Bézier Curves

Philip Peterson

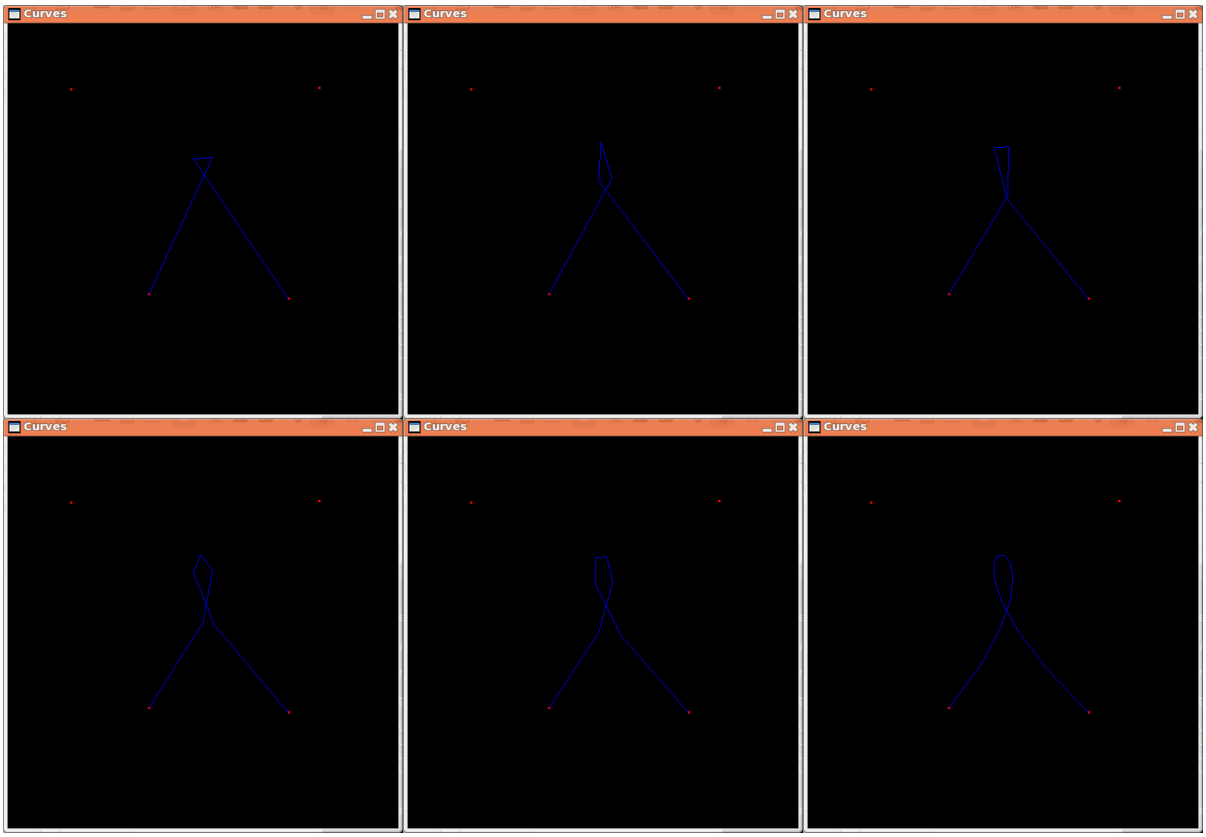
The implementations for drawing Bézier curves in this project all produce similar results. The subdivision method is rather unique, but because the de Casteljau algorithm produces essentially the same points as the blending functions (which in turn function identically to the OpenGL evaluator method), it is difficult to contrast the results of these approaches.

Nevertheless, provided below are some curves rendered using each of the implementations. The images are shown in order of increasing detail (number of steps or iterations).

