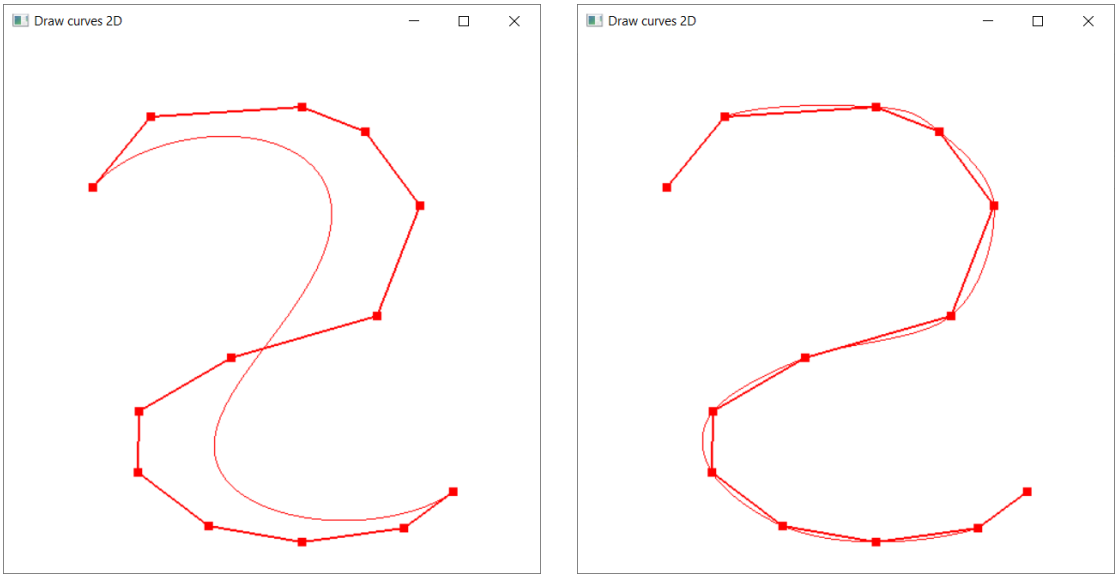
Fondamenti di Computer Graphics M (8 cfu)

Lab 01 - Disegno di curve di Bézier

Michele Righi

Settembre 2023



(a) De Casteljau con metodo uniforme (b) Catmull-Rom Spline

Figura 1: Demo Draw curves 2D

Traccia

3. Disegnare la curva di Bézier a partire dai punti di controllo inseriti, utilizzando l'algoritmo di de Casteljau.
4. Permettere la modifica della posizione dei punti di controllo tramite trascinamento con il mouse.
5. Integrare nel programma in alternativa uno dei seguenti punti:

(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.

1 Struttura del Progetto

Ho creato il progetto separando in file differenti le diverse implementazioni per disegnare le curve:

- `01_main.cpp`, contiene il main e tutte le funzioni riguardanti la logica di inizializzazione, input, e creazione/modifica dei control points;
- `01_3.h`, espone una funzione che applica l'algoritmo di de Casteljau;
- `01_5a.h`, espone una funzione che applica l'algoritmo Catmull-Rom Spline, per disegnare una curva interpolante a tratti;
- `01_5b.h`, espone una funzione che applica l'algoritmo di suddivisione adattiva;
- `01_5c.h`, sarà uno sviluppo futuro ed esporrà una funzione che applica l'algoritmo per disegnare una curva composta da tratti cubici;

2 Controlli

Ai comandi già presenti per aggiungere/rimuovere i control points, ho aggiunto i seguenti:

- Drag & drop tramite il cursore del mouse, sopra i control points per spostarli;
- `'i'`, stampa su standard output la lista delle coordinate dei control points;
- `'o'`, stampa su standard output la lista delle coordinate dei punti della curva;
- `'s'`, permette di attivare/disabilitare - a scorrimento - la visualizzazione di control points, segmenti, punti della curva;
- `'c'`/`'r'`, azzera la lista dei control point e quella dei punti della curva, pulendo il canvas;
- `'F1'` (default), seleziona l'algoritmo di de Casteljau base, aggiornando la curva;
- `'F2'`, seleziona l'algoritmo Catmull-Rom Spline, aggiornando la curva;
- `'F3'`, seleziona l'algoritmo di suddivisione adattiva, aggiornando la curva;
- `'F4'`, seleziona l'algoritmo di suddivisione a segmenti cubici.

