Appunti computer grafica

Lezioni precedenti al 30/09/2021 – Il sistema grafico

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 sovvrascrivere le informazioni di altri pixel, perdendo per sempre le informazioni della foto di partenza. Inoltre, per routare 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 vettoriali** (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 perdità 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 verra 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 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?

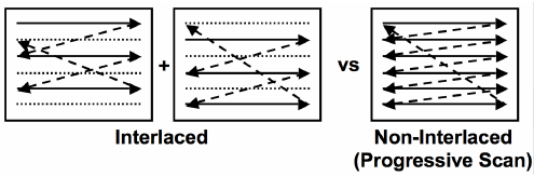
Nella Scheda Grafica, la GPU è nascosta sotto una grande ventola, siccome dissipa molto calore..

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 analofico 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 frame buffer**, che contiene le componeneti 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 transmette, 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

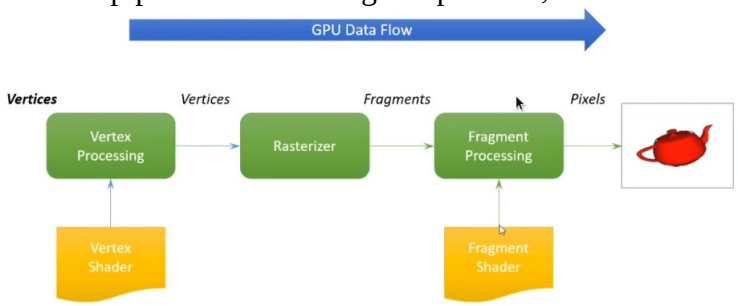
Cos'è OpenGL?

OpenGL è una **specifica** (ovvero un elenco di funzione 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 liberire 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 fornisci 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 tesselation 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 superiore.

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

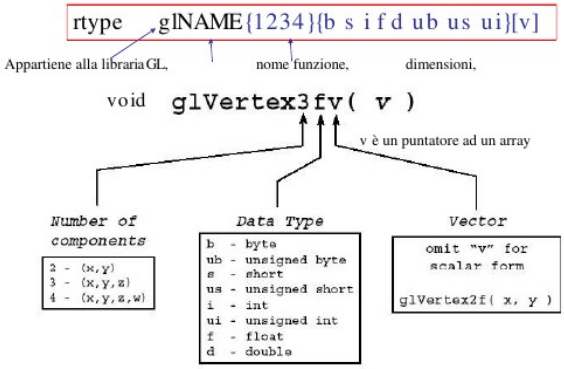
Macchina a stati

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

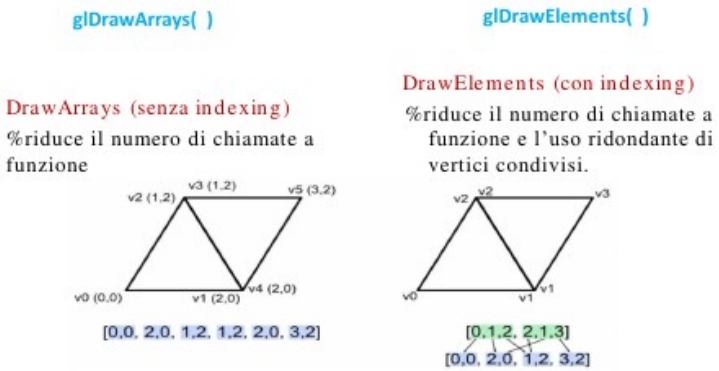



```
GLfloat    red,    greed,    blue;
Glfloat    coords[3];
glBegin (primType);          % primType Tipo della primitiva geometrica

for ( i = 0; i < nVerts; ++i )
{
    glColor3f( 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 array di dati di 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 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).

4. 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()`)

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

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

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

Durante il rendering:

4. aggiornare i VBO associati a questo VAO.
5. 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.

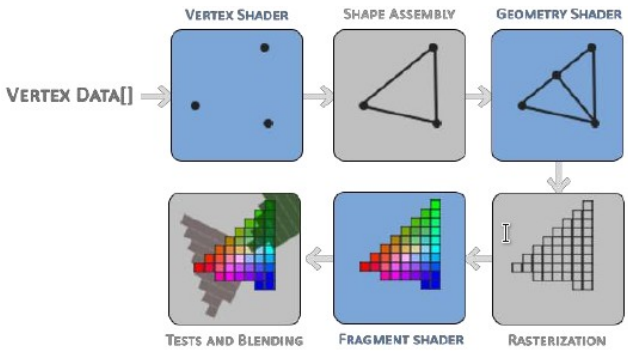
OpenGL: il lato pratico delle cose

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)**.

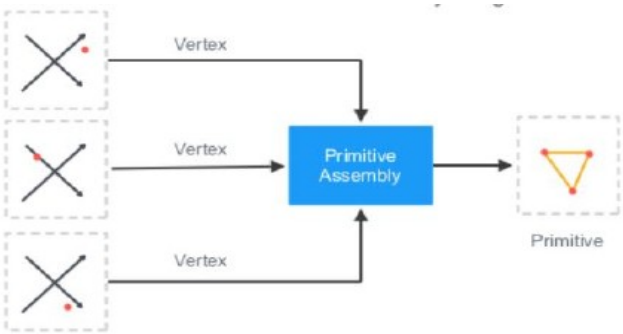
La pipeline OpenGL



Schematizzazione della 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 assembla tutti i punti nella forma primitiva data (per esempio, un triangolo).
Per capire come funzione la fase di Shape Assembly, facciamo riferimento a questa figura:



Diamo in pasto al Vertex Shader 3 vertici, che poi vengono riportate nella fase di assemblaggio delle primitive. Da qui, viene costruita una primitiva in un ordine specificato.

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 colleghiamo 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);
```

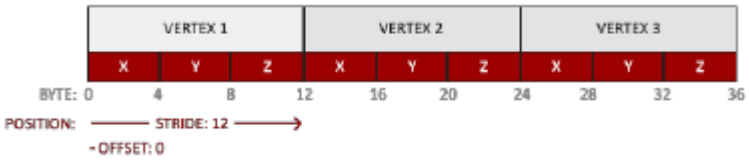
Il risultato sarà un programma che possiamo attivare chiamando:
`glUseProgram(shaderProgram);`
prima di ogni draw call.

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**. 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 inidica 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 `glEnableVertexAttributes(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`.

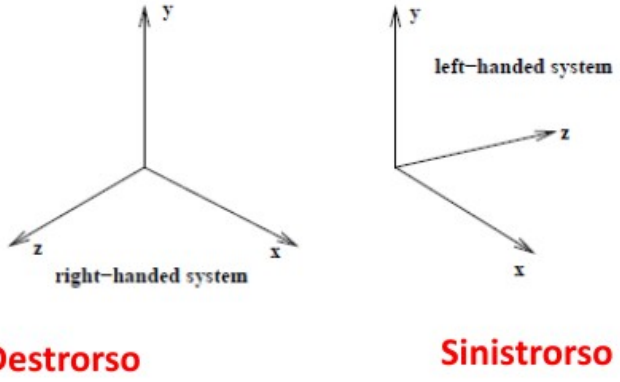
Lezione 04/10/2021 – Matematica per la computer grafica

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.



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 sono solamente vettori e scalari. Definiamo con R^n l'insieme dei vettori con n componenti reali. Per convenzione gli elementi di R^n sono rappresentati come dei **vettori colonna**.
Le operazioni che possiamo fare con dei vettori sono:

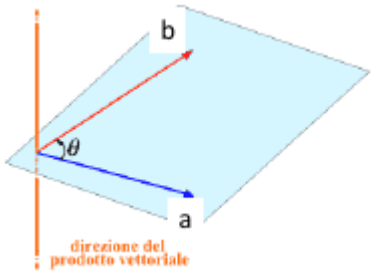
- Somma di due vettori, che viene fatta con la regola del parallelogramma ed è commutativa.
- Il prodotto per uno scalare, che cambia la lunghezza del vettore.

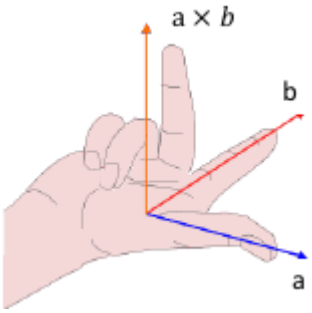
Rappresentiamo i vettori come dei segmenti liberi, senza un punto di applicazione specifico.

