Raccolta di appunti di PI (Programmazione di Interfacce)

La numerazione delle lezioni non segue quella reale, è solo a scopo organizzativo.

Lezione 1 – Introduzione e primi commenti

Wikipedia:
La **programmazione a eventi** è un paradigma di programmazione dell'informatica. Mentre in un programma tradizionale l'esecuzione delle istruzioni segue percorsi fissi, che si ramificano soltanto in punti ben determinati predefiniti dal programmatore, nei programmi scritti utilizzando la tecnica *a eventi* il flusso del programma è largamente determinato dal verificarsi di eventi esterni.

Il flusso di controllo nelle interfacce grafiche non è in mano al programmatore, è influenzato direttamente dagli eventi (click, mouse, touch, …) per questo le GUI si programmano seguendo il paradigma della programmazione ad eventi.

Wikipedia:
Invece di aspettare che un'istruzione impartisca al programma il comando di elaborare una certa informazione, il sistema è predisposto per eseguire all'infinito un loop di istruzioni all'interno del quale vi sono istruzioni che verificano continuamente la comparsa delle informazioni da elaborare (potrebbe trattarsi della creazione di un file in una certa cartella o della pressione di un tasto del mouse o della tastiera), e quindi lanciare l'esecuzione della parte di programma scritta appositamente per gestire l'evento in questione. Programmare "a eventi" vuol dire, quindi, ridefinire in modo personalizzato le azioni "di default" con cui il sistema risponde al verificarsi di un certo evento.

Gli eventi esterni a cui il programma deve reagire possono essere rilevati mediante polling (*interrogazione*) eseguito all'interno di un loop di programma, oppure in risposta ad un interrupt. In molte applicazioni si usa una combinazione di entrambe queste due tecniche.

I programmi che utilizzano la programmazione a eventi (denominati spesso *"programmi event-driven"*) sono composti tipicamente da diversi brevi sotto-programmi, chiamati gestori degli eventi (*event handlers*), che sono eseguiti in risposta agli eventi esterni, e da un dispatcher, che effettua materialmente la chiamata, spesso utilizzando una coda degli eventi che contiene l'elenco degli eventi già verificatisi, ma non ancora "processati". In molti casi i gestori degli eventi possono, al loro interno, innescare (*"trigger"*) altri eventi, producendo una cascata di eventi.

Gli event handler sono scritti dal programmatore e vengono eseguiti ad ogni occorrenza di un determinato evento, queste “funzioni” hanno spesso un parametro di input che trasporta informazioni relative all'evento (posizione del click, delta della rotella del mouse, …). Si possono registrare più handlers per uno stesso evento (almeno in f#), e cosa importantissima: **non si può conoscere a priori l'ordine di arrivo di un evento .**

Wikipedia:
La programmazione a eventi incoraggia l'utilizzo di tecniche di programmazione flessibili ed asincrone, e si basa sul principio di imporre al programmatore meno vincoli possibile (*"modeless"*). I programmi dotati di interfaccia grafica sono tipicamente realizzati secondo l'approccio *event-driven*. I sistemi operativi sono un altro classico esempio di programmi *event-driven* su almeno due livelli. Al livello più basso i gestori di interrupt gestiscono gli eventi innescati dall'hardware, con il processore principale che agisce da *dispatcher.* Ad un livello superiore i sistemi operativi fungono da dispatcher per i processi attivi, passando dati ed, eventualmente, altri interrupt ad altri processi, che possono, a loro volta, gestire ad un livello più alto gli stessi eventi.

Un sistema, o programma, *controllato da comandi* può considerarsi un caso particolare di sistema o programma *event-driven*, in cui il sistema, normalmente inattivo, aspetta che si verifichi un evento molto particolare, cioè l'invio di un comando da parte dell'utente.

Lezione 2 – Finestre e Contesto Grafico

Finestra: Area rettangolare dove il "programma può disegnare" (prendere leggittimamente possesso di un certo numero di pixel sullo schermo con il benestare del kernel), può contenere anche altre finestre e non include invece il bordo ed i pulsanti di servizio.

Contesto Grafico:

Wikipedia:
E' una struttura che contiene i parametri delle operazioni grafiche. Un contesto grafico include un colore di primo piano (foreground), un colore di sfondo (background), il font del testo e altri parametri grafici. Quando richiede un'operazione grafica, il client include un contesto grafico. Non tutti i parametri del contesto grafico sono necessari per l'operazione: per esempio, il font non ha alcun effetto nel disegnare una linea.

Wikipedia:
Il **framebuffer** è una memoria buffer della scheda video nella quale vengono memorizzate le informazioni destinate all'output per la rappresentazione di un intero fotogramma (*frame* in lingua inglese, da cui il nome *framebuffer*) sullo schermo.

Nel *framebuffer* sono contenute le informazioni sul colore di ciascun pixel, ovvero di ciascun punto dello schermo, che possono essere a 1-bit (bianco e nero), 4-bit o 8-bit (modalità a tavolozza), 16-bit (Highcolor) o 24-bit (Truecolor). È a volte presente un canale alfa che contiene informazioni sulla trasparenza del pixel. Le dimensioni del *framebuffer* dipendono dalla risoluzione dell'output video e dalla profondità di colore.

Nella grafica rastering, che è il modello tramite il quale stiamo programmando in questo corso, il sistema grafico (approfondire) non "ha memoria" dei disegni che stiamo realizzando, se vogliamo che essi siano persistenti dobbiamo ogni volta reinserire le direttive nel **framebuffer**, il quale verrà letto dalla scheda video per disegnare a schermo.

L'evento “onPaint” è un evento speciale nel modello a grafica rasterizzata, infatti l'handler che viene eseguito al momento della sua chiamata (causata ad esempio da un resize, messa a fuoco, invalidate, …) è quello che va a determinare i contenuti del **framebuffer** sulla scheda video.

Contrapposta alla grafica raster c'è la grafica a retention, quale tipologia scegliere sta al programmatore deciderlo a seconda delle caratteristiche del progetto.

Wikipedia:
La **grafica bitmap**, o **grafica raster** (in inglese bitmap graphic, *raster graphics*, mentre in italiano sarebbe traducibile come: *Grafica a griglia*), è una tecnica utilizzata per descrivere un'immagine in formato digitale. Un'immagine descritta con questo tipo di grafica è chiamata *immagine bitmap* o *immagine raster.*

La grafica bitmap si contrappone alla grafica vettoriale.

Wikipedia:
La bitmap è caratterizzata da due proprietà:

- risoluzione;
- profondità.

La prima è determinata dal numero di pixel contenuti nell'unità di misura considerata (in genere il pollice inglese, equivalente a 2,54 cm); si misura in *PPI* (*Pixel Per Inch*) oppure in *DPI* (*Dot Per Inch*, Punti per pollice). La seconda è definita dalla memoria che si dedica ad ogni pixel, ovvero dal numero di bit dedicati ad ogni pixel per descrivere il colore, e si misura in *BPP* (*Bit Per Pixel*); maggiore è il numero di bit, maggiore è il numero di colori che è possibile descrivere.

Un altro comune metodo per indicare la qualità delle immagini raster, tipico della Fotografia digitale, è moltiplicare il numero delle righe di pixel per quello delle colonne di pixel ed esprimere il valore in Megapixel.

La grafica bitmap non è vantaggiosa se l'utente necessita di apportare modifiche all'immagine, perché nel caso ad esempio di uno zoom, la risoluzione e quindi la qualità dell'immagine peggiora. I software grafici, per ridurre il problema, sono in grado di ripristinare la risoluzione inserendo nuovi pixel che vengono calcolati facendo una interpolazione, il processo inserisce, perciò, deliberatamente una quantità di informazioni presunte.

La grafica bitmap è invece ideale per rappresentare immagini della realtà, per modificare contrasti e luminosità di queste, per applicare filtri di colore.

Wikipedia:
I dati raster possono essere memorizzati attraverso tipologie di file che sfruttano algoritmi di compressione diversi, gravando in modo differente sul supporto di memorizzazione. I formati raster più comuni sono i seguenti:

Non compressi

Questi formati di file hanno richieste di elaborazione minima, non essendo necessari algoritmi di

compressione (in fase di scrittura) e decompressione (in fase di lettura), tuttavia, mancando di compressione, risultano particolarmente voluminosi, in termini di spazio occupato su disco (o altro dispositivo di memorizzazione), rispetto agli altri formati:

- raw
- bmp (in alcuni casi i file bmp sono compressi con un algoritmo RLE)

Con compressione lossless

Le immagini salvate con un algoritmo di compressione dati lossless occupano meno spazio nei dispositivi di memorizzazione, mantenendo inalterata tutta l'informazione originale:

- png (certe applicazioni permettono anche la scrittura di file png non compressi)
- tga
- tiff (sebbene questo sia l'uso più comune, questo formato permette diversi tipi di compressione)
- gif (per immagini fino a 256 colori)

Con compressione lossy

Le immagini memorizzate con un algoritmo di compressione lossy subiscono una perdita di informazione; pertanto questa tecnica non è adatta per *salvare* le immagini che vengono rielaborate coi programmi di fotoritocco (le continue modifiche comporterebbero un progressivo degrado dell'immagine ad ogni salvataggio e riapertura); invece, in virtù delle ridotte dimensioni del file, sono particolarmente indicate per la trasmissione di immagini o per ridurre le dimensioni di un'applicazione o di un prodotto da distribuire.

- jpeg
- gif (per immagini con più di 256 colori si ottiene una compressione **lossy** poiché vengono eliminate la maggior parte delle sfumature di colore)

La grafica a retention è detta anche grafica vettoriale. E' originata da primitive grafiche che producono “oggetti con proprietà” i quali vengono inseriti in un albero che li memorizza, quando si vuole memorizzare il disegno a schermo si va ad agire sul suddetto albero, non più direttamente sul disegno (approfondimenti tra qualche pagina).

Wikipedia:
La **grafica vettoriale** è una tecnica utilizzata in computer grafica per descrivere un'immagine. Un'immagine descritta con la grafica vettoriale è chiamata **immagine vettoriale**.

Nella grafica vettoriale un'immagine è descritta mediante un insieme di primitive geometriche che definiscono punti, linee, curve e poligoni ai quali possono essere attribuiti colori e anche sfumature. È radicalmente diversa dalla grafica raster in quanto nella grafica raster le immagini vengono descritte come una griglia di pixel opportunamente colorati.

Wikipedia:
I principali vantaggi della grafica vettoriale rispetto alla grafica raster sono i seguenti:

- possibilità di esprimere i dati in una forma direttamente comprensibile ad un essere umano (es. lo standard SVG);
- possibilità di esprimere i dati in un formato che occupi (molto) meno spazio rispetto all'equivalente raster;
- possibilità di ingrandire l'immagine arbitrariamente, senza che si verifichi una perdita di risoluzione dell'immagine stessa.

