REPORT PER LAB_01

Matteo giri

Introduzione e comandi

Questo è il report per l'esercizio di laboratorio **LAB_01**. E' possibile scegliere il tipo di curva di Bézier da disegnare decommentando la relativa funzione di draw nella drawScene.

```
if (NumPts > 2) {

    //for uniform bezier curve
    uniformBezierCurve();

    //for adaptive bezier curve
    //numeroTratti = 0;
    //adaptiveBezierCurve(PointArray, NumPts);

    //for catmull-rom spline curve
    //catmullRomBezierCurve(PointArray, NumPts);

    //for continuity spline curve
    //continuitySplineBezierCurve(PointArray, NumPts);

    //draw curve
    glBindVertexArray(VAO_2);
    glBindBuffer(GL_ARRAY_BUFFER, VBO_2);
    glBufferData(GL_ARRAY_BUFFER, sizeof(CurveArray), &CurveArray[0], GL_STATIC_DRAW);
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    glLineWidth(0.5);
    glDrawArrays(GL_LINE_STRIP, 0, 101); //for uniform bezier curve
    //glDrawArrays(GL_LINE_STRIP, 0, numeroTratti + 1); //for adaptive bezier curve
    //glDrawArrays(GL_LINE_STRIP, 0, (NumPts-1)*tsize+1); //for catmull-rom spline curve
    //glDrawArrays(GL_LINE_STRIP, 0, (NumPts - 1)*tsize + 1); //for continuity spline curve
    glBindVertexArray(0);
}
```

Per esempio, se si vuole disegnare una normale curva di Bézier con l'algoritmo di de Casteljau si decommenta *uniformBezierCurve()* e il relativo *glDrawArrays(),* come in figura. Per il punto 5 ho cercato di implementare tutti e tre i metodi, è quindi possibile scegliere che tipo di curva disegnare decommentando le relative righe di codice. È inoltre possibile cambiare la posizione dei punti di controllo cliccando sopra uno di essi e trascinandolo con il mouse su una nuova posizione. Per quanto riguarda il punto 5.c si può scegliere il tipo di continuità che si vuole ottenere premendo il tasto "0", "1", "2" prima di inserire uno o più nuovi punti della curva: il tasto "0" e "1" settano la continuità a due diversi tipi di C0, mentre il tasto "1" setta la continuità a C1 (per i prossimi tratti ovviamente, quelli già disegnati avranno la continuità settata precedentemente). Non sono riuscito con il metodo utilizzato a realizzare la scelta della continuità G1 (però se pensiamo che C1 comporta G1 allora scegliere "1" vale anche come continuità G1).

Dettagli realizzativi

Gli step 1 e 2 sono trascurati in quanto già implementati di base.

STEP 3: Disegnare la curva di Bézier utilizzando l'algoritmo di de Casteljau

Per disegnare la curva utilizzando de Casteljau ho realizzato la funzione *uniformBezierCurve* che viene chiamata nel *drawscene:*

```
//CALCULATE BEZIER CURVE (UNIFORM METHOD)
void uniformBezierCurve() {
    float result[3]; //result point array

    // for each step call deCasteljau algorithm
    for (int i = 0; i <= Steps; i++) {
        deCasteljau(PointArray,(GLfloat)i / Steps, result,NumPts);

        //put the results in the array
        CurveArray[i][0] = result[0];
        CurveArray[i][1] = result[1];
    }
}
```