Ripassino di algebra

Dati k vettori v_k e k scalari λ_k, la quantità $\lambda_1 v_1 + \lambda_2 v_2 + \dots + \lambda_k v_k$ 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 $\lambda_1 v_1 + \lambda_2 v_2 + \dots + \lambda_k v_k = 0$ 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,…, 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 × b è dato dalla matrice:

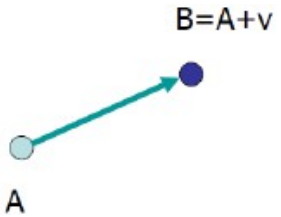
$$\begin{bmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix}$$

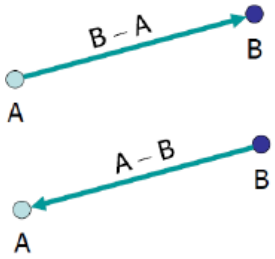
La norma a due del prodotto vettoriale rappresenta inoltre l'area del rettangolo individuato dai due vettori.

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 ance introdotti due nuove operazioni: [note su notazione: con le lettere maiuscole indichiamo dei punti]

- la **differenza punto-punto**, che defininisce 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:

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

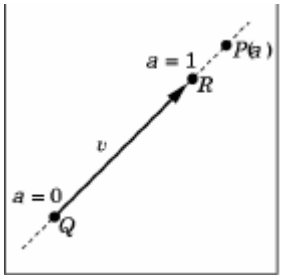
I coefficienti $\alpha_0, \alpha_1, \ldots, \alpha_N$ sono anche detti **coordinate baricentriche** (affini) di P nello spazio affine.

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

$$v = R - Q$$

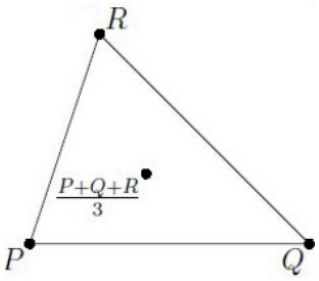
A sto punto consideriamo la loro combinazione affine, prendendo i coefficienti α, β tali che $\alpha + \beta = 1$. Dunque, possiamo scrivere P (che definisce un punto nella retta che va da Q a R) in questo modo:

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(α) 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 peso sono uguali a 1/n, il punto risultante si chiama **centroide** dell'insieme dei punti.



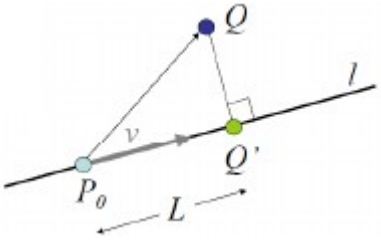
Calcolare la distanza fra un punto e una retta (ci serve per fare parallelismo coi piani)

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

$$L^2 + \text{dist}(Q, Q')^2 = \|Q - P_0\|^2$$

$$L = \frac{< Q - P_0, v >}{\|v\|} \qquad \text{Proiezione ortogonale di } Q - P_0 \text{ su } v$$

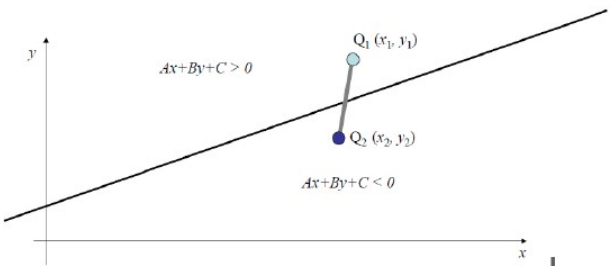
$$\Rightarrow \quad \text{dist}(Q, Q')^2 = \|Q - P_0\|^2 - L^2 = \|Q - P_0\|^2 - \frac{< Q - P_0, v >^2}{\|v\|^2}.$$



Altre proprietà della retta

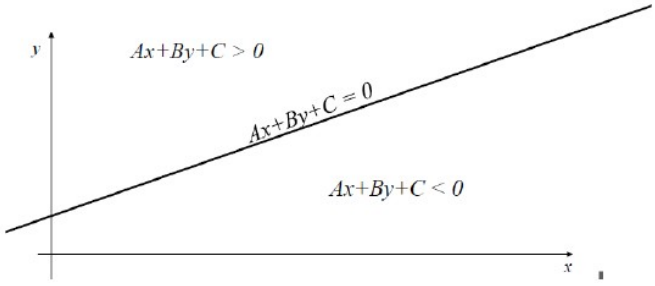
Il segmento $Q_1\ Q_2$ interseca la linea se e solo se

$$(Ax_1 + By_1 + C)(Ax_2 + By_2 + C) \leq 0$$



Modo semplice per vedere se un segmento interseca una retta

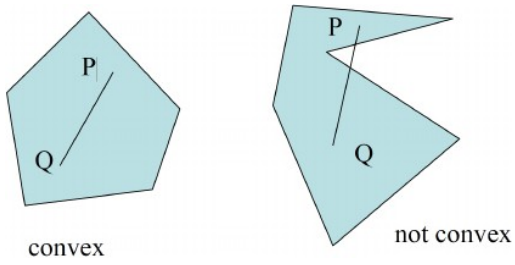
$$Ax + By + C = 0, \quad A, B, C \in \mathbb{R}, AB \neq 0$$



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.



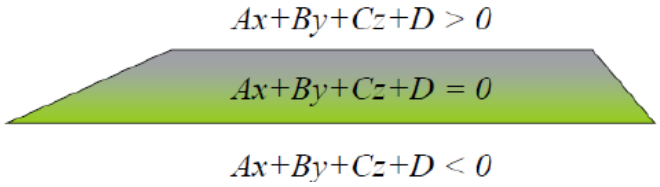
Dato un insieme di punti $P_1, P_2, \ldots, P_n$, l'insieme di tutti i punti P che possono essere rappresentati come combinazioni convesse è detto **inviluppo convesso** (guscio convesso) dell'insieme. Il guscio convesso (convex hull) di un insieme di punti è la più piccola regione convessa che contiene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

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



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:

$$P = P_0 + v = P_0 + v_1e_1 + v_2e_2 + v_3e_3$$
$$v = v_1e_1 + v_2e_2 + v_3e_3$$

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:

$$P = v_1e_1 + v_2e_2 + v_3e_3 + 1 \times P_0$$
$$v = v_1e_1 + v_2e_2 + v_3e_3 + 0 \times P_0$$

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

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:

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

In particolare, questa formula definisce le coordinate di ciascun punto della circonferenza. Siccome normalmente usiamo dei triangoli per costruitre 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 paramentriche** (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 funzoine, 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 stato 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:

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

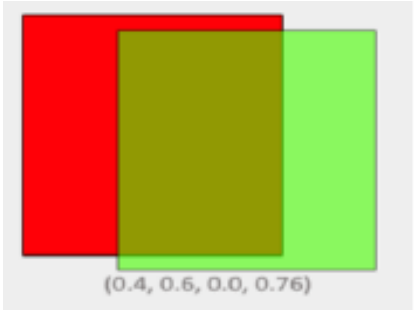
dove:
C_{source} = colore di output del fragment shader.
$C_{destination}$ = colore attualmente memorizzato nel frame color buffer
F_{source} = valore alfa del colore di output del fragment shader
$F_{destination}$ = valore alfa del colore attualmente memorizzato nel frame color buffer.

Dopo che il fragment shader è stato eseguito, l'equazione di blending viene eseguita tra l'output del colore del frammento generato dal fragmente 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, sostiuendo il colore precedente.

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

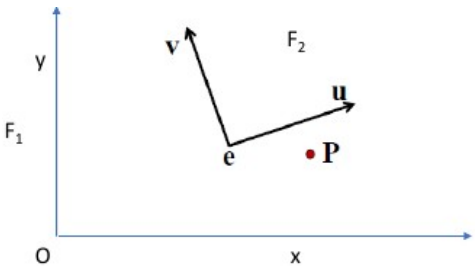


`glEnable(GL_BLEND);` che ci permette appunto di abilitare il blending.
La funzione **`glBlendFunc(Glenum sfactor, Glenum dfactor)`** specifica come vengono usati nell'equazione di blending i valori di F_{source} (`sfactor`) e di $F_{destination}$ (`dfactor`). In particolare, per ottenere lo stesso risultato dell'esempio dei quadrati, bisogna usare la funzione **`glBlendFunc(GL_SRC_ALPHA, GL_ONES_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.

