2 = r^2 \rightarrow \text{da cui } x^2 - 2ax + a^2 + y^2 - 2by + b^2 + z^2 - 2cz + c^2 = r^2$$

Sostituendo, al posto di x, y e z i valori

$$x = x_0 + t * \Delta x,$$

$$y = y_0 + t * \Delta y,$$

$$z = z_0 + t * \Delta z$$

Si ha

$$((\Delta x)^2 + (\Delta y)^2 + (\Delta z)^2)t^2 + 2t[\Delta x(x_0 - a) + \Delta y(y_0 - b) + \Delta z(z_0 - c)] + (x_0 - a)^2 + (y_0 - b)^2 + (z_0 - c)^2 - r^2 = 0$$

È una forma quadratica in t : se non esistono soluzioni reali allora il raggio e la sfera non si intersecano; nel caso in cui invece queste soluzioni esistano il risultato dipende dal numero delle stesse.

Se ne esiste una sola, raggio e sfera si sfiorano solamente; se le soluzioni sono due allora esiste una vera e propria intersezione ed i punti ottenuti non sono altro che la posizione delle intersezioni: una d'ingresso e una d'uscita. Quella che corrisponde al valore di t più piccolo è l'intersezione più vicina all'osservatore.

Intersezione di un raggio con un poligono

L'intersezione con questo oggetto 2D viene eseguita in due passi:

- Determinare l'intersezione del raggio con il piano contenente il poligono.
- Controllare se il punto di intersezione è interno o meno al poligono.

Data quindi l'equazione del piano del poligono:

$$Ax + By + Cz + D = 0$$

e sostituendo come nel caso precedente otteniamo:

$$A(x_0 + t\Delta x) + B(y_0 + t\Delta y) + C(z_0 + t\Delta z) + D = 0 \quad \text{da cui} \quad t = -\frac{Ax_0 + By_0 + Cz_0 + D}{A\Delta x + B\Delta y + C\Delta z}$$

A questo punto se il denominatore dell'equazione sopra è nullo, raggio e piano sono paralleli e pertanto non si intersecano; altrimenti vuol dire che esiste l'intersezione con il piano.

Ora dobbiamo vedere dove cade questa intersezione, se internamente o esternamente al poligono.

Per verificare ciò è sufficiente proiettare poligono e punto in modo ortogonale su uno dei tre piani che definiscono il sistema di coordinate, e verificare in 2D con algoritmi ad hoc se le coordinate del punto sono interne al poligono.

Efficienza del ray-casting

Un grosso problema relativo al ray-casting è dovuto all'efficienza. Infatti, anche la versione più semplice richiede di intersecare ogni raggio con ogni oggetto nell'ambiente.

Questo processo può diventare molto oneroso nel caso in cui gli oggetti siano numerosi.

L'unica cosa possibile per migliorarne l'efficienza è cercare di velocizzare i calcoli d'intersezione o cercare di evitarli in assoluto quando possibile.

Ottimizzazione delle intersezioni

Per ottimizzare il calcolo delle intersezioni, possiamo fare uso dei **bounding volumes**, che diminuiscono l'ammontare del tempo speso nei calcoli di intersezione.

Un oggetto, che è particolarmente difficoltoso da testare per le intersezioni, può essere racchiuso in un bounding volume, per il quale il test dell'intersezione è meno costosa.

Con l'uso dei bounding volumes, l'oggetto non deve essere testato se il raggio fallisce l'intersezione con il bounding volume che lo contiene.

I bounding volumes più comuni sono sfere, rettangoli solidi e ellissoidi (ma soprattutto le sfere).

Determinare se un raggio interseca una sfera è molto semplice: se la distanza tra il centro della sfera ed i raggio di luce è maggiore del raggio della sfera, allora non c'è intersezione: il raggio non interseca l'oggetto inscritto nella sfera.

Il test della sfera circoscritta si riduce alla determinazione della distanza di un punto da una retta.

Slabs

Kay e Kajiva hanno suggerito l'uso di un bounding volume che è formato da un poliedro convesso definito

dall'intersezione di 3 coppie di piani paralleli, una per ogni dimensione. Ciascuna di queste coppie è detta **slab**.

In questo modo delimitiamo ogni dimensione, creando così un bounding volume.

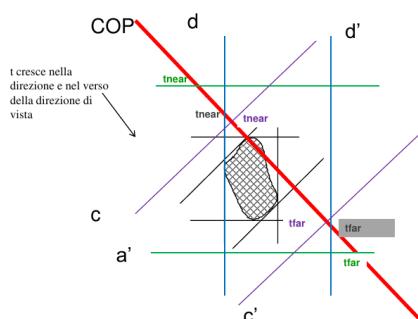
Dunque, quello che otteniamo è un simil parallelepipedo, che però è più "fine" e vicino alla forma dell'oggetto originale.

Essendo dei piani paralleli, ogni slab è rappresentata dall'equazione del piano:

$$Ax+By+Cz+D=0$$

Dove A, B, C sono costanti per ciascuno dei due piani che definisce lo slab e D può assumere il valore **Dnear** oppure **Dfar**.

Possiamo anche avere più coppie di piani per ogni dimensione, ottenendo così qualcosa del genere:



Cartman contiene 100,000 poligoni

Intersezione Raggio-Cartman = 100,000 intersezioni

raggio-poligono

Anche se il raggio non interseca Cartman

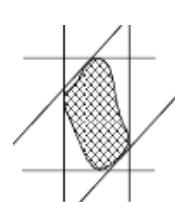
Soluzione

Circoscrivere Cartman con una sfera

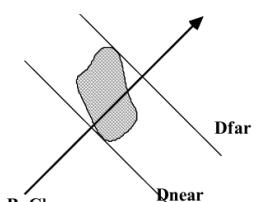


Se il raggio non interseca la sfera, allora il Raggio non interseca Cartman

Ho letteralmente aggiunto questa foto solo perché la prof ha messo un personaggio di South Park. È una serie tv estremamente divertente e scurrile e mi fa molto ridere. La consiglio a tutti. Oh mio dio hanno ammazzato Kenny, brutti bastardi!!! Ora c'è pure una stagione sul covid. Assurdo. Amo south park.



Una coppia per dimensione, come si può notare da questa figura

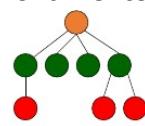
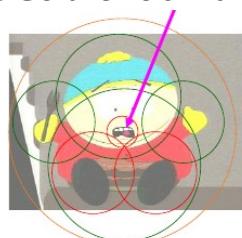


Questa coppia di piani paralleli va a delimitare una dimensione del mio oggetto

Gerarchie di Partizionamento Spaziale

L'ideale per il ray-casting sarebbe fare in modo che si evitino direttamente i calcoli di intersezione con gli oggetti che non vengono intersecati. Inoltre sarebbe bello che ogni raggio fosse testato solo con l'oggetto la cui intersezione con il raggio è più vicina all'origine del raggio.

Sono state formulate varie tecniche allo scopo di limitare il numero di intersezioni che devono essere realizzate: **Gerarchie di Partizionamento Spaziale**.



YEEEah altri esempi con Cartman! Se questo non è un momento di godimento, non so cos'altro sia.

In questo modo, i bounding volume possono essere organizzati in gerarchie annidate, in maniera tale che ogni nodo interno sia il bounding volume dei suoi figli, mentre le foglie corrispondono ai singoli oggetti.

Dunque, potrò andare a testare i figli solo se i padri sono stati colpiti.

Spatial partitioning

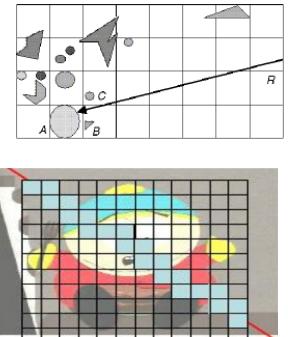
[abbiamo già visto questa tecnica [qui](#)].

Un'altra tecnica per ottimizzare il test delle intersezioni è quella di partizionare la scena in una griglia rettangolare di volumi della stessa dimensione.

Ad ogni cella viene assegnata una lista di oggetti interamente o parzialmente contenuti in essa.

In questo modo, un raggio deve essere intersecato solo con quegli oggetti che sono contenuti dentro le celle in cui passa.

Poiché le partizioni vengono esaminate in base all'ordine con cui il raggio le incontra, non appena ne trova una in cui esiste un'intersezione con l'oggetto, nessun'altra viene esaminata (questa modalità viene detta front-to-back).



Le caselle illuminate sono quelle che vengono attraversate dal raggio, e in si fa il test di intersezione solo con gli oggetti situati in queste caselle. (ommiddio hanno ammazzato cartman! Brutti bastardi!)

Antialiasing

Il ray casting analizzato fino a questo punto utilizza dei punti campionati su una griglia regolare, producendo il fenomeno dell'aliasing. Whitted, per ovviare a questo inconveniente, ha proposto un metodo adattivo che utilizza più raggi dove è richiesta una maggior precisione.

In tal caso i raggi non vengono più fatti passare per il centro del pixel, ma per i suoi vertici: invece di un unico raggio se ne considerano quattro.

Si calcolano i colori nei quattro vertici del pixel e, se questi valori non differiscono molto tra di loro, il colore del pixel viene calcolato come media pesata dei suoi vertici; in caso contrario il pixel viene suddiviso in quattro sottopixel ed i calcoli vengono eseguiti in funzione dei nuovi vertici.

Il processo procede fino a quando i valori non differiscono più di molto oppure si è raggiunta una certa profondità prefissata. Il pixel viene reso mediando opportunamente tutti i valori calcolati nei vari passaggi.