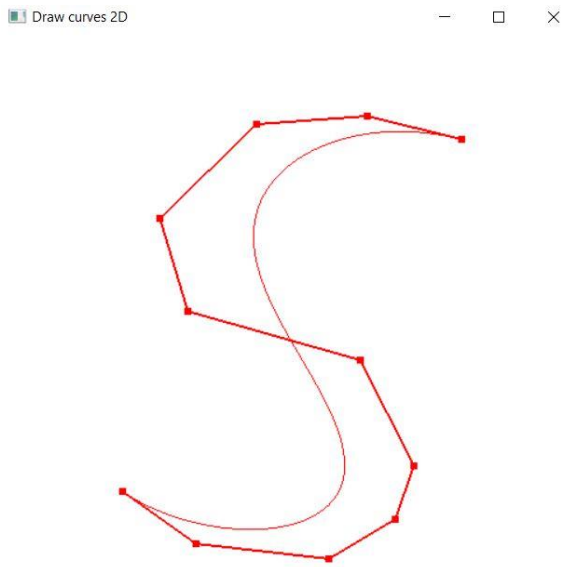
La funzione chiama la funzione *deCasteljau()* che implementa l'algoritmo, per *Steps* volte (Steps sarebbe il numero di punti con cui la curva di Bèzier verrà approssimata) e inserisce i risultati nell'array *CurveArray* che contiene tutti i punti della curva che verrà poi disegnata. La funzione *deCasteljau()* è implementata come segue:

```
//function that applies deCasteljau method to create a Bezier curve
void deCasteljau(float tempArray[MaxNumPts][2],float t,float (&result)[3],float NumPts) {
    float pi0[MaxNumPts][2]; //output array
    int n = NumPts-1; //curve degree

    for (int i = 0; i < MaxNumPts; i++) { //copy the array
        pi0[i][0] = tempArray[i][0];
        pi0[i][1] = tempArray[i][1];
    }
    for (int i = 1; i <= n;i++) {
        for (int j = 0; j <= n-i; j++) {
            pi0[j][0] = (1 - t)*pi0[j][0] + t * pi0[j+1][0];
            pi0[j][1] = (1 - t)*pi0[j][1] + t * pi0[j + 1][1];
        }
    }
    result[0] = pi0[0][0];
    result[1] = pi0[0][1];
}
```

Prende in ingresso l'array di punti di controllo *tempArray*, il parametro *t*, l'array dove verrà messo il risultato e il numero di punti di controllo. Tutti i risultati intermedi dell'algoritmo vengono inseriti nello stesso array, nel quale i risultati precedenti che non servono più vengono sovrascritti con i nuovi.

La curva viene infine disegnata nella drawscene andando a collegare i vari punti calcolati in *curveArray* con delle linee.

Esempio di esecuzione:



STEP 4: modifica della posizione dei punti di controllo tramite trascinamento del mouse

Per permettere la modifica della posizione dei punti di controllo per prima cosa ho modificato la funzione *myMouseFunc()*:

```
void myMouseFunc(int button, int state, int x, int y) {
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
        // (x,y) viewport(0,width)x(0,height)   -->   (xPos,yPos) window(-1,1)x(-1,1)
        float xPos = -1.0f + ((float)x) * 2 / ((float)(width));
        float yPos = -1.0f + ((float)(height - y)) * 2 / ((float)(height));

        selectedIndex = pointAlreadyExists(xPos, yPos);
        if (selectedIndex == -1) {
            addNewPoint(xPos, yPos);
            glutPostRedisplay();
        }
    }
    if (button == GLUT_LEFT_BUTTON && state == GLUT_UP)
    {
        selectedIndex = -1;
    }
}
```

La funzione ora controlla se nel punto cliccato con il tasto sinistro del mouse c'è già un control point tramite la funzione *pointAlreadyExists()* che controlla nell'array dei punti di controllo se c'è un punto in quella posizione (o nelle immediate vicinanze, per permettere di premere su un punto più facilmente), e in caso positivo ritorna l'indice del control point nell'array, che viene salvato nella variabile *selectedIndex*. Se il tasto sinistro viene rilasciato *selectedIndex* torna a -1 (che sta ad indicare nessun punto selezionato). In questo modo possiamo sapere se un punto di controllo è stato cliccato e soprattutto se il tasto del mouse è ancora premuto. Per effettuare il trascinamento vero e proprio ho realizzato la funzione *myMouseMotion()* che viene chiamata ogni volta che il mouse viene spostato sulla finestra (tramite glutMotionFunc):