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

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

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

Mentre P, nel frame F_2= (u, v, e), si esprime come

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

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

Siccome sappiamo che $\begin{bmatrix} u \\ v \\ e \end{bmatrix} = M \begin{bmatrix} x \\ y \\ \boldsymbol{O} \end{bmatrix}$

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

$$a^T \begin{bmatrix} x \\ y \\ \boldsymbol{O} \end{bmatrix} = b^T M \begin{bmatrix} x \\ y \\ \boldsymbol{O} \end{bmatrix}$$

Allora possiamo dire che:
Essenizialmente quindi:

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

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 trasormazioni 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

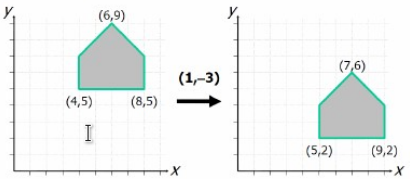
Traslare una primitiva geometrica nel piano significa muovere ogni suo punto P(x, y) di d_x unità lungo l'asse x e di d_y unità lungo l'asse y fino a raggiungere la nuova posizione P'(x', y') dove

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

In notazione matriciale:

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

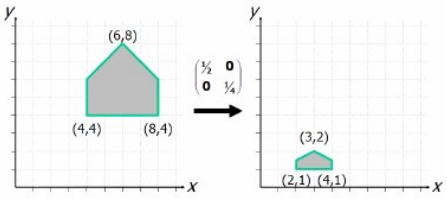
$$P' = P + \mathbf{T}$$



ebbe il
lazione.

Scelto un punto C (punto fisso) di riferimento, **scalare** una primitiva geomtrica 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 ottine in questo modo:

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



In notazione matriciale P' = **S**·P, dove

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

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!

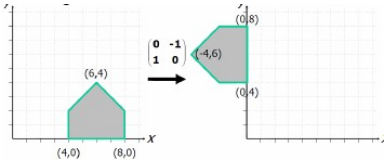
Una transazione, per essere affine, dev'essere espressa come il prodotto di una matrice. Per come l'abbiamo definito per ora, dunque, la transazione non è una trasformazione affine (e quindi nemmeno lineare).

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'orgine O degli assi è definita come:

$$x' = x \cdot \cos\theta - y \cdot \text{sen}\theta, \quad y' = x \cdot \text{sen}\theta + y \cdot \cos\theta$$

In notazione matriciale abbiamo: P' = **R** · P, dove

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



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

Coordinate omogenee e translazione

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:

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

***Traslazione**: notare come sia cambiato in modo radicale.*

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

Scalatura

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

Rotazione

Altre operazioni: Riflessioe 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:

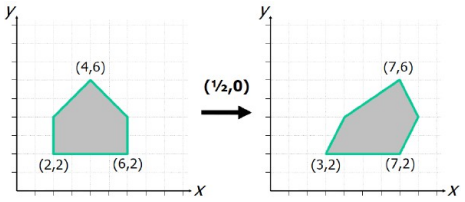
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

E la deformazione rispetto a un'asse (**shear**), che essenzialmente corrisponde alle relazioni

$$x' = x + ay, \quad y' = y + bx$$

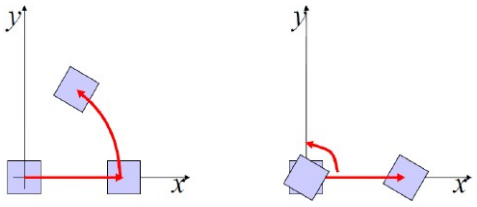
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

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



Composizione di Trasformazioni

L'ordine di concatenazoine è importante siccome le trasformazioni sono associative ma **NON commutative**, (infatti ruotare e poi traslare è diveso 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.

$$\mathbf{R}_g = \begin{bmatrix} 1 & 0 & P_x \\ 0 & 1 & P_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -sen\,\theta & 0 \\ sen\,\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -P_x \\ 0 & 1 & -P_y \\ 0 & 0 & 1 \end{bmatrix}$$

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.

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

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 $P = M^{-1}P'$.

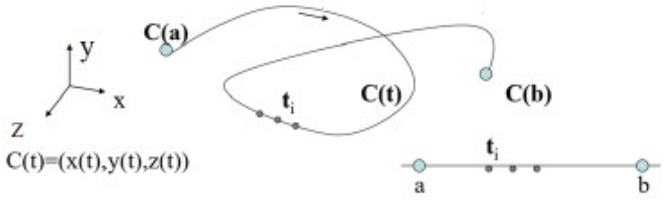
Fortunatamente per le trasformazioni viste <u>le matrici inverse sono particolarmente semplici da calcolare</u>.

Infatti:

- L'inversa della matrice di trasformazione di rotazione R di un angolo θ è data dalla matrice di rotazione di un angolo −θ: $R(-\theta) = R^T(\theta) = R^{-1}(\theta)$ (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 $\mathbf{R^3}$**, è un'applicazione $t \rightarrow (x(t), y(t), z(t))$ dove x(t), y(t), z(t) sono funzioni continue del parametro $t \in [a,b] \subseteq R$, 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 P0 = (x0 , y0) e P1 = (x1 , y1), 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 R3 tramite C dell'intervallo $I= [a,b] \subseteq R$, 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, la lunghezze delle le lunghezze delle poligonali associate formano una successione monotona non decrescente che converge all' integrale, che rappresenta quindi la lunghezza della traiettoria:

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

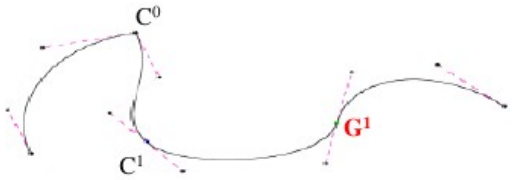
Dunque, siano x'(t), y'(t), z'(t) continue in [a, b] le derivate prime delle componenti parametriche della curva C(t) = (x(t), y(t), z(t)), allora la lunghezza di C tra C(a) e C(b) è definita dalla seguente formula:

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

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** C^0 che assicura l'assenza di salti.
Se due tratti di curva si uniscono in un punto P_0 e, inoltre, detti v_1 e v_2 i vettori velocità in P_0 del primo e secondo tratto di curva rispettivamente, si definisce

- **Continuità parametrica C^1**: le direzioni e i moduli dei vettori tangenti v_1, v_2 dei due segmenti curvi nel punto di contatto P_0 sono uguali.
- **Continuità geometrica G^1**: le direzioni dei vettori tangenti v_1, v_2 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: $C(t_i) = p_i \equiv \begin{pmatrix} x_i = x(t_i) \\ y_i = y(t_i) \end{pmatrix}$
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:

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

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

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

Dobbiamo quindi risolvere 3 problemi di interpolazione.
$C(t_i)$ sarà la nostra curva interpolata finale.

Lezione 14/10/2021 – glm e GLSL

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 boolenai)

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 costrutire 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 pemette 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 gloabali, 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 nostra 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 primitva

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 alla 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.glsl", "f.glsl");
/* 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 funzoini 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 comuqnue 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 bisonga 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 trasformaizione 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 matriche 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));
// radiants 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

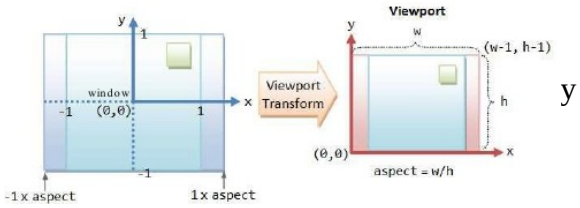
Come sappiamo, le trasformazioni ci permettono di passare da uno spazio ad un altro, e vengono usate per realizzare una scena 3D e la pipeline del rendering. **Usiamo trasformazioni rappresentate da matrici 4x4.**

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

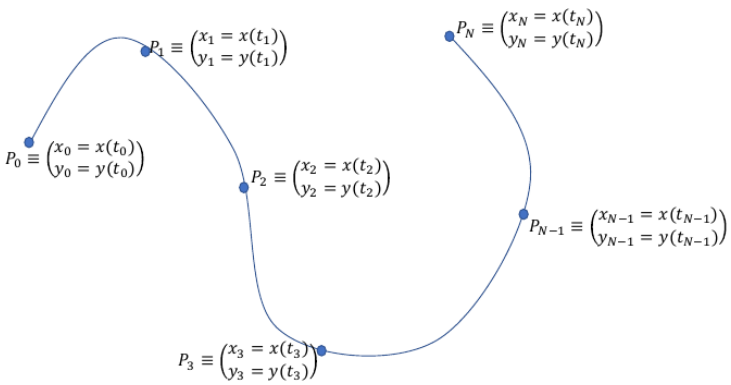
Un ultima trasformazione molto importante è la **trasformazione di Viewport**. La **Viewport** è la regione rettangolare dello schermo dove l'immagine viene proiettata. Di default è la regione che coincide con la finestra aperta sullo schermo. Tuttavia, è possibile modificare la regione della finestra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

