Programming Exercise 6: Animating the drawing of Bezier curve

Introduction

In this exercise, you will implement two algorithms "uniform subdivision" and "recursive subdivision" to draw the Bezier curve of arbitrary degree. You could animate the drawing process of Bezier curve as you like (for example the animation effect http://upload.wikimedia.org/wikipedia/commons/f/ff/Bezier_3_big.gif). Before starting on this programming exercise, we strongly recommend watching the lectures.

To get started with the exercise, you will need to download the starter codes. There is a skeleton file "***Bezier.cpp***" which provided all necessary code for you to start .

To run these programs ,the glut enviroment should be setup. You can get the glut from glut-3.7.6-bin.zip (117 KB)(http://user.xmission.com/~nate/glut/glut-3.7.6-bin.zip) ,unzip it and copy three files to different locations on your window as following:

- glut.h – This is the file you'll have to include in your source code. The common place to put this file is in the gl folder which should be inside the include folder of your system. for example ,your vc2010 include folder "*c:\program files\Microsoft Visual Studio 10.0\VC\include\gl*" or "*c:\program files(x86)\Microsoft Visual Studio 10.0\VC\include\gl*" in(my computer)
- glut32.lib (Windows version) – This file must be linked to your application so make sure to put it your lib folder. example ,your vc2010 lib folder "*c:\program files\Microsoft Visual Studio 10.0\VC\lib*" or "*c:\program files(x86)\Microsoft Visual Studio 10.0\VC\lib*" in(my computer)
- glut32.dll (Windows) – You could place the dll file in your exe's folder.for example,the sustem folder: "*c:\windows\system32*" or "*c:\windows\sysWOW64*" in my computer

The representation of a Bezier curve in the skeleton program

We provided a simple C struct Point2f to define a point on the 2D plane:

```
struct Point2d{
    double x;
    double y;
};
```

A Bezier curve could be represented by its control points which is just an array of 2D points:

```
typedef vector<Point2d> Bezier;
```

In the skeleton program ,a global variable called " Bezier b_curve " is then used to store the control points of a Bezier curve and a helper function "init()" is used to input these control points.

Evaluate a point on the curve at a parameter t

To draw a Bezier curve ,you should be able to evaluate a point on the curve at a parameter u. The simple way to do this is to use the Bezier equation to directly evaluate the point at a parameter t. For example, the equation of Bezier curve with degree 3 is :

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2 t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3 \ , t \in [0,1].$$

So you could get a point at a parameter using this eqaution by writing the following code piece:

```
inline Point2d bezier(const Bezier &curve,const double u){
    Point2d P;
    const Bezier &C = curve;
    P.x=pow(1-u,3)*C[0].x+3*pow(1-u,2)*u*C[1].x+3*(1-u)*pow(u,2)*C[2].x+pow(u,3)*C[3].x;
    P.y=pow(1-u,3)*C[0].y+3*pow(1-u,2)*u*C[1].y+3*(1-u)*pow(u,2)*C[2].y+pow(u,3)*C[3].y;
    return P;
}
```

As we mentioned in the class, this method to evaluate a point on the curve is not effective .This method is also not general enough to evaluate a point on a Bezier curve of arbitrary degree.

So a more effective method is to use the DeCasteljau's constructive method to evaluate a point on a a Bezier curve of arbitrary degree (see figure 1 the evaluation process for a 3 degree Bezier curve).

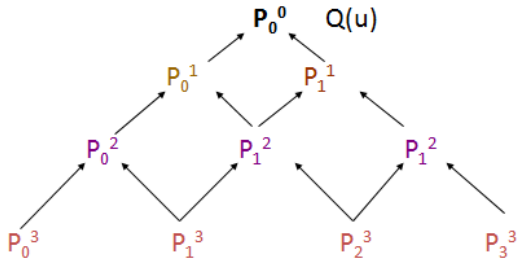


figure1 DeCasteljau's constructive method

The DeCasteljau Implementation for evaluating a point could be like the follwing (figure 2). We recommend you to use the to evaluate a point on a Bezier curve. We have written the function " Point2d DeCasteljau_Bezier(const Bezier &curve,const double u)" for you, Your **first task** is to fill the missing code in this function. Then you can modify the code in function " draw_uniform_subdivided_Bezier " from

```
        Point2d Q = bezier(curve,u); //Bezier(curve,u)
//      Point2d Q = DeCasteljau_Bezier(curve,u);
```

to

```
        //Point2d Q = bezier(curve,u); //Bezier(curve,u)
        Point2d Q = DeCasteljau_Bezier(curve,u);
```

Input: Control points C_i with $0 \leq i \leq n$ where n is the degree.
Output: $f(u)$ where u is the parameter for evaluation