Il primo punto si traduce nella possibilità, per una persona, di intervenire direttamente sull'immagine anche senza fare uso di programmi di grafica o addirittura senza conoscenze approfondite in merito. Ad esempio, per tradurre il testo presente in un'immagine SVG, spesso è sufficiente aprire il file con un editor di testo e modificare le stringhe lette nel file.

Tale sistema di descrizione delle informazioni grafiche presenta inoltre l'indubbio vantaggio di una maggiore compressione dei dati: in pratica una immagine vettoriale occuperà molto meno spazio rispetto ad una corrispondente raster, con una riduzione dell'occupazione di RAM e memoria di massa, principalmente nelle forme geometriche o nei riempimenti a tinta piatta. Risulta, inoltre, più facile da gestire e da modificare, essendo minore la quantità di dati coinvolti in ogni singola operazione di aggiornamento. Questo rende il vettoriale particolarmente adatto per gestire grandi quantità di dati come quelli cartografici che sono tipicamente gestiti in modalità vettoriale; infine l'ingrandimento o la riduzione delle misure e proporzioni del soggetto prodotto in vettoriale non incide in maniera significativa sul *peso* dell'immagine stessa, il riempimento di forme con *tinte piatte* è generato da semplici funzioni matematiche e risulta, quindi, estremamente leggero in termini di memoria utilizzata.

Wikipedia:
La grafica vettoriale, essendo definita attraverso equazioni matematiche, è indipendente dalla risoluzione, mentre la grafica raster, se viene ingrandita o visualizzata su un dispositivo dotato di una risoluzione maggiore di quella del monitor, perde di definizione. Una linea che percorre lo schermo trasversalmente se viene rappresentata utilizzando la grafica raster viene memorizzata come una sequenza di pixel colorati disposti a formare la linea. Se si provasse ad ingrandire una sezione della linea si vedrebbero i singoli pixel che compongono la linea. Se la medesima linea fosse memorizzata in modo vettoriale la linea sarebbe memorizzata come un'equazione che parte da un punto identificato con delle coordinate iniziali e termina in un altro punto definito con delle coordinate finali. Ingrandire una sezione della linea non produrrebbe artefatti visivi o la visualizzazione dei singoli pixel componenti l'immagine, dato che la linea sarebbe visualizzata sempre con la massima risoluzione consentita dal monitor.

Ad esempio, prendendo un'immagine vettoriale grande 2x24 pixel e aumentando la risoluzione fino a 1024x768, si otterrà una immagine che ha la stessa definizione di quando era 2x24. Se noi procediamo con lo zoom su di un file di tipo bitmap adattato a 800x600 px e lo ingrandiamo a 1024x768 px la definizione risulta inferiore, questo perché il processore non ricalcola la definizione dell'immagine come succede nelle immagini vettoriali ma utilizza le stesse informazioni (pixel) dell'immagine su una superficie maggiore.

Il principale svantaggio della grafica vettoriale rispetto alla grafica raster è che la realizzazione di immagini vettoriali non è una attività intuitiva come nel caso delle immagini raster. I programmi vettoriali dispongono di molti strumenti che, per essere sfruttati pienamente, richiedono svariate conoscenze. Un altro difetto è legato alle risorse richieste per trattare le immagini vettoriali: una immagine vettoriale molto complessa può essere molto *corposa* e richiedere l'impiego di un computer molto potente per essere elaborata. Inoltre, le risorse richieste per trattare l'immagine non sono definibili a priori e quindi ci si potrebbe trovare nell'impossibilità di elaborare un'immagine per la mancanza di risorse sufficienti. Nel caso di un'immagine raster, invece, una volta definita la risoluzione ed il numero di colori, è abbastanza semplice definire le risorse massime necessarie per trattare l'immagine stessa; al contrario di quanto accade con le *tinte piatte*, i riempimenti sfumati o complessi generati in vettoriale comportano un alto impiego di risorse.

La grafica vettoriale ha un notevole utilizzo nell'editoria, nell'architettura, nell'ingegneria e nella grafica realizzata al computer. Tutti i programmi di grafica tridimensionale salvano i lavori definendo gli oggetti come aggregati di primitive matematiche. Nei personal computer l'uso più evidente è la definizione dei font. Quasi tutti i font utilizzati dai personal computer vengono realizzati in modo vettoriale, per consentire all'utente di variare la dimensione dei caratteri senza perdita di definizione.

Lezione 3 – Prime Trasformazioni

1/71 di pollice è un punto tipografico, importante quando si misurano i font.

Lo schermo è composto da una matrice di punti, per disegnare si specificano delle coordinate bidimensionali (x , y). Nella visione fisica di questo mondo i sistemi grafici usano l'angolo in alto a sx come (0,0) o punto di origine, con l'esempio dell'orologio abbiamo visto che ci sarebbe tornato meglio avere il centro in un altro posto per facilitare i calcoli, quindi abbiamo traslato il centro da (0,0) a (100,100) questa cosa avviene normalmente nei sistemi grafici, ma non solo la traslatura, anche la scalatura e la rotazione: le cosiddette TRASFORMAZIONI AFFINI DEL PIANO.

n.b. Nell'esempio dell'orologio, per introdurre il concetto di trasformazioni affini, invece di ammazzarci di calcoli per trovare la posizione esatta di ogni tick del quadrante, avevamo ruotato il contesto grafico e disegnato il tick all'interno di un loop (l'effetto è quello di girare il foglio su cui stiamo lavorando di un tot. gradi, disegnando sempre nel solito punto.

n.b. Il contesto grafico è una struttura dati, e come tale si può memorizzare, ripristinare e modificare (entro certi limiti).

Una **trasformazione lineare** è una funzione fra spazi che preserva le combinazioni lineari (operazioni di somma e di moltiplicazione per scalare) mentre una **trasformazione affine** è una trasformazione esprimibile come somma di una trasformazione lineare seguita da una traslazione .

Usando le **coordinate omogenee**, tutte le trasformazioni affini possono essere espresse come una semplice moltiplicazione tra matrici. (coord cartesiane : x,y ; coord omogenee: x,y,1).

Esempio: moltiplicazione fra matrici 3x3

[1 2 0 3 0 2]×[2 0 3 2 0 1]=[7 4 6 2]

TRASLAZIONE(nel punto x+a, y+b):

Coordinate cartesiane → p=[x y], d=[a b]: p + d=[x+a | y+b]

Coordinate omogenee → delta=[1 0 a | 0 1 b | 0 0 1], p=[x y 1]: deltap=[xa | yb | 1]

SCALATURA UNIFORME(di fattore a):

Coordinate cartesiane → p=[x y]: ap=[ax ay]

Coordinate omogenee → A=[a 0 0 | 0 a 0 | 0 0 1],p=[x y 1]: Ap=[ax | ay | 1]

ROTAZIONE(di angolo t):

Coordinate cartesiane → p=[x y]: p t=[xcos(t)−ysin(t) | xsin(t) + ycos(t)]

Coordinate omogenee → P=[cos(t) − sin(t) 0 | sin(t) cos(t) 0 | 0 0 1],p=[x y 1]: Pp=[xcos(t) − ysin(t) | xsin(t) + ycos(t) | 1]

Il vantaggio delle coordinate omogenee è che tutte le operazioni di trasformazioni diventano moltiplicazioni, che possono essere combinate tra loro e salvate in matrici che effettuano una o più trasformazioni simultaneamente (ma comunque mantenendo un ordine). Molte operazioni di disegno non modificano la posizione dei punti che lo costituiscono, ma si limitano a cambiare la matrice di trasformazione associata alla figura.

n.b. Salvare e ripristinare il controllo grafico ad ogni repaint (eseguita potenzialmente anche a 60fps) è niente in confronto alle operazioni che svolge di routine la GPU, non se ne accorge nemmeno.

Lezione 4 – Controlli Grafici e Matrici di Trasformazione

Un **controllo grafico** è un oggetto (è sottinteso quindi che ha uno stato interno) che ha associata una finestra (**window**: rettangolo di pixel che il sistema grafico mette a disposizione per il disegno) in cui possiamo disegnare liberamente e ricevere eventi. In pratica è una componente di un interfaccia grafica che reagisce agli input dell'utente. Sono astrazioni di programmazione, che noi abbiamo ridefinito in Light Weight Controls: in pratica è un'interfaccia che rappresenta un controllo grafico ma non deriva direttamente dalla struttura (UserControl in f#, Qwidget in Qt, …) che il sistema su cui stiamo lavorando ci fornisce, trattiene delle informazioni che permettono al suo contenitore – un Light Weight Controls Container – di disegnarlo ogni volta che è richiesto un refresh (LWCC eredita direttamente da UserControl).

n.b. E' un errore grave non supportare interfacce multi canale! Ci sono aspetti di accessibilità ed usabilità che vanno sempre tenuti in considerazione, per non parlare di produttività: la tastiera è molto più produttiva rispetto al mouse.

Le matrici di trasformazione “vista-mondo” → v2w e “mondo-vista” → w2v fanno parte di una tecnica utilizzata spesso nella programmazione grafica, queste matrici sono legate al contesto

grafico, gli vengono applicate al momento del disegno delle componenti dell'interfaccia ed al momento della lettura delle coordinate degli eventi occorsi.

In una applicazione grafica avremo solitamente uno spazio di **coordinate del modello** (lo spazio cui ogni singolo oggetto modellato viene riferito), uno spazio di **coordinate del mondo** (descrive in modo unificato l'intero universo) ed uno **spazio di coordinate del dispositivo** (caratteristico della periferica di output utilizzata, gestito quindi dalla GPU).

Generalmente le trasformazioni di vista sono precedute dall'operazione di **clipping** (ritaglio degli elementi fuori della scena), esattamente la stessa che effettua la GPU prima di colorare i pixel sullo schermo.

Dunque la tecnica che abbiamo usato a lezione (esercizio sull'Editor di Beziér) consiste nell'associare due matrici di trasformazione al contesto grafico dell'applicazione (potrebbero benissimo essere più di una – in sostanza nel 2D non sono altro che rettangoli da mappare uno dentro l'altro), abbiamo inserito degli oggetti nella “scena”, quando vogliamo spostare uno di questi quello che facciamo nella “onPaint” dell'Editor di Beziér è:

- salvare il contesto grafico
- applicare le trasformazioni alla w2v
- inserire la w2v come matrice del contesto grafico
- disegnare le curve (che si trovano in coordinate mondo)
- ripristinare il contesto grafico

n.b. La proprietà "Transform" del contesto grafico, che come suggerisce il nome non è altro che una matrice di trasformazione (3x3 poichè lavoriamo in 2D), viene applicata ad ogni punto che il contesto grafico andrà a disegnare sullo schermo.

