Relazione laboratori

Corso di Computer Graphics - Università di Bologna

Daniele Polidori
daniele.polidori2@studio.unibo.it

a.a. 2022-23

1 LAB-01

Punto 4b. Nello svolgimento del primo laboratorio, per quanto riguarda il punto 4, ho scelto di svolgere l'opzione *b*.
Oltre ai VAO già in uso per la curva di base, ho creato una seconda coppia di VAO per disegnare il tratto cubico che di volta in volta viene creato. Questo, una volta completato, viene attaccato alla curva di base: i punti relativi al tratto cubico vengono aggiunti a quelli della curva di base, così da liberare lo spazio per un possibile nuovo tratto cubico, e così via.
L'utente può scegliere la continuità con cui attaccare i tratti cubici alla curva di base: la continuità C^0 premendo il tasto *0* (scelta di default), C^1 premendo il tasto *1* e G^1 premendo il tasto *g*. Tale continuità rimarrà selezionata finché non ne verrà scelta una differente.
I tratti successivamente creati verranno raccordati con la continuità selezionata. Siano $p_0, \ldots, p_n$ i punti della curva di base e $v_0, \ldots, v_3$ i punti del tratto cubico, le continuità vengono applicate nella maniera seguente. v_0 viene eliminato, sarà infatti sostituito da p_n. A partire dalle coordinate di p_{n-1} e p_n vengono calcolate le coordinate che deve assumere il punto v_1, così che la continuità venga soddisfatta. Viene applicata la formula (1) per ottenere la continuità C^1 e la formula (2) per la continuità G^1: tali formule vengono applicate separatamente sulle coordinate x e sulle y dei rispettivi punti, con $\Delta = p_3 - p_2$. Per la continuità G^1, il punto verrà posizionato in modo tale che il tratto $v_1 - p_n$ sia lungo la metà del tratto $p_n - p_{n-1}$. Infine, i punti v_1, v_2 e v_3 vengono aggiunti alla curva di base, in qualità di, rispettivamente,

p_{n+1}, p_{n+2} e p_{n+3}.

$$v_1' = p_n + \Delta = 2 \cdot p_n - p_{n-1} \tag{1}$$

$$v_1' = p_n + \frac{\Delta}{2} = p_n + \frac{p_n - p_{n-1}}{2} \tag{2}$$

Punto 5. Per realizzare lo spostamento dei punti tramite trascinamento con il mouse, mi sono servito delle funzioni callback di OpenGL: `glutMouseFunc()` e `glutMotionFunc()`.
Tramite la prima, quando viene premuto il tasto destro del mouse, controllo se mi trovo sopra un punto. Per facilitare la presa, considero un intorno delle coordinate dei punti: dato un punto (x, y), anziché le coordinate strette x e y, cerco dei valori più laschi, rispettivamente negli intervalli $[x - 0.01, x + 0.01]$ e $[y - 0.01, y + 0.01]$. Se il mouse si trova sopra un punto, questo viene "agganciato" (i.e. un puntatore punta alla sua cella di memoria) e quando poi rilascio il tasto, il punto viene "sganciato".
Tramite la seconda, invece, catturo la posizione del mouse durante il trascinamento del punto. Se un punto è stato agganciato, sostituisco le coordinate di tale punto con le coordinate del mouse in ogni istante. I valori continuano a mutare fino a che il punto non viene sganciato, ovvero finché il tasto del mouse non viene rilasciato.

2 LAB-02

Nello svolgimento del secondo laboratorio ho deciso di riprodurre un cielo notturno dove si trovano dieci piccole fonti di luce (detti aloni) e un cerchio di luce più grande (detto sole), che segue il movimento del mouse.
Quando il sole (i.e. il mouse) passa sopra un alone, quest'ultimo scompare (dando l'effetto di venire inglobato dal sole). Se il sole tocca uno dei bordi dello schermo, si incorre in una penalità: tutti gli aloni inglobati tornano alla loro posizione iniziale (in questo modo la partita ricomincia da capo).
Lo scopo del gioco è inglobare (i.e. passare con il sole sopra) tutti gli aloni.
Presentiamo ora il modo in cui sono stati realizzati i vari oggetti presenti nella scena: il cielo, gli aloni, il sole e i sistemi particellari.