Raytracing (RTX ON) e Radiosity

Il modello di rendering che abbiamo finora studiato è detto **locale**, siccome ogni primitiva è trattata indipendentemente dalle altre. Il concetto di scena, la presenza di altri oggetti nella scena, interessano solo il programmatore e non il renderer.

Dunque, il modo con cui viene visualizzato un poligono dipende solo dalle caratteristiche del poligono e del raggio di luce che lo colpisce direttamente.

Questo tipo di modello ha molti vantaggi:

- Semplicità
- Parallelismo a livello di primitiva
- Costo costante per primitiva.

Nel mondo reale, però, la situazione è diversa. Per esempio:

- Se un oggetto blocca la luce proveniente da una sorgente, gli oggetti al di là di esso restano in ombra.
- Se un oggetto è riflettente, la luce che si riflette da esso illumina gli altri oggetti.

Un modello di illuminazione locale non può produrre nessuno di questi effetti.

Il modello di illuminazione di Phong, anche se accoppiato con la tecnica di shading di Phong (che è il massimo che possiamo ottenere fin'ora), non permette la rappresentazione visiva corretta di alcuni fenomeni:

- Luci ed osservatori posizionati all'interno della scena.
- Valutazione delle componenti emissive e trasmissive della radiazione luminosa.

Dunque, un modello di illuminazione globale è invece il **Raytracing**, che mette assieme eliminazione delle superfici nascoste e shading per gestire le ombre, la riflessione e la rifrazione. Così la riflessione e la trasmissione speculare globale completano l'illuminazione locale di tipo ambientale, speculare, diffusa (possiamo vederlo come una evoluzione del raycasting, a cui aggiungiamo la gestione della riflessione, della rifrazione e delle ombre). Un altro modello globale è quello della **radiosity**, che separa la fase di shading dalla fase di eliminazione delle superfici nascoste. Modella tutte le interazioni dell'ambiente con le sorgenti luminose prima in una fase indipendente dalla vista, poi calcola una o più immagini dal punto di vista desiderato usando uno tra i classici algoritmi di eliminazione delle superfici visibili e di shading interpolativo.

Distinzione tra Ray-tracing e Radiosity

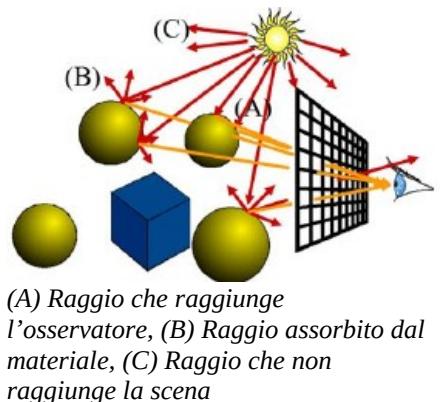
- **Raytracing** è un algoritmo **view-dependent**. Gli algoritmi view-dependent discretizzano il piano di vista per determinare i punti in cui valutare l'equazione di illuminazione, fissata la direzione di vista. Gli algoritmi view-dependent sono adatti per trattare fenomeni speculari che sono altamente dipendenti dalla posizione dell'osservatore, ma realizzano lavoro extra quando modellano fenomeni diffusivi, che cambiano poco su grandi aree dell'immagine.
- **Radiosity** è un algoritmo **view-independent**. Gli algoritmi view-independent discretizzano l'ambiente e lo elaborano allo scopo di fornire abbastanza informazione per valutare l'equazione dell'illuminazione in ogni punto e da ogni direzione di vista. Al contrario dei view-dependent, gli algoritmi view-independent modellano efficientemente il fenomeno **diffusivo**, ma richiedono eccessive quantità di memoria per catturare abbastanza informazione riguardo ai fenomeni speculari.

Algoritmo del raytracing e ray-casting shadows

Come abbiamo detto prima, possiamo estendere l'algoritmo del raycasting per l'eliminazione delle superfici nascoste in modo da gestire ombre, le riflessioni, le rifrazioni, generando così il metodo del raytracing.

Il metodo del **raytracing** è basato sull'osservazione che, di tutti i raggi luminosi che partono da una sorgente, i soli che contribuiscono all'immagine sono quelli che raggiungono l'osservatore.

I raggi luminosi possono raggiungere l'osservatore sia direttamente, sia per effetto delle interazioni con le altre superfici.



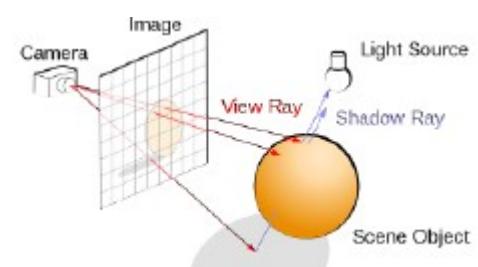
Tuttavia, la maggior parte dei raggi che lascerà la sorgente non raggiungerà l'osservatore e dunque non contribuirà all'immagine. Dunque, determiniamo i raggi che contribuiscono all'immagine invertendo la traiettoria dei raggi e considerando solo quelli che partono dall'osservatore. Questa è l'idea alla base del metodo Ray-tracing, che simula all'indietro il cammino compiuto dalla radiazione luminosa per giungere all'osservatore.

Il processo di lanciare i raggi dall'occhio verso la sorgente di luce, per disegnare un'immagine, viene chiamato a volte backwards ray tracing (o "ray tracing inverso"), dal momento che i fotoni viaggiano in senso opposto a quello usuale.

Poiché si deve assegnare un colore ad ogni pixel, si deve considerare almeno un raggio luminoso per ogni pixel. Questo raggio è detto **raggio primario** (o view-ray) del pixel.

Ciascun raggio primario può intersecare o una superficie, o una sorgente luminosa, o può andare all'infinito senza intersezioni, e in questo caso verrà assegnato il colore dello sfondo.

Per determinare la gradazione di colore (shading) del punto di intersezione del raggio con una superficie è necessario applicare un modello di illuminazione e di shading scelto tra quelli descritti precedentemente (es. [Phong shading](#)).



Per gestire le ombre, il ray-tracing fa uso di **shadow rays**, che essenzialmente partono dal punto di intersezione fra la superficie dell'oggetto e il view-ray e arrivano fino ciascuna delle sorgenti luminose. In questo modo, se ci sarà un oggetto che ti interseca lo shadow ray, vorrà dire che la sorgente luminosa non contribuirà all'illuminazione in quel punto. L'intensità (colore) del pixel viene calcolata dall'applicazione del modello di illuminazione di Phong nel punto intersezione commando il contributo di ogni sorgente luminosa visibile dal punto di intersezione (ignorando quindi il contributo della sorgente luminosa oscurata).

Un modello di illuminazione più semplificato del raytracing si chiama **Shadow Raycasting** (o Raycasting Shadows) [ovviamente non c'entra con il raycasting visto prima] che gestisce sia l'illuminazione diretta che la visibilità delle superfici e le ombre (ma senza gestire riflessioni o rifrazioni). Le ombre vengono gestite come detto prima.

Con il Shadow Raycasting, la formula dell'illuminazione di Phong diventa:

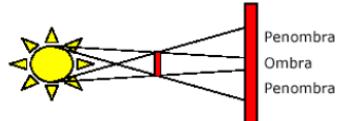
$$I_r = I_a k_a + \sum_{i=1}^m S_i f_{att_i} I_{p_i} (k_d (N \cdot L_i) + k_s (R_i \cdot V)^n)$$

Il parametro S_i vale 0 o 1 a in base al fatto che la sorgente sia bloccata o no.

Il prezzo che si deve pagare per l'introduzione delle ombre è quello di dover sparare un raggio ombra per ogni punto della superficie che abbiamo determinato essere visibile e per ogni luce. Dunque si ha una maggiore complessità computazionale.

Limiti dello Shadow Raycasting

Il modello di illuminazione permette solo delle ombre nette. Infatti, lo Shadow Raycasting assume che la sorgente di luce sia puntiforme, cosa inesistente in natura, perciò ci sarà una assenza di penombra.



Se si usano più sorgenti di luce però, si possono modellare sorgenti di luce non puntiformi. Tuttavia, in questo modo il numero di raggi ombra aumenta e quindi aumenta anche il tempo di rendering.

Riflessi fatti col Raytracing

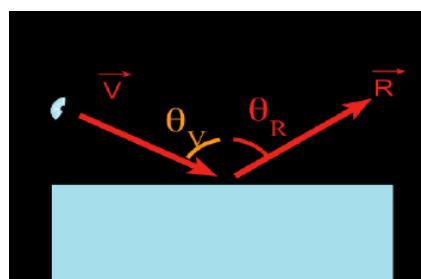
Fin'ora si sono considerati raggi luminosi che hanno origine dalla risorsa luminosa colpiscono una superficie e si riflettono verso l'osservatore.

In realtà la luce compie percorsi molto più articolati, rimbalzando in scena molte volte prima di raggiungere l'osservatore grazie ad oggetti riflettenti presenti, e apportando così un contributo luminoso su tutti gli oggetti che vengono incontrati nel percorso dei raggi (inter-riflessioni). Per tener conto almeno parzialmente di questo effetto, ad ogni intersezione raggio-oggetto viene generato un nuovo raggio (raggio riflesso) nella direzione della riflessione speculare perfetta.

I raggi riflessi 'rimbalzano' da superficie a superficie contribuendo all'intensità luminosa di ciascuna superficie che intersecano, finché intersecano una risorsa luminosa o raggiungono un massimo numero di 'rimbalzi'.

Questi calcoli sono fatti **ricorsivamente** e devono tener conto dell'attenuazione dovuta alla distanza, cioè dell'assorbimento della luce da parte delle superfici.

La direzione di riflessione è gestita esattamente come se fosse il raggio dall'osservatore (raggio primario).