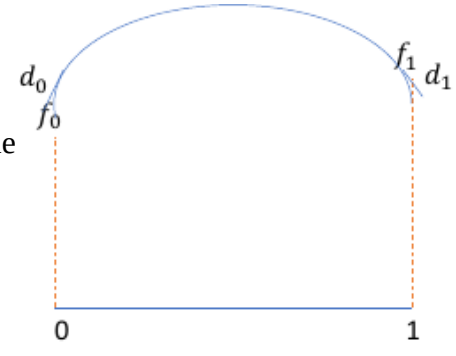
$$X_N(t_i) = x_i$$

Supponiamo che ogni punto P abbia delle coordinate (x_i,y_i) che supponiamo essere ottenute come la valutazione di un certo polinomio sconusciuto X_N (come direbbe Asperti, 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:

$$P_3(0) = f_0 \qquad P_3(1) = f_1$$
$$P_3'(0) = d_0 \qquad P_3'(1) = d_1$$

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

$P_3(0)$
$P_3(1)$

$$\begin{cases} a_0 = f_0 \\ a_0 + a_1 + a_2 + a_3 = 1 \\ a_1 = d_0 \\ a_1 + 2a_2 + 3a_3 = d_1 \end{cases}$$

$P_3'(0)$
$P_3'(1)$

$$P'_3(t) = a_1 + 2a_2x + 3a_3x^2$$

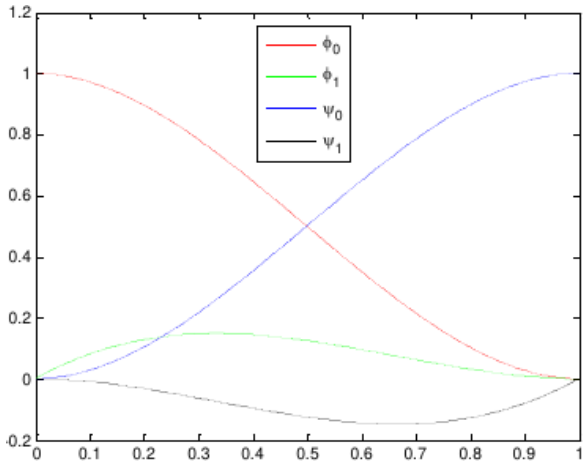
Hermite dimostra che il polinomio $P_3(t) = a_0 + a_1t + a_2t^2 + a_3t^3$ che interpola questi dati e che rispetta i vincoli prima definiti si può esprimere come:

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

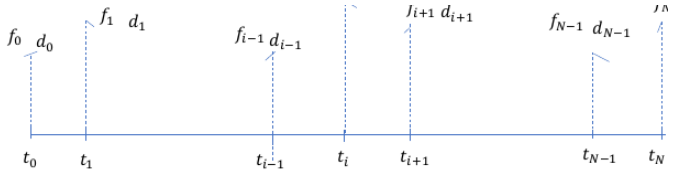
dove:

$$\varphi_0(t) = 2 \cdot t^3 - 3 \cdot t^2 + 1$$
$$\varphi_1(t) = t^3 - 2 \cdot t^2 + t \qquad 0 \le t \le 1$$
$$\psi_0(t) = -2 \cdot t^3 + 3 \cdot t^2$$
$$\psi_1(t) = t^3 - t^2$$

Possiamo essenzialmente usare queste equazioni quindi per definire la funzoine 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 != 0. Questo perché in t=0, le altre formule si annullano tranne φ_0, come richiede il vincolo della notra 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 abbiamo 0 sotto intervalli.



Se però abbiamo i valori (t_i, f_i, d_i) con i = 0, …, 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, ..., 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 ricacolare 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 $\varphi_{1,i}(t)$, $\psi_{1,i}(t)$ 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 $t \in [0, 1]$. 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:

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

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 lolollol), quello che useremo di più sarà il metodo del **rapporto incrementale**

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

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 facilemente dimostrare che la curva che creaiamo 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.

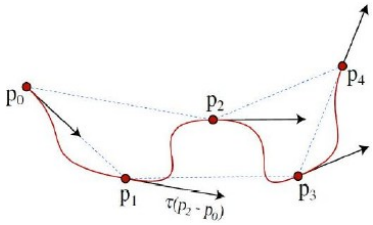
$$d_i = \frac{1}{2}(d_i + d_{i-1}) = \frac{p_{i+1} - p_i}{2(t_{i+1} - t_i)} + \frac{p_i - p_{i-1}}{2(t_i - t_{i-1})}$$

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

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



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 inziale p_i ed un punto finale p_{i+1} con tangenti d_i e d_{i+1}, le formule della derivata corrispondono a:

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

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

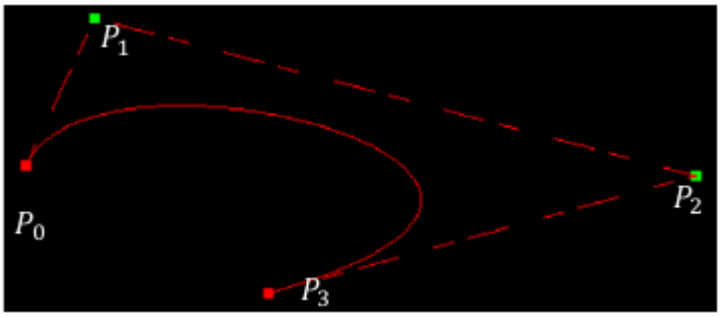
Dove il parametro tension (T) varia la lunghezza del vettore tangente (ovvero quanto è schiacciata la curva), il parametro bias (B) cambia la direzione del vettore di tangente e il valore continuity (C), varia la continuità.
In pratica corrisponde il metodo che abbiamo implementato nella esercitazione del giorno prima, anche se avevamo posto i parametrici TBC a 0, rendendolo quindi pari al metodo delle differenze finite.

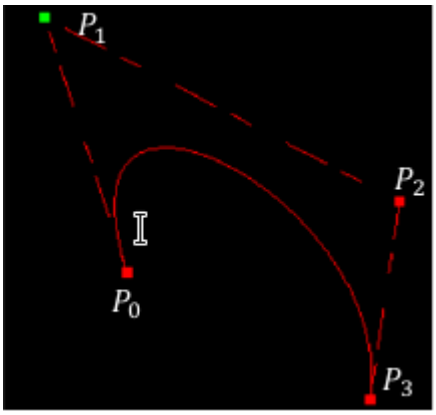
Differenza fra spline e le altre interpolazioni/curve.
Le spline sono delle funzioni polinimiali a tratti, definiti su un certo numero di sottointervalli, e nel punto di contatto tra due intervalli successivi hanno una certa regolarità. La loro caratteristica è che sono definite da polinomi di grado 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.





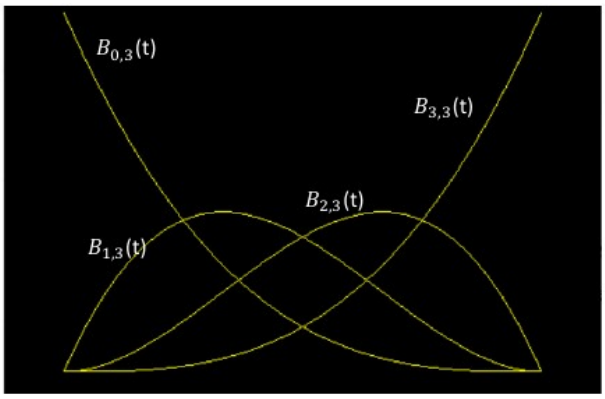
Come per le curve di Hermite, abbiamo delle funzioni base che permettono di ottenere il segmento di curva cubica di Bezier. Queste sono **i polinomi di Bernstein** (di grado 3), e sono:

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

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

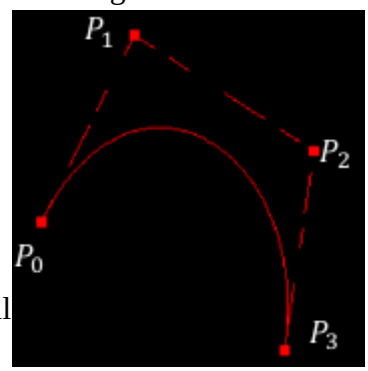
$$B_{2,3}(t) = 3t^2(1-t)$$

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



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:

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



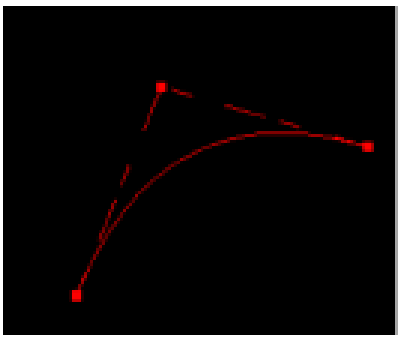
I punti P_i sono detti **punti di controllo della curva** (o **maniglie**) ed individuano il poligono di controllo della curva. [N.B. ovviamente nella formula bisogna mettere un n al posto del 3 per renderla generale per n punti].