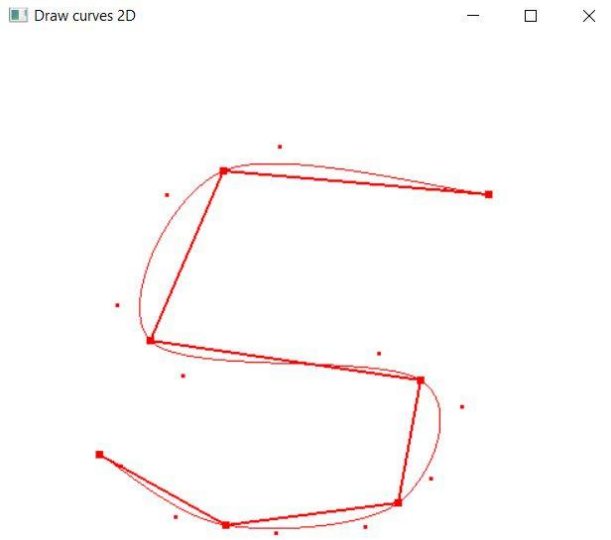
```
void myMouseMotion(int x, int y) {

    if (selectedIndex != -1) // If we have a selected control point
    {
        //change the coordinates of the control point to be the ones of the mouse
        float xPos = -1.0f + ((float)x) * 2 / ((float)(width));
        float yPos = -1.0f + ((float)(height - y)) * 2 / ((float)(height));

        PointArray[selectedIndex][0] = xPos;
        PointArray[selectedIndex][1] = yPos;
        glutPostRedisplay();
    }
}
```

Se il mouse viene spostato mentre *selectedIndex* non è -1, la posizione del control point corrispondente all' indice *selectedIndex viene aggiornata con i valori della posizione del puntatore del mouse.*

STEP 5.a: disegno di una curva di Bézier interpolante a tratti (Catmull-Rom Spline)

Per disegnare la curva interpolante tramite Catmull-Rom Spline il procedimento è simile a quello per disegnarla tramite approssimazione visto nel punto 3. Nella drawscene viene chiamata la funzione *catmullRomBezierCurve()* che prende in ingresso i punti di controllo e il numero di punti di controllo (non metto la funzione qui perché è lunga). Per prima cosa nella funzione definisco le variabili per la stima delle derivate (m1, m2) e per gli unknown control points (p1, p2), e poi ciclo su tutti i control points andando a lavorare a coppie (control point Pi e Pi+1) usando il metodo spiegato nelle slide: per prima cosa calcolo m1 e m2, poi tramite m1 e m2 ottengo i 2 unknown control points tra i e i+1 che inserisco in un HiddenPointArray insieme a Pi e Pi+1. Infine, disegno i control points calcolati ed effettuo deCasteljau su questi 4 control points per ottenere i punti della parte della curva completa relativa a questa coppia e li inserisco in *CurveArray*. Quando tutte le coppie di control points saranno state completate all'interno di *CurveArray* sarà presente l'intera curva creata tramite interpolazione, che viene disegnata nella drawscene collegando i vari punti come visto nel punto 3.

Esempio di esecuzione:



Nella figura, i punti più piccoli sono i punti di controllo intermedi ottenuti tramite l'algoritmo (gli unknown control points).

STEP 5.b: disegno di una curva di Bézier mediante suddivisione adattiva

Come per gli altri punti, ho realizzato una funzione chiamata *adaptiveBezierCurve()* che viene chiamata nella drawscene. La funzione implementa il metodo della suddivisione adattiva visto a lezione e prende in ingresso i punti di controllo e il numero di punti di controllo:

```
//CALCULATE BEZIER CURVE (ADAPTIVE SUBDIVISION METHOD)
void adaptiveBezierCurve(float tempArray[MaxNumPts][2],int NumPts) {
    //Extract external control points
    float p1[2];
    float p2[2];
    for (int i = 0; i < 2; i++) {
        p1[i] = tempArray[0][i];
        p2[i] = tempArray[NumPts - 1][i];
    }

    //Calculate segment between p1 and p2
    float m = (p2[1] - p1[1]) / (p2[0] - p1[0]);
    float b = p1[1] - m * p1[0];

    //flatness test
    int test_planarita = 1;
    for (int i = 1; i < NumPts - 1; i++) {
        float distanza = distanza_punto_retta(tempArray[i][0], tempArray[i][1], m, b);
        if (distanza > tol_planarita)
            test_planarita = 0;
    }

    //if flatness test is good we can draw the segment between p1 and p2
    if (test_planarita == 1) {
        CurveArray[numeroTratti][0] = p1[0];
        CurveArray[numeroTratti][1] = p1[1];
        CurveArray[numeroTratti+1][0] = p2[0];
        CurveArray[numeroTratti+1][1] = p2[1];
        numeroTratti++;
    }
    //else split curve into 2 and repeat process for both
    else {
        float subd_1[MaxNumPts][2];
        float subd_2[MaxNumPts][2];

        deCasteljauSplit(tempArray, 0.5, NumPts, subd_1, subd_2);

        adaptiveBezierCurve(subd_1, NumPts);
        adaptiveBezierCurve(subd_2, NumPts);
    }
}
```

