Interpolating Piecewise Quadratic Bezier Curves

HIMANSHU RAJ, Indraprastha Institute of Information Technology Delhi, India

1 ABSTRACT

This project involves modifying an existing piecewise linear curve drawing application to generate interpolating piecewise quadratic Bezier curves. The primary objective was to enforce G1 continuity at the curve joins and provide user-friendly control for designing the curves interactively. The implemented solution successfully integrates user-driven control points with real-time curve generation, achieving both functional and geometric correctness.

2 INTRODUCTION

Bezier curves are a fundamental tool in computer graphics for creating smooth, scalable curves. Bezier curves provide good local control and C1/G1 continuity at joins. In this project, the task was to modify an existing linear curve drawing application to implement interpolating piecewise quadratic Bezier curves, which are known for their ability to provide smooth transitions with local control. The goal was to maintain G1 continuity at the junctions between curve segments, ensuring that the curves appear visually smooth. Additionally, the solution was designed to allow real-time user interaction for adding and moving control points.

3 STRATEGY

To create the interpolating piecewise quadratic Bezier curve and to enforce G1 continuity at joins, we will break down the problem and build upon the approach. In further discussion, points drawn by the user will be referred to as the control points (P_i, i from 0) and points assumed by us to make bezier curves will be referred to as the phantom points (p_j, j from 1).

When we have the first control point, we just need to draw the point on the canvas. When we have the second control point, we just draw a linear bezier curve between the two points as instructed.

When we have our third control point (this is where our implementation starts), we need to draw two bezier curves that have G1 continuity at P_1. For this, we assume one phantom point p_1 between P_0 and P_1, and another phantom point p_2 between P_1 and P_2. Here we will have 3 quadratic bezier curves and 3 phantom points.

When we have our fourth control point, we assume one more phantom p_3 between P_2 and P_3, so that G1 continuity is maintained at P_2.

To maintain continuity, we are ensuring that tangents at joins are parallel (implementation required tangents to be equal as well, which ensures C1 continuity, which is stronger than G1 continuity, implying G1 continuity is held.)

Generally, we need to assume the first phantom point for the first quadratic bezier curve, and then the rest of the phantom points are calculated with respect to the previous control point and the previous phantom point. The curve is plotted after the phantom point is decided, and then the next curve is decided in a similar way. The continuity is maintained by ensuring parallel tangents of both curves at joins.

4 IMPLEMENTATION

Major code implementation is done in 'main.cpp/calculatePiecewiseQuadraticBezier()' to update bezier curves if control points are updated and 'main.cpp/reCalculatePiecewiseQuadraticBezier()' to update bezier curves if phantom points are updated.

Author's address: Himanshu Raj, himanshu22216@iiitd.ac.in, Indraprastha Institute of Information Technology Delhi, India.

Minor code changes are done in ‘main.cpp/main’ function to create, update and delete buffers and array objects for bezier curve and phantom points, and to handle events in the rendering and display loop.
Helper functions named ‘addPhantomPoints’, ‘searchNearestPhantomPoint’ and ‘editPhantomPoint’ are defined in ‘utils.cpp’ to give more control over curve joins (bonus part).
Helper functions and other minor changes are pretty similar and inspired by the starter code provided in the assignment.

In first phase of program execution when user can add points and see the interpolation, ‘calculatePiecewise-QuadraticBezier()’ helps to find the curve each time a new control point is added. It interpolates all the points again by assuming first phantom points, and subsequentially plotting next curves. We assume the first phantom point for the first quadratic bezier curve (on the perpendicular bisector at a proportionate length), and then the rest of the phantom points are calculated with respect to the previous control point and the previous phantom point. The curve is plotted after the phantom point is decided, and then the next curve is decided in a similar way.

In second phase of program execution when user can move control points as well as phantom points (bonus part), ‘calculatePiecewiseQuadraticBezier()’ helps to plot updated curve if a control point is updated (moved), and ‘reCalculatePiecewiseQuadraticBezier()’ helps to plot the updated curve if a phantom point is updated. We update all the phantom points except the one updated by user and all segments are plotted after this calculation of new phantom points.