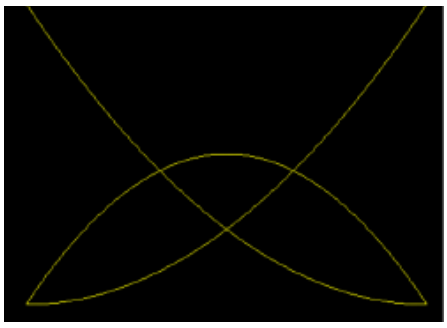
Nel caso più generale di curve di Bezier di grado n abbiamo a che fare con n+1 punti, di cui il primo e l'ultimo sono interpolati, il secondo e il penultimo danno la direzione della derivata negli estremi, mentre gli altri punti vengono "approssimati" e servono a modellare la forma della curva. Dunque, 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:

Coefficiente binomiale

$$B_{i,n}(t) = \binom{n}{i}(1-t)^{n-i}t^i \qquad i = 0,..n$$

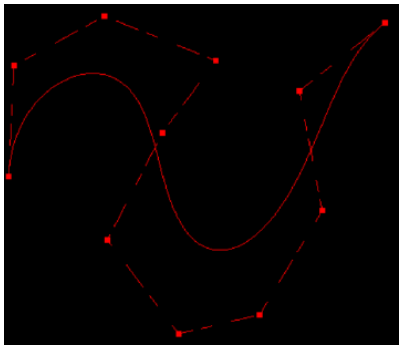
Vediamo un esempio della curva di Bernstein di grado 2:

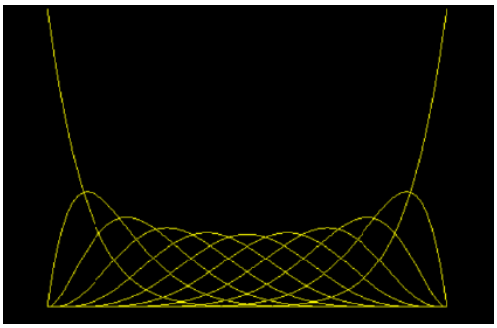




Sulla sx abbiamo la curva vera e propria, a dx invece abbiamo il grafico delle curve che compongono la curva.

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 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) influenzerò 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 comunique 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, ..., n] sono **le funzioni di base di Bernstein di grado n** definiti su [a, b] dalla relazione:

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

e c_0 , c_1, ..., 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 Bernestein con la rappresentazione mediante di un polinomio mediante la base monomiale (ovvero la classica somma di monomi per costruire un polinomio $1,x,x^2 \ldots x^n$.):

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

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 e definito in [0, 1] è legato al polinomio di Bernestein 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 convoilge sempre somme di quantità positive (abbiamo la definizione di stabilità numerica a calcolo numerico, dove abbiamo visto che la somme fra numeri con segno opposto sono invece instabili).

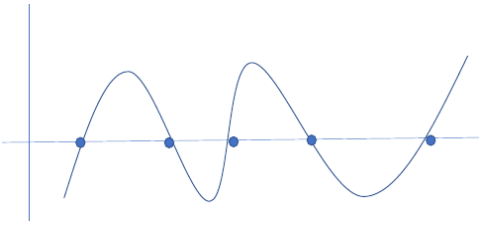
3) I polinomi di Bernstein sono una **partizione dell'unità**. Questo indica che, fissato un valore di t, la somma di tutte le funzioni base su quel punto è pari a 1.

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

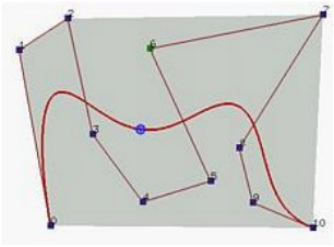
4) Il polinomio $p(x) = \sum_{i=0}^{n} c_i B_{i,n}(x)$ è una combinazione lineare convessa dei valori dei suoi coefficienti (siccome la loro somma è 1, come abbiamo potuto vedere dalla *prima* e *terza proprietà*), e quindi:

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

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. Questo ci consente di avere per il polinomio un limite superiore per il numero dei suoi zeri semplici in [a,b].

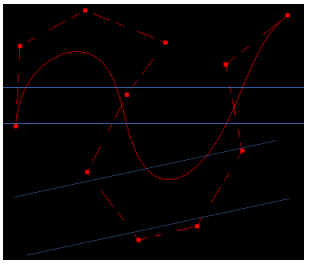


Proprietà dell'inviluppo convesso



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



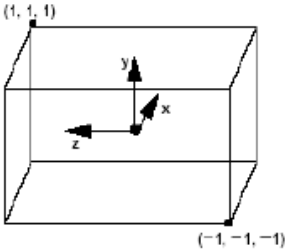
Il numero di intersezioni di una qualsiasi retta con il poligono di controllo è maggiore o uguale del numero di intersezioni della stessa retta con il polinomio.

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.

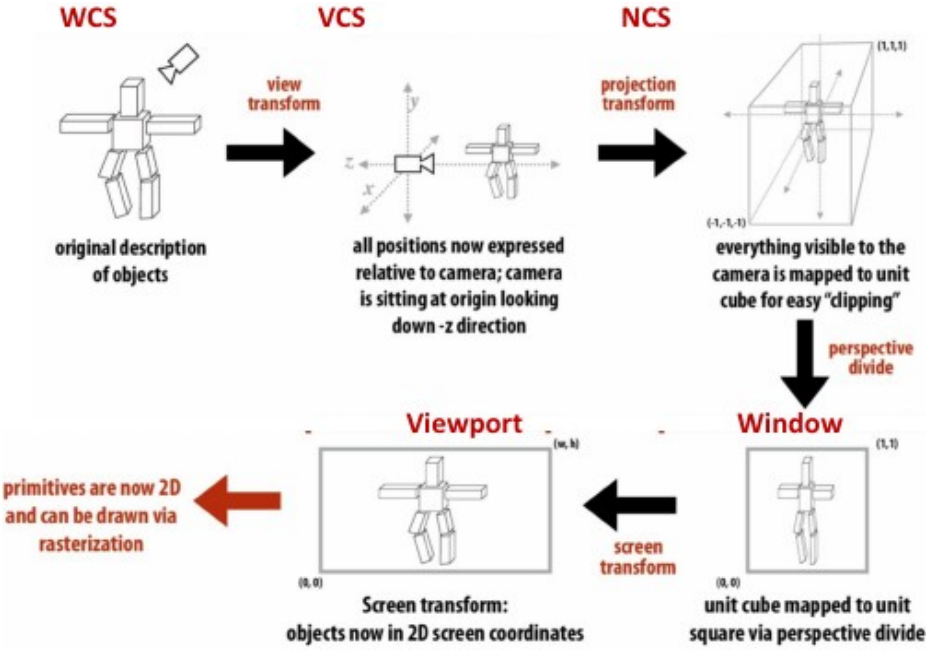
Poichè solo i modelli ’visti’ dalla camera virtuale sono resi, tutti i modelli sono quindi trasformati mediante una **trasformazione di vista** nel sistema di riferimento della camera o camera frame (**View Coordinate System – VCS**).

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, NDC**). 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 trasfomazione affine 4x4** (T_M).

2 - Trasformazioni di Vista (da WCS a VCS)

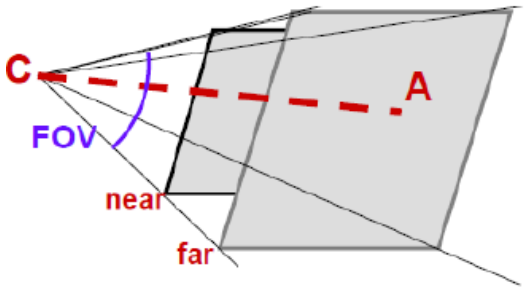
Questa fase dello stadio geometrico prende in input i vertici in 3D definiti in WCS, e da in output le coordinate in **VCS (View Coordinate System)**, ovvero un sistema di coordinate della telecamera sintetica della nostra scena. La matrice di vista (View Matrix) sarà quindi utilizzata per trasformare 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. Dopodiché, dovremo costruire la trasformazione di vista a partire dal posizionamento e orientamente 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 trasfomazione ad ogni vertice dell'oggetto.