Cielo. Ho preso l'oggetto del cielo dal programma `2D_Jumping_Ball.cpp` e l'ho collegato a `VAO_CIELO` e a `VAO_ANIMAZIONECIELO`. Ne ho modificato la posizione e la scala (modificando l'input delle funzioni `translate()` e `scale()`, nella funzione `drawScene()`), in modo tale da riempire tutto lo schermo a disposizione. Ne ho modificato anche il colore (cambiando il valore

delle variabili `col_top` e `col_bottom`, nella funzione `init()`), così da rendere il cielo notturno.
Ho aggiunto poi un'animazione al cielo. Per farlo, ho preso l'oggetto dell'alone del sole dal programma `2D_Jumping_Ball.cpp`. Ne ho modificato la scala (modificando l'input della funzione `scale()`, nella funzione `drawScene()`), in modo tale da ingigantire l'alone nello scermo. Ne ho modificato il colore e la trasparenza (cambiando il valore della variabile `col_bottom_sole`, nella funzione `disegna_luce()`), così da renderlo un chiarore vagamente riconoscibile. Infine, nella funzione `drawScene()`, ho creato contemporaneamente cinque aloni nel cielo, in posizioni casuali. Questi, a intervallo di tempo regolare, cambiano posizione, sempre in maniera casuale. L'effetto ottico che ne deriva è quello di un cielo dinamico, mostrando un vago movimento indistinguibile sullo sfondo.

Aloni. Ho preso l'oggetto dell'alone del sole dal programma `2D_Jumping_Ball.cpp` e l'ho collegato a `VAO_SOLE`. Ne ho modificato la scala (modificando l'input della funzione `scale()`, nella funzione `drawScene()`), in modo tale da renderlo una piccola fonte di luce. Ne ho modificato anche il colore (cambiando il valore della variabile `col_bottom_sole`, nella funzione `disegna_sole()`), rendendolo più chiaro.
All'inizio della partita, nella funzione `drawScene()`, creo contemporaneamente dieci aloni nel cielo, in posizioni casuali. Successivamente vengono mostrati soltanto quelli che ancora non sono stati inglobati. Se il sole tocca un bordo dello schermo, vengono mostrati nuovamente tutti gli aloni.

Sole. Ho preso l'oggetto del sole (con il suo alone) dal programma `2D_Jumping_Ball.cpp` e l'ho collegato a `VAO_SOLE`. Ne ho modificato la scala (modificando l'input della funzione `scale()`, nella funzione `drawScene()`), per renderlo leggermente più piccolo. Ne ho modificato anche il colore (cambiando il valore delle variabili `col_top_sole` e `col_bottom_sole`, nella funzione `disegna_sole()`), rendendolo più chiaro.
Infine, ho agganciato il sole al movimento del mouse: ho sostituito l'input della funzione `translate()` con delle variabili che memorizzano la posizione attuale del mouse (il loro valore viene continuamente aggiornato attraverso la funzione callback di OpenGL `glutPassiveMotionFunc()`).
Quando un alone viene inglobato, l'alone del sole viene ingrandito, per mostrare il progresso nella dinamica di gioco. Se il sole tocca un bordo dello schermo, l'alone del sole torna alla sua dimensione iniziale.
In caso di vittoria, l'alone del sole viene ingrandito progressivamente (senza mai fermarsi). Per realizzare questo effetto ho usato cinquanta chiamate

ricorsive alla funzione `aumentaScala_aloneSole()`, attraverso la funzione `glutTimerFunc()`, ciascuna delle quali applica un piccolo incremento alla scala dell'alone del sole.