La direzione del raggio che stiamo seguendo è determinata a partire dal vettore v (che proviene o direttamente dall'osservatore o in generale da un qualunque altro punto di intersezione con altri oggetti) e dalla normale n alla superficie nel punto.

Limiti delle riflessioni col raytracing

Il calcolo degli effetti di riflessione ha un **costo** (si spara un altro raggio) e un limite:

- Modella accuratamente solamente superfici perfettamente lisce parzialmente riflettenti.
- Nella realtà la maggior parte delle superfici non sono perfettamente lisce, ma riflettono in un insieme di direzioni (il riflesso è sfocato).
- Si può modellare sparando molti raggi (è molto costoso).

Lezione 13/12/2021 – Lezione su Blender

Non penso metterò appunti sulle lezioni su Blender, a parte su alcune cose per cui magari farò una sezione a parte: una miniguida per blender. Lo farei più per me che per gli altri però.

Lezione 14/12/2021 – Rifrazione raytracing, cenni radiosity

Gestione Rifrazione con raytracing

Il ray tracing è in grado di gestire sia superfici riflettenti che superfici che lasciano passare parzialmente o totalmente la luce.

La luce che colpisce un oggetto trasparente/ semitrasparente viene in parte trasmessa (i raggi sono **rifratti**).

Cos'è la rifrazione?

Quando i raggi luminosi passano da un mezzo "trasparente" a un altro, cambiano direzione subendo una deviazione della loro traiettoria.

Quando la luce viaggia nel vuoto, viaggia alla velocità della luce, indicata con la costante c .

Ma quando viaggia attraverso un altro mezzo, la sua velocità diminuisce.

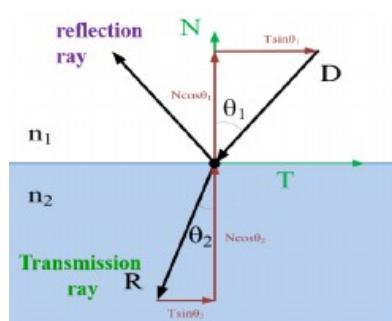
Se denotiamo la velocità della luce in questo mezzo con v , l'**indice di rifrazione** (IOR) è il rapporto di c e v .

$$\eta = \frac{c}{v}$$

Per fare un esempio: la luce viaggia più velocemente nell'acqua che nel vetro, ma più lentamente che nell'aria (l'aria ha un indice di rifrazione molto vicino a 1 e in CG trattiamo quasi sempre l'aria come se fosse un vuoto).

La rifrazione è gestita dalla **legge di Snell**. Per capire meglio, consideriamo questo situazione: abbiamo due mezzi trasmissivi con indice di rifrazione η_1 (in alto) e η_2 (in basso) in contatto tra loro attraverso una superficie, che viene chiamata **interfaccia** (linea orizzontale in figura).

Il raggio incidente, il raggio rifratto e la normale alla superficie di separazione dei due mezzi, nel punto di incidenza, giacciono sullo stesso piano.



D – direzione del raggio incidente
N – normale unitaria alla superficie
R – direzione raggio rifratto
T – tangente unitaria alla superficie
(nel piano N e D)

La **legge di Snell** è data dalla seguente formula:

$$\eta_1 \sin(\theta_1) = \eta_2 \sin(\theta_2)$$

E da essa riusciamo a descrivere le modalità di rifrazione di un raggio luminoso nella transizione tra due mezzi con indice di rifrazione diverso.

Se il primo mezzo è meno rifrangente del secondo, allora $\eta_1 < \eta_2$ e, di conseguenza:

$$\frac{\eta_2}{\eta_1} > 1 \rightarrow \frac{\sin(\theta_1)}{\sin(\theta_2)} > 1 \rightarrow \sin(\theta_1) > \sin(\theta_2) \rightarrow \theta_1 > \theta_2$$

Quindi il raggio rifratto si avvicina alla normale alla superficie. Altrimenti, se il primo mezzo è più rifrangente del secondo, allora $\eta_1 > \eta_2$ e, di conseguenza:

$$\frac{\eta_2}{\eta_1} < 1 \rightarrow \frac{\sin(\theta_1)}{\sin(\theta_2)} < 1 \rightarrow \sin(\theta_1) < \sin(\theta_2) \rightarrow \theta_1 < \theta_2$$

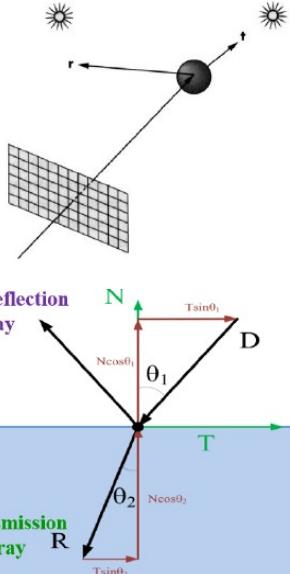
Quindi il raggio rifratto si allontana dalla normale alla superficie.

La rifrazione è anche il motivo per cui se un osservatore guarda in direzione di un oggetto in acqua, lo vedrà a una profondità inferiore rispetto a quella reale, a causa del cammino dei raggi luminosi che, provenendo dall'acqua, che ha un indice di rifrazione maggiore di quello dell'aria, si rifrangono allontanandosi dalla normale alla superficie di separazione dei due mezzi.

Gestione Rifrazione tramite raytracing

Per gestire la trasparenza degli oggetti con ray tracing ad ogni intersezione raggio-oggetto semitrasparente viene generato un nuovo raggio nella direzione della trasparenza (secondo la legge di rifrazione) e calcolato il contributo trasmesso.

Ogni volta che un raggio colpisce una superficie produce, in generale, un raggio riflesso ed uno rifratto.



Vediamo ora geometricamente come la rifrazione venga simulata attraverso il raytracing. Sia R il raggio rifratto. Proiettiamo R su $-T$ (sia a la proiezione) e su $-N$ (sia b la proiezione). Otteniamo R in questo modo:

$$R = a + b = -T \sin(\theta_2) - N \cos(\theta_2)$$

È una somma vettoriale in questo caso.

Ora, se non conosciamo T , possiamo ricavarlo usando questa formula:

$$T = \frac{-D - N \cos(\theta_1)}{\sin(\theta_1)}$$

Possiamo poi calcolare R così quindi:

$$R = (D + N \cos(\theta_1)) \frac{\sin(\theta_2)}{\sin(\theta_1)} - N \sqrt{1 - \sin^2(\theta_2)}$$

Per le leggi di Snell sappiamo poi che:

$$\eta_1 \sin(\theta_1) = \eta_2 \sin(\theta_2) \quad \sin(\theta_2) = \frac{\eta_1}{\eta_2} \sin(\theta_1) \quad \frac{\sin(\theta_2)}{\sin(\theta_1)} = \frac{\eta_1}{\eta_2}$$

Otteniamo così:

$$R = (D + N \cos(\theta_1)) \frac{\eta_1}{\eta_2} - N \sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - \cos^2(\theta_1))}$$

Notare come qui l'angolo sia θ_1 e non θ_2

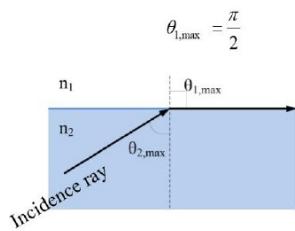
Infine, poiché $\cos(\theta_1) = N \cdot D$, la formula finale sarà:

$$R = D \frac{\eta_1}{\eta_2} + N \frac{\eta_1}{\eta_2} (N \cdot D) - N \sqrt{1 - \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - (N \cdot D)^2)}$$

Se il radicando è negativo, allora non c'è rifrazione.

Si parla di **total internal reflection**.

Questo accade quando la luce tenta di passare da un mezzo più rifrangente (vetro) a un mezzo meno rifrangente (aria) con un angolo basso.



Albero del ray-tracing

Il raytracing è un algoritmo ricorsivo, con un processo che per ogni pixel, esegue queste operazioni in modo ricorsivo:

- Trova la prima intersezione del raggio con la scena; se il raggio non colpisce alcun oggetto, viene restituito il "colore di sfondo"; se il raggio interseca la luce, viene restituito il colore della luce;
- Genera raggi d'ombra alle fonti luminose
- Calcola l'illuminazione di quel punto in base al modello di illuminazione locale
- Genera un raggio di riflessione **r** e un raggio di trasmissione **t** appropriatamente
- Richiama se stesso ricorsivamente con **r** e **t** (ovvero richiama sé stesso, non più con la direzione dell'occhio, ma con la direzione del raggio di trasmissione e del raggio di riflessione).
- Combina i contributi dell'illuminazione risultanti ed esce

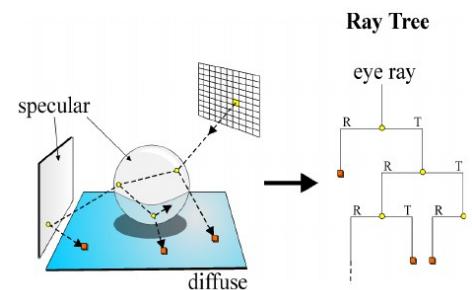
Ogni volta che usiamo il raytracing, costruiamo **un albero binario**.

Nell'albero binario ogni nodo corrisponde all'intersezione tra un raggio e una superficie. Ogni nodo può avere un ramo per il raggio di riflessione ed un ramo per il raggio di rifrazione.

Come si può notare dall'immagine qui a fianco, il raggio appena incontra una superficie diffusiva, diventa una foglia dell'albero (la prima foglia a sinistra).

Per determinare i colori nel punto dello schermo, si ripercorre l'albero dalle foglie alla radice, accumulando i contributi del modello locale di illuminazione di Phong a ciascuno dei punti intersecati.