3 Implementazione

Per l'implementazione delle funzioni che generano i punti della curva, ho utilizzato un *puntatore a funzione*:

```
// Function pointer to calculate the curve
static void (*getCurve)(Point2D* ctrlPts, int numCtrl, Point2D* curvePts, int* numCurve);
```

Avendo a disposizione diversi algoritmi, ciò ne semplifica il passaggio da uno all'altro, in quanto basta semplicemente cambiare la funzione a cui il puntatore è indirizzato.

La funzione `update()`, che viene chiamata ad ogni frame, in caso siano presenti almeno due control points, utilizza la funzione salvata nel puntatore `*getCurve`, per disegnare la curva, salvandola in `curvePointArray`:

```
static void update(int value) {
    // Update curve: draw the curve only if there are at least 2 control points
    if (numCtrlPts > 1) {
        if(getCurve != NULL)
            getCurve(ctrlPointArray, numCtrlPts, curvePointArray, &numCurvePts);
    }

    glutPostRedisplay();
    glutTimerFunc(UPDATE_DELAY, update, 0);
}
```

3.1 Punto 3: De Casteljau

L'algoritmo di de Casteljau è un metodo ricorsivo che permette di ottenere, per costruzione geometrica, un polinomio $f(t)$. Questo può essere usato per calcolare la curva di Bézier: scegliendo un insieme di valori abbastanza grande, al variare di t il polinomio forma un'approssimazione della curva sufficientemente densa da sembrare appunto una curva, invece che una sequenza di segmenti.

```
static Point2D deCasteljau(Point2D* points, int numPoints, float t) {
    // Auxiliary array
    Point2D pointArrayAux[MAX_NUM_PTS] = { 0.0f, 0.0f };

    // Copy to the auxiliary array
    for (int i = 0; i < numPoints; i++) {
        pointArrayAux[i] = points[i];
    }
```

```
9
10      // "Depth" loop
11      for (int i = 1; i < numPoints; i++) {
12          // Points loop
13          for (int j = 0; j < numPoints - 1; j++) {
14              pointArrayAux[j] = lerp(pointArrayAux[j], pointArrayAux[j + 1], t);
15          }
16      }
17      return pointArrayAux[0];
18  }
19
20  void getCurve_deCasteljau(Point2D* ctrlPts, int numCtrl, Point2D* curvePts, int* numCurve) {
21      if (numCtrl < 2)
22          return;
23
24      *numCurve = MIN(100, MAX(MAX_NUM_PTS, numCtrl * 2));
25
26      for (int i = 0; i < *numCurve; i++) {
27          curvePts[i] = deCasteljau(ctrlPts, numCtrl, (float)i / *numCurve);
28      }
29  }
```

3.2 Punto 4: Drag & Drop

Poiché col tasto sinistro del mouse si può sia aggiungere un nuovo control point, che trascinarlo, ho deciso di non permettere all'utente di aggiungere un control point alle stesse coordinate in cui se ne trova già un altro.

Dunque, quando l'utente muove il mouse sul canvas, se il cursore si posiziona in uno spazio vuoto, il click sinistro aggiungerà un nuovo control point; altrimenti, se il cursore si posiziona su un control point, premere il tasto sinsitro del mouse permetterà di trascinarlo.