Sistemi Particellari. Ho preso l'oggetto del sistema particellare dal programma `2D_PS.cpp` e l'ho collegato a `VAO_SISTEMAPARTICELLARE`. Tale oggetto è stato utilizzato in più punti.
Come prima cosa, l'ho collegato al movimento del mouse (attraverso la funzione `glutPassiveMotionFunc()`) e ne ho cambiato il colore (impostando il giallo nel valore della variabile `rgb`, presente nella funzione).
Inoltre, nella funzione `drawScene()`, ho creato, in ciascuna posizione degli aloni, un piccolo sistema particellare di colore arancione (indicato nel valore della variabile `rgb`, presente nella funzione). Tale sistema particellare viene mostrato soltanto nelle posizioni degli aloni che non sono stati ancora inglobati.
Infine, ho utilizzato questo oggetto per segnalare visivamente un'azione negativa, quando essa viene compiuta dall'utente. Infatti, quando il sole tocca un bordo dello schermo, il cielo si riempie di un sistema particellare di colore rosso (indicato nel valore della variabile `rgb`, nella funzione `glutPassiveMotionFunc()`), diffuso uniformemente in tutta l'area dello schermo.

3 LAB-03

Punto 1a. La scelta dello shading da applicare in scena è riferito solo all'oggetto *Bunny* (`Mesh/bunny.obj`).
Nel menu principale del programma, ho creato un sottomenu (nella funzione `buildOpenGLMenu()`) per consentire la modifica dello shading. Quando viene selezionato uno shading differente, viene invocata la funzione `init_mesh()`, che sostituisce il vecchio oggetto (memorizzato nella variabile `objects`) con uno nuovo. Questo, se la scelta è diversa dal Flat shading, conterrà le normali ai vertici (anziché quelle alle facce), mentre per il resto rimarrà identico.
Il calcolo delle normali ai vertici $\overrightarrow{Nv}$, viene eseguito nella funzione `loadObjFile()`.
Cerco, per ciascun vertice i, le n facce che lo condividono, quindi applico la formula per il calcolo della normale al vertice:

$$\overrightarrow{Nv_i} = \frac{\sum_{k=1}^{n} \overrightarrow{Nf_{i,k}}}{n}$$

con $\overrightarrow{Nf_{i,k}}$ la normale alla k-esima faccia condivisa dal vertice i.
Nella variabile `vertexIndices` vengono memorizzati gli indici dei vertici

(decrementati di 1), ordinati seguendo le dichiarazioni delle facce nel file (con estensione `.obj`) del rispettivo oggetto. Dal momento che le normali ai vertici vengono calcolate e memorizzate seguendo i vertici in ordine crescente, l'indice di ciascuna di esse corrisponde a quello del rispettivo vertice (già decrementato di 1). La variabile `normalIndices` viene quindi riempita con i valori presenti in `vertexIndices`.

Punto 1b. Ho definito i valori di un nuovo materiale, che ho chiamato *My Material*. Poi, nella funzione `init()`, l'ho aggiunto nella variabile `materials`. Infine, ho aggiunto la rispettiva voce nel sottomenu dei materiali, per renderlo selezionabile.

Punto 1d. Ho creato il vertex shader `Shaders/v_wave.glsl`, per l'oggetto *Wave*. Lo shader prende in input i vertici e le normali (nelle variabili `vPosition` e `vNormal`). Dall'applicazione gli vengono passati (nelle funzioni `initShader()` e `drawScene()`) i valori delle variabili `time`, `P`, `V`, `M`, `light` e `material`. In output viene restituita la variabile `Color`.
L'illuminazione viene gestita all'interno del vertex shader. Il fragment shader che gli ho associato (nella funzione `initShader()`) è quello relativo al Gouraud shading (`Shaders/f_gouraud.glsl`), che ha l'unica funzione di prendere in input `Color` e restituirlo come `FragColor`.
Per una resa migliore del movimento ondoso, ho aumentato il valore in input alla funzione `refresh_monitor()`, nella funzione `main()`: in questo modo il moto viene rallentato, rendendolo più piacevole alla vista.