Quindi il modello di illuminazione di Phong viene usato localmente per calcolare l'illuminazione in un punto, però vengono sommati tutti i contributi risalendo dalle foglie alle radici.



A destra abbiamo la matrice dello schermo, da cui parte un raggio. Questo in parte viene riflesso, e in parte rifratto dalla sfera di vetro, dando vita così ad altri due raggi. Uno dei due raggi incontra una superficie diffusiva e quindi si ferma, siccome non riflette.

L'equazione dell'illuminazione tramite raytracing sarà quindi:

$$I = I_{local} + K_r I_{reflect} + K_t I_{transmit}$$

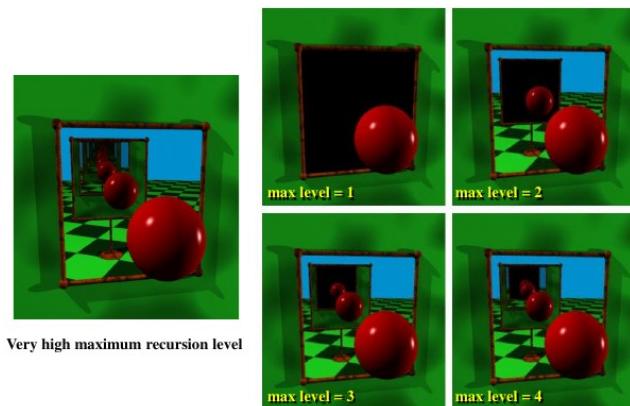
dove:

- K_r in $[0,1]$: fattore che specifica quale frazione della luce dalla direzione della riflessione viene riflessa;
- K_t in $[0,1]$ proprietà materiale, specifica la frazione della luce trasmessa attraverso la superficie
- $I_{reflect}$: intensità della luce riflessa in entrata, calcolata in modo ricorsivo nella direzione del raggio di riflessione;
- $I_{transmit}$: intensità della luce riflessa in entrata, calcolata in modo ricorsivo nella direzione di trasmissione;
- Illuminazione I_{local} calcolata mediante modello di illuminazione locale (Phong)

Il criterio di stop del nostro raytracing ricorsivo può essere dato da:

- dopo un certo numero di rimbalzi

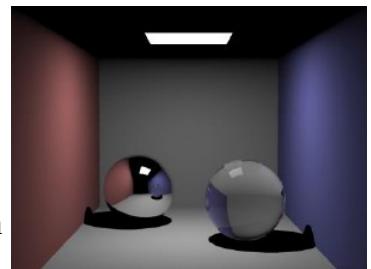
- se i contributi della riflessione e rifrazione diventano troppo piccoli.



Il metodo è tuttavia caratterizzato da un elevato costo computazionale, dovuto principalmente al calcolo delle intersezioni. Il calcolo delle intersezioni è problematico per molti tipi di superficie, perché la maggior parte dei ray tracers trattano solo superfici piane o quadriche.

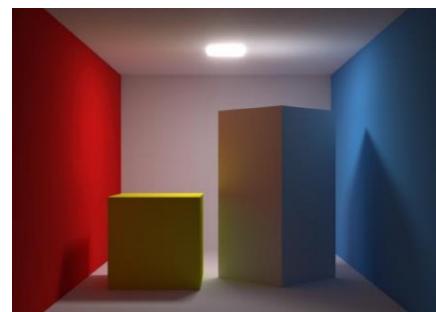
Il ray tracer usa il modello di Phong per includere un termine diffusivo al punto di intersezione tra un raggio ed una superficie. Tuttavia esso ignora la luce che viene distribuita per diffusione su quel punto, altrimenti dovrebbe considerare un tal numero di punti da diventare impossibile.

Perciò la tecnica del ray tracing è più adatta per ambienti altamente riflessivi.



Radiosity: cenni

Abbiamo visto come il raytracing sia un algoritmo di illuminazione globale view-dependent, siccome fissa il punto di vista (l'occhio dell'osservatore), poi determina le superfici visibili di quel pixel va a studiare l'effetto non solo della sorgente luminosa che colpisce l'oggetto, ma anche i rimbalzi dei raggi di luce. Tuttavia, questo algoritmo tratta bene solo i fenomeni caratterizzati da riflessione speculare, ma tratta male i fenomeni dove abbiamo superfici diffuse.

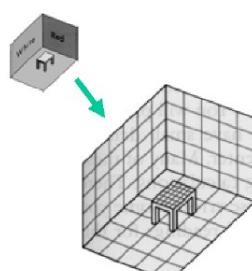


Il **modello del radiosity** è stato studiato proprio per superare questo difetto: il radiosity è stato ideato per approssimare le interazioni tra superfici dotate di riflessioni diffuse.

Gli algoritmi del radiosity determinano prima tutte le interazioni della luce in un ambiente, in maniera indipendente dalla posizione dell'osservatore. Poi una o più viste (immagini proprio) possono essere renderizzate utilizzando una tra le tecniche dell'eliminazione delle superfici nascoste ed una fra le tecniche di shading interpolativo.

Nel metodo del radiosity, la scena viene suddivisa in pezzi (patches), ovvero in molti poligoni piatti e di dimensioni limitate, ciascuno dei quali è considerato perfettamente diffusivo. Ciascun patch emette una propria luce, senza che ci sia una luce nella scena.

A questo punto, si vuole determinare come una patch che lascia una pezza va ad influenzare le altre pezzi, che viene descritto dai fattori di forma (**form factors**).



I form factor definiscono quanta parte dell'energia che esce da una patch arriva su un'altra patch, tenendo in considerazione occlusioni, orientamento delle patch, distanza.

Il calcolo dei form factor delle patch di una scena sarebbe quadratico, ma la maggior parte dei form factor sono praticamente nulli: patch lontane non si influenzano.

Le ipotesi che si fanno per il calcolo del radiosity sono:

- Conservazione dell'energia luminosa in un ambiente chiuso: tutta l'energia trasmessa o riflessa da qualsiasi superficie è uguale all'energia riflessa o assorbita da ogni altra superficie.
- Ogni superficie emette luce, così tutte le sorgenti luminose hanno un'area.

Supponiamo di suddividere la scena in un numero finito di n patches, ognuna delle quali è supposta avere un'area finita. Ognuna di queste patches quindi emette e riflette luce uniformemente sull'intera area.

Supponiamo poi che ogni patch sia emettitore e riflettore diffuso Lambertiano. La luce che è riflessa da una superficie è attenuata dalla reflectivity della superficie. Definiamo “reflectivity” il colore della superficie.

L'equazione della radiosity descrive la quantità di energia luminosa che può essere emessa da una superficie, come la somma dell'energia intrinseca della superficie e dell'energia che colpisce la superficie ricevuta da qualche altra superficie.

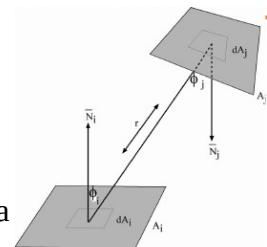
L'energia che lascia una superficie (superficie "j") e colpisce un'altra superficie (superficie "i") è attenuata da due fattori:

- Il **"form factor"** tra le superfici "i" e "j", che tiene conto della relazione fisica tra le due superfici.
- La **riflettività della superficie "i"** (quindi il suo colore), che assorbirà una certa percentuale dell'energia luminosa che colpisce la superficie.

Com'è fatto un form factor

Il "form factor" descrive la frazione di energia che lascia una superficie ed arriva su una seconda superficie. Tiene conto:

- della distanza tra le superfici, calcolata come distanza tra i centri di ognuna delle superfici
- del loro orientamento, calcolato come l'angolo tra il vettore normale alla superficie e un vettore disegnato dal centro di una superficie al centro dell'altra superficie.



Il form factor da un'area differenziale dA_i ad un'area differenziale dA_j è dato da:

$$F_{dA_i dA_j} = \frac{\cos \phi_i \cos \phi_j dA_j}{\pi r^2} \quad H_{ij} \quad H_{ij} = \begin{cases} 1 & \text{se } dA_j \text{ è visibile da } dA_i \\ 0 & \text{altrimenti} \end{cases}$$

Equazione del radiosity

$$\mathbf{B}_i = \mathbf{E}_i + \rho_i \sum_{j=1}^n \mathbf{B}_j \mathbf{F}_{ji} \frac{\mathbf{A}_j}{\mathbf{A}_i}$$

- Questa equazione mostra che l'energia che lascia un'unità di area della superficie è la somma della luce emessa dalla patch i (E_i) più la luce riflessa.
- La luce riflessa viene calcolata facendo la somma della luce incidente, che non è altro che la somma della luce che lascia un'unità di area della patch j-esima (A_j) e che raggiunge la patch i-esima. In particolare, F_{ji} è il **form factor**, e specifica la frazione di energia che lasciando la patch j raggiunge la patch i (dipende dall'orientazione delle due patch e dall'occlusione tra le patch).
- $B_j A_j$ è l'energia totale emessa dalla patch j con area A_j (radiosità per area), dove B_j è la radiosità di j.
- La luce riflessa totale viene poi scalata rispetto al coefficiente di riflessione della patch (ρ_i).

Quando calcoliamo la radiosity, quello che normalmente conosciamo/settiamo noi sono i form

factor (siccome li possiamo facilmente calcolare per ciascuna patch rispetto alle altre) e l'energia emessa E_i .

