Interpolating Piecewise Cubic Bézier Curves with C[1] Continuity (Progressive Drawing)

AMAN SHARMA, IIIT Delhi, India

This report describes a simple strategy to draw an *interpolating* piecewise curve through user-provided points using *cubic Bézier* segments. C^1 continuity at joins is obtained by placing the inner control points along endpoint tangents computed from finite differences. The system draws the full curve *progressively* while points are being added, and supports moving points after placement; the curve updates immediately. Implementation details match the provided OpenGL/ImGui starter code.

1 PROBLEM SETTING

The starter program collects 2D points (x, y) from the user and, by default, shows the polyline. The goal is to replace the polyline display with a *piecewise cubic Bézier* interpolant that passes through all points and is C^1-continuous at joins. The curve must update in real time both while new points are added and when existing points are dragged.

2 CUBIC BÉZIER AND BERNSTEIN BASIS

A cubic Bézier with control points $B_0, \dots, B_3 \in \mathbb{R}^2$ is

$$B(t) = (1-t)^3 B_0 + 3(1-t)^2 t\, B_1 + 3(1-t)t^2\, B_2 + t^3 B_3, \quad t \in [0, 1].$$

The Bernstein basis functions are

$$b_0 = (1-t)^3, \quad b_1 = 3(1-t)^2 t, \quad b_2 = 3(1-t)t^2, \quad b_3 = t^3.$$

We render each segment by sampling $t_k = \frac{k}{N-1}$, $k = 0, \dots, N-1$, and emitting the points $B(t_k)$ as a line strip (`SAMPLES_PER_BEZIER = N`).

3 INTERPOLATION AND C[1] CONTINUITY

Let the user's points be $P_0, \dots, P_n$. Between consecutive points (P_i, P_{i+1}) we build one cubic segment with

$$B_0 = P_i, \qquad B_3 = P_{i+1}, \qquad B_1 = P_i + \tfrac{1}{3} T_i, \qquad B_2 = P_{i+1} - \tfrac{1}{3} T_{i+1},$$

where T_i is the tangent at P_i. This guarantees interpolation: $B(0) = P_i$ and $B(1) = P_{i+1}$.

The derivative of a cubic Bézier is

$$B'(t) = 3(1-t)^2 (B_1 - B_0) + 6(1-t)t(B_2 - B_1) + 3t^2 (B_3 - B_2),$$

hence $B'(0) = 3(B_1 - B_0)$ and $B'(1) = 3(B_3 - B_2)$. Using the *same* T_i for the segment ending at P_i and the segment starting at P_i makes the first derivative continuous at P_i, i.e. the curve is C^1.

Tangent estimation. We estimate tangents by finite differences:

$$T_0 = P_1 - P_0, \qquad T_i = \tfrac{1}{2}(P_{i+1} - P_{i-1}) \ (1 \le i \le n-1), \qquad T_n = P_n - P_{n-1}.$$

Interior tangents use a central difference; endpoints use forward/backward differences.

Author's address: Aman Sharma, IIIT Delhi, New Delhi, India.

4 PROGRESSIVE DRAWING AND INTERACTION

Progressive drawing. As soon as the user places two points, the first cubic segment is drawn. Each additional point P_k adds a new segment (P_{k-1}, P_k). On every add or drag operation we recompute the required tangents and rebuild the concatenated line strip.

Interaction.

- **Left click** on the canvas: add a new interpolation point.
- **Right click**: finish adding points; subsequent **left click + drag** selects and moves a point.
- The curve refreshes immediately in both modes (adding and editing).

5 IMPLEMENTATION NOTES

We store all positions in normalized device coordinates (NDC) and render with a pass-through shader. Three VAO/VBO pairs are used: control points, reference polyline (optional), and the piecewise Bézier line strip. When points change, we:

(1) upload control points,
(2) (optionally) rebuild the polyline samples,
(3) rebuild the Bézier samples via the Bernstein basis and upload.

5.1 Key Function: `calculatePiecewiseBezier()`

The core routine follows exactly the strategy above: derive tangents, build inner controls as $P \pm T/3$, sample with the Bernstein basis, and emit vertices. (Listing abridged for clarity.)

```
void calculatePiecewiseBezier() {
  piecewiseBezier.clear();
  const int m = (int)controlPoints.size();
  if (m < 2*3) return;

  struct Pt { float x, y; };
  std::vector<Pt> P; P.reserve(m/3);
  for (int i=0; i+2<m; i+=3) P.push_back({controlPoints[i],controlPoints[i+1]});
  const int n = (int)P.size() - 1; if (n <= 0) return;

  // Tangents (one-sided at ends, central inside)
  std::vector<Pt> T(P.size());
  if (P.size()==2) { T[0]={P[1].x-P[0].x, P[1].y-P[0].y}; T[1]=T[0]; }
  else {
    T[0] = {P[1].x-P[0].x, P[1].y-P[0].y};
    T[n] = {P[n].x-P[n-1].x, P[n].y-P[n-1].y};
    for (int i=1;i<n;++i)
      T[i] = {0.5f*(P[i+1].x-P[i-1].x), 0.5f*(P[i+1].y-P[i-1].y)};
  }

  const int samples = std::max(2, SAMPLES_PER_BEZIER);
  const float dt = 1.0f / float(samples-1);

  for (int i=0;i<n;++i) {
    Pt B0=P[i], B3=P[i+1];
    Pt B1={P[i].x + T[i].x/3.0f,     P[i].y + T[i].y/3.0f};
    Pt B2={P[i+1].x - T[i+1].x/3.0f, P[i+1].y - T[i+1].y/3.0f};
```

```
    for (int k=0;k<samples;++k) {
      float t = k*dt, u = 1.0f-t;
      float b0=u*u*u, b1=3*u*u*t, b2=3*u*t*t, b3=t*t*t;
      float x=b0*B0.x + b1*B1.x + b2*B2.x + b3*B3.x;
      float y=b0*B0.y + b1*B1.y + b2*B2.y + b3*B3.y;
      if (i>0 && k==0) continue; // avoid duplicate joint vertex
      piecewiseBezier.insert(piecewiseBezier.end(), {x,y,0.0f});
    }
  }
}
```

6 RESULTS AND DISCUSSION

The method produces a visually smooth curve that interpolates all user points and preserves C^1 continuity by construction. Because tangents come from local finite differences, the curve responds well to local edits. Sampling with $N = 10$ already gives a clean line strip; increasing N refines the rendering.

Limitations / Extensions. This version does not expose handles for manual tangent editing. Straightforward extensions include: interactive tangent handles, and an optional tangent-length optimization step to better align the curve with the control polyline.

7 CONCLUSION

We implemented an interpolating, C^1-continuous piecewise cubic Bézier curve that draws progressively while the user adds or moves points. The construction uses endpoint tangents and inner control points at $\pm T/3$, and a Bernstein-basis sampler for rendering. The approach is simple, robust, and integrates cleanly with the provided OpenGL/ImGui framework.