1 for $(level = n\ ;\ level \geq 0\ ;\ level--)$ {
2 if $(level == n)$ { // Initial control points
3 $\forall i : 0 \leq i \leq n : p_i^{level} = C_i$; continue ; }
4 for $(i = 0\ ;\ i \leq level\ ;\ i++)$
5 $p_i^{level} = (1-u) * p_i^{level+1} + u * p_{i+1}^{level+1}$;
6 }
7 $f(u) = p_0^0$

figure2 DeCasteljau Implementation

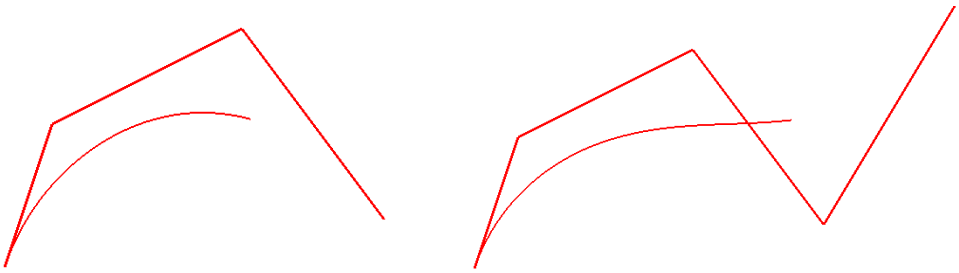
```
//==============DeCasteljau=====================
Point2d DeCasteljau_Bezier(const Bezier &curve,const double u)
{
    int n = curve.size()-1; //where n is the degree of Bezier,and control points are n+1
    vector<Bezier> pi_level(n+1);
    for(int level = n;level>=0;level--){ //
        pi_level[level].resize(level+1);
          //there will be 0,1...level controls points in this level,the total number of points is level+1
    }

    for(int level = n;level>=0;level--){
        if(level==n){
            for(int i = 0 ; i<=n;i++)
                pi_level[level][i] = curve[i];
            continue;
        }

        for(int i = 0;i<=level;i++){
            // ---please fill code here----
            //.............
        }
    }
    return pi_level[0][0];
}
```

We then implemented the uniform subdivision algorithm in the function " void draw_uniform_subdivided_Bezier(const Bezier &curve,int t = 0, int m=1000)" to draw the Bezier curve by subdivided the parameter range[0,1] into 'm' segments and connected these subdivided points on curve with lines to approximate the curve. You could set different values for 'm' to check the effect of subdivision. Then we use a function "

void animation_Bezier(const Bezier &curve,const int m = 200)" to animation the process of drawing the curve. When you run the program with different number of control points as following:

```
Point2d CP;
CP.x = 50;CP.y=100;          b_curve.push_back(CP);
CP.x = 100;CP.y=250;         b_curve.push_back(CP);
CP.x = 300;CP.y=350;         b_curve.push_back(CP);
CP.x = 450;CP.y=150;         b_curve.push_back(CP);
// CP.x = 600;CP.y=400;      b_curve.push_back(CP);
```

you will see different results as figure 3:



4 control points (degree3) 5 control points (degree3)

figure3 The animation of drawing of a Bezier curve

You second task is then to modify the function " animation_Bezier " to make it display a result like the following:

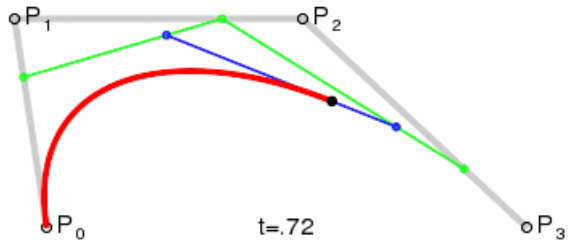


figure4 **The animation of constructive drawing of a Bezier curve**

Your three task is to code two functions "bool Flatness(const Bezier &curve,double flatness)" and " void Subdivision(const Bezier &curve,double u,Bezier &L,Bezier &R)" in the function " void draw_Bezier_Subdivision(const Bezier &curve,double flatness = 0.1)" . Similar to the function" draw_uniform_subdivided_Bezier ", the function " draw_Bezier_Subdivision " will draw the Bezier curve by recursively subdividing the curve. The code is written for you:

```
void draw_Bezier_Subdivision(const Bezier &curve,double flatness = 0.1){
  if(Flatness(curve,flatness) ) {
       // ---please fill code here----
       //draw V0V1, V1V2,V2V3 ,…-
       return;
  }
  Bezier L,R;   double u   =0.5;
  Subdivision(curve,u,L,R);
  draw_Bezier_Subdivision(L);
  draw_Bezier_Subdivision(R);
```

}

You should fill the missing code in function" draw_Bezier_Subdivision" too.

You could do a little modification to the function " DeCasteljau_Bezier " to finish the function "Subdivision" to generate two subdivided Beziers (in fact the control points for each). The function " Flatness " could be written depending on the the difference between the length of the control polygon and the length of the segment between endpoints (figure 5).

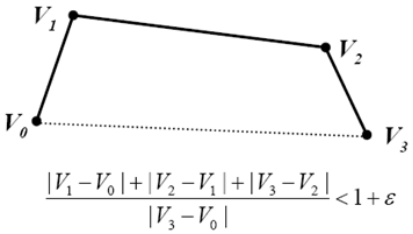


figure 5 flatness based on the difference ...

All parts of this programming exercise are due 2012/11/15 PM at 23:59