Quando invece andiamo a leggere le coordinate di un evento, come “onMouseDown” ecco il workflow:

- trasformiamo il punto del click con la matrice v2w per modificare le coordinate dell'evento in coordinate del mondo (le coordinate degli eventi sono sempre in coordinate vista!)
- effettuiamo i controlli necessari (ad esempio un matching con i controlli o i disegni presenti nella “scena” per vedere se siamo in una posizione particolare o interessante)

Perchè usare i LWC? Quando si alloca un controllo grafico si chiede al sistema grafico di riservare una finestra grafica che lo contiene, ciò causa un overhead, inoltre sto portandomi dietro dei vincoli (ho una finestra sopra l'altra, gestione eventi, ecc ecc). Allora si introduce un sistema chiamato “light weigth controls” (si tratta in sostanza di rifare un pezzetto di sistema grafico in casa) utilizzato per rendere i controlli utente meno pesanti e per avere più libertà di utilizzo e più efficienza.

Nota sulle matrici di trasformazione: Le due matrici w2v e v2w servono per le fasi di trasformazione dei punti, se per esempio voglio traslare il punto a=(0,0) di 10 lungo l'asse x, prima di disegnarlo nella OnPaint dico al sistema grafico di moltiplicarlo per la matrice w2v(10,0);

quando invece mi posiziono sul punto (che ora si trova in 10,0) prima di "leggerlo" moltiplico a per v2w(-10,0) in modo tale che "legga" il punto a=(0,0).

Lezione 5 – Il paradigma MVC ed altre nozioni

L'assunzione in pixel delle coordinate non funziona più, in virtù dell'aumentata potenza di calcolo si possono costruire interfacce che usano coordinate virtuali. Le densità di pixel sugli schermi odierni fanno si che un singolo pixel non sia neanche visibile.

Ogni evento è costituito da un “trigger” ed una coda di handlers che vengono eseguiti al momento dello “scatto”. Il “trigger” è una funzione che richiama tutti i metodi registrati nella coda di handlers chiamata “publish”.

In genere il click deve avvenire solo nel caso in cui mouseDown e mouseUp vengono effettuati sopra il solito controllo, la soluzione più comune è una tecnica detta : **mouseCapturing.** Quando avviene mouseDown, il controllo che lo ha ricevuto continua a ricevere eventi del mouse anche se sono fuori dalla sua finestra di controllo, poi sta a chi programma gli handlers degli eventi che stiamo definendo scegliere cosa fare alla ricezione di tali eventi.

Il paradigma MVC: **model – view – controller** è un pattern di programmazione utilizzato nelle interfacce grafiche (ma non solo). La vista non contiene necessariamente tutte le informazioni che vengono manipolate dal sistema grafico, l'applicazione grafica ha un modello che rappresenta tutto l'insieme dei dati necessari a produrre la vista. Dato il modello, si riesce a proiettare l'informazione necessaria al momento sulla vista, alcune di queste informazioni saranno transienti (editor di curve: era necessario ricordarsi che quando veniva premuto il bottone, il mouseDown veniva gestito in maniera diversa) quindi il modello contiene sia informazioni che possono finire sul disco ma contiene anche informazioni utili alla produzione della vista che magari sono di scarso rilievo rispetto allo scopo finale dell'applicazione. Il **controllo** è relativo alla gestione dell'input, l'utente 'vede' la vista ed interagisce con il controllo (che può essere parecchio sofisticato, alcune app consentono di manipolare il modello attraverso vari comandi che possono essere eseguiti secondo modalità diverse come tastiera, mouse, voce, ...) il quale a sua volta va a modificare il **modello** che a sua volta riporta i cambiamenti (se ci sono) sulla **vista**. Il MVC è un pattern molto diffuso (anche se non è l'unico, si veda Document View Controller per esempio, Model View View Controller MVVC che è una variante del MVC).

Si noti che quando si gioca online il modello sta sul server, mentre la vista, una copia locale del modello ed il controller sono locali all'utente. Quando la rete è intasata ed il modello locale del gioco non riesce a speculare più, ad esempio, sulle reali posizioni degli altri giocatori, si ha il classico effetto 'teletrasporto'. Il MVC è a volte usato esplicitamente, esistono delle classi che hanno nome 'Model', 'Control' e 'View' in alcuni framework (sopratutto applicazioni web graphic).

Wikipedia:
Il **Model-View-Controller** (**MVC**, talvolta tradotto in italiano **Modello-Vista-Controllo**), in informatica, è un pattern architetturale molto diffuso nello sviluppo di sistemi software, in particolare nell'ambito della programmazione orientata agli oggetti, in grado di separare la logica di presentazione dei dati dalla logica di business.

Questo pattern si posiziona nel livello di presentazione in una Architettura multi-tier.

Wikipedia:
Il pattern è basato sulla separazione dei compiti fra i componenti software che interpretano tre ruoli principali:

- il **model** fornisce i metodi per accedere ai dati utili all'applicazione;
- il **view** visualizza i dati contenuti nel model e si occupa dell'interazione con utenti e agenti;
- il **controller** riceve i comandi dell'utente (in genere attraverso il view) e li attua modificando lo stato degli altri due componenti.

Questo schema, fra l'altro, implica anche la tradizionale separazione fra la logica applicativa (in questo contesto spesso chiamata "logica di business"), a carico del *controller* e del *model*, e l'interfaccia utente a carico del *view*.

I dettagli delle interazioni fra questi tre oggetti software dipendono molto dalle tecnologie usate (linguaggio di programmazione, eventuali librerie, middleware e via dicendo) e dal tipo di applicazione (per esempio se si tratta di un'applicazione web, o di un'applicazione desktop).

Lezione 6 – Double Buffering e Animazioni

A lezione abbiamo modellato un'applicazione che “lancia palline rimbalzanti” per analizzare alcuni aspetti legati all'animazione degli oggetti. Una volta implementato del basi del modello MVC abbiamo potuto notare il famoso fenomeno del “flickering” durante le animazioni della pallina, perché accade? Pochi sanno che è un fenomeno **inevitabile** ed è attenuato solo grazie a tecniche come quella del Double Buffering che descriveremo tra poco, tale fenomeno è legato all'asincronia tra eventi di paintBackground ed onPaint, il “**framebuffer**” viene aggiornato dalla scheda video in maniera diversa da quella richiesta dall'intervallo di animazione della palla.

Il Double Buffering in sostanza implementa una pipeline, prima disegna “offline” senza toccare il framebuffer, una volta completato il disegno buttiamo la bitmap nel framebuffer ed il flickering scompare. Ogni buon framework fornisce la propria implementazione del Double Buffering.

n.b. In parole povere il problema grosso è che l'intervallo di tempo che trascorre tra uno svuotamento del framebuffer ed il successivo non è prevedibile ne programmabile dall'utente. La scheda video effettua la stampa a video ad intervalli che dipendono da come è stata programmata per ottimizzare le operazioni di calcolo ed il carico di lavoro, quindi se non usiamo il DB può capitare che mentre “noi” siamo impegnati a disegnare elementi da inserire nel framebuffer, la GPU decida che è il momento di stampare a video i contenuti, dunque prima dipinge l'area interessata con il colore di sfondo, poi stampa il contenuto del fb, ecco spiegato in breve il flickering: ovvero l'alternarsi del colore di background al disegno animato.

Nell'esempio precedente avevamo realizzato le **animazioni** andando a modificare direttamente la posizione della pallina di un determinato offset ad ogni tick di un timer impostato da noi. La posizione di un oggetto non è l'unica proprietà animabile, possiamo agire sulla dimensione ad esempio, colore, … etc etc. Il problema di base delle animazioni è trovare una maniera per descrivere come alcune determinate proprietà di un insieme di controlli debba modificarsi in

funzione del tempo.

Il movimento inteso come traiettoria può essere decomposto lungo i due assi cartesiani (parlando di 2D), in questo caso si parla di **forma parametrica** : x(t) = fx(t); y(t) = fy(t). Le forme parametriche sono insiemi di funzioni (sistemi) regolate da (almeno) un parametro comune a tutte. L'**automa temporizzato** è il modo naturale di descrivere l'evoluzione delle proprietà, una volta stabilite le equazioni che regolano, ad esempio, il moto di un oggetto le approssimiamo con delle forme parametriche.

Un metodo alternativo per descrivere le animazioni consiste nell'utilizzo di **storyboard** (linee guida a fumetto per le riprese nei films) : una volta impostati i keyframe il calcolatore **interpola** i valori delle proprietà tra un kf e l'altro usando le cosiddette **easing functions**.

Wikipedia:
In matematica l'**equazione parametrica** o **letterale** è un'equazione matematica in cui le variabili (indipendente e dipendente) sono espresse a loro volta in funzione di uno o più parametri. Un tipico parametro potrebbe essere il tempo (*t*): esso, in equazioni riguardanti la cinematica, è utilizzato per stabilire la velocità, l'accelerazione e altri aspetti del movimento. Il contrario di equazione parametrica è equazione numerica.

Per esempio, una generica retta di equazione cartesiana

$$ax + by + c = 0$$

come equazione parametrica diventa:

$$\begin{aligned} x &= x_0 + \alpha t \\ y &= y_0 + \beta t \end{aligned}$$

e il parametro *t* è dato da: $t = \frac{x - x_0}{\alpha}$ ($\alpha = b$ e $\beta = -a$)

L'equazione di una parabola, $y = x^2$ può essere parametrizzata in funzione del parametro *t*, ponendo

$$\begin{aligned} x &= t \\ y &= t^2 \end{aligned}$$

La parametrizzazione di una circonferenza di raggio *r* e centro nell'origine ($x^2 + y^2 = r^2$) è:

$$\begin{aligned} x &= r\cos(t) \\ y &= r\sin(t) \end{aligned}$$

Wikipedia:
In matematica, e in particolare in analisi numerica, per **interpolazione** si intende un metodo per individuare nuovi punti del piano cartesiano a partire da un insieme finito di punti dati, nell'ipotesi che tutti i punti si possano riferire ad una funzione *f*(*x*) di una data famiglia di funzioni di una variabile reale.

Nelle attività scientifiche e tecnologiche, e in genere negli studi quantitativi di qualsiasi fenomeno, accade molto spesso di disporre di un certo numero di punti del piano ottenuti con un campionamento o con apparecchiature di misura e di ritenere opportuno individuare una funzione che passi per tutti i punti dati o almeno nelle loro vicinanze (vedi curve fitting).

L'attività di **inbetwining** o **tweening** è il processo che porta alla generazione di frames intermedi tra due keyframe della storyboard. Questa attività è supportata dalle **easing functions**: funzioni che passano per determinati punti noti (keyframes) – ne esistono varie tipologie che simulano più o meno determinati "effetti fisici" - .

n.b. La **collision detection** è l'attività di decidere ad ogni tick del timer, se alcuni oggettti sono in collisione con altri oggetti.