A questo punto quindi posso disporre l'equazione in questo modo:

$$\mathbf{B}_i - \rho_i \sum_{j=1}^n \mathbf{B}_j \mathbf{F}_{ij} = \mathbf{E}_i$$

E otteniamo un'equazione con incognite B_i e B_j . Se questa operazione lo facciamo per ogni patch, otteniamo un sistema di equazioni, che possiamo risolvere con una matrice.

$$\begin{bmatrix} 1-\rho_1 F_{11}, -\rho_1 F_{12}, \dots, -\rho_1 F_{1n}, \\ -\rho_2 F_{21}, 1-\rho_2 F_{22}, \dots, -\rho_2 F_{2n}, \\ \vdots \\ -\rho_n F_{n1}, -\rho_n F_{n2}, \dots, 1-\rho_n F_{nn}, \end{bmatrix} \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \\ \vdots \\ \mathbf{B}_n \end{bmatrix} = \begin{bmatrix} \mathbf{E}_1 \\ \mathbf{E}_2 \\ \vdots \\ \mathbf{E}_n \end{bmatrix}$$

Questa matrice è una matrice sparsa, siccome molti degli elementi sono = 0. Infatti, abbiamo detto che per degli elementi lontani, F_{ij} è nullo (o trascurabile).

Normalmente questa matrice viene risolta con il metodo di Gauss-Sidel.

La soluzione del sistema lineare porta ad un valore B_i di radiosity per ogni patch. Utilizzando il valore di radiosity, ogni patch può essere renderizzata da qualsiasi punto di vista con un qualsiasi algoritmo di eliminazione delle superfici visibili.

Cose in più che potrebbe chiedere – Nurbs e Shading oltre Phong

Curve Nurbs – spline razionali

Il modo più semplice per definire le curve Nurbs 2D è quello di vederle come la proiezione sul piano proiettivo (quello in cui tutti i punti hanno coordinate omogenee $(x,y,1)$) di una curva spline 3D.

Consideriamo i punti di controllo $P_i^w = \begin{pmatrix} w_i \cdot x_i \\ w_i \cdot y_i \\ w_i \end{pmatrix}$, con $i = 1, \dots, m+K$ ottenuti dai punti $P_i = (x_i, y_i)$ e

dai pesi w_i tali che $w_i > 0$ per $i = 1, \dots, m+K$.

Fissata una partizione nodale associata Δ di $[a,b]$ ed una partizione nodale estesa Δ^* consideriamo la curva spline 3D che ha punti di controllo P_i^w , con $i = 1, \dots, m+K$, la formula della nostra curva spline sarà:

$$C(t) = \frac{\sum_{i=1}^{m+K} w_i P_i N_{i,m}(t)}{\sum_{i=1}^{m+K} w_i N_{i,m}(t)} = \begin{cases} \frac{\sum_{i=1}^{m+K} w_i x_i N_{i,m}(t)}{\sum_{i=1}^{m+K} w_i N_{i,m}(t)} \\ \frac{\sum_{i=1}^{m+K} w_i y_i N_{i,m}(t)}{\sum_{i=1}^{m+K} w_i N_{i,m}(t)} \end{cases}$$

Esattamente come per la curva di Bezier razionali, aggiungiamo un denominatore con cui dividiamo la curva per w_i .

Possiamo definire le funzioni base di una curva Nurbs in questo modo (che è lo stesso che abbiamo visto per le curve di Bezier, qui).

$$R_{i,m}(t) = \frac{w_i N_{i,m}(t)}{\sum_{i=1}^{m+K} w_i N_{i,m}(t)}$$

Ottenendo così questa formula finale:

$$C(t) = \sum_{i=1}^{m+K} P_i R_{i,m}(t)$$

dove $R_{i,m}(t)$ sono funzioni Nurbs di base, cioè la base delle b-spline razionali. Queste funzioni base dipendono dalla partizione nodale e dai pesi w_i $i=1, \dots, m+K$.

La valutazione di una curva Nurbs può essere effettuata mediante l'algoritmo di De-Boor (che abbiamo visto [qui](#)) applicato ai punti P_i^w con $i = 1, \dots, m+K$, (similarmente si può realizzare

l'algoritmo di knot insertion). Si tratta quindi di utilizzare le coordinate dei punti P_i^w e alla fine di effettuare la proiezione della curva sul piano proiettivo $w = 1$.

Proprietà curve Nurbs

Le proprietà delle curve Nurbs sono **sicuramente quelle delle curve b-spline, con in più l'invarianza rispetto alle proiezioni prospettiche.**

Le proprietà infatti sono mantenute, in quanto le basi delle b-spline razionali $R_{i,m}(t)$ godono delle medesime proprietà della b-spline classica.

[se si sanno le proprietà delle spline, che possiamo trovare [qui](#), si può skippare questo paragrafo]
 Se $K = 0$, nella partizione nodale con nodi aggiunti abbiamo solo i nodi coincidenti agli estremi coincidenti e nessun nodo vero, mentre se $N = m$, la curva Nurbs coincide con una curva di Bezier razionale. Le curve Nurbs godono delle **proprietà di controllo locale**: alla modifica di un punto P_i o di un peso, si modificano solo gli span relativi a quel vertice di controllo.
 Esse godono delle proprietà di **strong convex-hull**.

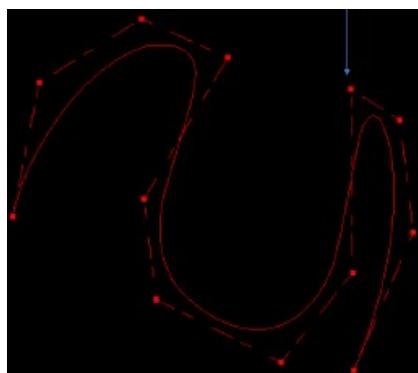
Se la partizione nodale estesa Δ^* ha nodi aggiuntivi agli estremi fra loro coincidenti, la curva Nurbs interpola il primo ed ultimo punto del poligono di controllo e sarà tangente al primo ed ultimo lato del poligono di controllo.

Il poligono di controllo rappresenta una approssimazione lineare delle curve Nurbs, approssimazione che può essere migliorata introducendo dei nodi.

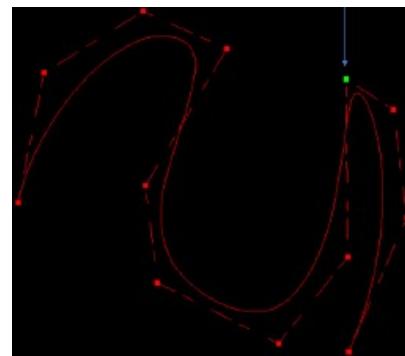
Le curve Nurbs sono invarianti per trasformazioni affini (e prospettiche appunto).

Esse consentono di **realizzare circonferenze** o altre figure geometriche intervenendo sul valore dei pesi w_i .

Anche qui, i pesi w_i rappresentano un ulteriore **strumento di modellazione** che influisce notevolmente sull'andamento della curva, e, ancora, i pesi modificano la nostra curva facendola "attrarre" sempre di più ai nostri vertici di controllo. Nel caso tutti i pesi siano pari ad 1, le funzioni base coincidono con le normali funzioni di base di Cox.



Curva con $w_i = 1$



Curva con $w_i = 2.3$

Proprietà delle funzioni base delle Nurbs $R_{i,m}(t)$

- Le formule $R_{i,m}(t)$ sono **positive** su tutto il loro supporto, $R_{i,m}(t) \geq 0$ per $t \in [t_i, t_{i+1}]$.
- Sono a **supporto compatto**, il che significa che sono diverse da 0 per $t \in (t_i, t_{i+m})$.
- Costituiscono una **partizione dell'unità**.

Il passaggio dalla curva omogenea 3D $C(t)$ alla sua proiezione $C(t)$ può creare qualche problema riguardo alla nozione di continuità sia parametrica che geometrica, che abbiamo visto sino a questo momento. È infatti necessario introdurre il concetto di continuità parametrica razionale (che abbiamo visto [qui](#)) e di continuità geometrica razionale, che sono generalizzazioni di concetti che già conosciamo.

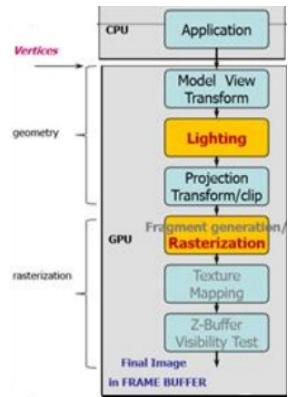
Shading vs illuminazione

Il **calcolo dell'illuminazione** viene fatto in coordinate di Vista, prima di applicare la trasformazione di Proiezione che porta nelle coordinate di clip e potrebbe alterare le normali. Viene dunque fatto **prima della rasterizzazione**.

Lo **shading**, invece, avviene nel **rasterization stage**.

Metodi di Shading

Con il termine di **shading** si intende il processo di determinare il colore di tutti i pixel che ricoprono una superficie usando un modello di illuminazione.



Il metodo più semplice consiste nel:

- Determinare per ogni pixel la superficie visibile.
- Calcolare la normale alla superficie in quel punto.
- Valutare l'intensità della luce ed il colore usando un modello di illuminazione.

Questo modello di shading è molto costoso.

Modelli di shading

Adesso descriveremo i modelli di shading più efficienti per superfici definite da poligoni o da mesh di poligoni (in generale, infatti, gli oggetti più complessi sono rappresentati mediante mesh poligonali).

Constant shading

Il modello di shading più semplice per un poligono è il **constant shading** noto anche come **flat shading**.



Si determina il valore dell'equazione dell'illuminazione (quindi si applica il modello di illuminazione) una volta sola per ogni poligono e poi questo valore si assegna a tutto il poligono.

Questo approccio è valido se sono vere diverse ipotesi:

- La sorgente luminosa è posta all'infinito, così $N \cdot L$ è costante per ogni faccia del poligono.
- L'osservatore è posto all'infinito così $L \cdot V$ è costante per ogni faccia del poligono.
- Il poligono rappresenta la superficie da modellare e non è un'approssimazione di una superficie curva.