5 RESULTS

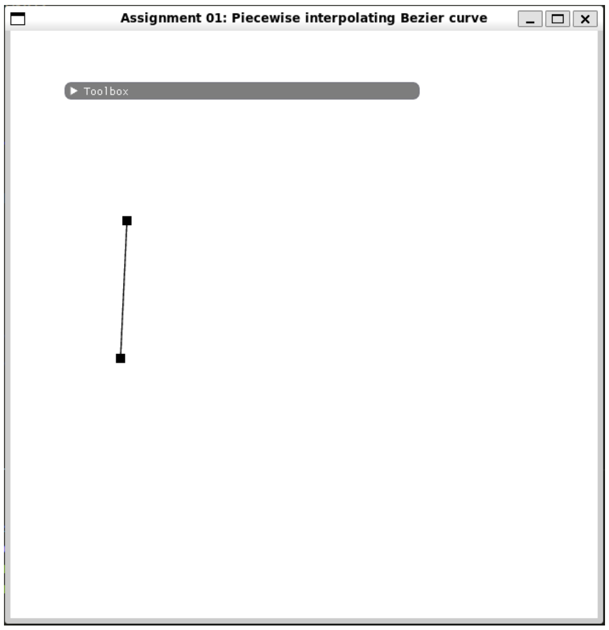

Fig. 1. First phase: 2 points

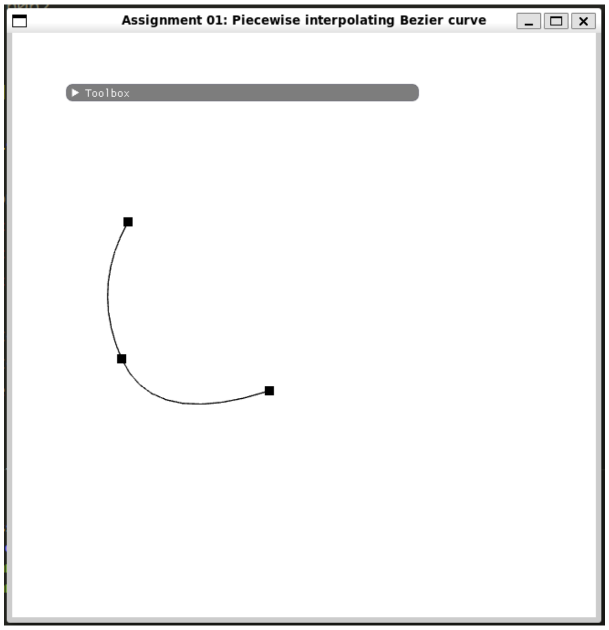

Fig. 2. First phase: 3 points

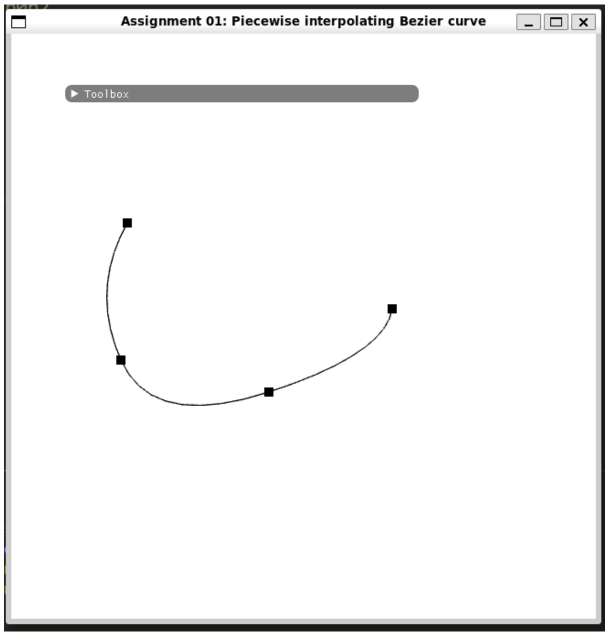

Fig. 3. First phase: 4 points