Lezione 7 – Sistema grafico [IUM]

Il sistema grafico (oggi praticamente ovunque) è parte del sistema operativo, ma è nato come modulo indipendente da quest'ultimo, su unix ad esempio è un processo (x-server) che agisce da server con i processi "cliente". I sistemi grafici forniscono i componenti di base per le interfacce grafiche ovvero il disegno, lo spostamento delle finestre sullo schermo e l'interazione con le periferiche di input quali mouse e tastiera. X ad esempio non gestisce invece l'interfaccia grafica utente cioè lo stile grafico delle applicazioni che vengono gestite invece dall'ambiente desktop scelto dall'utente e in uso sul computer.

Wikipedia su X11 Windows System:

La macchina dove girano i programmi (client) non deve essere necessariamente la macchina locale (*display server*). I termini *server* e *client* vengono spesso confusi: per X il server è il display locale dell'utente, non una macchina remota. Questo permette anche di visualizzare sullo stesso display applicazioni che vengono eseguite su diversi host, oppure che su un host vengano eseguite applicazioni la cui interfaccia grafica finisce su diversi display.

X è ormai usato secondariamente anche da sistemi operativi che non lo supportano nativamente (es. Mac OS X, che usa nativamente Quartz), per permettere il funzionamento del software progettato per questo sistema grafico, come la suite da ufficio OpenOffice.org e GIMP.)

Ultimamente però diversi OS unix-like hanno introdotto specifici moduli nel kernel proprio come le controparti Windows e MacOS.

La grafica richiede potenza e cicli macchina ed ogni volta che si attraversa la barriera user/kernel si sprecano cicli, per cui oggi sta praticamente tutto nel kernel (anche se così quest'ultimo diventa più instabile). Oggi tutto cresce in fretta, il cavo hdmi (uscito solo pochi anni fa) ad esempio non supporta il 4K, solo pochi anni fa l'ecosistema dei dispositivi attorno al microprocessore era estremamente più lento di quello odierno, adesso non è più vero, ormai il collo di bottiglia sta diventando proprio il processore stesso. Il sistema oggi non può permettersi di sprecare cicli macchina, l'hw che abbiamo a disposizione in questo momento è veramente al limite.

Il sistema operativo ci fornisce delle bellissime astrazioni : i file. Queste astrazioni ci vengono messe a disposizione direttamente dal file system, il sistema grafico svolge un ruolo analogo: prende una risorsa comune e la rende disponibile ai vari processi, ha a disposizione dei pixel invece

di blocchi di dati da scrivere, ma da punto di vista filosofico il sistema grafico non è altro che un componente software del sistema operativo che svolge lo stesso lavoro del file-system. Il sistema grafico è un modulo che si inserisce tra framebuffer, input devices, etc etc … e queste sono le risorse che lui deve sezionare tra i vari clienti (processi). Il processo p che vuole utilizzare funzionalità grafiche fa esattamente quello che fa per aprire un file: chiede al modulo grafico una risorsa grafica: createWindow() che è una primitiva di sistema, ed al processo p viene restituita un handle che rappresenta la finestra associata (la handle è come se fosse un file). Tutto ciò deve essere sensibile agli eventi (sotto forma di interrupts), contrariamente agli eventi e comunicazioni come abbiamo visto a SO, gli interrupts sono la normalità nella gestione della grafica, non rappresentano situazioni eccezionali, sono la normalità.

La coda degli eventi, allo stato attuale delle cose, è sempre associata ad un processo, cosa succede se inserisco una sleep nell'handler di un evento? La finestra si blocca, ma il resto del sistema procede regolarmente!

Una volta la coda degli eventi era unica per tutto il sistema, se una finestra si bloccava, tutto il sistema grafico si bloccava. Oggi ad ogni processo P è associata una coda degli eventi, questa funziona esattamente come funzionano i LWC, c'è una correlate direttamente nel sistema che associa gli eventi e la loro posizione alle finestre relative. Oggi è possibile fare una finestra od un bottone non rettangolare grazie alle osservazioni fatte sopra.

Importante: Non è responsabilità del sistema grafico consumare gli eventi nella coda di un processo, il sistema grafico funziona in modalità pull, è responsabilità del processo interrogare la propria coda per controllare la presenza di eventi. Un thread del processo è designato apposta per eseguire gli handler registrati: il famoso **event loop**:

```
while(peekMessage(w)!= quit){
        dispatch(w)
}
```

E' un loop(in un thread del processo) che usa la libreria del sistema grafico (una API che contiene funzioni che fanno il dispatch del messaggio all'interno del processo) con cui interagisce (Application.DoEvents() esegue un ciclo degli eventi in F#). Se il thread della user interface non gestisce la coda degli eventi, la paint non arriva! Il mouse non arriva! ecc ecc... lo sleep nel bottone che abbiamo visto nell'esempio è una simulazione propria di questo caso.

Quando si fanno interfacce grafiche che devono eseguire calcoli pesanti, è una PESSIMA idea eseguirle nell'handler di un evento! Se devo eseguire un calcolo pesante posso allocare un thread, farlo calcolare e nel frattempo seguire il corso degli eventi. Tanti frameworks però non sono thread safe -> se proviamo a far cambiare un proprietà del controllo grafico da un thread ci viene sollevata un'eccezione perchè non c'è una sincronizzazione tra thread grafici. Ci sono degli helper (Application.Dispatch(fun ->) che ci permettono di simulare un thread postponendo l'esecuzione della funzione specificata.

IMPORTANTISSIMO → La domanda ESSENZIALE: chi invoca l'evento click? RISPOSTA:

un thread del processo stesso, quindi il processo stesso!

Il sistema grafico è tecnologia risalente agli anni 70, dentro il processo c'è la coda degli eventi che arrivano e gli handlers collegate alle finestre grafiche che il sistema grafico gli ha assegnato (possono essercene anche più di una). Assumiamo che arrivi l'evento (window1, mousedown, x, y) nella coda, quando facciamo il dispatch chiamiamo una funzione del sistema grafico WindowProcedure realtiva alla finestra che ha ricevuto l'evento. Ogni finestra ha una **WindowProcedure**, ovvero un puntatore a funzione. Quando viene invocata gli viene passato l'evento associato. Se noi creiamo eventi fittizi e li inviamo alle applicazioni questi vengono eseguiti come se fossero veri (si vedano quei programmi che registrano macro). La window procedure (ogni finestra ne ha una per tutti i tipi di evento che possono avvenire) ha come argomento l'evento e viene chiamata dal thread che gestisce la coda eventi. Adesso la finestra sa che è appena successo qualcosa. Gli eventi vanno gestiti tutti: non ce ne accorgiamo ma il framework ha i suoi default handler.

```
WindowProcedure(m){
 switch(m.type):
  case MouseDown: mouseDown(m)
  case Paint: ...
  case ... : ...
}
```

n.b. le librerie grafiche ed i sistemi grafici sono praticamente tutti scritti in C

Nello stack delle librerie grafiche c'è una parte della libreria grafica di sistema, sopra di esso troviamo un livello che fornisce un po' di astrazioni sulle primitive tipo Windows.Forms, GTK, Python, etc ...

Gli handlers con cui noi vogliamo gestire gli eventi vengono chiamati dalla winproc --> chiamata dal thread del processo --> che la prende dalla coda eventi del processo --> nella coda è stata messa dal sistema grafico.

Dentro il sistema grafico ci sono altre due strutture dati: **l'albero delle finestre** e **l'albero dell'ereditarietà**. Gli eventi si propagano su entrambi gli alberi, un evento in generale può galleggiare nell'albero delle finestre attraverso il processo di **bubbling** (se non gestisco io magari gestisce il mio contenitore) o affondare tramite il **tunnelling** (inverso del bubbling).

Finestre: Desktop(Form1(Textbox,Bottone),Form2(...),...)

Ereditarietà: Control(UserControl(LWContainer(Editor(...))))

Il dispatching degli eventi che abbiamo implementato quando abbiamo scritto l'interfaccia LWC (la funzione correlate in sostanza) è in pratica ciò che fa il sistema grafico quando deve distribuire gli eventi occorsi.

Lezione 8 – Interazione ed Input [IUM]

L'input alle interfacce grafiche non è più costituito dai soli mouse e tastiera, kinect, leap motion, etc. Il paradigma di programmazione ad eventi però non verrà sconvolto, solo complicato, perchè il numero di dispositivi e di eventi sta aumentando a dismisura (riconoscimento di una sequenza di eventi e non solo il singolo evento come pinch & zoom, movimenti composti della mano, ...)

Nell'equazione interazione uomo macchina, l'uomo è una costante, non si può aggiornare il firmware umano. Una volta, nell'immaginario collettivo, l'interazione uomo macchina veniva concepita come l'uomo che si adatta alle macchine (basti pensare che per caricare un videogioco su commodore si dovevano lanciare dei comandi macchina (magari vedere libro: the second machine age - Andrew McAfee). La prima rivoluzione industriale ha automatizzato la produzione di energia moltiplicando la capacità produttiva dell'uomo in maniera significativa, nelle mani dell'uomo è rimasto il controllo, l'automazione industriale ha distrutto la classe operaia. Molti processi industriali prevedono che l'uomo sia relegato solo a controlli di qualità e mansioni di controllo sull'operato delle macchine. Basta pensare anche ai piloti di aerei che stanno scomparendo sostituiti da droni più veloci nelle reazioni.

Oggi stiamo assistendo all'automazione del controllo, una seconda rivoluzione industriale, una volta i grafici facevano il make up della interfaccia con photoshop, una volta decisa si passava il modello al programmatore che iniziava a lavorare sopra per realizzare l'interfaccia vera e propria, inizia una negoziazione vera e propria tra varie figure.

Lezione 9 – La pipeline di rendering [IUM]

Cosa succede da Drawline(black, 0, 0 100, 100) al disegno effettivo della linea? La **rendering pipeline** è un sistema di processing a stadi (una catena) in cui ciascuno stadio prende in ingresso uno stadio più basso, lo raffina e manda fuori qualcosa da dare in pasto ad uno stadio successivo (vedere il capitolo associato sul libro di Cignoni).

Wikipedia:
Una **pipeline grafica**, nella computer grafica tridimensionale, è una pipeline dati designata alla trasformazione dei modelli tridimensionali del mondo in immagini bitmap bidimensionali.

Per realizzare l'immagine bitmap la pipeline può implementare uno o più algoritmi come lo Z-buffering, il reyes rendering e il ray tracing e altri algoritmi.