Ho rimesso l'immagine in modo da ricordare di cosa stiamo parlando.

Se una delle prime due ipotesi non è verificata, allora per continuare ad usare questo tipo di shading è necessario un metodo per determinare un valore singolo per ognuno degli L e V. Per esempio, i valori possono essere calcolati per il centro del poligono, o per il primo vertice del poligono.

Questa tecnica di shading è **veloce** perché richiede pochi calcoli.

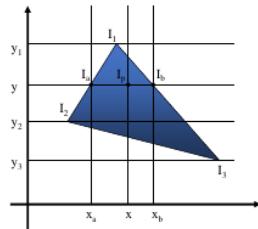
Se i poligoni sono molto piccoli (larghi quanto un pixel) quando vengono proiettati sullo schermo, allora il risultato è indistinguibile da quello di qualsiasi altra tecnica.

Solitamente si usa per avere una vista grossolana della scena: infatti il risultato visivo non è del tutto soddisfacente, in quanto lascia visibile la suddivisione tra i poligoni (effetto Mach Banding), senza rendere nell'immagine l'andamento della superficie approssimata geometricamente dalla mesh di poligoni.

Interpolated shading

Come alternativa a quella di valutare l'equazione dell'illuminazione in ogni punto del poligono, è stato introdotto l'uso dell'**interpolated shading**:

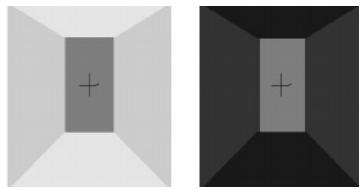
Si applica l'equazione dell'illuminazione ad ogni vertice del triangolo e questi valori vengono interpolati linearmente lungo i lati del triangolo.



Effetto Mach Banding

Se lo shading viene fatto indipendente su ogni poligono (sia esso costante o interpolato) si ha comunque una netta visibilità, non voluta, dei bordi tra due poligoni adiacenti, causati dalla brusca variazione della normale alla superficie.

A causa del cosiddetto effetto Mach banding anche una maggiore finezza della griglia dei poligoni non riduce la discontinuità di shading tra poligoni adiacenti. Questo effetto è quello per cui un oggetto messo vicino ad uno più chiaro risulta più scuro e messo vicino ad uno più scuro risulta più chiaro.



Gouraud Shading

Per ovviare a questo inconveniente si sono sviluppati dei metodi di shading che tengono conto delle informazioni date da poligoni adiacenti.

Uno di questi è il **Gouraud shading**, nel quale si tiene conto della geometria effettiva che si sta visualizzando: se la griglia di poligoni rappresenta una superficie curva, per ogni vertice della griglia non si utilizza la normale al poligono, ma la normale alla superficie.

In questa maniera il calcolo dello shading produce lo stesso valore su entrambi i lati di poligoni che hanno bordi in comune rendendo lo shading complessivo privo di salti.



Il metodo richiede che sia nota la normale alla superficie che si approssima in ogni vertice.

Se non è disponibile, la si approssima con la media delle normali ai poligoni che condividono il vertice.

Se uno spigolo deve essere effettivamente visibile (ad esempio lo spigolo di raccordo tra la superficie dell'ala e della fusoliera di un aereo) si generano due insiemi di normali su ciascuno dei due lati dello spigolo.

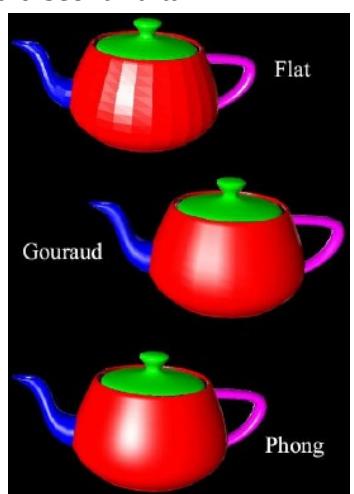
Una volta calcolate le normali in ogni spigolo, si applica il modello di illuminazione per calcolare il valore del colore nel vertice e si interpola linearmente, all'interno dei poligoni, con lo stesso procedimento descritto in precedenza (ovvero lo shading interpolato).

Efficienza del metodo di Gourad

Dal punto di vista implementativo il Gouraud shading è **mediamente efficiente**, poiché l'equazione di illuminazione va calcolata una volta sola per vertice.

Per poter individuare i vettori normali, necessari per poter calcolare la normale nei vertici, occorre una struttura dati che rappresenti l'intera mesh di poligoni.

Il risultato visivo prodotto dal Gouraud shading è perlopiù quello desiderato: non è visibile la suddivisione in mesh dei poligoni, e si ottiene una rappresentazione liscia e senza discontinuità della superficie approssimata.



Phong shading

Sebbene il Gouraud shading sia più che sufficiente per la maggior parte delle applicazioni, non risulta particolarmente realistico quando si vogliono rappresentare superfici dotate di un alto coefficiente di riflessione speculare. In questo caso il modello più sofisticato e più costoso computazionalmente è il **modello di Phong (Phong shading)** o di **interpolazione delle normali**.

In questo modello le normali nei vertici vengono calcolate nella stessa maniera con cui si calcolavano nel Gouraud Shading, ma il calcolo dello shading dei

pixel del poligono viene effettuato usando normali calcolate interpolando linearmente (e poi rinormalizzando) all'interno dello spazio delle normali.

Il Phong shading risulta più costoso in termini computazionali rispetto al Gourad shading: si interpolano vettori e non valori interi e l'equazione di illuminazione viene calcolata per ogni pixel.

Phong Shading (per-fragment illumination) (non ci interessa per la teoria)

Per l'implementazione del Phong-Shading, applicheremo alla normale di ogni vertice in coordinate dell'oggetto la trasposta dell' inversa della matrice model view (M^{-1})^T (dove $M = \text{View}^* \text{Model}$). La normale trasformata sarà data in output e rappresenterà quindi l'input del fragment shader.

Quindi le normali saranno interpolate sui lati della primitiva prima di arrivare nel fragment shader e sarà nel fragment shader che si calcolerà l'equazione dell'illuminazione e quindi il colore per ogni frammento della scena.

It's been a wild ride...

Finalmente abbiamo finito le lezioni. E io godo. Mi ci è voluto un bel po' per scrivere sti appunti, ma non penso sarà mai peggio di sistemi.

Until next time,

Angelo.

Cose in più che non chiede (ma che sono molto utili)

Mesh Poligonali

[up]

Textures

Possiamo rendere gli oggetti nella scena più realistici mappando delle texture su ciascuno dei poligoni della mesh. Possiamo applicare la texturing sia durante la fase di shading, che dopo la fase di shading e perturbandone così il risultato.

PBR e Texture

Sono esempi di texture:

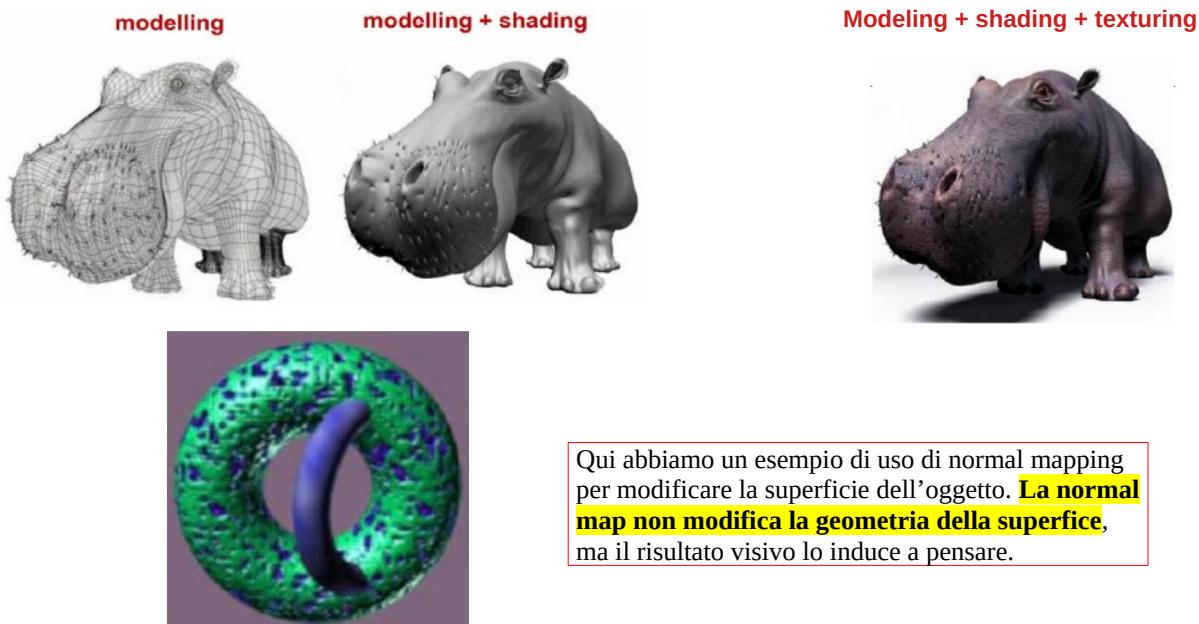
- **color-map** (o albedo): ogni pixel è un colore (componenti: R-G-B, o R-G-B-A). L'intensità di colore di un punto della superficie dipende dal colore del punto corrispondente della immagine di texture.
- **alpha-map**: ogni pixel è un valore di trasparenza alpha.
- **“normal-map” o “bump-map”** ogni pixel è una normale (componenti: X-Y-Z); le variazioni dei livelli di grigio della mappa daranno luogo a variazioni di altezza sulla superficie.
- **shininess-map**: ogni pixel contiene un valore di specularità.