Per implementare questa feature ho usato le callback di GLUT `glutPassiveMotionFunc()`, `glutMouseFunc()` e `glutMotionFunc()`:

- con `glutPassiveMotionFunc()`, quando l'utente posiziona il mouse su un control point, salviamo in una variabile `iHoverCtrlPt` l'indice del punto, altrimenti assegnamo il valore -1;
- con `glutMouseFunc()`, al click del tasto sinistro del mouse controlliamo se `iHoverCtrlPt` contiene un indice. In caso affermativo, entriamo in modalità "dragging". Al rilascio, usciamo dalla modalità "dragging";
- con `glutMotionFunc()`, se siamo in modalità "dragging", aggiorniamo le coordinate del control point a cui l'indice `iHoverCtrlPt` fa riferimento.

A questo workflow ho aggiunto anche il cambio di cursore per hovering di un control point e dragging.

3.3 Punto 5a: Catmull-Rom Spline

A differenza dell'algoritmo di de Casteljau, con Catmull-Rom la curva passa per ciascun control point (approssimazione). Con questo algoritmo, per calcolare ciascun punto della curva, abbiamo bisogno di 4 punti: una coppia per ciascuna parte ($t = 0.0$ e $t = 1.0$). Dati due control point P_i e P_{i+1} (Fig. 3), ci serve trovare i punti P_i^+ e P_{i+1}^-. P_i^- e P_{i+1}^+ sono noti, e sono rispettivamente il control point precedente e quello successivo (per poter disegnare una curva con questo algoritmo abbiamo bisogno di almeno 4 control points).

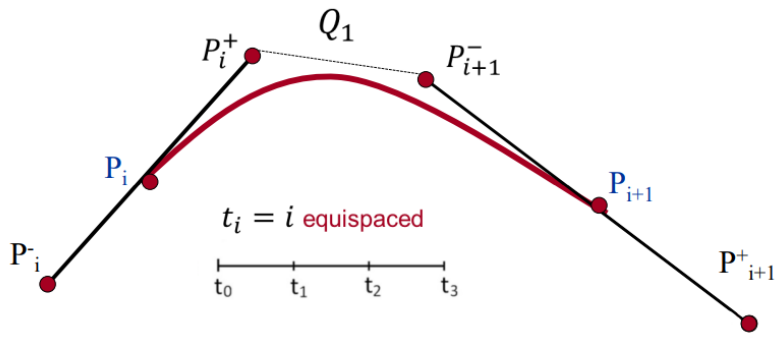


Figura 2: Catmull-Rom spline

Per trovare i control point mancanti possiamo usare la stima della derivata in P_i:

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

Da cui ricaviamo:

$$P_i^+ = P_i + \frac{1}{3} m_i$$

$$P_i^- = P_i - \frac{1}{3} m_i$$

Ponendola in un altro modo, considerando 4 control point $P0$, $P1$, $P2$, $P3$, possiamo ottenere il punto sulla curva al variare di t, tramite la seguente formula:

$$q(t) = 0.5*((2*P1)+(-P0+P2)*t+(2*P0-5*P1+4*P2-P3)*t2+(-P0+3*P1-3*P2+P3)*t3)$$

```
1 static Point2D catmullRomSpline(Point2D p0, Point2D p1, Point2D p2, Point2D p3, float t) {
2     Point2D res;
```

```
    res.x = 0.5 * (
        (2 * p1.x) +
        (-p0.x + p2.x) * t +
        (2 * p0.x - 5 * p1.x + 4 * p2.x - p3.x) * t * t +
        (-p0.x + 3 * p1.x - 3 * p2.x + p3.x) * t * t * t
    );
    res.y = 0.5 * (
        (2 * p1.y) +
        (-p0.y + p2.y) * t +
        (2 * p0.y - 5 * p1.y + 4 * p2.y - p3.y) * t * t +
        (-p0.y + 3 * p1.y - 3 * p2.y + p3.y) * t * t * t
    );

    return res;
}

void getCurve_CatmullRomSpline(Point2D* ctrlPts, int numCtrl, Point2D* curvePts, int* numCurve) {
    // NB: we need at least 4 control points
    if (numCtrl < 4)
        return;

    Point2D p1, p2, p3, p4;
    int ptsPerSpline = ((MAX_NUM_PTS * 2) - 2) / (numCtrl * 4);

    *numCurve = 0;

    for (int i = 1; i < numCtrl - 2; i++) {
        p1 = ctrlPts[i - 1];
        p2 = ctrlPts[i];
        p3 = ctrlPts[i + 1];
        p4 = ctrlPts[i + 2];

        for (int j = 0; j <= ptsPerSpline; j++) {
            float t = (float)j / ptsPerSpline;
            curvePts[*numCurve] = catmullRomSpline(p1, p2, p3, p4, t);
            (*numCurve)++;
        }
    }
}
```