Definire la vista – camera

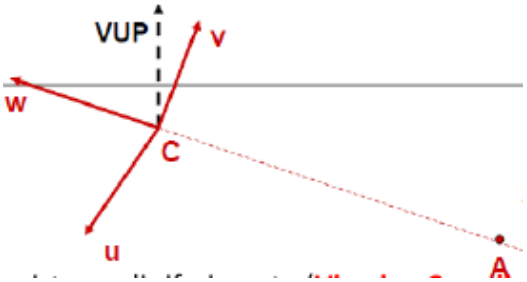
Abbiamo bisogno di sapere quattro cose sul nostro modello di fotocamera sintetica per costruire la trasformazione della vista



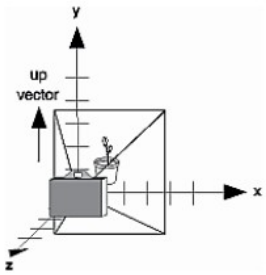
- **Punto C**: posizione della telecamera in WCS (da dove si sta guardando).
- **Punto A**: il punto A permette il calcolo del vettore C-A, che specifica in quale direzione sta puntando la telecamera (dove A è il centro della scena).
- field of view o **FOV** (**grandangolo**, normale...)
- posizionamento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

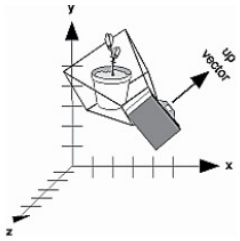
Abbiamo poi altri 3 assi/vettori molti importanti, calcolati a partire dai parametri descritti prima:

- 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: $w = \frac{C - A}{\|C - A\|}$. [Norare 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 specificar 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 trasfomazione 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):

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

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 trasfomazioni:

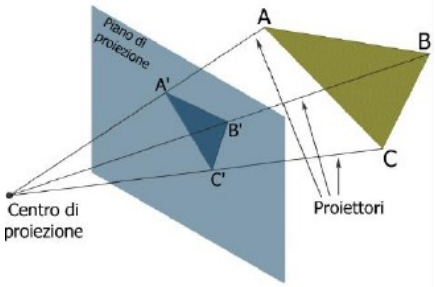
- L'oggetto viene posizionato in **coordinate mondo (WCS)** con matrice T_M (ovvero la matrice di modellazione), la camera è posizionata in coordinate mondo con matrice T_v (ovvero la matrice di vista).
- La seguente trasformazione prende ogni vertice dell'oggetto dal sistema di coord. locali (che chiamiamo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



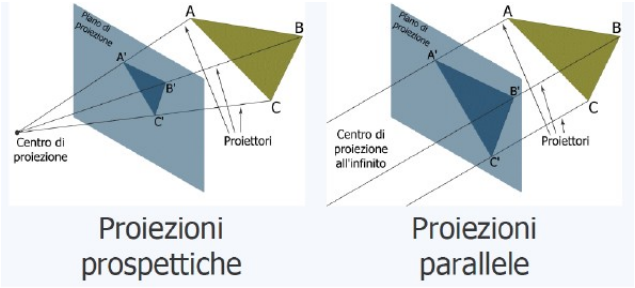
Proiezioni geometriche – alcune definizioni

Si dice **proiezione** una trasfomazione geometrica con il dominio in uno spazio di dimensione n ed il codominio in uno spazio di dimesione 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)

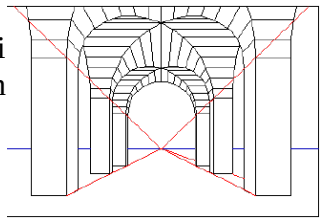
Le proiezioni geometriche piane sono suddivisibili in due classi base: **prospettiche** e **parallele**. La differenza fra le due classi è data dalla distanza tra il centro di proiezione ed il piano di proiezione: se tale distanza è finita, allora la proiezione è prospettica, altrimenti se è infinita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



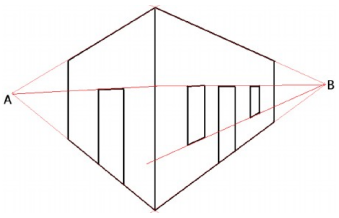
Normalmente la proiezione prospettica è la più realistica, in quanto riesce a riprodurre il modo con cui nella realtà vediamo gli oggetti: oggetti più grandi sono più vicini all'osservatore, oggetti più piccoli sono più lontani. Inoltre le distanze uguali lungo una linea non vengono proiettate su distanze 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 prospettiche

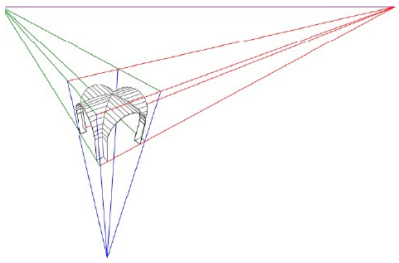
Le proiezioni prospettiche sono caratterizzate dai **vanishing point**: la proiezione di ogni insieme di linee parallele (non parallele al piano di proiezione) converge in un punto detto vanishing point (punto di convergenza). Il numero di questi punti è infinito, come il numero delle possibili direzioni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



2 punti di fuga


3 punti di fuga

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 si parla di proiezione ortografiche. Altrimenti, si parla di **proiezione obliqua**.

Proiezione ortografica

Proiezione obliqua

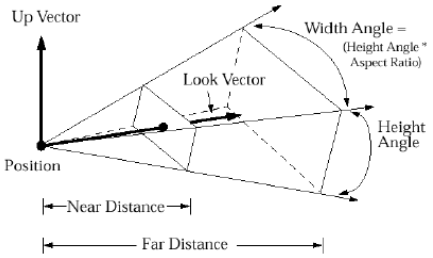
Abbiamo poi le proiezioni ortografiche assonometriche, che sono ortografiche, ma siccome lla direzione di proiezione non è allineata con un asse principale, vengono mostrano facce diverse di un oggetto, assomigliando in questo alle proiezioni prospettiche.
Un tipo speciale di proiezione assonometrica 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 proieizone 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] × [-1,1] × [-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 (detto **frostum**)*

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

In questa lezione di laboratorio, abbiamo espanso l'esercizio vecchio del funghetto.

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 segnamo 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.

poichè $B_{-1,n-1}(t) = B_{n,n-1}(t) = 0$

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

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].

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

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,...,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:

Sostituiamo con la formula della funzione base di Bernstein.

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

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

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).

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

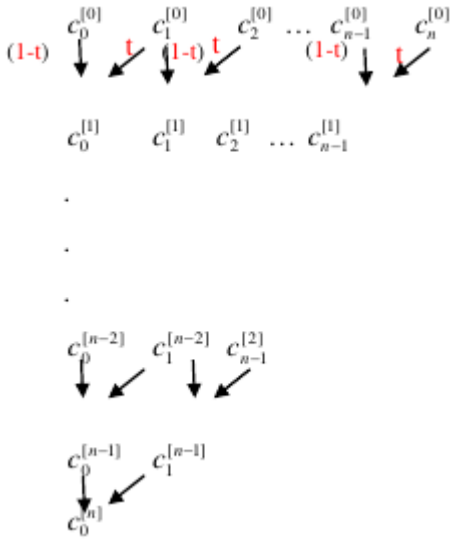
A sto punto quindi posso applicare la stessa situazione di prima, con $c_i^{[2]} = t \cdot c_{i+1}^{[1]} + (1-t) \cdot c_i^{[1]}$, e otterremo così nuovamente una formula del tipo: $P(t) = \sum_{i=0}^{n-2} c_i^{[2]} B_{i,n-2(t)}$

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

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

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 $c^{[i]}$. 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:

$$Lerp(t, a, b) = (1-t)a+t(b)$$

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

Per capire come funziona effettivamente l'algoritmo di de Casteljau, vediamo l'interpretazione geometrica del procedimento in sé.
Consieriamo una curva di Bezier cubica (quindi definita da 4 punti P_0...P_3).
Consideriamo i CP (ovvero i punti di controlli 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

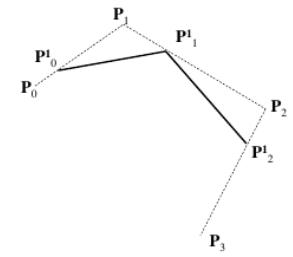
Calcoliamo 3 punti P_0^1, P_1^1, P_2^1, ottenuti rispettivamente come l'interpolazione lineare fra i punti P_0 e P_1 (ovvero Lerp(t, P_0, P_1)), P_1 e P_2, P_2 e P_3.