Wikipedia:
La pipeline può essere realizzata in software o in hardware sebbene per questioni di velocità e di prestazioni tutte le moderne schede grafiche dispongono di diverse pipeline grafiche più o meno avanzate. Sebbene esistano molte implementazioni di una pipeline grafica tutte queste implementazioni suddividono il lavoro in quattro operazioni principali:

- Modellazione: Durante questa fase vengono generati, come insieme di vertici, gli oggetti da rappresentare; ad esempio linee, poligoni, punti.
- Elaborazione geometrica: In questa fase si attuano principalmente tre elaborazioni:
    - Normalizzazione (o Viewing): ovvero l'adattamento delle coordinate degli oggetti a quelli della camera virtuale.
    - Clipping: vengono rimosse tutte le parti degli oggetti non visibili, perché fuori dalla vista.
    - Ombreggiatura (Lighting and shading (Illumination)): in questa fase vengono calcolati i colori e i riflessi degli oggetti tenendo conto delle proprietà dei singoli poligoni e delle luci incidenti e riflesse.
- Proiezione: L'immagine 3d è proiettata sulla superficie 2d.
- Rasterizzazione o Scan Conversion: La scena è convertita da un insieme di vertici ad un insieme di pixels (bitmap o immagine raster).

Wikipedia:
La pipeline grafica può essere gestita direttamente dal programma tramite accesso diretto all'hardware o può essere gestita tramite librerie grafiche che forniscono delle primitive di manipolazione che vengono utilizzate dal programma. Nella maggior parte dei casi vengono utilizzate le librerie grafiche sebbene queste introducano una leggera penalizzazione delle prestazioni permettano al programma di sfruttare le schede grafiche in commercio senza dover scrivere una versione apposita del programma per ogni tipologia di scheda grafica. Le più diffuse librerie grafiche tridimensionali sono OpenGL e DirectX.

Wikipedia:
Un ipotetico processo di funzionamento potrebbe essere il seguente:

1. La CPU invia le istruzioni e le coordinate 3D della scena alla GPU
2. Nel vertex shader programmabile viene trasformata la geometria e vengono applicati alcuni effetti di illuminazione
3. Il geometry shader, se presente, trasforma ulteriormente la geometria della scena
4. La geometria viene riprodotta in triangoli, che vengono ulteriormente trasformati in *quad* (ogni quad è una primitiva di 2x2 pixel)
5. Vengono applicati ulteriori effetti tramite il pixel shader
6. Viene effettuato il test di visibilità (z-test): se un pixel è visibile, viene scritto nel framebuffer per l'output su schermo.

Il tutto comincia con delle primitive grafiche in ingresso, il primo stadio lo abbiamo già visto: **trasformazione delle coordinate**. Devo trasformare (0,0,100,100) in quale spazio? Dipende dalla matrice di trasformazione associata al contesto grafico (per default è la matrice identica), nel primo passo si normalizzano le coordinate in un unico spazio, in sostanza si risponde alla domanda: “quali pixel vanno colorati?”

Il secondo stadio è quello di **clipping**, le primitive grafiche vengono ritagliate -> in ingresso si ha

un'area di clipping e si fanno andare avanti solo le primitive visibili totalmente o parzialmente(in questo caso la primitiva va cambiata e va ritagliata solo la parte visibile)

Dopo il clipping c'è lo stadio di rastering: qui si trasformano primitive vettoriali in sequenze di punti.

n.b. Non è detto che la pipeline venga eseguita dal processore -> oggi si usano GPU persino sul mobile.

n.b. Funzioni parametriche: utilizzando lo strumento delle funzioni esprimono per esempio traiettorie in spazio piano che non sono funzioni! (ad una x non possono corrispondere due y detto in termini orribili) sono funzioni che invece di y = f(x) si esprimono come

x(t) = g(t)

y(t) = h(t)

Se volessi decrivere un moto circolare uniforme di raggio unitario: (Faccio variare alfa che è l'angolo tra raggio e asse x)

x(t) = sen t cos t

y(t) = sen t dove t = alfa ed ecco fatta una traiettoria circolare con una funzione parametrica.

Le funzioni parametriche serviranno sia per il clipping che per fare rastering:

Il problema del clipping è questo: data una regione(rettangolare per semplicità ma può essere arbitraria) voglio sapere quale primitive grafiche giacciono all'interno, quali no e quali solo parzialmente. Vedremo due algoritmi, il primo è il Cohen-Sutherland: una qualsiasi curva può essere approssimata con tanti piccoli segmenti di densità variabile (si sa dal calcolo integrale e lo sviluppo in serie) quindi ci limiteremo a studiare segmenti e poligoni.

Sono algoritmi progettati negli anni 70 quando i cicli macchina si pagavano caro, nativamente i processori eseguivano solo ADD e SHIFT o comunque operazioni bit a bit come i CONFRONTI, il problema iniziale era eliminare i segmenti che sicuramente non si vedevano, l'idea dietro all'algoritmo è osservare i semipiani ottenuti dal prolungamento dei lati del rettangolo di clipping ottenendo nove aree diverse, stabilire se un punto sta al di la di uno di questi vincoli costa un confronto, quindi con solo 4 confronti (perchè quattro sono i prolungamenti del rettangolo) io so in quale semipiano sta il mio punto. Sutherland prese una maschera di 4 bit, ogni bit vale uno se sta esterno al vincolo o zero se sta all'interno del prolungamento, si numerano a partire da sinistra e dal basso -> ordine dei vincoli sx - dx - giù - sù

Quindi l'algoritmo CS consiste nell'assegnare maschere di 4 bit ad ogni estremo di segmento (tipo p1:0100 e p2:0101) si esegue l'and bit a bit e si scopre (nell'esempio alla lavagna ovviamente non presente qui) che i due punti stanno dalla parte sbagliata (non disegnabile) di almeno un vincolo -> è un test che garantisce solo il caso negativo, ovvero quando sicuramente il segmento sta in tutto e per tutto fuori dall'area di clipping.

Algoritmo di Lian-Barsky: Prendiamo un segmento parzialmente visibile, se volessi determinare l'equazione della retta su cui giace tale segmento potrei definirla usando un'equazione parametrica:

x(t) = x0 + Dx * t , y(t) = y0 + Dy * t -> ovviamente t deve essere compreso tra 0 e 1 (è una sorta di interpolazione anche questa). Quindi costruiamo quattro diseguaglianze (uno per ogni lato del rettangolo di clipping) :

x(t) >= xmin è sottinteso che il parametro t è indipendente, può essere diverso nello stesso momente nelle quattro diseuaglianze

x(t) <= xmax

y(t) >= ymin

y(t) <= ymax

in questo modo si riesce ad individuare il punto di intersezione del segmento ed il lato del rettangolo di clipping

quindi xmax >= x0 + Dt* t >= xmin (stessa cosa per y)

otteniamo xmax - x0 >= t*Dx >= xmin - x0

tutto ciò può essere ridotto a : t*Pk <= Qk e può essere fatto per tutte e 4 le diseguaglianze. Notare come si usino solamente operazioni "facili e veloci" che non coinvolgono divisioni

quindi t >= Pq / Qk e t = Pk / Qk solo nel punto di intersezione

Cosa succede nel clipping di poligoni? (tutte le aree irregolari si possono approssimare con poligoni)

Ogni poligono è dato dai punti che individuano i segmenti del poligono, sposto ogni punto fuori dal poligono sul punto di intersezione dei segmenti che lo compongono, adesso però è aperto -> basta collegare i punti in prossimità delle aperture.

A seconda del livello di accelerazione grafica della scheda video, molti di questi algoritmi di base vengono eseguiti direttamente in GPU.

Procediamo con il **rastering**: Se avessi una bitmap e volessi un algoritmo per accendere i pixel di un segmento da A a B come farei? Potrei scrivere l'equazione della retta y = (yb - ya) / (xb - xa) * x + q poi scrivere l'algoritmo:

foreach x in xa .. xb

    setPixel(x,y(x))

E' corretto ma non è giusto dal punto di vista dell'approssimazione! Se la linea è quasi verticale, rischio di accendere solo un paio di pixel e così facendo la linea non si vede proprio.

Se io sono in xa, ya dove sarà la y nel punto xa+1? Qual'è la differenza della y tra xa e xa+1?

y(xa+1) - y(xa) = p(xa+1 - xa) = p dove p è la pendenza -> per verificare la formula basta sostituire con quella sopra

quindi con un'addizione posso trovare i punti successivi da accendere (si nota che se p > 1 lasciamo dei buchi sulla linea -> quindi si usa il reciproco di p che sicuramente è minore di uno se p > 1, se p < 1 si usa p)

Quella vista sopra è una tecnica chiamata **differential data analisys**: DDA. Un problema noto di questo approccio è che p è un float, quindi coinvolge operazioni in virgola mobile che costano assai e soprattutto sono lente, inoltre se p è ad esempio 0.1, nei passaggi successivi la y rimane molto piccola 0.2 0.3 0.4 0.5 ... siccome è una variabile discreta, la y in sostanza rimane sempre la solita per molti passaggi, quindi va approssimata (arrotondare o troncare) -> se tronco, verticalmente produrrò disallineamenti -> se arrotondo invece produco un effetto scalino. Esiste un algoritmo migliore:

Bresenham (dal nome del creatore) -> gli algoritmi che vediamo, fanno capire quali sono i meccanismi di ragionamento che stanno dietro il mondo 2D

Prendiamo la griglia dei pixel dello schermo e supponiamo di voler disegnare una linea che attraversa lo schermo in diagonale da basso a sinistra in alto a destra, vorremmo andare a vedere dove la linea interseca i pixel, alcuni saranno attraversati in pieno, altri invece non saranno così ovvi perchè la linea potrebbe passare precisamente nel punto di intersezione tra due o più pixels. Posso usare come discriminante della scelta di accendere i pixel indecisi, non guardando il punto x+1,y ma guardando il punto dove sta la linea rispetto al punto x+1/2 -> questo algoritmo è molto apprezzato perchè si usa solamente aritmetica intera:

la retta si può esprimere con: F(x,y) = ax + by + c per ricavare i coefficienti basta osservare y = Dy / Dx * x + q quindi a = Dy, b = Dx, c = q*Dx

Cosa posso dire se F(x,y) < 0? posso dire che quel punto sta sul semipiano superiore della retta, a questo punto posso chiedermi: che succede a F(x+1/2,y)? se è = 0 sono sulla retta quindi accendo sicuramente il pixel giusto, se < 0 il punto sta sopra quindi accendo xa+1,ya+1 altrimenti sta sotto ed accendo x+1,y

nel caso in cui y(x+1)=y(x) vado a calcolare F(x+3/2,y) - F(x+1/2,y) = a

nel caso in cui y(x+1)=y(x)+1 vado a calcolare F(x+3/2,y +1) - F(x+1/2,y) = a + b

Come si fa per riempire le superfici? Per ora abbiamo visto solo lo stroking(disegnare il contorno di una primitiva)

Un poligono si esprime come una lista di vertici, deve essere un poligono chiuso, altrimenti sarebbe una linea spezzata e non sarebbe riempibile. Fare il filling è un operazione costosa perchè è quadratica, si usa l'algoritmo di ScanLine (vedere libro Cignoni).