Punto 1e. Il Toon shading viene applicato solo all'oggetto *Bunny*.
Ho creato il vertex shader `Shaders/v_toon.glsl` e il fragment shader `Shaders/f_toon.glsl`. Il primo prende in input i vertici e le normali (nelle variabili `vPosition` e `vNormal`); dall'applicazione gli vengono passati (nelle funzioni `initShader()` e `drawScene()`) i valori delle variabili `P`, `V`, `M` e `light`; in output vengono restituite le variabili `N`, `L` ed `E`. Queste vengono prese in input dal fragment shader, che restituisce in output la variabile `FragColor`.
Anche per il Toon shading vengono calcolate le normali ai vertici.
Ho aggiunto la rispettiva voce nel sottomenu degli shading, per renderlo selezionabile.

Punto 2. Nelle funzioni `moveCameraLeft()` e `moveCameraRight()`, per realizzare la funzionalità *Pan oriz. camera (sx/dx)*, ho calcolato il vettore direzione $\vec{A} - \vec{C}$ (nella variabile `direction`) tra la posizione della camera

($\vec{C}$) e del punto di riferimento in scena ($\vec{A}$). Ho poi calcolato, nella variabile `slide_vector`, il prodotto vettoriale tra il suddetto vettore direzione e l'Up Vector della camera. Tale vettore, che definisce lo spostamento da compiere, viene sottratto, in `moveCameraLeft()`, e rispettivamente sommato, in `moveCameraRight()`, ai vettori $\vec{A}$ e $\vec{C}$.

Punto 3. Nella funzione `modifyModelMatrix()` applico le trasformazioni di traslazione, rotazione e scalatura degli oggetti in scena. L'oggetto e il sistema di riferimento, rispetto al quale operare (WCS o OCS), sono quelli che in quel momento sono selezionati. Le trasformazioni vengono applicate alla matrice (la variabile `M`) dell'oggetto (contenuto nella variabile `objects`).

4 LAB-04

Punto 1. Per generare le hard shadow, creo, per mezzo della funzione `Ray()`, nella funzione `TraceRay()`, un raggio che parte dal punto d'intersezione `point` e va in direzione (`dirToLight`) della luce `pointOnLight`. Lancio tale raggio tramite la funzione `CastRay()`.
Successivamente aggiungo il raggio, come shadow ray, al ray tree, attraverso la funzione `AddShadowSegment()`, così da essere visualizzato durante la modalità di debug.
Infine, controllo se l'oggetto colpito dal raggio è una sorgente luminosa: se è così, calcolo e aggiungo il suo contributo luminoso al colore di quel punto, altrimenti quella luce non contribuirà alla sua luminosità. Per eseguire tale controllo, dato il punto luce `pointOnLight` con coordinate (x, y, z), verifico se le coordinate del punto colpito dallo shadow ray sono comprese, rispettivamente, negli intervalli $[x - 0.00001, x + 0.00001]$, $[y - 0.00001, y + 0.00001]$ e $[z - 0.00001, z + 0.00001]$.

Punto 2. Per gestire i reflection ray, se la superficie colpita è riflettente, creo, nella funzione `TraceRay()`, per mezzo di una chiamata ricorsiva alla stessa, il raggio riflesso, che parte dal punto d'intersezione `point`.
Quando invoco ricorsivamente `TraceRay()`, passo il valore della variabile `bounce_count` decrementato di uno. Per verificare se la superficie colpita è riflettente, controllo se la variabile `reflectiveColor`, riferita all'oggetto, ha almeno un valore (tra R, G e B) maggiore di zero. Siano $\vec{v}$ la direzione del raggio incidente e $\vec{n}$ la direzione normale alla superficie nel punto d'intersezione, per calcolare la direzione del raggio riflesso $\overrightarrow{r_v}$, utilizzo la formula:

$$\overrightarrow{r_v} = \vec{v} - 2(\vec{n} \cdot \vec{v})\vec{n}$$

Infine, aggiungo il contributo riflesso al colore del punto d'intersezione, differenziando tra i due tipi di sfera.
Se si tratta della sfera grande bianca, il contributo corrisponde al valore di ritorno della chiamata ricorsiva a `TraceRay()` (i.e. riflette esattamente il colore dell'ambiente esterno).
Se si tratta della sfera piccola rossa, invece, corrisponde al prodotto della media dei valori R, G e B (quindi nell'intervallo $[0, 1]$) di tale contributo per la variabile `reflectiveColor`: in questo modo, a un valore RGB basso, e rispettivamente alto, dell'ambiente esterno corrisponde un valore basso, e rispettivamente alto, del colore rosso della sfera (i.e. `reflectiveColor`).

Punto 3. Per gestire le soft shadow, nella funzione `TraceRay()`, lancio 128 raggi (shadow ray), dal punto d'intersezione `point`, verso punti casuali della risorsa luminosa area light e conto quanti di questi la raggiungono.
Se tutti i raggi raggiungono la luce, allora il punto avrà il pieno contributo di tale luce al suo colore.
Invece, se non tutti i raggi arrivano alla luce, vengono gestiti due casi: quello di ombra e quello di penombra. Se il numero di raggi è inferiore o pari a 15, il contributo della luce in quel punto è nullo: ci troviamo in una situazione di ombra piena. Se, invece, è superiore a 15 (ma almeno un raggio non arriva alla luce), si ha un contributo graduale in base al numero di raggi: più shadow ray arrivano alla luce, più quel punto sarà luminoso. Essendo il numero di raggi in questione, per esprimere la situazione di penombra, corrispondente a 113 ($= 128 - 15$), il contributo previsto per ognuno di questi raggi è pari a 0.009 ($= 1/113$): tale contributo viene applicato per ciascun raggio, dal sedicesimo in poi, che arriva alla luce. Quindi, in questo caso, il contributo della luce in quel punto sarà pari al prodotto del numero di shadow ray che arrivano alla luce, decrementato di 15, per il singolo contributo.
Ricapitolando, sia n il numero di shadow ray che arrivano alla luce, il contributo di questa in quel punto, nei tre casi di ombra, penombra e luce piena, corrisponde a:

$$\begin{cases} 0 & \text{se } n \leq 15 \\ (n - 15) \cdot 0.009 & \text{se } 15 < n < 128 \\ 1 & \text{se } n = 128 \end{cases}$$

Sono giunto a ideare questa strategia seguendo alcuni passi.
Inizialmente, ho lanciato 4 raggi dal punto d'intersezione: uno verso ciascuno dei quattro vertici della faccia della luce. Il risultato però era scadente, dal momento che l'ombra appariva divisa in fasce sovrapposte facilmente distinguibili. Aumentando il numero di raggi, l'ombra veniva resa in maniera più uniforme, ma comunque risultava divisa in fasce.

Invece, lanciando i raggi in punti casuali della luce, l'ombra risultava uniforme, con un effetto più realistico di gradualità verso la luce. Infatti, in questo modo, a punti adiacenti nell'ombra non corrispondeva più necessariamente lo stesso livello d'intensità. Lanciando pochi raggi, però, l'ombra risultava puntiforme e quindi ancora poco realistica. Infine, aumentando il numero di raggi a 128, ho ottenuto un effetto di ombra diffusa, più attinente alla realtà. Ho scelto di considerare in ombra anche i punti che hanno fino a 15 shadow ray che arrivano alla luce, così da estendere la zona d'ombra e ridurre quella di penombra.

Riferimenti bibliografici

[1] `https://github.com/danielepolidori/LabComputerGraphics` - repository GitHub di questo lavoro.