Passo 2

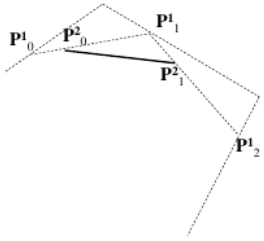
Calcoliamo 2 punti P_0^2, P_1^2, ottenuti rispettivamente come l'interpolazione lineare fra i punti P_0^1 e P_1^1 (ovvero Lerp(t, P_0^1, P_1^1)), P_1^1 e P_2^1.

Passo 3, ovvero il passo finale

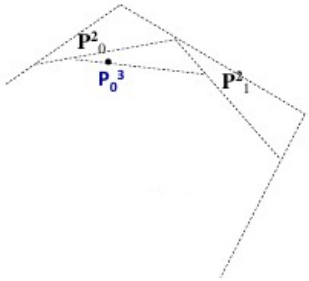
A questo punto, ci basta calcolare l'ultimo punto P_0^3, che rappresenta il valore della curva di Bezier per t=0.4



Passo 1



Passo 2

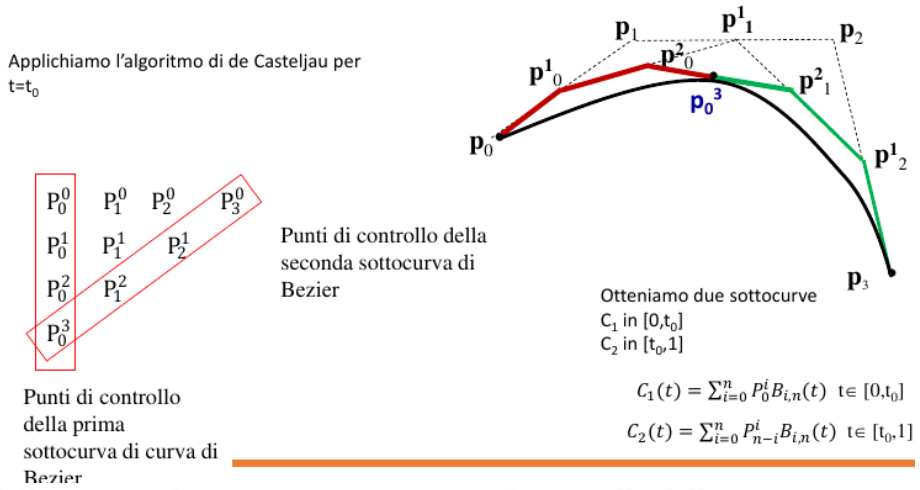


Passo 3

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 **suddivere** una curva di Bezeier di grando n in due sottocurve di grando 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:



I coefficienti della prima colonna rapprentano 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 trasfomazione affini**, quindi possiamo applicare le trasfomazioni ai punti di controllo e poi applicare de Casteljau e ottenere la stessa curva trasfomata.
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:

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

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

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

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]:

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

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

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

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

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

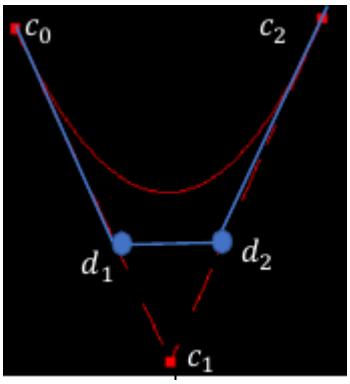
A questo punto, consideriamo il polinomio di Bernstein di grado n+1:

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

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

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

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:



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

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

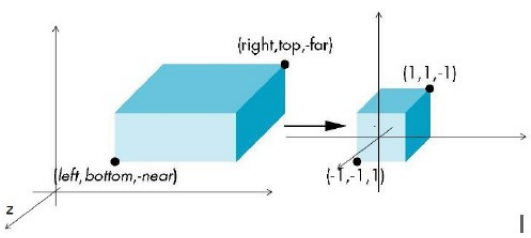
In questo modo, come prima, il polinomio acquisisce un punto di controllo in più, che lo raffina.

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 piano 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)



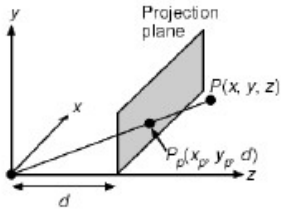
$$\mathbf{v}' = \mathbf{P} \cdot T_v \cdot T_m \cdot \mathbf{v}$$

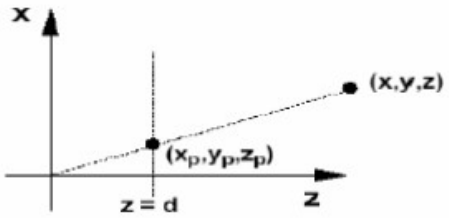
$$\mathbf{P}(left, right, top, bottom, near, far) = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{near-far} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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 matriche 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

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

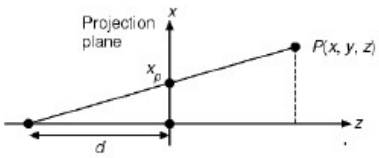
Analogamente, dobbiamo fare la stessa cosa con il piano yz per trovare la coordinata del punto y_p, usando la proporzione $y : y_p = z : d$.
La coordinata z del punto sarà semplicemente d, dunque le coordinate del punto finale saranno (x_p, y_p, d).

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.

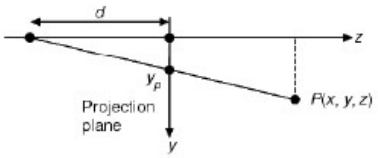
$$\begin{bmatrix} x*d/z \\ y*d/z \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

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.



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



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

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

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

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

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 punto possiamo rifare le proporzioni viste in precedenza.

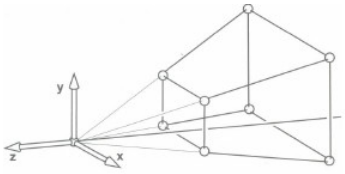
Notare come nella matrice z sia uguale a 0, siccome appunto il piano che andiamo a prendere in considerazione è apputno z = 0, e per fare in modo che z+d/d compaia nella formula, dovremo mettere 1/d e 1 nell'ultima riga della matrice.

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]:

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

Window Transformations e trasformazione window-vieport

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]×[-1,1]×[-1,1] (spazio immagine 3D) in una regione rettangolare (x, y) in [-1,1]×[1,1] (finestra 2D) [in poche parole, elimina la terza dimensione].

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

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

Abbiamo poi un'ultima [FINALMENTE] trasformazione rimanente da eseguire: la **trasformazione window-vieport**.
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 × ny) pixel. Alle coordinate reali di ogni vertice nella window dovranno corrispondere coppie 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

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 espreime 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:

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

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 $B_{n,n-1}(t) = 0$.

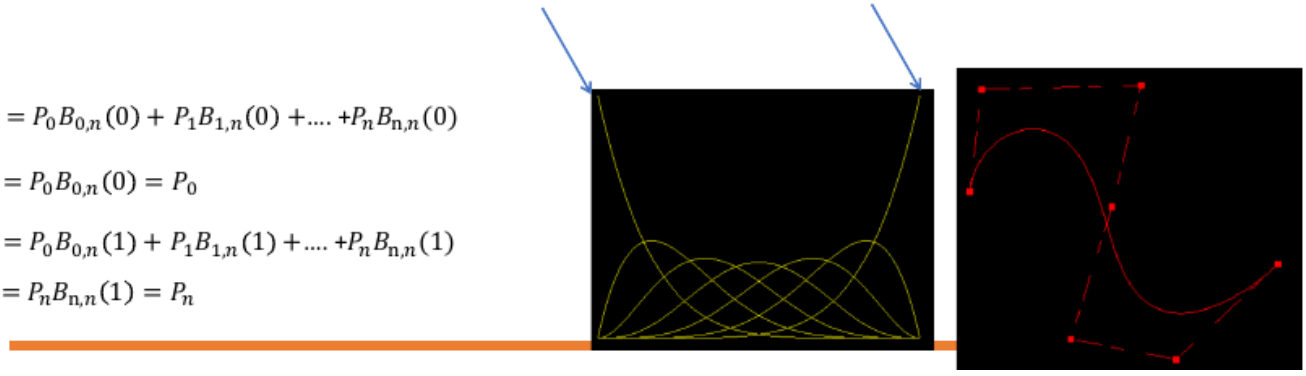
Possiamo così arrivare alla conclusione che il vertice di controllo P_0 corrisponde al valore del parametro t=0, mentre il vertice di controllo P_n corrisponde al valore del parametro t=1.

