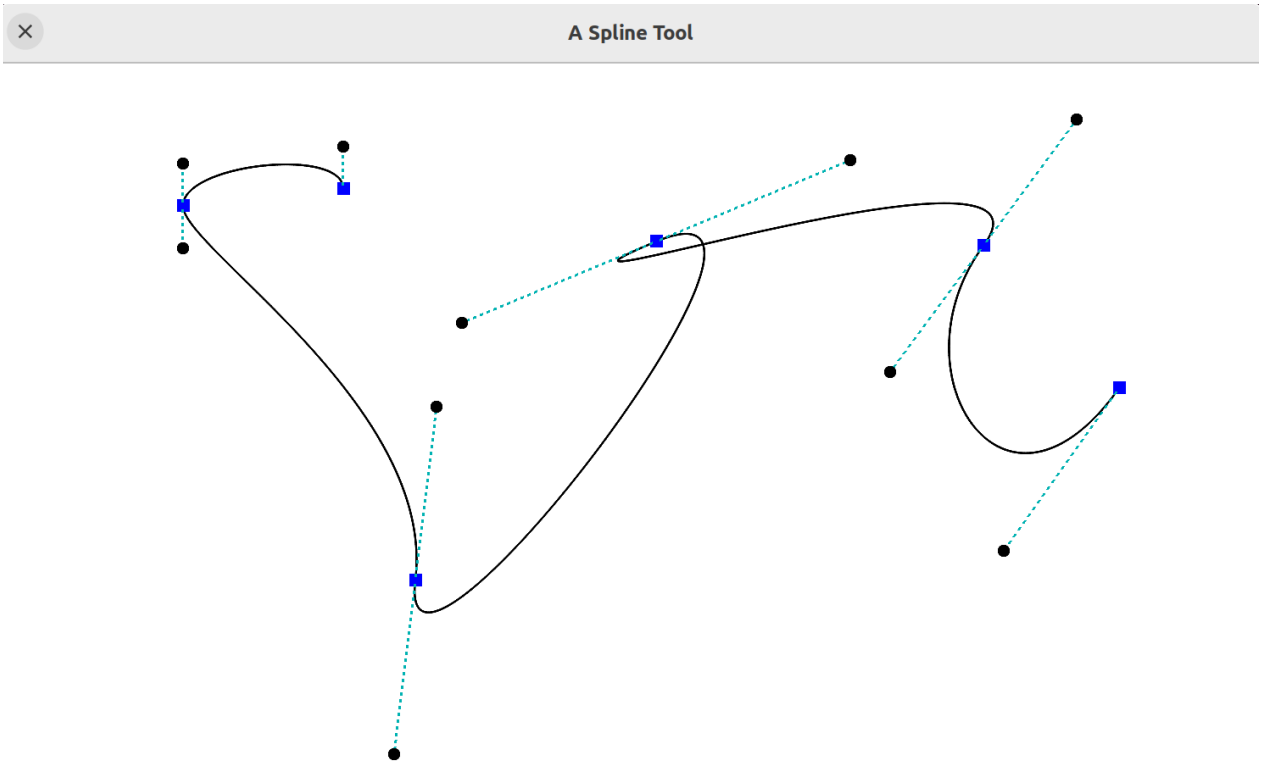
University of Western Ontario, Computer Science Department
CS3388B, Computer Graphics

Assignment 3 **Due: Thursday, February 16, 2022**

General Instructions: This assignment consists of 5 pages, 1 exercise, and is marked out of 100. For any question involving calculations you must provide your workings. *Correct final answers without workings will be marked as incorrect.* Each assignment submission and the answers within are to be solely your individual work and completed independently. Any plagiarism found will be taken seriously and may result in a mark of 0 on this assignment, removal from the course, or more serious consequences.

Submission Instructions: The answers to this assignment (code, workings, written answers) are to be submitted electronically to OWL. Ideally, any workings or written answers are to be typed. At the very least, clearly *scanned* copies (no photographs) of hand-written work. If the person correcting your assignment is unable to easily read or interpret your written answers then it may be marked as incorrect without the possibility of remarking. Include a **README** with any code.



Exercise 1. The goal of this assignment is to build a sort of limited *pen tool*, which creates and manipulates a cubic Bezier spline. The user adds points, or *nodes*, by clicking on the window. The program connects these nodes together as a spline and renders the spline. The user can move around existing nodes and their control points to modify the spline.

At a high-level, your program should perform the following:

- If the user clicks on an empty space within the window, a new node is added to the spline and this new node becomes one of the endpoints of the spline.
- If the user clicks and holds on an existing node, they should be able to "drag" that node around the window.
- If the user clicks and holds an on existing control point (often called a "handle"), then the handle moves around the window.