Grazie alle texture, possiamo alcune proprietà dell'oggetto, tra cui:

- **Colore**: viene modificato il colore (diffuso o speculare) nel modello di illuminazione di Phong con il corrispondente colore dalla texture map;
- **Colore riflesso** (environment mapping): simula la riflessione di un oggetto speculare che riflette l'ambiente circostante; permette di creare immagini che apparentemente sembrano generate da un ray tracer (vedi [cubemaps](#)).
- **Perturbazione della normale** (bump mapping): perturbazione della normale alla superficie in funzione di un valore corrispondente in una funzione 2D;

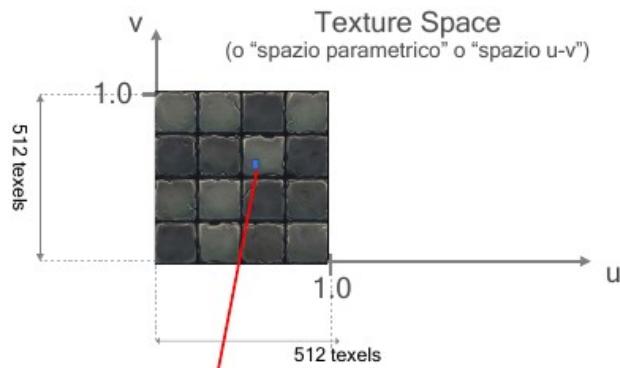
- Trasparenza: controllare l'opacità di un oggetto trasparente.

Ecco alcuni esempi visivi:



Notazione delle texture, e texture mapping

Una texture è definita nella regione $[0, 1] \times [0, 1]$ dello “spazio parametrico”, parliamo quindi di **Coordinate di Texture space** (o spazio u-v).

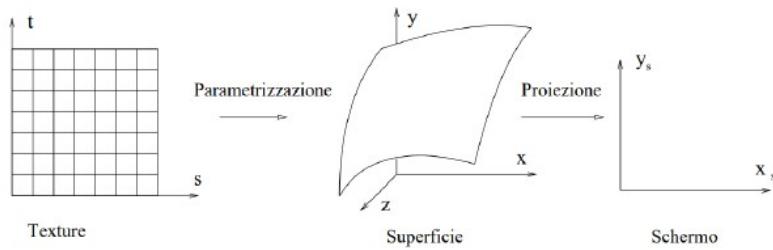


Consideriamo una texture come una matrice di $n \cdot m$ elementi, il texture mapping (o uv mapping) è una **funzione** matematica che trasforma le coordinate texture (u, v) in coordinate del mondo (X, Y, Z) e poi in coordinate schermo (x_s, y_s). Essenzialmente quindi:

- Associa i texel ai punti della superficie di un oggetto geometrico.
- Proietta i punti sullo schermo.

Dunque, è composta da 3 passi:

- Definire la texture nello spazio (u, v)
- Associare durante la fase di modellazione a ciascun vertice di ciascun triangolo ((X_i, Y_i, Z_i)) il corrispondente punto nello spazio (u, v)
- Calcolare durante la fase di rendering per ogni pixel generato all'interno di una faccia il valore di tessitura per interpolazione



Parametrizzazione delle texture – mappatura in un passo

Nella mappatura in un passo si definisce (in forma analitica) la funzione W che definisce la corrispondenza tra i pixel della texture ed i punti della superficie.

- Questo si può fare quando si ha la descrizione parametrica della superficie sottostante alla maglia poligonale.
- Altrimenti si specifica tabularmente la corrispondenza W^{-1} tra vertici della maglia e punti della texture.

Se la superficie è data in forma parametrica, ad ogni suo punto P sono associate due coordinate (parametri) (u, v) . Per ottenere W basta specificare la mappa che va da (u, v) sulla superficie a (s, t) nella texture.

È opportuno che W sia invertibile, quindi spesso è l'identità (con qualche fattore di normalizzazione).

- Ad esempio si consideri una sfera di raggio r e centro l'origine:

$$S(\theta, \phi) = \begin{cases} r \sin(\phi) \cos(\theta), & \theta \in [0, 2\pi] \\ r \cos(\phi) & \phi \in [0, \pi] \\ r \sin(\phi) \sin(\theta) \end{cases}$$

Si può scrivere come

$$S(u, v) = \begin{cases} r \sin(2\pi u) \cos(\pi v), & u, v \in [0, 1] \\ r \cos(2\pi u) & \\ r \sin(2\pi u) \sin(\pi v) \end{cases}$$

Parametrizzazione delle texture – mappatura in due passi

Una tecnica più generale, che si può usare senza conoscere l'equazione parametrica della superficie, è la **mappatura in due passi**.

Si mappa la texture su una superficie intermedia semplice, in modo che la parametrizzazione (corrispondenza punti-superficie con pixel-texture) sia immediata; questa prende il nome di S-mapping

- Quindi si mappa ogni punto della superficie intermedia in un punto della superficie in esame; questa prende il nome di O-mapping
- La concatenazione dei due mapping genera la corrispondenza W tra i pixel della texture ed i punti dell'oggetto
- Il primo passaggio (S-mapping) è in genere semplice; basta scegliere superfici facili da parametrizzare
- Ad esempio si puo` prendere come superficie intermedia un cilindro
- Oltre al cilindro è facile fare l'S-mapping con cubi, piani e sfere.
- In genere si considera la superficie intermedia come esterna all'oggetto da "tessitura".

[..]

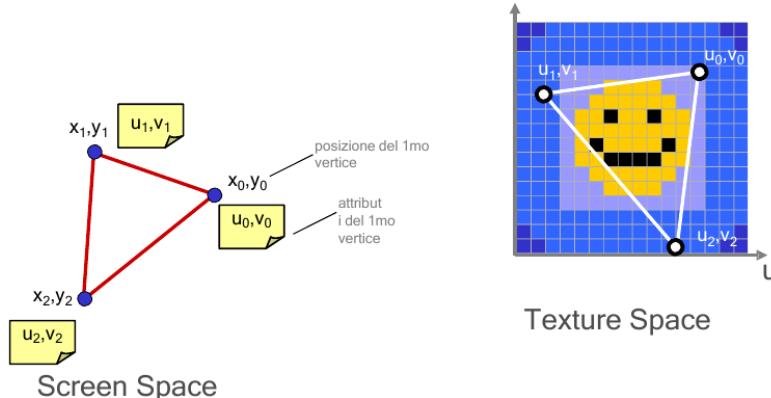
Proiezioni per il texture mapping

Come trasformare le coordinate di una mappa di texture 2D in una superficie 3D generica?

Si fa uso di:

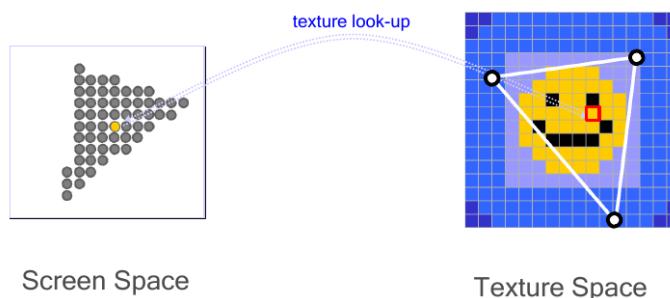
- Trasformazione piana (ad es. bilineare)
- Proiezione cilindrica
- Proiezione sferica

Ad ogni vertice (di ogni triangolo) si associa un punto di coordinate u,v nello spazio della texture.



Si definisce quindi un mapping fra il triangolo (poligono della mesh) e il triangolo della texture.

Dopodiché, ad ogni vertice di ogni triangolo si associa un punto di coordinate u,v nello spazio della texture (quando facciamo la proiezione dal modello con le texture al nostro schermo si intende, ovvero quando le renderizziamo).



Interpolazione delle coordinate texture
[...]

Come usare le texture in OpenGL

Per prima cosa, dobbiamo caricare un'immagine. Possiamo usare la libreria stb_image.h, che è un file single header che ci permette di caricare delle immagini.

Per usarla, dovremo specificare

```
#define STB_IMAGE_IMPLEMENTATION  
#include stb_image.h
```

Nel nostro file in cui la usiamo.

Dopodiché, dovremo effettivamente generare la texture.

Come a qualsiasi altro oggetto in OpenGL, ad una texture è associato un ID. Creiamone uno:

```
unsigned int texture;  
 glGenTextures(1, &texture);
```

glGenTextures prende come input quante texture vogliamo generare e le memorizza in un unsigned int array dato come secondo argomento (nel nostro caso solo una singola unsigned int, siccome vogliamo generare una singola texture).

Poi, dobbiamo attivare la texture:

```
glActiveTexture(GL_TEXTURE0);
 glBindTexture(GL_TEXTURE_2D, texture);
 glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
 GL_UNSIGNED_BYTE, data);
```

Ed ecco fatto. Abbiamo generato la texture.

Gli argomenti di `glTexImage2D` sono:

- Il primo argomento specifica il target della texture : `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D` (ovvero, se la texture è 1D, 2D, 3D, NON SE LO È LA SUPERFICIE A CUI VIENE APPLICATA. Esatto, possiamo avere anche texture 3d.)
- Il secondo argomento specifica il livello di mipmap per il quale vogliamo creare una texture nel caso in cui si desidera impostare manualmente ciascun livello di mipmap (poniamolo a zero).
- Il terzo argomento dice a OpenGL in quale tipo di formato vogliamo memorizzare la texture. La nostra immagine ha solo RGB valori, quindi memorizzeremo anche la texture con valori RGB.
- Il quarto e il quinto argomento impostano la larghezza e l'altezza della texture risultante.
- L'argomento successivo dovrebbe essere sempre 0 (per legacy).
- Il settimo e l'ottavo argomento specificano il formato e il tipo di dati dell'immagine sorgente. Abbiamo caricato l'immagine con RGB valori e li abbiamo archiviati come chars (byte), quindi passeremo i valori corrispondenti.
- L'ultimo argomento sono i dati dell'immagine reale.