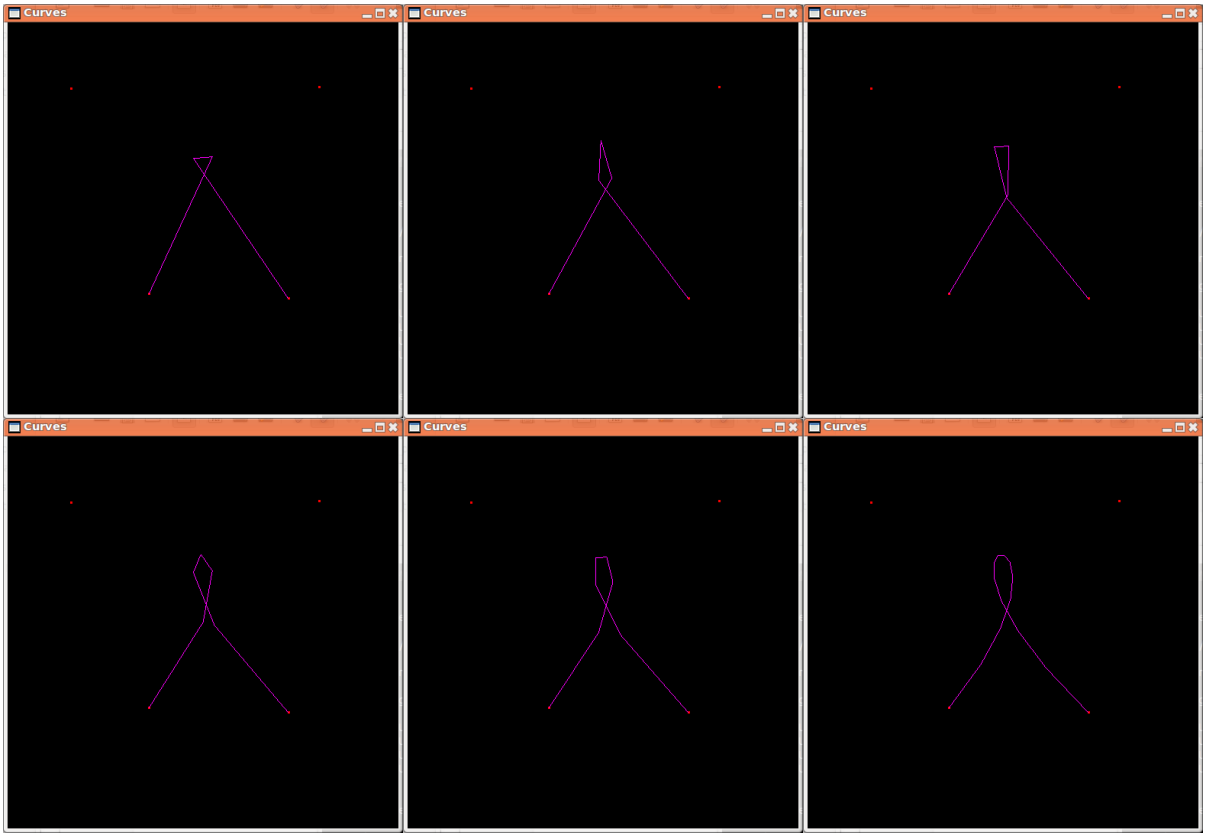
Renderings

Curve One

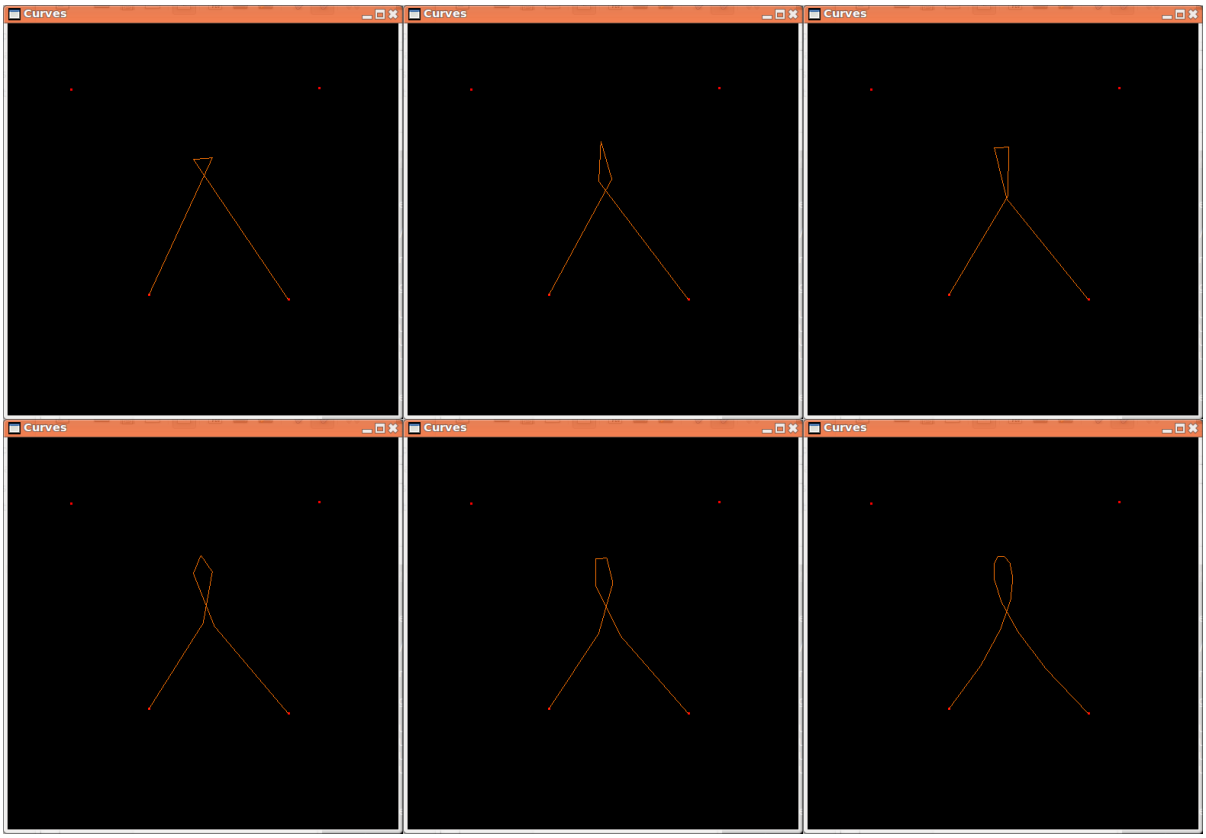
Blending Functions



De Casteljau's Algorithm



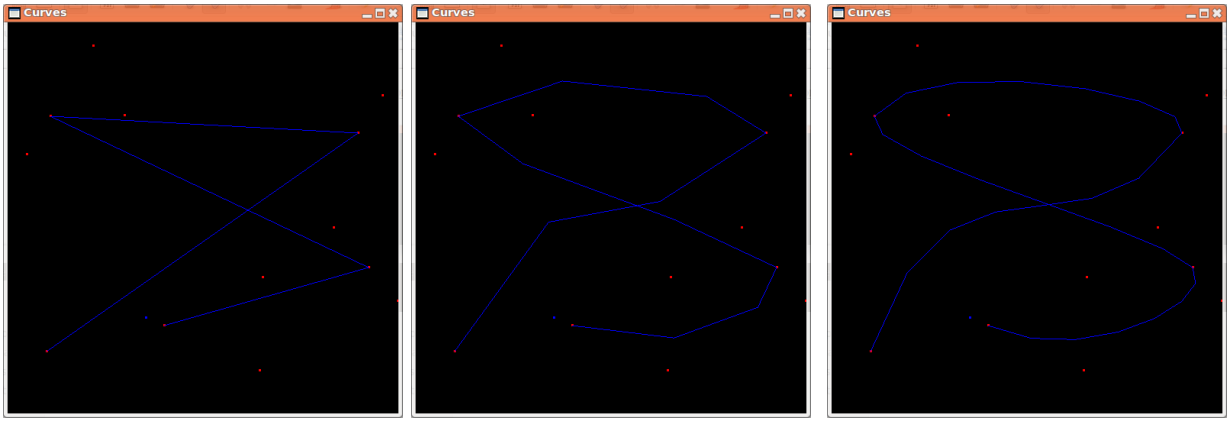
OpenGL Evaluators