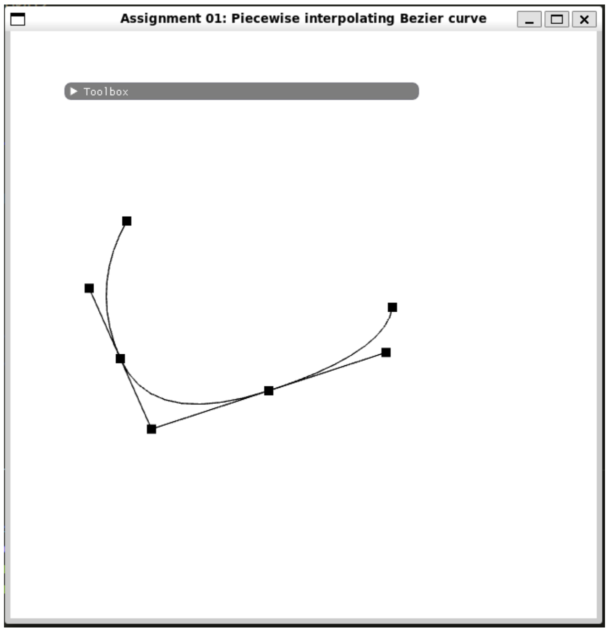

Fig. 4. Second phase: Mode switch

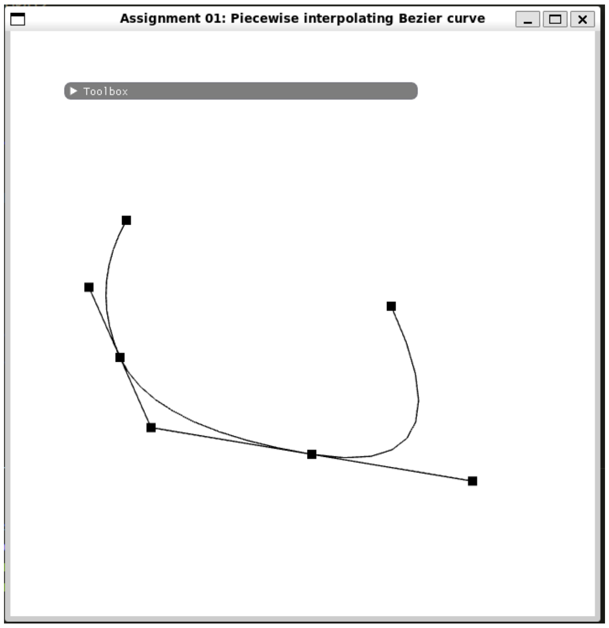

Fig. 5. Second phase: editing P_2

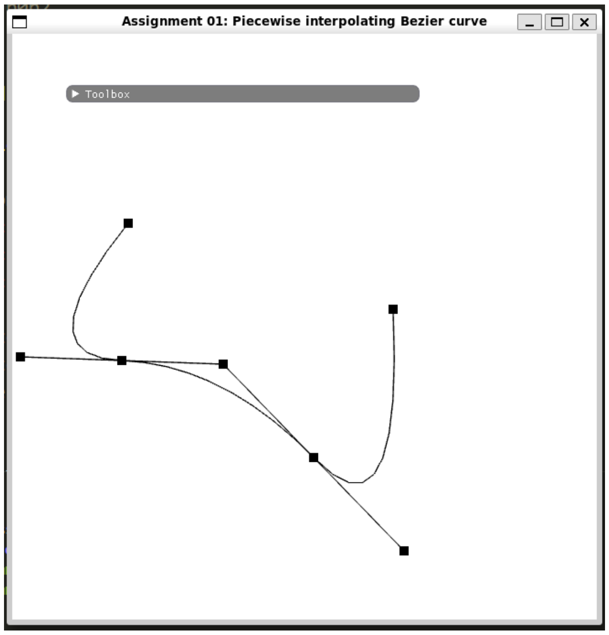

Fig. 6. Second phase: editing p_2