Texture Wrapping

Le coordinate della texture di solito vanno da (0,0) a (1,1), ma cosa succede se specifichiamo coordinate al di fuori di questo intervallo?

Il comportamento predefinito di OpenGL è di ripetere le immagini delle textures (sostanzialmente ignoriamo la parte intera della coordinata delle textures in virgola mobile), ma ci sono più opzioni che OpenGL offre:

- `GL_REPEAT`: comportamento predefinito per le textures. Ripete l'immagine della texture.
- `GL_MIRRORED_REPEAT`: uguale a `GL_REPEAT` ma rispecchia l'immagine ad ogni ripetizione.
- `GL_CLAMP_TO_EDGE`: blocca le coordinate tra 0 e 1. Il risultato è che coordinate più alte vengono bloccate sul bordo, risultando in un modello di bordo allungato.
- `GL_CLAMP_TO_BORDER`: alle coordinate al di fuori dell'intervallo viene ora assegnato un colore del bordo specificato dall'utente.



Ognuna di queste opzioni può essere impostata per ogni asse delle coordinate (s, t [e r se si sta utilizzando 3D texture] equivalenti a x, y, z) con la funzione `glTexParameter`.

Filtraggio delle texture e mipmapping

Se si ha una texture a bassa risoluzione, OpenGL ci fornisce delle tecniche di filtraggio delle textures. Ad esempio:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

GL_LINEAR (noto anche come filtro (bi) lineare) prende un valore interpolato dai texel vicini della coordinata texture, approssimando un colore tra i texel, mentre GL_NEAREST fa uso di più texel in un pixel eseguendo la media fra essi.

OpenGL inoltre ci permette di creare manualmente delle **mipmap**.

Per capire cosa sono, supponiamo di avere una grande stanza con migliaia di oggetti, ognuno con una texture attaccata. Ci saranno oggetti lontani che hanno la stessa texture ad alta risoluzione, rispetto a quella che hanno gli oggetti vicini all'osservatore.

Ciò produrrà artefatti visibili su piccoli oggetti, per non parlare dello spreco della larghezza di banda della memoria usando textures ad alta risoluzione su piccoli oggetti.

Per risolvere questo problema OpenGL utilizza un concetto chiamato mipmap: si tratta di una raccolta di immagini di texture in cui ogni texture successiva è due volte più piccola rispetto alla precedente.

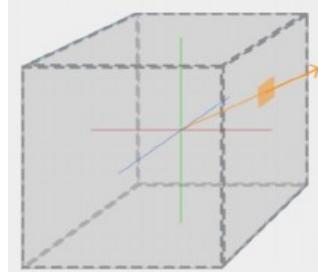
L'idea alla base delle mipmap è questa: dopo una certa soglia di distanza dal visualizzatore, OpenGL utilizzerà una diversa texture mipmap che si adatta meglio alla distanza dall'oggetto.

Poiché l'oggetto è lontano, la risoluzione più piccola non sarà evidente per l'utente. OpenGL è quindi in grado di campionare i texel corretti e c'è meno memoria cache durante il campionamento di quella parte delle mipmap.

Cube Maps

Una cube map è una texture che contiene 6 singole texture 2D che formano ciascuna una faccia di un cubo. Le cube map hanno le proprietà di essere campionate usando un **vettore di direzione**.

Supponiamo di avere un cubo unitario $1 \times 1 \times 1$ ed un vettore direzione che ha l'origine nel centro del cubo. Il campionamento di un valore di texture della cubemap con un vettore direzione avviene nel seguente modo:



Se immaginiamo di avere un cubo a cui applichiamo una cubemap, questo vettore di direzione è uguale alla posizione (interpolata) del vertice locale del cubo. In questo modo possiamo campionare la cubemap usando i vettori di posizione effettiva del cubo, purché il cubo sia centrato sull'origine. Quando campioniamo una cubemap, consideriamo quindi **tutte le posizioni dei vertici del cubo come coordinate di texture**.

Creare una cube-map

Una cubemap è una texture come qualsiasi altra, quindi possiamo creare una generando una texture e legandola al target **GL_TEXTURE_CUBE_MAP** prima di eseguire ulteriori operazioni sulla texture.

Per creare una cube-map, quindi, possiamo usare i comandi:

```
unsigned int textureID;  
 glGenTextures(1, &textureID);  
 glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
```

Poiché una cubemap contiene 6 texture, una per ogni faccia del cubo, dobbiamo chiamare **glTexImage2D 6 volte**, impostando i parametri. Ogni volta, però, **dovremo specificare un target diverso**, in modo che corrisponda a una faccia specifica della mappa cubica e far capire ad OpenGL per quale parte della mappa cubica stiamo creando una texture.

Texture target	Orientation
GL_TEXTURE_CUBE_MAP_POSITIVE_X	Right
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	Left
GL_TEXTURE_CUBE_MAP_POSITIVE_Y	Top
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	Bottom
GL_TEXTURE_CUBE_MAP_POSITIVE_Z	Back
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	Front

Volendo, potremmo eseguire un ciclo for cominciando da **GL_TEXTURE_CUBE_MAP_POSITIVE_X**, e aumentando di 1 ad ogni iterazione per generare una texture per ogni faccia.

Il codice che otterremo sarà quindi qualcosa del genere:

```
vector<string> texture_faces {  
    "right.jpg",  
    "left.jpg",  
    "top.jpg",  
    "bottom.jpg",
```

```

    "front.jpg",
    "back.jpg"
};

int width, height, nrChannels;
unsigned char *data;
for(GLuint i = 0; i < textures_faces.size(); i++)
{
    data = stbi_load(faces[i].c_str(), &width, &height, &nrChannels, 0);
    glTexImage2D( GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width, height, 0,
    GL_RGB, GL_UNSIGNED_BYTE, data );
}

```

Ci sono poi alcuni parametri speciali di wrapping e filtering che dobbiamo usare:

```

glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);

```

Essenzialmente, S, T, R sono le 3 coordinate della texture, e impostiamo per ogni sua coordinata impostiamo il metodo CLAMP_TO_EDGE, siccome le coordinate di texture che sono esattamente tra due facce potrebbero non colpire una faccia esatta (a causa delle limitazioni hardware), quindi con GL_CLAMP_TO_EDGE, OpenGL restituisce sempre i loro valori di bordo ogni volta che campioniamo le facce.

Fragment Shader per cubemap

All'interno del fragment shader dobbiamo usare un diverso campionatore del tipo samplerCube, ma questa volta usando un vettore di direzione vec3 invece di un vec2.

Ecco quindi cosa avremo:

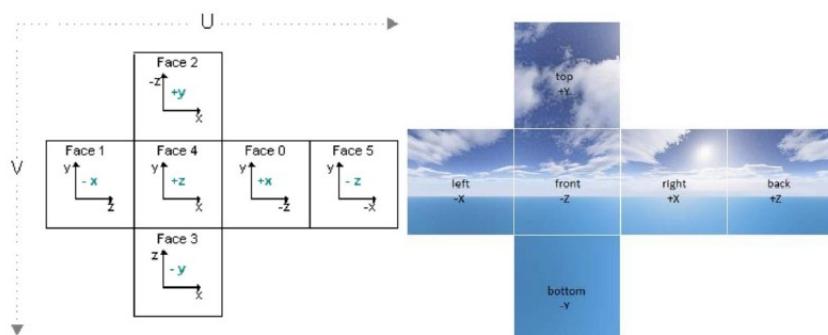
```

in vec3 textureDir; → vettore direzione che
uniform samplerCube cubemap; → rappresenta una coordinata 3D
                                di texture
void main()
{
    FragColor = texture(cubemap, textureDir); → campionatore di texture di
}                                         tipo cubemap

```

Usare una cubemap per creare uno skybox

Uno **skybox** è un cubo (grande) che racchiude l'intera scena e contiene 6 immagini di un ambiente circostante, dando al giocatore l'illusione che l'ambiente in cui si trova sia in una realtà molto più grande di quella reale.



Mappatura ambientale (Riflessione)

Possiamo usare le cubemaps per simulare l'effetto di riflessioni su oggetti della nostra scena. Per farlo semplicemente facciamo in modo che i colori dell'oggetto siano più o meno uguali al suo ambiente in base all'angolo di visualizzazione.

Sarà necessario modificare **il fragment shader associato all'oggetto riflettente**, in modo che rifletta l'ambiente circostante in base all'angolo di visualizzazione.

```
I = normalize(Position - cameraPos);  
R = reflect(I, normalize(Normal));  
FragColor = vec4(texture(skybox, R).rgb, 1.0);
```

R è la direzione lungo cui campionare la cubemap

Rifrazione con le cubemaps

Possiamo anche simulare la rifrazione usando le cubemaps. La rifrazione è il cambiamento di direzione della luce dovuto al cambiamento del materiale attraverso il quale scorre la luce del sole, come abbiamo visto [qui](#).

Possiamo implementarla molto facilmente usando la funzione `refract` incorporata in GLSL, che prevede un vettore normale, una direzione di vista e un rapporto tra gli indici di rifrazione di entrambi i materiali. L'indice di rifrazione determina quanto si distorce la luce passando da un materiale ad un altro. Ogni materiale ha il proprio indice di rifrazione.

Per creare questo effetto, dovremo modificare il fragment shader in questo modo:

```
void main()  
{  
    float ratio = 1.00 / 1.52;  
    vec3 I = normalize(Position - cameraPos);  
    vec3 R = refract(I, normalize(Normal), ratio);  
    FragColor = vec4(texture(skybox, R).rgb, 1.0);  
}
```