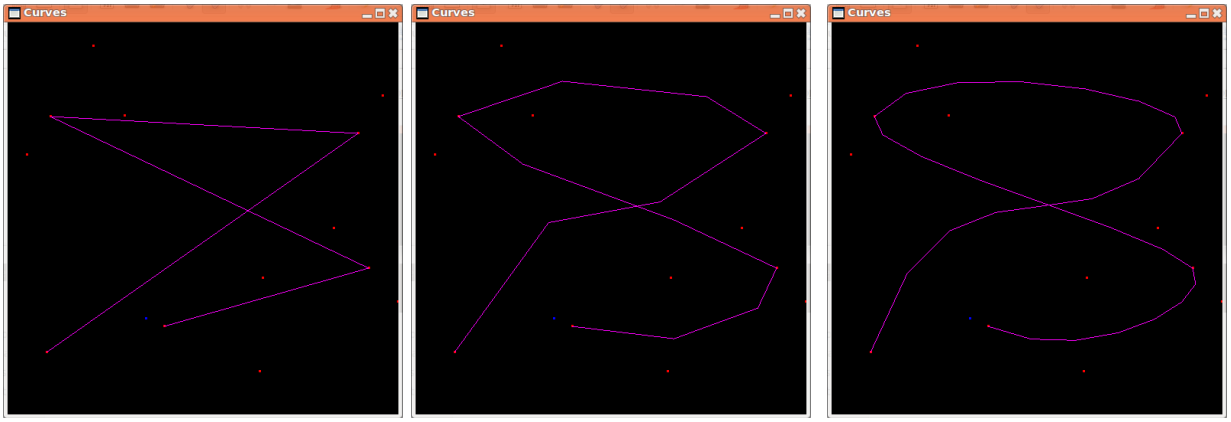
Subdivision

Curve Two

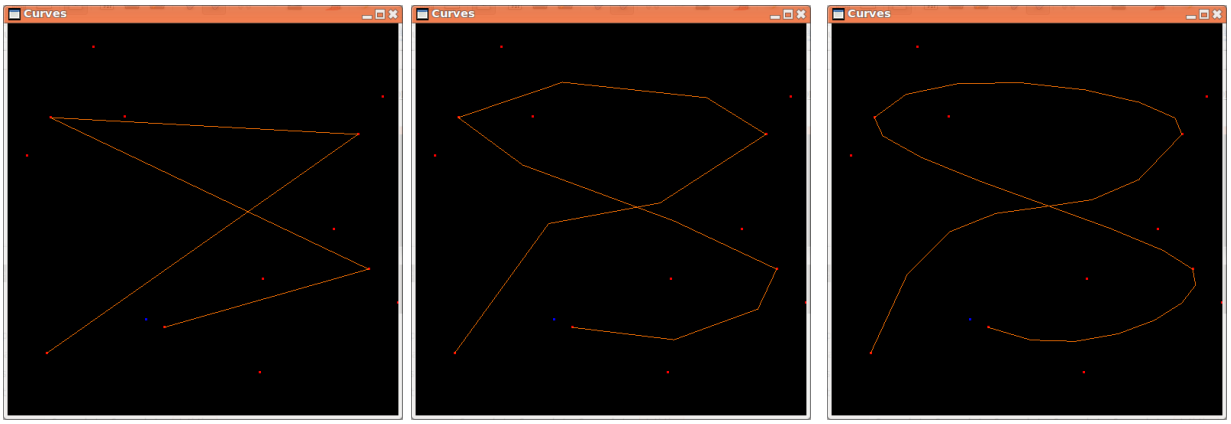
Blending Functions



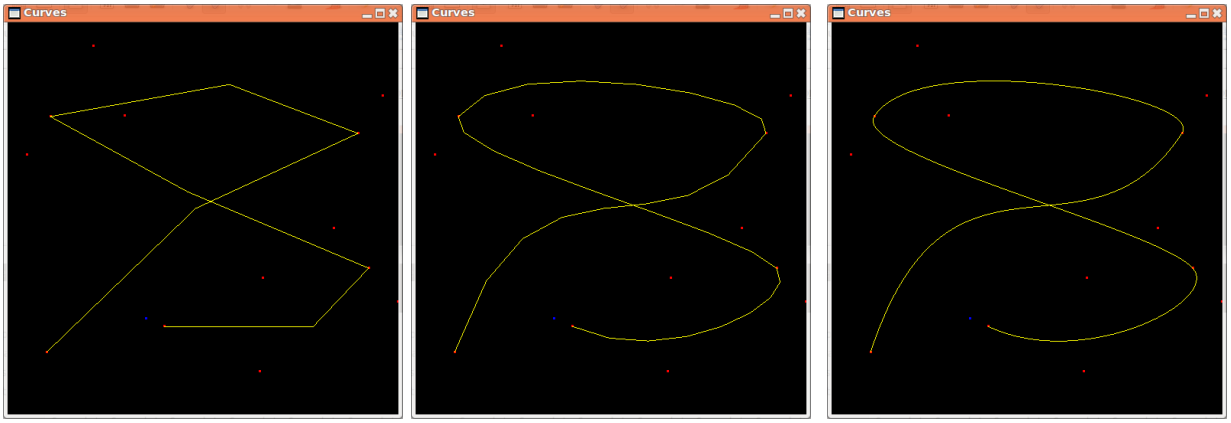
De Casteljau's Algorithm



OpenGL Evaluators



Subdivision



Duplicate points

When a control point is repeated, this reduces the number of degrees of freedom of the curve. This is not problematic in some cases (P1 = P2, P2 = P3, or P1=P3), as can be shown by renders of some such curves:

However, when the third and fourth control points of a curve are specified to be identical, this makes the collinearity constraint difficult. Since the third and fourth point are the same point, there is no single line to be drawn between them that can be used to constrain the next (fifth) point. One solution would be to fall back on using the second and third points for this constraint, but that might fail as well. In fact, there could be infinitely many points in the same location, and thus infinitely many failures. This implementation solves the problem by simply backtracking until two points are found that are not in the same position, and those two are used as the collinearity constraint.

However, because the user can place an arbitrary number of points in the same location, the user would be able to break the continuity constraint by placing multiple identical points in certain configurations. To re-enforce the collinearity constraint, this implementation requires that if there is a “black hole” (or sequence of consecutive points which are all in the same position) and the Bézier curve that “enters” the black hole is not the same curve as that which the user is currently constructing, then the next point must be constrained so that it is collinear with the two most recent points that differ in position. This allows the continuity constraint to be enforced despite these black holes.

Cusps

Suppose the user has just drawn a segment, and he/she is just about to draw a fifth control point. If the user places this point such that the vector from the fourth point to the fifth point is antiparallel to the vector from the third point to the fourth point, a cusp would be rendered. Thus, if the user places his/her cursor in such a manner, this implementation solves the problem by placing the new point a fixed distance in the opposite direction from the fourth point. This allows the user to click to draw a point, whereas an error message being displayed might cause the program to enter an impassable state, since the user might have drawn the curve such that the next point cannot be clicked.

Comparison of Methods

The obvious odd method out is the subdivision algorithm. This algorithm has the advantages of converging very quickly to an accurate representation of the curve as well as the ability to have the number of iterations dynamically adjusted to draw a pixel-perfect representation by measuring convex hull thickness. The subdivision method does have the disadvantage that medium numbers of iterations (approximately 40) are very costly and may halt the renderer if not limited. Thankfully, numbers of iterations this large are not normally necessary for visual display.

The remaining algorithms, while they produce similar visual representations, are somewhat different. Unlike the subdivision method, these algorithms' suitability for on-screen display of curves very much depends on the step size chosen. However, these algorithms would be ideal for when an object's parameter (such as position of a camera) is controlled by a Bézier curve. Subdivision would make little sense in this use case.

The OpenGL method has the drawback of requiring that the points already be laid out contiguously in an array of at least 3 components per point, for maximum efficiency of implementation. This approach is not (easily) an option if the goal is to pipe the results to a program other than OpenGL, since the evaluators only produce OpenGL calls.

Thus, for when OpenGL calls are not the desired result, we are left with the parametric (blending functions) method and the De Casteljau algorithm. These two approaches are almost identical for cubic Bézier curves. If implemented recursively, the De Casteljau algorithm has the advantage that it may produce points of any order of a Bézier curve. (This is also possible with the OpenGL method.) If the order is too high, of course, there could be a stack overflow.

If the blending functions method is implemented naively using matrix multiplication, many operations are spent

doing multiplications and additions with zero, which is a pointless performance hit. When the matrix multiply is precomputed as in this implementation, it is approximately 21 floating point operations per 2D cubic Bézier segment. The De Casteljau algorithm, in contrast, is approximately 48 floating point operations. However, if hardware is available which can perform linear interpolations as a single operation, comparable to addition and multiplication, then the De Casteljau algorithm shrinks to 6 floating point operations. Thus, the De Casteljau algorithm is probably preferable for GPU programs such as shaders.

Resources Used

Symbolic computations were facilitated by Wolfram|Alpha
http://www.wolframalpha.com/

MSDN OpenGL Reference
http://msdn.microsoft.com/en-us/library/windows/desktop/dd368808(v=vs.85).aspx

OpenGL Documentation
https://www.opengl.org/resources/libraries/

xkcd Color List
http://xkcd.com/color/rgb/