More details on how the program is expected to perform is broken into parts below.

The Setup.

1. To make things easy, you should setup your projection matrix and viewport to exactly match window coordinates. Thus, if you create a window with width W and height H, setup your projection matrix to be `glOrtho(0, W, 0, H, -1, 1)` and viewport to be `glViewport(0, 0, W, H)`.
2. Let screen width and screen height be command-line arguments for your program.
3. Enable 4 times multisampling in your window for anti-aliasing.

The Rendering.

In this step, assume you have a list of nodes and a list of control points already created.

1. Render the spline itself by rendering each piece as an independent cubic Bezier curve. Each curve should be a polyline with 200 line segments. Consider using `glLineWidth` of greater than 1.0. You may wish to "smooth" the line. This can be done with three simple functions:

```
glEnable(GL_LINE_SMOOTH);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

2. Render each node as a square point by simply specifying a reasonably large `glPointSize` (10-20) and rendering GL_POINTS.
3. Render each control point as a *round point*. It is the same as the nodes except now you will enable smoothing with the following three functions:

```
glEnable(GL_POINT_SMOOTH);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glBegin(GL_POINTS); ... glEnd();
```

4. Render a *dotted line* connecting the control point to its associated node. Recall that a cubic Bezier curve has two end points and two control points. In a cubic Bezier spline, each end point has 1 control point and each intermediate point has two control points (one for the "previous" curve and one for the "next" curve). To render a dotted line, use `GL_LINES` with a *stipple pattern*.

```
glEnable(GL_LINE_STIPPLE);
glLineStipple(scaleFactor, stipplePattern);
glBegin(GL_LINES); .... glEnd();
```

The Interaction.

Now we need a way for the user to interact with the program. This will involve mouse and keyboard controls.

1. If the user left-clicks and holds on an existing node, then that node should be continuously re-positioned to be under the cursor for as long as the left mouse button is held. All of the node's associated control points should move around as well, maintaining the same *offset* with the node at all times.
2. If the user left-clicks and holds on an existing control point, then that control point should be continuously re-positioned to be under the cursor for as long as the left mouse button is held. If the node whose control point we are manipulating is an interior node, then that node has a second control point which must also be moved as follows:
   (a) The distance from one control point to the node and the distance from the node to the other control point must always be equal.
   (b) The two control points and the node must all be *co-linear*. That is, they all fall on a straight line. For example, moving one control point left moves the other one right.
3. If the user left-clicks in the window and it is not on an existing node, then a new node should be added to the spline as follows.
   (a) Create a new node at the current cursor position. This node has one control point which should be placed 50 pixels above the node.
   (b) Find which of the two end points of the spline is closer to the new node. Let that closer end point now be an intermediate node and the new node by the new end point.
   (c) Add a second control point to the now intermediate node so that this new control point follows the rules of 2.a and 2.b, above.
4. If the user ever presses the "E" key on their keyboard, delete all nodes and control points and reset the program to its initial state.

Tips and Tricks.

1. In GLFW, you can use `glfwGetCursorPos`, `glfwGetMouseButton` and `glfwGetKey` to get interactions from the user.
2. GLFW uses a top-left origin for its cursor position. You can easily convert this to the bottom-left origin coordinate system of `glOrtho` using `mouseY = screenH - mouseY`

3. Remember that a spline is *implied* by the positions of its nodes and its control points. Thus, you should only explicitly store the nodes and control points as their (x, y) positions. The rendering takes care of actually drawing the Bezier curve between the nodes.

4. Organize your code into two distinct phases. First, process inputs and manipulate your data structure of nodes and control points. Second, render everything for this frame. Repeat forever.

5. You may organize your node data any way you wish, although I recommend the following.

```
struct Point {
    float x;
    float y;
};

struct Node : Point {
    bool hasHandle1, hasHandle2;
    Point handle1;
    Point handle2;
};

std::vector<Node> nodes;
```

6. Given a line $y = mx + b$ and a point on that line (x_0, y_0), there are two unique points at a distance d away from (x_0, y_0). Their x-coordinates are given by the formula:

$$x = x_0 \pm \sqrt{\frac{d^2}{1 + m^2}}$$

7. To find out if a user clicked on a node, take the current cursor position and iterate through your list of nodes, checking if the cursor position is less than a certain distance away from the node's position. A "tolerance" of about 10 pixels is reasonable. Maybe your tolerance here should be proportional to the `glPointSize` you chose earlier?