Report LAB01

Nicola Modugno

prof.ssa Serena Morigi

Abstract—In questo elaborato viene descritta l'estensione del programma `LAB_01.cpp`, mirato alla manipolazione interattiva di punti di controllo su un piano 2D tramite OpenGL e GLFW. A partire da una versione base che consente l'inserimento e la cancellazione di punti, sono state implementate una curva di Bézier mediante l'algoritmo di de Casteljau e un sistema di modifica dinamica dei punti tramite trascinamento. Inoltre, è stata integrata l'interpolazione Catmull-Rom per ottenere una curva fluida che attraversa i punti di controllo.

1. Introduzione

Il programma di partenza `LAB_01.cpp` permette all'utente di inserire punti di controllo (Control Points, CP) in una finestra grafica 2D mediante clic del mouse, visualizzando un poligono di controllo che connette i punti tramite segmenti. I punti vengono memorizzati in un array bidimensionale `vPositions_CP[MaxNumPts][2]` contenente le coordinate (x, y).

La gestione dell'input avviene tramite callback di GLFW, che permettono il rilevamento e la gestione in tempo reale degli eventi dell'utente. Il sistema consente la cancellazione dei punti di controllo premendo il tasto `f` (rimozione del primo punto) o `l` (rimozione dell'ultimo punto), attraverso la funzione `key_callback()`.

Le specifiche dell'esercizio richiedevano di:

1. Testare i controlli da tastiera e mouse.
2. Analizzare l'uso delle callback GLFW per la cattura degli eventi.
3. Implementare la curva di Bézier tramite l'algoritmo di de Casteljau.
4. Consentire il trascinamento dei punti di controllo.
5. Integrare nel programma in alternativa una tra le seguenti
   (a) disegno di una curva di Bézier interpolante a tratti (Catmull-Rom Spline)
   (b) disegno di una curva di Bézier mediante algoritmo ottimizzato basato sulla suddivisione adattiva
   (c) disegno di una curva di Bézier composta da tratti cubici, dove ogni tratto viene raccordato con il successivo con continuità C0,C1, o G1 a seconda della scelta utente (da keyboard).

2. Utilizzo delle Callback di OpenGL e GLFW

Il progetto è strutturato in due file principali: `LAB_01.cpp` che gestisce la scena e contiene le strutture dati e `gestione_callback.cpp` che contiene le callback per la gestione dell'input. Le due parti comunicano attraverso variabili globali, come `vPositions_CP`, `mouseOverIndex` e `isMovingPoint`. La base del funzionamento interattivo è rappresentata dall'impiego delle callback offerte da GLFW, che consentono di reagire agli eventi generati dall'utente, come il movimento del mouse, la pressione di un tasto o un clic. Queste callback sono state assegnate a funzioni specifiche: `mouse_button_callback()` rileva i clic del mouse per inserire nuovi punti o attivarne la modifica; `cursor_position_callback()` permette l'aggiornamento in tempo reale delle coordinate durante il trascinamento; infine, `key_callback()` interpreta la pressione dei tasti `f` e `l` per eliminare i punti. L'interazione tra utente e grafica è resa possibile da un ciclo di rendering continuo che aggiorna la scena dopo ogni modifica. Per rilevare se il mouse si trova sopra un punto esistente, si scorre l'array `vPositions_CP` e si confronta la distanza con il cursore, secondo la formula euclidea:

$$d = \sqrt{(x_{mouse} - x_i)^2 + (y_{mouse} - y_i)^2}$$

Se $d < 0.06$, il punto viene considerato "attivo" e modificabile. In questo caso, `mouseOverIndex` viene aggiornato con l'indice del punto selezionato. Durante il trascinamento, la variabile `isMovingPoint` è impostata a `true`, e `cursor_position_callback()` richiama la funzione `modifyPoint()` per aggiornare in tempo reale la posizione del punto selezionato. Al rilascio del tasto del mouse, `isMovingPoint` viene settata a `false`, confermando la nuova posizione.