La **GPU** è una scheda che ha un po' di funzioni ed un po' di RAM, è collegata con la CPU tramite un bus PCI che (per quelli di fascia alta) ha una portata di circa 15GBps (giga BIT) - i cuda core di una GPU sono le ALU in buona sostanza, per questo su schede di fascia alta si trovano circa 2000 cudacore - perciò contando che ogni core ha circa 10 alu abbiamo circa 150-200 core reali - simd (single instruction multiple data) -> per questo servono tanti core, devono eseguire calcolo parallelo.

Rasterizzare curve di Beziér (esempio fatto a lezione da integrare con appunti Ciste)**:**

Come ultimo esempio vediamo come si esegue il rastering delle curve di Beziér (segmentazione)

Le curve di Beziér esistono in molte varianti a seconda dei punti di controllo che utilizzano, le più conosciute sono le quadratiche (3 punti di controllo) - non permettono la rappresentazione di flessi -

e le cubiche - permettono la rappresentazione di flessi - (4 punti di controllo) -> vedere Wikipedia. Naturalmente in grafica si usano le cubiche perchè sono più interpretabili dall'utente e sono sufficientemente espressive per essere composte assieme ad altre primitive grafiche per formare path più complessi.

Tecnicamente le curve di Beziér sono una famiglia di curve che definiscono come vincolare una curva polinomiale dato un insieme di punti di controllo.

E' possibile ricavare una curva di Beziér da una canonical spline calcolando i punti di intersezione tra i segmenti tangenti alla curva nei punti di controllo di quest'ultima, i punti ricavati dalle intersezioni saranno proprio i punti di controllo della curva di Beziér che volevamo ottenere.

Inannzi tutto consideriamo una curva di Beziér come una curva parametrica -> vedere lezioni passate

x(t) = axt^3 + bxt^2 + cxt + dx

y(t) = ayt^3 + byt^2 + cyt + dy -> dove le x e le y sono in pedice per differenziare i parametri

Dunque la curva è rappresentata da due equazioni con 8 parametri (ax, ..., dx, ay, ..., dy) Perciò per fornire un'equazione parametrica con cui segmentare la nostra curva di Beziér cubica dovremo determinare ognuno degli otto parametri:

Identifichiamo p0,p1,p2,p3 come i nostri 4 punti di controllo della curva, come quando studiavamo le interpolazioni associamo il punto p0 con il valore 0 ed il punto p3 con il valore 1

Dunque x(0) = 0 + 0 + 0 + dx = dx

y(0) = 0 + 0 + 0 + dy = dy perciò x(0) = dx , y(0) = dy

Dal vincolo p3 ricaviamo altri due vincoli:

x(1) = ax + bx + cx + dx

y(1) = ay + by + cy + dy

Per calcolare l'equazione del segmento p0p1 usiamo la formula di derivazione:

xv01(t') = x0 + (x1 - x0)t' = x0 + (x1 - x0)(1/3t)

yv01(t') = y0 + (y1 - y0)t' = y0 + (y1 - y0)(1/3t)

(questo perchè si può approssimare t' = 1/3t per motivi matematici...)

dunque xv01(t) = (x1-x0)/3

yv01(t) = (y1-y0)/3

deriviamo l'eq parametrica in t ottenendo:

x'(t) = 3axt^2 + 2bxt + cxt

y'(t) = 3ayt^2 + 2byt + cyt -> dove le x e le y sono in pedice per differenziare i parametri

dunque:

x'(0) = (x1-x0)/3 = cx

y'(0) = (y1-y0)/3 = cy (INTEGRARE CON APPUNTI CISTE APPENA DISPONIBILI!!!)

Stessa cosa con il segmento p2p3 ottenendo:

x'(t) = (x3-x2)/3 = 3ax + 2bx + cx

y'(t) = (y3-y2)/3 = 3ay + 2by + cy

Dunque infine abbiamo 8 equazione per risolvere un sistema ad 8 variabili... (INTEGRARE!!!!)

Lezione 10 – Digital Signal Processing [IUM]

La DSP è una tecnica di analisi ed elaborazione digitale dei segnali elettrici che si basa sull'utilizzo di processori dedicati con un elevato grado di specializzazione: i processori di segnali digitali. Essi sono stati un'innovazione importante, un primo esempio è il radar, basti pensare che prima del suo avvento le torri di controllo verificavano la presenza di aerei nella loro zona tramite delle cornette amplificanti che gli operatori usavano come cuffie. (Vedere il libro “The scientist and engineer's guide to digital signal processing”)

Differenza tra accuratezza e precisione in parole povere: si esegue una misura svariate volte, se le mettiamo assieme in un grafico i valori misurati si distribuiranno attorno ad una distribuzione normale con la media centrale e la campana che ha a che fare con la varianza, tanto più **precise** sono le misure tanto più la campana è stretta. L'**accuratezza** di una misura è la distanza tra il valore che misuriamo ed il valore reale che si può solo stimare.

Campionare: ho un segnale continuo ed un misuratore che mi permette di vedere il valore per un attimo e leggerlo con una certa frequenza, tale frequenza è la **frequenza di campionamento**.

Il processo di campionamento è il seguente: si parte dal segnale originale e si "discretizza" in due direzioni (x,y) la direzione y ha a che fare con la grandezza di scala con cui rappresentiamo i segnali -> campionatori a 10 bit producono valori da 0 a 1023. la direzione x ha a che fare direttamente con la frequenza.

Le seguenti nozioni risalgono alla 2guerra mondiale, Shannon e Turing (che fece la sua tesi con Church, quello della tesi church-turing)

Teorema di campionamento di Shannon: ci da la frequenza giusta di campionamento per poter convertire un segnale analogico in digitale e viceversa senza che il segnale venga disturbato eccessivamente: la frequenza di campionamento deve essere almeno doppia rispetto a quella del senale che vogliamo campionare.

Convoluzione: operazione matematica definita su funzioni che permette di prendere un segnale in input e moltiplicarlo per un altro segnale ed ottenere un terzo segnale (il quale non rappresenta una funzione necessariamente invertibile)

Dato un segnale (oscillatorio e periodico) è interessante scomporre questo segnale in sotto-segnali (grazie alla trasformata di Fourier) in modo da filtrare quelle frequenze disturbanti. La **trasformata di Fourier** prende un segnale nel dominio nel tempo e lo trasforma in una serie di segnali nel dominio delle frequenze.

In parole povere la TDF consente di scomporre un'onda qualsiasi, anche molto complessa e

rumorosa (segnale telefonico, trasmissione tv, musica, voce) in più sotto-componenti. Più precisamente la TDF permette di calcolare le diverse componenti – **ampiezza** , **fase**, **frequenza**-delle onde sinusoidali che, sommate tra loro, danno origine al segnale di partenza (una volta effettuata la trasformata si leggono le frequenze sull'asse x e le ampiezze sull'asse y).

n.b. Il filtro adsl effettua semplicemente una TDF: elimina le frequenze > 30Khz, il segnale adsl viaggia mescolato alla voce ed il filtro divide la voce dalla frequenza adsl > 30Khz

Vediamo come sono trattate le **immagini**: sono rappresentate come matrici di punti, ciascun punto ha un colore associato. Una immagine può essere memorizzata in vari formati che comunque sono oggetti lineari (mettere le righe/colonne delle matrici una di fianco all'altra), quanti bit usiamo per memorizzare un colore? 32 bit di solito (argb) ma adesso anche (sony) a 64 bit. In sostanza un formato differisce dall'altro per la quantità di informazione sul colore, per la scelta di memorizzazione della matrice (righe o colonne) e per vari altri fattori. Per ogni punto dunque si hanno varie possibilità: rappresentare fisicamente il colore (con un codice convenzionale) o rappresentare un indice di una tabella colori che contiene tutti i colori (formato di immagine indicizzato come il gif), usare gli indici è in effetti una forma di compressione perché un indice occupa solo un byte.

Come già detto le compressioni sono “lossless” (gif, png, ...) o “lossy” (jpg, …). Le tecniche usate per comprimere sono RLE (i pixel adiacenti del solito colore vengono compressi in blocchi), LZW (sistemi di compressione basati su dizionari che vengono aggiornati ad ogni compressione) e DCT (quella che usa jpeg: fa una trasformata: spezza il segnale dell'immagine e lo scompone in sotto-segnali).

Lezione 11 – Il Web e la Grafica Retention

Fin'ora abbiamo lavorato unicamente con grafica Raster e framebuffer: scegliamo una primitiva grafica che tramite la pipeline viene trasformata nei pixels opportuni, al termine del processo di drawing il sistema grafico non sa più niente della primitiva, per lui è come se non ci fosse più, lui si limita a tenere la **bitmap** con il disegno nel framebuffer scaricando sull'utente tutte le policy per trattenere in memoria eventuali frames (in qualche modo c'è una perdita di informazione). I vantaggi di questo approccio sono: costo in memoria costante per il sistema grafico (è in effetti una sorta di compressione), il modello nella applicazone è problem tailored (ritagliato per la particolare app che stiamo facendo), è un metodo compatto a scapito di cicli macchina in più (ad ogni raster vanno riprodotte le immagini, una sorta di decompressione).

Oggi nella maggior parte dei casi risparmiare cicli macchina non è più un problema di primo livello (basta pensare al Raspberry p0: un computer completo contenuto nella scocca di una USB Key).

E' il momento di analizzare la **grafica a retention**: un vecchio design che sta nuovamente prendendo piede soprattutto in 3D. Possiamo riassumere il tutto come un processo in memoria il cui output è un albero che viene preso in pasto dal sistema grafico che stavolta conosce tutte le primitive grafiche. Il sistema grafico si prende in carico un aspetto che prima era nelle mani del programmatore. **Un esempio di grafica a Retention è il web browser!** Non riesce a mandare i giochi però, infatti è stato introdotto il canvas in html5 per reintrodurre la grafica immediata. L'albero è di proprietà del sistema grafico, io programmatore non dirò più “drawline”, ma “aggiungi

una linea all'albero di visualizzazione", io do le informazioni al sistema grafico, dico cosa disegnare ed al resto ci pensa lui, compresa la gestione di onpaint, resize, eventi etc etc etc. Se voglio far sparire cose dalla grafica devo esplicitamente toglierla dall'albero, non è più come prima che il sistema grafico la dimenticava se non la disegnavo ogni volta. Un altro esempio interessante è il WPF (Windows Presentation Foundation) creato da Microsoft intorno al 2005, Windows Forms era nato alla fine degli anni 90. Loro volevano proiettare in html le primitive grafiche che si usano nel design di applicazioni grafiche, non ce la fecero e divisero il progetto in due parti WebForms e WindowsForms, ma non si diedero per vinti e continuarono a lavorare sul progetto di un modello di grafica a retention valido per tutti (browsers, apps, ...) e così naque il WPF: esiste una unica finestra, si butta via tutto il passato e si reingegnerizza tutto il processo grafico da capo, le Universal App usano grafica fatta in WPF (una sua variante) dunque puramente a retention.