Per prima cosa estraggo i due punti di controllo esterni passati come parametro e calcolo il segmento tra questi. Questo segmento mi serve per fare il test di planarità, ovvero andare a vedere se la distanza tra i punti di controllo e questo segmento sono minori di un certo valore. Questo è fatto tramite la funzione *distanza_punto_retta* che va a calcolare appunto la distanza di ogni singolo punto con il segmento calcolato. Se tutte le distanze calcolate sono minori di una certa soglia allora salvo i due punti estremi in *CurveArray* (che verranno poi utilizzati per disegnare la curva). Se il test di planarità non viene passato, vado a dividere in due la curva di Bézier formata da tutti i control points utilizzando l'algoritmo di splitting con de casteljau e vado a chiamare ricorsivamente la funzione con i due nuovi set di punti di controllo ottenuti dallo splitting.

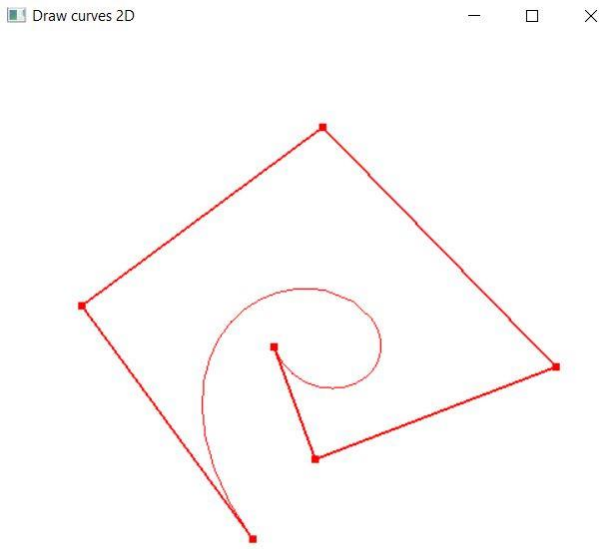
Lo split è effettuato dalla funzione *deCasteljauSplit* che prende in ingresso l'insieme dei punti di controllo, il punto di split, il numero di punti di controllo e i due array dove andranno messi i control points delle due nuove curve:

```
//function that applies deCasteljau method to split a curve
void deCasteljauSplit(float tempArray[MaxNumPts][2], float t, int NumPts, float (&curve1)[MaxNumPts][2], float(&curve2)[MaxNumPts][2]) {
    float pi0[MaxNumPts][2];
    int n = NumPts - 1; //curve degree

    for (int i = 0; i < MaxNumPts; i++) { //copy the array
        pi0[i][0] = tempArray[i][0];
        pi0[i][1] = tempArray[i][1];
    }

    curve1[0][0] = pi0[0][0];
    curve1[0][1] = pi0[0][1];
    curve2[NumPts - 1][0] = pi0[NumPts - 1][0];
    curve2[NumPts - 1][1] = pi0[NumPts - 1][1];

    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= n - i; j++) {
            pi0[j][0] = (1 - t)*pi0[j][0] + t * pi0[j + 1][0];
            pi0[j][1] = (1 - t)*pi0[j][1] + t * pi0[j + 1][1];
            //if its the first lerp of the iteration I save the point to the first curve
            if (j == 0) {
                curve1[i][0] = pi0[j][0];
                curve1[i][1] = pi0[j][1];
            }
            //if its the last lerp of the iteration I save the point to the second curve
            if (j == n - i) {
                curve2[NumPts-i-1][0] = pi0[j][0];
                curve2[NumPts-1-i][1] = pi0[j][1];
            }
        }
    }
}
```

Per prima cosa copio l'array dei punti di controllo e poi salvo il primo punto di controllo come primo punto della nuova prima curva, e l'ultimo come ultimo punto della nuova seconda curva. Poi effettuo l'algoritmo di de casteljau e salvo i punti di controllo trovati nelle rispettive curve.

