# Bezier Reconstruction Design Doc

## Executive summary

Curve retrieval today is vertex-based and at times relies on heuristics to determine when vertices should be included in the curve. Unfortunately, focusing on vertices creates ambiguity when dealing with intersecting chains, causing us to act conservatively and lose reconstruction info. On top of that, the heuristics being used are just best guesses, and can produce false-positives and false-negatives, leading to assertions and sub-optimal geometry generation.

I’d like to move to an edge-base curve retrieval algorithm, which will eliminate ambiguity at intersections and free us from any reliance on heuristics. The end result will be more predictable and better looking output. As an added bonus, speed of the new algorithm should be improved, since the heuristics used involve expensive Newton-Ralphson methods (although the cost of the actual scanner operation is still expected to dwarf the cost of the reconstruction).

## Gory details

Today, all of the scanner operations (like Union, Outline, etc…) operate on polygons. Whenever a geometry is encountered that contains Bezier segments, that geometry must first be tessellated (“flattened”) into a polygon before being passed to the scanner. Unfortunately, this means that the geometry suddenly becomes resolution dependent – zooming in on the resulting geometry will reveal the individual line segments used to approximate the Beziers. This is bad for customers like Text and XPS who use scanner operations in intermediate steps (before knowing the resolution of the device). To work around this, the scanner has an optional post-processing step that attempts to reconstruct pieces of the Beziers from the polygons. This process is (unsurprisingly) known as Bezier reconstruction.

Unfortunately, Bezier reconstruction as it exists today is rather fragile: There are times when it fails to reconstruct a Bezier and times when it reconstructs a Bezier that it should not. The document enumerates the shortcomings of the current Bezier reconstruction algorithm and proposes a modification that overcomes these deficiencies.

### Current algorithm

Bezier reconstruction works by reading metadata spewed by the Bezier tesselator on input. When acting on a Bezier, the tesselator marks each of the line segments generated as belonging to that Bezier. It also records how far along in the Bezier the endpoint of the Bezier occurs (the *Bezier parameter*):



Note that the first point in Bezier is not marked. This is because the first point is technically part of the *previous* segment.

During the scanner operation, if two chains intersect (exactly or inexactly) then all curve retrieval information (the Bezier parameter and the pointer to the Bezier object itelf) at the intersection point is removed. This removal is currently necessary since there are currently no well-defined semantics on Bezier retrieval at intersections.

The output of a scanner operation is a non-self-intersecting geometry that (in the case of Outline) has the same fill as the input geometry:



The Bezier reconstructor walks the vertices of this output, looking for one originating from a Bezier. As soon as it encounters such a vertex it starts reconstructing the corresponding Bezier. As more vertices from the Bezier are encountered, it keeps track of the lowest and highest Bezier parameter encountered. Note that since the scanner operation may permute or reverse certain sections of the geometry, the Bezier vertices may not be encountered in the order that they were input. Reconstruction stops as soon as it encounters a vertex not associated with the Bezier; however, it may extend the Bezier to include this last vertex if it determines that the vertex may have in fact come from the same Bezier. To decide this, the reconstructor calculates the point on the full Bezier closest to the vertex. If that point is within some small delta of the vertex and the Bezier parameter of the point is monotonic with the parameters of the other vertices, the Bezier is extended.

The heuristic particularly helps when a number of Beziers happen to meet at a single vertex:



Since the vertex in red isn’t marked with any Bezier information, we are forced to use heuristics to hook it up to each of the four Beziers.

Unfortunately, relying on the Bezier extension heuristic is somewhat risky, as it can produce false-positives as in the following example:



Here, the Bezier intersects itself (at Bezier parameters .2 and .8) and the intersection is exact (i.e. the points at .2 and .8 both appear in its polygonal approximation). The reconstruction algorithm encounters its first Bezier vertex at .1 (recall that in the current implementation the start point is never associated with the Bezier). The next vertex the algorithm encounters is at .2. Only, the Bezier information at this vertex has been cleared out, because it is an intersection point. Hence, the algorithm attempts a Bezier extension. That is, it attempts to calculate the Bezier parameter at the vertex. Now, in this case the retriever *should* return .2, but it might just as easily return .8, an incorrect extension.