3. Curva di Bézier: Algoritmo di de Casteljau

La funzione seguente implementa l'algoritmo di de Casteljau per approssimare la curva di Bézier partendo dai punti di controllo e dal parametro t.

```
void deCasteljau(float t, float* result) {
    float coordX[MaxNumPts], coordY[MaxNumPts];

    for (int i = 0; i < NumPts; i++) {
        coordX[i] = vPositions_CP[i][0];
        coordY[i] = vPositions_CP[i][1];
    }

    for (int i = 1; i < NumPts; i++) {
        for (int k = 0; k < NumPts - i; k++) {
            coordX[k] = (1 - t) * coordX[k] + t * coordX
    ↪ [k + 1];
            coordY[k] = (1 - t) * coordY[k] + t * coordY
    ↪ [k + 1];
        }
    }

    result[0] = coordX[0];
    result[1] = coordY[0];
}
```

Code 1. Algoritmo di de Casteljau

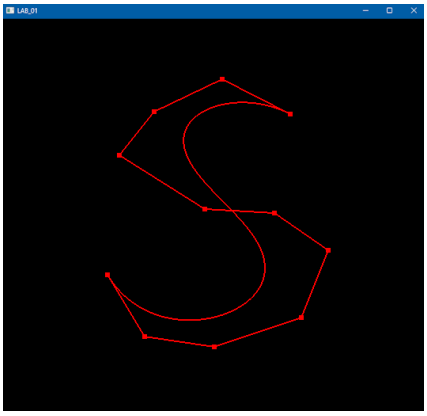

Figure 1. Resa della curva con Bézier

L'algoritmo funziona copiando inizialmente le coordinate dei punti in due array temporanei. Successivamente, esegue iterazioni di interpolazione lineare tra coppie di punti adiacenti, fino a ottenere un singolo punto, che corrisponde alla posizione finale della curva per un determinato parametro t.

4. Catmull-Rom Spline

In base alla traccia, è stata scelta l'opzione **(a)**: la curva Catmull-Rom interpolante.

La curva Catmull-Rom è una spline interpolante a tratti che garantisce continuità nella prima derivata (C1: vettori tangenti con stessa direzione e verso) ed è progettata per passare attraverso i punti di controllo. A differenza delle curve di Bézier, che tendono a "stare

dentro" il *convex-hull* dei punti di controllo, approssimando la curva, la Catmull-Rom attraversa i punti generando una curva più aderente al profilo desiderato. In questo contesto, la curva è detta **interpolante** nei punti di controllo. La differenza più significativa risiede nell'introduzione di nuove strutture dati nella versione modificata per supportare la gestione dinamica delle curve di Bézier. Di seguito verranno mostrati alcuni *snippet* del file `LAB01.cpp`.

```
// Aggiunta nella versione modificata
float vPositions_Bezier[MaxNumPts][2];  // Punti di
    ↪ controllo Bezier
int NumBezierPts = 0;
int mouseOverIndex = -1;
bool isMovingPoint = false;
```

Code 2. Nuove strutture dati nella versione modificata

Queste strutture supportano la generazione dinamica di punti di controllo per curve cubiche di Bézier, moltiplicando per tre la capacità di memorizzazione per gestire la tripla di punti P_i, P_{i+1}, P_{i-1} necessaria per ogni segmento di curva.

Inoltre, l'espressione utile a calcolare la distanza euclidea sono state incapsulate in un'unica funzione:

```
float findDistance(float firstPoint_xPos, float
    ↪ firstPoint_yPos,
                   float secondPoint_xPos, float
    ↪ secondPoint_yPos) {
    return sqrt(pow(firstPoint_xPos - secondPoint_xPos,
    ↪ 2) +
                pow(firstPoint_yPos - secondPoint_yPos,
    ↪ 2));
}

void findPoint(float xPos, float yPos) {
    float dist;
    for (int i = 0; i < NumPts; i++) {
        dist = findDistance(vPositions_CP[i][0],
    ↪ vPositions_CP[i][1],
                            xPos, yPos);
        if (dist < 0.06) {
            mouseOverIndex = i;
            printf("Mouse over CP %i\n", i);
            return;
        }
        else
            mouseOverIndex = -1;
    }
}
```

Code 3. Sistema di rilevamento dei punti di controllo

4.1. Costruzione dei Punti di Controllo Bézier

La funzione `buildBezierControlPoints()` rappresenta l'implementazione per la generazione dei punti per ottenere una spline Catmull-Rom in cui approssimiamo la derivata in P_i con il coefficiente:

$$m_i = \frac{P_{i+1} - P_{i-1}}{2}$$

e definiamo i control points da individuare come:

$$P_i^+ = P_i + \frac{m_i}{3}$$

e

$$P_i^- = P_i - \frac{m_i}{3}$$

Di seguito è mostrata parte del codice utilizzato per implementare la formula.

```
void buildBezierControlPoints() {
    if (NumPts < 4) {
        NumBezierPts = 0;
        return;
    }

    NumBezierPts = 0;

    // Calcoliamo le tangenti m_i per ogni punto
    ↪ interpolato
    // I punti interpolati sono P_0, P_3, P_6, P_9, ...
    ↪ (ogni 3 punti a partire dal primo)

    for (int i = 0; i < NumPts && NumBezierPts <
    ↪ MaxNumPts - 3; i += 3) {
        // Punto corrente da interpolare
        float P_i[2] = { vPositions_CP[i][0],
    ↪ vPositions_CP[i][1] };
        // Calcolo della tangente m_i = (P_{i+1} - P_{i-
    ↪ 1}) / 2
        float m_i[2];

        if (i == 0) {
            // Primo punto: usiamo differenza in avanti
            if (i + 1 < NumPts) {
                m_i[0] = (vPositions_CP[i + 1][0] - P_i[
    ↪ 0]) * 0.5f;
                m_i[1] = (vPositions_CP[i + 1][1] - P_i[
    ↪ 1]) * 0.5f;
            }
        }
        else if (i >= NumPts - 1) {
            // Ultimo punto: usiamo differenza all'
    ↪ indietro
            m_i[0] = (P_i[0] - vPositions_CP[i - 1][0])
    ↪ * 0.5f;
            m_i[1] = (P_i[1] - vPositions_CP[i - 1][1])
    ↪ * 0.5f;
        }
        else {
            // Punto intermedio: differenza centrata
            m_i[0] = (vPositions_CP[i + 1][0] -
    ↪ vPositions_CP[i - 1][0]) * 0.5f;
            m_i[1] = (vPositions_CP[i + 1][1] -
    ↪ vPositions_CP[i - 1][1]) * 0.5f;
        }

        // Se non  il primo segmento, aggiungiamo P-_i (
    ↪ punto di controllo entrante)
        if (i > 0 && NumBezierPts < MaxNumPts) {
            float Pminus_i[2] = { P_i[0] - m_i[0] / 3.0f,
    ↪  P_i[1] - m_i[1] / 3.0f };
            vPositions_Bezier[NumBezierPts][0] =
    ↪ Pminus_i[0];
            vPositions_Bezier[NumBezierPts][1] =
    ↪ Pminus_i[1];
            NumBezierPts++;
        }
        // Aggiungiamo il punto interpolato P_i
        if (NumBezierPts < MaxNumPts) {
            vPositions_Bezier[NumBezierPts][0] = P_i[0];
            vPositions_Bezier[NumBezierPts][1] = P_i[1];
            NumBezierPts++;
        }

        // Se non  l'ultimo segmento, aggiungiamo P+_i (
    ↪ punto di controllo uscente)
        if (i < NumPts - 3 && NumBezierPts < MaxNumPts)
    ↪ {
            float Pplus_i[2] = { P_i[0] + m_i[0] / 3.0f,
    ↪  P_i[1] + m_i[1] / 3.0f };
            vPositions_Bezier[NumBezierPts][0] = Pplus_i
    ↪ [0];
            vPositions_Bezier[NumBezierPts][1] = Pplus_i
    ↪ [1];
            NumBezierPts++;
        }
    }
}
```