Alla fine di tutto questo processo avremo un insieme di punti della curva ottenuta tramite suddivisione adattiva nell'array *CurveArray,* che verrà poi disegnato a schermo.

Esempio di esecuzione:

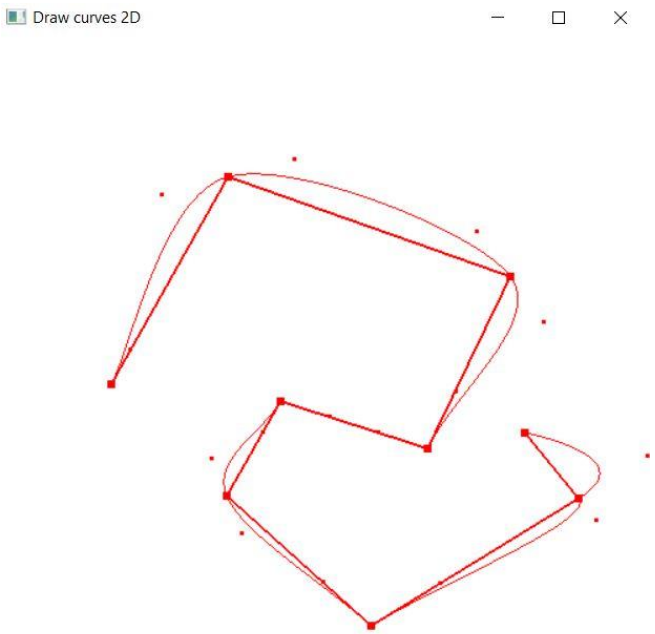


STEP 5.c: disegno di una curva di Bézier composta da tratti cubici, dove ogni tratto viene raccordato con il successivo con continuità C0, C1, o G1 a seconda della scelta utente da keyboard

Per disenare una curva di bezier a tratti dove l'utente può scegliere la continuità del collegamento tra ogni tratto ho scelto di utilizzare la TCB Spline come visto nelle slide (ridotta solo al parametro c per evitare di appesantire troppo il codice). Per realizzarla ho creato un nuovo array *continuityArray* che viene popolato con il valore di continuità selezionato ogni volta che un nuovo punto di controllo viene creato. Quindi se al momento è selezionato un valore di continuità pari a c=0 per esempio, se creo un nuovo punto di controllo verrà aggiunto 0 nella posizione dell'array relativa al nuovo punto. L'utente può cambiare il valore di continuità c premendo il tasto "0" per c=1, "1" per c=0 e "2" per c=-1. Premetto che non sono riuscito a codificare la continuità G1 con questo metodo, però la C1 e C0 si (in quanto se c=0 la continuità sarà C1, perché le formule per le derivate si trasformano in quelle della Catmull-Rom spline che crea una curva di continuità C1, mentre se c=1 o c=-1 si ha continuità C0).

La funzione che realizza la TCB spline con scelta di continuità è *continuitySplineBezierCurve.* La funzione è molto simile a quella per realizzare una curva con Catmull-Rom (step 5.a), con la differenza che in questo caso utilizzo le formule viste su slide per trovare ri (tangente di destra del punto di controllo corrente) e li (tangente di sinistra del punto di controllo successivo), con i quali posso trovare i due unknown control points che permettono di realizzare la curva cubica. La formula per trovare ri usa un valore di c pari a quello dell'array *continuityArray* alla posizione del punto corrrente, mentre per trovare li si usa un valore di c pari a quello del punto successivo. La procedura per disegnare il tutto è poi equivalente a quella dello step 5.a.

Esempio di esecuzione:



Come si può vedere dall'esempio, il primo e secondo tratto sono collegati con continuità C1, stessa cosa per il secondo con il terzo, mentre il terzo con il quarto e il quarto con il quinto sono in continuità C0. Il quinto con il sesto sono continuità C1, mentre il sesto con il settimo e il settimo con l'ottavo sono in continuità C0.