3.4 Punto 5b: Suddivisione Adattiva

L'idea su cui si basa questo algoritmo è suddividere ricorsivamente una curva, data dai segmenti tra i vari control points, in sotto-curve sempre più piccole, fino ad ottenere un'approssimazione adeguata, in base ad un "flat test".

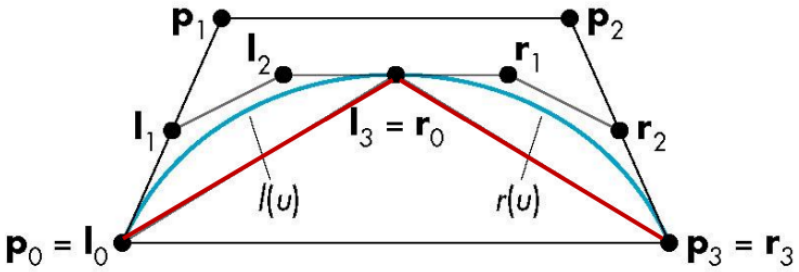


Figura 3: Adaptive subdivision

Per implementare questo algoritmo, ci serve una versione di de Casteljau leggermente modificata che, per ciascuna ricorsione, restituisca i nuovi control point (verranno usati per le due sotto-curve).

```
static void adaptiveSubdivision(Point2D* tmpPtsArray, int numPts) {
    bool isFlat = true; // if true we consider the segment as "flat"

    // Extract external Control Points p1 and p2: 0, N-1
    Point2D p1 = tmpPtsArray[0];
    Point2D pN = tmpPtsArray[numPts - 1];

    // Flat Test: if each point between p1 and pN is <= planary tollerance,
    // then we consider the segment (p1,pN) flat
    for (int i = 1; i < numPts - 1; i++) {
        // Flat Test: Calculate distance tmpPtsArray[i] from line
        float dist = distance(p1, pN, tmpPtsArray[i]);
        if (dist > flatTolerance) {
            isFlat = false;
            break;
        }
    }

    if (isFlat) {
        // Draw the segment, as we consider it for the curve
        curveSegments[numSeg] = p1;
        curveSegments[numSeg + 1] = pN;
```

```
23
24          numSeg++;
25      }
26      else {
27          // Divide the curve into 2
28          Point2D res[(MAX_NUM_PTS * 2) - 1];
29          int num;
30          Point2D subdLeft[MAX_NUM_PTS] = { 0.0f, 0.0f };
31          Point2D subdRight[MAX_NUM_PTS] = { 0.0f, 0.0f };
32
33          // Apply deCasteljau in t=0.5 saving CtrlPts of the 2 new curves
34          deCasteljauControlPoints(tmpPtsArray, numPts, 0.5f, res, &num);
35
36          for (int i = 0; i < num; i++) {
37              if (i < numPts) {
38                  subdLeft[i] = res[i];
39              }
40              if (i >= numPts-1) {
41                  subdRight[i - numPts+1] = res[i];
42              }
43          }
44
45          adaptiveSubdivision(subdLeft, numPts);
46          adaptiveSubdivision(subdRight, numPts);
47      }
48  }
49
50  void getCurve_adaptiveSubdivision(Point2D* ctrlPts, int numCtrl, Point2D* curvePts, int* numCurve) {
51      if (numCtrl < 2)
52          return;
53
54      numSeg = 0;
55      adaptiveSubdivision(ctrlPts, numCtrl);
56
57      for (int i = 0; i < numSeg; i++) {
58          curvePts[i] = curveSegments[i];
59      }
60      *numCurve = numSeg;
61  }
```