Parliamo di **HTML5**: il renderer (che è il componente che disegna html: il browser vero è il renderer! Quello che implementa la visualizzazione vera e propria che è integrabile nel browser, html5 è proprio un renderer, quindi un controllo grafico il cui modello può essere manipolato via scripting) ha una struttura dati, ovvero il suo modello (il DOM: document object module), e consente la modifica di quest'ultimo via scripting (javascript, CSS ed ovviamente dal loader della rete). Quando si apre una pagina web il DOM è vuoto, man mano che si scaricano informazioni il modello viene cambiato e la vista aggiornata ogni volta. La struttura del browser non è altro che un controllo grafico che legge un modello e produce una vista.

Dunque è bene conoscere un po' di **JavaScript**: la struttura di JS è decisamente innovativa, ad oggi è cambiato pochissimo dai suoi albori, gli è venuto bene (quasi) al primo colpo. JS è un linguaggio funzionale [le funzioni sono valori, sono cittadine di primo livello etc etc etc], fortemente tipato, dinamicamente tipato, senza costruttori di tipo, di scripting (prevede che una parte delle funzioni primitive di js possa venire dall'esterno, può essere integrato all'interno di un processo che ospita alcune librerie), asincrono da poco (fino a poco fa era singletrhead), con garbage collector e scoping dinamico (nota a margine: il C non è un linguaggio fortemente tipato ma è staticamente tipato). Cosa vuol dire "senza costruttori di tipo"? Java ha costruttori di tipo(class) anche il C(struct) il C++ (class e struct) mentre JS ha un numero di tipi finito, non si può estendere (e Python l'ha copiato) anche se l'utente è ingannato e crede che si possa farlo. I tipi predefiniti sono: numeri, stringhe, tipi nativi (esposti come oggetti ma non lo sono, come date ed altra roba) e dizionari(strutture che associano a nomi dei valori, dove il nome deve essere di tipo "string"). Di recente hanno aggiunto anche il tipo Array (che comunque sono sparsi: se io scrivo a = [] posso dire che a[0] = 1 e a[100] = 3 senza definire niente in mezzo, dunque non sono aree contigue in memoria come accade in C, dove se provo a prendere a[3] ho un comportamento non definito). **L'ultimo e fondamentale tipo sono le FUNZIONI**. Essendo un linguaggio funzionale, JS supporta le **chiusure lessicali** (sono una struttura fondamentale: scoping statico e linguaggi funzionali in pratica lo producono come effetto):

```
function counter() {
 var c = 0;
 return function () {c++;}
}
```

Attenzione! Se io faccio var f = Counter(); alert(f()); Questo restituisce 0! Perché il ++ è un

operatore di post incremento, prima restituisce la variabile c, poi la incrementa! Se io adesso chiamo alert(Counter()()) restituisce ancora 0 perché è una chiamata differente, quindi un ambiente differente, quindi se io chiamo adesso alert(f()) stamperà 1 ... e così via.

Questo è valido perché la funzione che restituiamo si porta dietro l'ambiente di dichiarazione! Ecco la famosa **closure** (sono nate con lisp). La closure esegue un lavoro molto complicato, deve far si che l'ambiente venga salvato (Non è a costo zero! vanno di moda ora perché ci sono tanti cicli macchina da sprecare).

Un **dizionario** lo si può introdurre così var a = {}; Poi si può popolare come a['Nome'] = 'Antonio'; (in JS sia virgolette che apici denotano le stringhe). Ora la cosa più carina è che posso scrivere anche a.Eta = 2; che è assolutamente analogo alla riga sopra, il mio dizionario adesso è:

Nome: Antonio

Eta : 2

E' la stessa cosa che scrivere a['Eta'] = 2; L'autore di JS ha ingannato il popolo di tonti che pensa che se scrivi a.Piffero allora a deve per forza essere un oggetto, invece stai assegnando un'HashTable guada un po'!!!

Tutti gli pseudo oggetti che in realtà sono dizionari hanno una chiave che è **prototype**

a.prototype = {'Test':0}

a.Test vale proprio 0! Tutto questo vuole approssimare una sorta di proprietà “is a” per emulare il "**principio di sostituzione**".

quindi a.prototype = un'altro dizionario è come se dicessi che a deriva dall'altro dizionario.

nota: se io faccio a.Test=2, nell'implementazione di a il valore test è 2 ma nel prototipo rimane 0!

Proviamo a scrivere un esempio:

```
function S = { 'sp':0;
  'push':function(v){this[this.sp++]=v;}; // uso sp attuale poi lo incremento
  'pop':function(){if(!this.sp) throw "stack vuoto"; var v = this[--this.sp]; /*decremento sp attuale e lo uso*/ delete this[this.sp];
        return v;}
}
```

Abbiamo appena scritto uno stack. Non posso usare direttamente s perchè è unico, ma posso usarlo come matrice per creare altri stack con il metodo dei prototypes:

dico function B = {} //diz vuoto

poi B.prototype = S ed ecco: anche B è uno stack

infine b = new B;

n.b. This è una keyword speciale che assume il valore dell'ultimo nome chiamato a runtime: se io eseguo b.push(2) nel prototipo è come se fosse scritto b[b.sp++]=2 il quale altro non fa che dire b[0]

= 2 dunque prima il dizionario era b={'sp':1}, ora è b={'sp':1, 0:2} se facessi per esempio b.push(3) avrei b={'sp':2, 0:2, 1:3}.

// scriviamo una funzione che cloni uno stack

```
function Stack(o) {
  o.sp = s.sp;
  o.push = s.push;
  o.pop = s.pop;
  //tutto ciò sarebbe stato equivalente a for(k in s) o[k] = s[k];
}
var o = {} // stack vuoto
Stack(o); // adesso o è uno stack
o.push(2);
```

avrei potuto fare in alternativa:

```
function Stack() {
  this.sp = 0;
  this.push = function {...}
  this.pop = function {...}
};
var o = new Stack();
```

Il **prototype si aggancia alle funzioni** non ai tipi

Stack.prototype = {'a':2} // non è retroattivo: il proto va dichiarato prima di fare new o = new Stack();

o.a = 2;

che succede se faccio o.sp = 'a' ?

Corrompo lo stato del pseudoggetto -> JS non permette di fare information hiding!! Questo è una delle principali deficienze di JS, adesso sp vale in effetti 'a', non viene sollevato nessun errore -> ergo stato corrotto!

L'unico modo di avere uno "stato privato" è usare la parola chiave var durante la locazione:

```
function Stack() {
  var sp = 0;
  this.push = function {qui mi riferisco a sp non this.sp ...}
  this.pop = function {idem ...}
```

};

usciti dallo scope sp non si vede più ma push e pop faranno sempre riferimento allo stesso sp per via del legame -> scoping

es:

(function () {

  a = 2;

  ...

})() la var a che sarà messa dentro funzioni pubbliche non si potrà vedere fuori dallo scope

Un altro linguaggio interessante secondo Cisternino è **TypeScript** (E' MS anche se è open source) il bootstrap è stato scritto in JS (le successive compilazioni possono essere fatti con il solito linguaggio, è una pratica comune), è una sorta di estensione di JS che aggiunge le classi e il tipaggio statico. E' un preprocessor il cui output è proprio JS.

La distinzione tra linguaggi di alto e basso livello ormai è diventata inutile, anche i linguaggi di scripting possono assumere bassi livelli al contrario di ciò che pensano tutti.

Alcuni esempi: LLVM è low level virtual machine, progetto molto supportato da Apple, in sostanza è un compilatore che è capace di trasformare codice in codice intermedio. Il gcc usa una rappresentazione interna che si chiama RTL quindi llvm non è una cosa nuova. Emscrpiten è un progetto su cui ha lavorato Mozzilla Foundation, è un compilatore cpp, js, webgl, webaudio, ... prende in ingresso cpp + openjl e lo compila in JS! Non solo, in realtà hanno specificato anche asmjs (edge lo implementa) in sostanza è una restrizione del linguaggio JS con cui si può fare type inference e convertire JS direttamente in linguaggio macchina ottenendo miglioramenti estremi. Con il contributo di unreal hanno compilato unreal engine 3 in JS così da giocare a giochi superfighi scritti in cpp direttamente nel browser con html5 ! Unity3d è un game engine che nasce su apple ma da subito usa .net come macchina virtuale, all'inizio si basava su un plugin. Mozzilla ha dichiarato guerra ai plugin e quello che succederà entro breve è che il plugin di unity non andrà più. Quindi si sono messi a lavorare, prima scrivevano in c# poi compilavano in msil (bytecode virtual machine in .net), dopo hanno scritto il 2cpp che genera cpp, poi lo passano in emscripten che lo compila in JS. Adesso è in fondo alla gerarchia, quindi JS è di basso livello! Una cosa ottima di cpp: offre supporto ad oggetti senza overhead significativo a runtime.

Chiudiamo la parentesi dei linguaggi e continuiamo a parlare di browser: abbiamo già detto che il core del browser è il renderer(leggere codice html e fare la paint in base a quello). Il browser non è altro che uno user control il cui modello è programmabile con JS, in origine non era così (visual basic, perl, ...) ma poi è stato deciso praticamente all'unanimità di lasciare solo JS. Implementa un modello di grafica a retention, infatti ha all'interno un DOM che può essere modificata da noi interagendo con il controllo (eventi) oppure da script che leggono il DOM e lo cambiano, poi il browser si accorge che il dom è cambiato ed aggiorna la vista (ricordarsi MVC).

Cos'è html? E' un linguaggio di marcatura e non di programmazione (non è turing equivalente, non esprime operazioni, si limita solo a marcare).