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

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

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

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



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

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

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

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

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

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

Per quanto riguarda i coefficienti avremo che:

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

Ma,

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

Segue quindi:

$$[c_0 \quad c_1 \quad \dots \quad c_n] = [a_0 \quad a_1 \quad \dots \quad a_n]\Lambda$$

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

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 tangenti 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,...,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), sono non sono diverse da zero su tutto l'intervallo [a,b], ma hanno un supporto locale che dipende dal loro ordine.

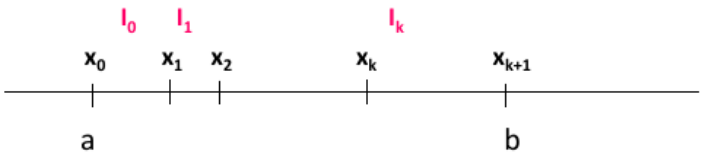
Spline polinomiali a nodi multipli

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

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

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]):

$I_i = [x_i, x_{i+1})$ per i=0,…k-1.
$I_k = [x_k, x_{k+1}]$

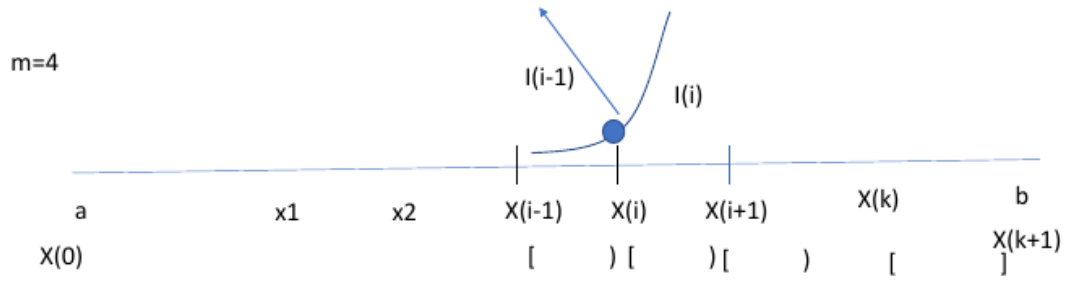


Sia poi **m** un intero positivo tale che m < k e sia M = (m_1, m_2, ..., m_k) un vettore di interi positivi tali che $1 \le m_i \le m \quad \forall\, i = 1, \ldots, k$

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

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

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 – mi – 1). È questo ciò che ci dice la formula sopra.

Possiamo **decidere noi** fino a che grado di continuità si può arrivare in ciascun xi, giocando con il valore di mi.
Maggiore sarà la molteplicità m, minori saranno le condizioni di raccordo sui nodi.

Se poniamo m_i=1 (i=1, ...,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:

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

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 cuspide!).

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, ..., x_k di molteplicità M = (m_1, ..., 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, …, m-m_i-1 che fanno in totale m-m_i condizioni) imposte su ogni nodo x_i, i = 1, ..., 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 ...< x_1 < x_2 <... semplici, si definisce B-spline normalizzata di ordine m relativa al nodo x_i (con x ∈ [a, b]), una funzione $N_{i,m}(x)$ che gode delle seguenti proprietà:

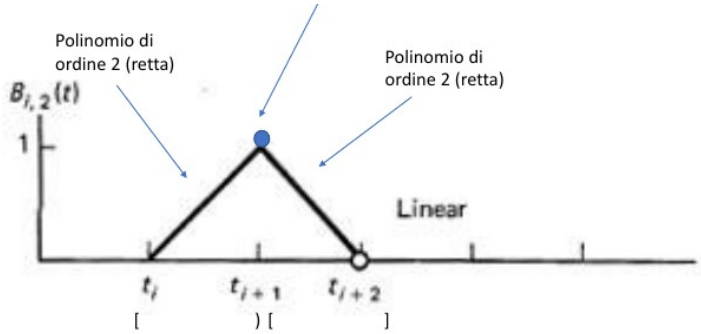
$N_{i,m}(x) = 0$ per $x < x_i \;\; x > x_{i+m}$. (supporto compatto);

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

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.

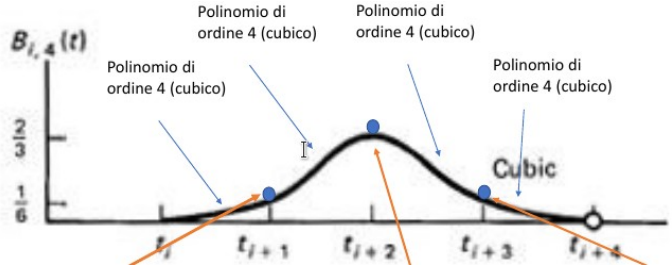
2) la condizione di normalizzazione, che indica che l'area della funzione dev'essere pari a tale funzione.

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:



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 ,…, 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].

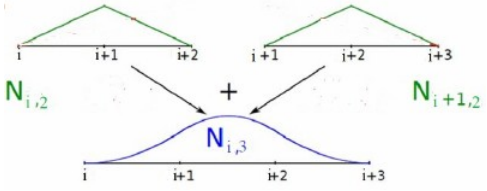
Formule di Cox per la valutazione di una B-spline (poco importante)

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:

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

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

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

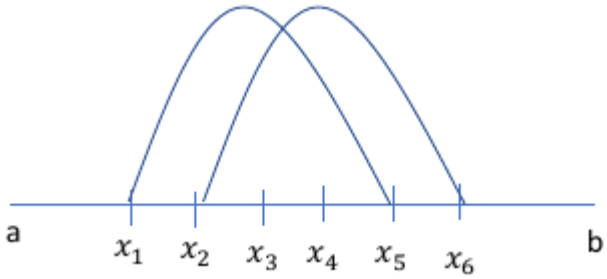


Formule di Cox per la valutazione di una B-spline (poco importante)

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].

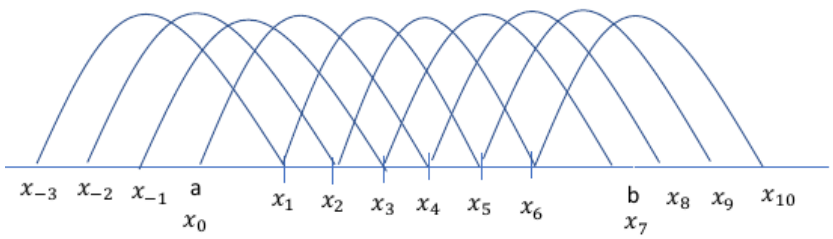
Quindi, dire che lo spazio delle spline ha dimensione m+K, vuol dire che la sua base deve essere costituita da m+k funzioni base.

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.



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

Tuttavia non è così… ci mancano ancora degli strumenti per costruire una base per $S_m(\Delta,M)$, non possiamo solo con i k nodi x_i.
Aggiungo quindi 2m nodi fittizi. . 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 esempoi 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 $\Delta = \{a = x_0 < x_1 < ... < x_k < x_{k+1} = b\}$ ed il vettore di molteplicità $M=(m_1, ..., m_k)$, 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) $t_1 \leq t_2 \leq t_3 \leq ... \leq t_{2m+K}$ i nuovi nodi sono semplici, ma possono coincidere.
b) $t_m \equiv a$, $t_{m+K+1} \equiv b$
c) $t_{m+1} \leq t_{m+2} \leq t_{m+3} \leq ... \leq t_{m+K}$ sono scelti in modo tale che siano coincidenti con:
$$(x_1 \equiv x_1 \equiv ... \equiv x_1) < (x_2 \equiv x_2 \equiv ... \equiv x_2) < ... < (x_k \equiv x_k \equiv ... \equiv x_k)$$
Per m_1 volte, m_2 volte … m_k volte rispettivamente.
d) $t_i \leq a$ con $i = 1, ..., m-1$ (possono coincidere tutti con a)
$t_i \geq b$ con $i = m + K + 2, ... , 2m + K$ (possono coincidere tutti con b)

I punti x_i, $i = 1, . . . , k$ vengono anche chiamati **break points**.

[49]

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 – bho

Lorem Ipsum

Lorem Ipsum

Lezione 25/11/2021 – bho

Lorem Ipsum

Lorem Ipsum

Lezione 29/11/2021 – bho

Lorem Ipsum

Lorem Ipsum