Because the scanner throws out reconstruction information at intersections, the curve retrieval algorithm can also return false-negatives (i.e. fail to reconstruct a Bezier that it should have). Moreover, this can happen even with simple (non-self-intersecting) geometries.

### Proposed solution

Most of the problems encountered when performing Bezier reconstruction center on the fact that we’re currently treating vertices as the primary datatype. This, I believe, is fundamentally wrong. When flattening we’re not really creating a bunch of points that lie on the Bezier -- what we’re really doing is creating a bunch of line segments that closely approximate it. Hence, our focus should be on the segments (“edges” in scanner terminology) not the vertices.

The new Bezier reconstruction algorithm would store start and stop information about each edge:



Note that unlike the current algorithm, the new algorithm causes no loss of information in the flattening process. The entire Bezier, including the first segment can be reconstructed without heuristics. Admittedly, the same benefit could be achieved by storing potentially two Beziers references on each vertex (since the start point of one Bezier could be the endpoint of another). The real benefit of edge-centric reconstruction presents itself when we consider chain intersection:



Since information is stored on the edges and not the vertices, there is no ambiguity about multiple arising from the fact that the red vertex is shared by four different Beziers.

### Optional improvement

Even in the new model there are cases where Bezier reconstruction may be incomplete. For instance, when a Bezier and a line segment transversely intersect, a new vertex is constructed:



This new vertex, since it did not come from the flattener, does not have a Bezier parameter associated with it. Hence, the edges highlighted in green will not be reconstructed. One way to mitigate this would be to linearly interpolate the Bezier parameter from the two known parameters on either side. This value could be significantly off, but in the common case it would produce a reasonable result.

### Limitations

Even the proposed Bezier reconstruction algorithm still has its limits. For instance, if the flattening tolerance is set so high that the entire Bezier maps to a single edge and the Bezier is self-intersecting, then the scanner will not have an opportunity to remove said intersection. In fact, given a large enough tolerance, the following geometry would get completely optimized away:



The only sure-fire way to address this and similar issues is to build Bezier intersection support directly into the scanner. We are considering this for future versions, but this would be a major undertaking.

### Implementation details

The scanner doesn’t actually have an object model for edges – chains are just linked-lists of vertices. Hence, edge data will actually be stored on vertices. A simple way of doing this, and one that already has some precedent (see, for instance, Attributes), is to store the edge info on the vertex terminating the edge. Here the “terminating” vertex is the one lower in YX-lexicographic order. It is *not* necessarily the one with the greater Bezier parameter. Indeed, care will have to be taken for Beziers that are not YX-monotonic, as the “terminating” vertex will swap from preceding to following:



Observant readers may wonder what happens when the above Bezier is flipped over:



Here it seems we have an issue: We have two pieces of Bezier information and one vertex to store it on. Fortunately, this is not the case, as the bottom-most (gray) point is occupied by two vertices: one for each chain.

During reconstruction, when two adjacent edges are encountered that come from the same Bezier and the endparameter of one is the startparameter of the other, then the two edges can be coalesced into a single Bezier segment. Since no transforms of any kind will be applied to the Bezier parameters after construction, I don’t see any issue with a simple floating-point equality test (i.e. there should be no need for a “fuzzy” compare).

### Performance considerations

The amount of computation required by the new algorithm should be substantially less than the old algorithm, since no numerical calculations are performed. In particular, the NewtonRalphson iterations necessary for the curve extension heuristic are completely removed.

The only negative effect is a slightly larger working set, since we need to store the beginning *and* end parameters for each edge. This should be completely dwarfed by the size of the rest of the geometry data.