Code 4. Algoritmo di costruzione dei punti di controllo

La funzione `computeBezierCurve` unisce all'array dei control points inseriti dall'utente quelli calcolati con la funzione `buildBezierControlPoints`. Anzitutto, verifica che siano stati aggiunti quattro punti e, successivamente, calcola il numero di tratti dal numero di punti. Ottenuto il numero di tratti, copia i punti necessari alla realizzazione della `Catmull-Rom Spline` in un array temporaneo per ciascun tratto. I punti presenti nell'array `vPositions_CP` vengono memorizzati in un array temporaneo e sostituiti con quelli nuovi realizzati tramite le formule matematiche precedenti. Infine, l'array viene ripopolato inserendo i punti di controllo originali. I punti vengono renderizzati chiamando l'algoritmo di `deCasteljeau()`.

```
void computeBezierCurve() {
    if (NumBezierPts < 4) return;

    int curveIndex = 0;
    int numSegments = (NumBezierPts - 1) / 3;

    // Calcola punti per segmento in base al numero totale di segmenti
    // per non superare MaxNumPts
    int pointsPerSegment = (MaxNumPts - 10) / numSegments; // -10 per sicurezza
    if (pointsPerSegment < 10) pointsPerSegment = 10; // minimo 10 punti per segmento
    if (pointsPerSegment > 50) pointsPerSegment = 50; // massimo 50 punti per segmento

    // Per ogni tratto di curva cubica (4 punti di controllo alla volta)
    for (int tratto = 0; tratto < numSegments && curveIndex < MaxNumPts - pointsPerSegment; tratto ++) {
        int startIdx = tratto * 3;

        // Copiamo i 4 punti di controllo del segmento corrente in vPositions_CP temporaneamente
        float tempCP[4][2];
        for (int i = 0; i < 4 && startIdx + i < NumBezierPts; i++) {
            tempCP[i][0] = vPositions_Bezier[startIdx + i][0];
            tempCP[i][1] = vPositions_Bezier[startIdx + i][1];
        }

        // Salviamo i punti originali
        float originalCP[MaxNumPts][2];
        int originalNumPts = NumPts;
        for (int i = 0; i < NumPts; i++) {
            originalCP[i][0] = vPositions_CP[i][0];
            originalCP[i][1] = vPositions_CP[i][1];
        }

        // Impostiamo temporaneamente i 4 punti di controllo per deCasteljau
        NumPts = 4;
        for (int i = 0; i < 4; i++) {
            vPositions_CP[i][0] = tempCP[i][0];
            vPositions_CP[i][1] = tempCP[i][1];
        }

        // Generiamo punti sulla curva usando deCasteljau
            int actualPointsThisSegment = (tratto == numSegments - 1) ? pointsPerSegment + 1 : pointsPerSegment;

        for (int i = 0; i < actualPointsThisSegment && curveIndex < MaxNumPts; i++) {
            float t = (float)i / (float)(pointsPerSegment);
            if (t > 1.0f) t = 1.0f; // clamp a 1.0

            float result[2];
            deCasteljau(t, result);
            vPositions_C[curveIndex][0] = result[0];
            vPositions_C[curveIndex][1] = result[1];
            curveIndex++;
            if (NumPts >= MaxNumPts - 10) {
                removeFirstPoint();
            }
        }

        // Recuperiamo i punti originali
        NumPts = originalNumPts;
        for (int i = 0; i < NumPts; i++) {
            vPositions_CP[i][0] = originalCP[i][0];
            vPositions_CP[i][1] = originalCP[i][1];
        }
    }

    NumPoints = curveIndex;
}
```

Code 5. Algoritmo di computazione delle curve a tratti

Infine, la funzione `drawScene()`, chiama le funzioni `buildBezierControlPoints` e `computeBezierCurve`.

```
if (NumPts > 3) {
    buildBezierControlPoints();
    computeBezierCurve();

    if (NumPoints > 0) {
        glBindVertexArray(vao_2);
        glBindBuffer(GL_ARRAY_BUFFER, vposition_Curve_ID );
        glBufferData(GL_ARRAY_BUFFER, sizeof( vPositions_C),
                     &vPositions_C[0], GL_STREAM_DRAW);
        glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE,
                              2 * sizeof(float), (void*)0) ;
        glEnableVertexAttribArray(0);
        glPointSize(0.5);
        glDrawArrays(GL_POINTS, 0, NumPoints);
        glLineWidth(2.0);
        glDrawArrays(GL_LINE_STRIP, 0, NumPoints);
        glBindVertexArray(0);
    }
}
```

Code 6. Rendering nella versione modificata

Nella Figura 2 è mostrata la resa della curva ottenuta mediante Catmull-Rom Spline. Confrontandola con la Figura 1 si può osservare come la curva Bézier tende a non passare per i punti di controllo, limitandosi risultando molto più approssimativa, mentre la Catmull-Rom li attraversa, rendendo l'interazione visiva più coerente con l'input dell'utente.

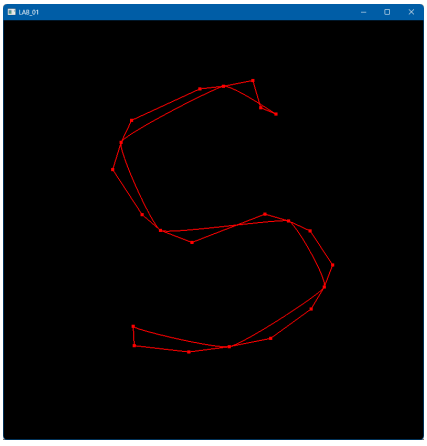

Figure 2. Resa della curva con Catmull-Rom Spline