Esempio:

<!DOCTYPE html>

<html lang="it"> // lang è un attributo -> ogni elemento può avere più attributi

  <body>Ciao</body> // body non è un tag, ma un elemento

</html>

L'html vede gli elementi come un "apro - contenuto(lista di figli) - chiudo", ogni elemento è un unicum (all'inizio non era cosi, c'erano pseudoelementi come br o hr), se voglio scrivere à non scrivo à perchè i set di caratteri non sono uguali in tutto il mondo, scrivo à (ne esistono molti altri per tutti i simboli), queste sono entità! html è quindi elementi con attributi, testo ed entità

L'XML (extendible markup language) E' un metalinguaggio che permette di definire nuovi linguaggi di markup. Lo standard XML definisce una serie di elementi, una serie di entità, una serie di direttive (<! è una direttiva per il processor del metalinguaggio) e basta. Definisce una sintassi standard, prevede che esista una sola radice, ogni xml può essere rappresentato come albero (vedere anche dtd, xsd, xmlns).

Json è una versione ristretta della sintassi simil object oriented di JS (infatti json: js objet notation), serve per lo scambio di messaggi nel web. Nei mondi più complicati si usa sempre xml (Come insegna il grande Tito Flagella.

CSS (cascading style sheet) è un rule language (un linguaggio a regole) che serve a definire pattern sul DOM. Il significato di un css è : tutti gli elementi del DOM che soddisfano il pattern che ti do, voglio che tu gli assegni le proprietà che descrivo (in sostanza esegue un “foreach”). Se sapessi che h1 corrisponde a head1 in html potrei dire h1 {color : red;} tutti gli elementi nel DOM che si chiamano h1 impostano la proprietà color a rosso. Si può usare l'attributo id per diversificare anche elementi dal solito nome.

Lezione 12 – Il mondo del 3D

Il problema del 3D è costruire un controllo grafico che rappresenti un mondo 3D (modello) e che proietti il modello sulla vista della camera (vista), la questione principale del 3D è: **come si fa a rappresentare un mondo tridimensionale?**

L'occhio guarda uno schermo che contiene la proiezione 2D del mondo 3D, il Kinect (sensore a luce strutturata a basso costo) proietta un pattern noto ad infrarossi sulla stanza e determina come il pattern si deformi dalle due camere (una infrarosso ed una a colore), usando una tecnica detta **post-processing.**

Anche il cervello fa post-processing, la retina ha un buco che coincide con l'innesto del nervo ottico, il cervello interpola i "pixels" mancanti, inoltre sulla retina la visione è capovolta e curva.

Noi vediamo la proiezione della luce che rimbalza sugli oggetti, non gli oggetti in se. Il modello della grafica 3D si appoggia esattamente su queste nozioni.

Il modello a cui ci si ispira per realizzare il 3D è il **modello "pinhole camera"** : un buco infinitesimale che idealmente faccia passare un solo raggio di fotoni alla volta (replicabile con una scatola ed una fiammella)

Dobbiamo modellare la luce che segue alcune proprietà:

-colore: legato alla capacità dei materiali di riflettere / assorbire colori

-proprietà geometriche

-fenomeni ottici: riflessione, rifrazione, (la luce cambia riflessione quando cambia il materiale con cui è a contatto, per questo se infiliamo un bastone nell'acqua per metà lo vediamo spezzato, un fenomeno della stessa natura è l'arcobaleno), diffusione (tecnicamente è un effetto collettivo), interferenza (la luce è un'onda e come tale ha una certa frequenza, ma non solo oscilla, ha un movimento tridimensionale, i fotoni ruotano)

Tutto questo viene implementato con la tecnica di **raytracing**.

Come si rappresentano le superfici? Si approssimano tramite poligoni (in genere mesh di triangoli), si noti che una mesh di triangoli è di più rispetto ad una nuvola di punti (cloud of points). Perchè si usano i triangoli? Perchè datri tre punti almeno un piano passa in due punti, è l'unica figura che garantisce alla vista di essere sempre consistente. Tutte le superfici 3D sono approssimate con mesh di trinagoli (mesh = zuppa, misto, insieme).

Oggi la grafica 3D è fatta in **real-time** (videogiochi,...) o **non real-time** (films,... -> nel 2008 quando uscì king kong facvano un frame in 24 ore di calcolo, le mesh erano talmente grosse che invece di essere salvate sui dischi (lenti) venivano ricalcolate perché così risparmiavano tempo. Cameron ha sempre fatto film con una trama di poco spessore ma ad alto contenuto tecnologico, fissando sempre dei punti di riferimento.

Quello che va regolamentato nella luce è la parte geometrica, come si riflettono i raggi. La seconda cosa importante è il colore.

Mi posiziono su una superficie piccola a piacere, voglio calcolare il contributo luce che quella superficie da alla camera. Come posso riuscirci? Grazie all' **equazione di rendering.**

Wikipedia:
In computer grafica, l'**equazione di rendering** descrive il flusso dell'energia luminosa attraverso una scena. È basata sulla fisica della luce e fornisce risultati teoricamente perfetti, in contrasto con le varie tecniche di rendering, le quali approssimano questo ideale.

Matematicamente, l'equazione viene espressa nel modo seguente:

$$L_o(x,\vec{w}) = L_e(x,\vec{w}) + \int_\Omega f_r(x,\vec{w}',\vec{w}) L_i(x,\vec{w}')(\vec{w}' \cdot \vec{n}) d\vec{w}'$$

Dove:

$L_o(x,\vec{w})$ è la luce uscente in una particolare posizione x e direzione $\vec{w}$.
$L_e(x,\vec{w})$ è la luce emessa nella stessa posizione e direzione.

$\int_\Omega ...d\vec{w}'$ è una somma infinitesimale calcolata su un emisfero di direzioni entranti.
$f_r(x, \vec{w}', \vec{w})$ è la percentuale di luce riflessa in quella posizione (dalla direzione entrante a quella uscente).
$L_i(x, \vec{w}')$ è la luce entrante da posizione e direzione $\vec{w}'$.
$(\vec{w}' \cdot \vec{n})$ è l'attenuazione della luce entrante dovuta all'angolo d'incidenza.

In poche parole: **In una particolare posizione e direzione, la luce uscente (L_o) corrisponde alla somma della luce emessa (L_e) e di quella riflessa. La luce riflessa, a sua volta, è la somma della luce entrante (L_i) da tutte le direzioni, moltiplicata per la superficie riflettente e per l'angolo incidente.**

Wikipedia:
Due interessanti caratteristiche sono: la sua linearità (è composta solo di moltiplicazioni e addizioni), e la sua omogeneità spaziale (è la stessa in tutte le posizioni e direzioni). Questo significa che un gran numero di fattorizzazioni ed arrangiamenti sono (facilmente) deducibili per giungere alla soluzione.

L'equazione di rendering è il concetto chiave accademico/teorico nel campo del rendering. È utile come espressione formale astratta degli aspetti impercettibili del rendering. Unendo la luce uscente a quella entrante, attraverso un punto d'interazione, questa equazione rappresenta l'intero *trasporto di luce* presente nella scena. Tutti i più complessi algoritmi possono essere visti come soluzioni a particolari formulazioni di questa equazione.

Ancora due note sul **Ray Tracing:**

Wikipedia:
Il **Ray tracing** è una tecnica generale di geometria ottica che si basa sul calcolo del percorso fatto dalla luce, seguendone i raggi attraverso l'interazione con le superfici. È usato nella modellazione di sistemi ottici, come lenti per fotocamere, microscopi, telescopi e binocoli. Il termine viene utilizzato anche per un preciso algoritmo di Rendering nel campo della Computer grafica 3D, in cui le visualizzazioni delle scene, modellate matematicamente, vengono prodotte usando una tecnica che segue i raggi partendo dal punto di vista della telecamera piuttosto che dalle sorgenti di luce. Produce risultati simili al ray casting ed allo scanline rendering, ma semplifica alcuni effetti ottici avanzati, ad esempio un'accurata simulazione della riflessione e della rifrazione, restando abbastanza efficiente da permetterne l'uso in caso si voglia ottenere un risultato di alta qualità.

Wikipedia:
Il ray tracing descrive un metodo per la produzione di immagini costruite in sistemi di computer grafica 3D, con maggior realismo di quello che si potrebbe ottenere con l'uso di ray casting o scanline rendering. Lavora tracciando, all'inverso, il percorso che potrebbe aver seguito un raggio di luce prima di colpire un'immaginaria lente. Mentre la scena viene attraversata seguendo il percorso di numerosi raggi, le informazioni sull'aspetto della scena vengono accumulate. La riflessione del raggio, la sua rifrazione o l'assorbimento sono calcolate nel momento in cui colpisce un qualsiasi oggetto.

Le scene, nel ray tracing, vengono descritte matematicamente, solitamente da un programmatore, o da un grafico, utilizzando particolari programmi. Le scene possono anche includere immagini e modelli creati attraverso varie tecnologie, per esempio usando la fotografia digitale. Seguendo i raggi in senso inverso, l'algoritmo viene alleggerito di molti gradi di magnitudine (matematica), il che rende possibile una precisa simulazione di tutte le possibili interazioni presenti nella scena. Questo è dovuto al fatto che la maggior parte dei raggi che parte da una sorgente non fornisce dati significativi all'occhio di un osservatore. Potrebbero invece rimbalzare finché si riducono al nulla, andare verso l'infinito o raggiungere qualche altra camera. Una simulazione che parta seguendo tutti i raggi emessi da tutte le sorgenti di luce non è fisicamente praticabile.

La scorciatoia utilizzata nel raytracing, quindi, presuppone che un dato raggio termini sulla camera, e ne cerca la sorgente. Dopo aver calcolato un numero fisso di interazioni (già deciso in precedenza), l'intensità della luce nel punto di ultima intersezione viene calcolata con un insieme di algoritmi, inclusi il classico algoritmo di rendering ed altre tecniche (come la radiosity).

Ad oggi il mondo 2D si tratta meglio con algoritmi dedicati che con un proiezione del 3D quindi conviene trattare i due argomenti separatamente, come due problemi diversi. Buona parte del "raffinamento 3D" viene eseguito proprio dalla funzione di rendering.

Legge di Fresnel: su uno specchio ideale il raggio incidente è uguale all'angolo di uscita rispetto alla normale della superficie.

Legge di Lambert: (riguarda la diffusione) è un tentativo di dare una spiegazione collettiva che a livello microscopico richiederebbe RayTracing ed un microscopio per comprenderlo, ci fornisce un modello più a grandi linee. Ogni superficie ha una componente riflessiva a cui si applica Fresnel ed una componente diffusiva (poiché superfici perfettamente lisce non esistono, sono tutte scabre anche se in minima parte) .

Modello di illuminazione/approssimazione di Phong : luce = luce ambientale + sommatoria sorgenti luminose, questo è il modo con cui si stabilisce quanta luce esce da ogni punto del piano di proiezione. La luce è un informazione di luminanza, non di colore. Per cui si ha bisogno di un modello simile a quello visto per la luce, che però si occupi dello "shading" (ombreggiatura).

La superficie viene approssimata con mesh di triangoli come abbiamo detto in precedenza.