Fun with Curvature-Driven Textured Lines

David A. Hostetler
Graphics Algorithms and 3D Technologies (G3D)
Intel Architecture Labs (IAL)

Abstract

The idea behind curvature-driven texured lines is to emulate curvature in the rendered outline of a model by generating a quadrilateral for each individual silhouette edge, positioning the quad over the edge and texturing it with an image that represents a stroke with some degree of curvature. The appropriate degree of curvature would be determined by the perceived angles that the edge forms with any other edges connected to it, when the edge segments are projected onto the projection plane.

1. Introduction

Objects in life tend to be curvy. In computer graphics, we basically imitate objects using lines. Given enough lines, it works pretty well; objects assume curvature. The curviness in a model is dependent upon the number of polygons allocated to that model. Less polygons means less curve. A tried and true method of fooling the observer into thinking that there is more geometric detail than actually exists is to use texturing. An appropriately drawn texture can be projected onto a single polygon and provide the illusion of a very complex surface. Unfortunately, one of the strongest visual indicators of a model's complexity is its perceived outline, or silhouette. Human vision is easy to fool, as it is quick and eager to assume depth and motion. Unfortunately, it is equally quick to identify discontinuities and patterns. The traditional use of textures (i.e. texturing the faces of the polygons in a model) has no impact on a model's outline, and the discontinuous nature of faceted silhouettes is a serious impediment to the visual fidelity of digitally rendered images. The concept of curvature-driven textured (CDT) lines is aimed directly at improving the fidelity of rendered outlines of low polygon models.

Fidelity might seem a contradictory goal of nonphotorealistic rendering (NPR). It is important to note that while, as the name implies, the intent is not to emulate photo-realism, there is still a realism that NPR is pursuing. The realism sought after is that of artistic images, created via some two-dimensional medium.

2. The Intent

The basic tenet of CDT lines is that silhouettes (and other important edges) will look better if we assume that they represent some curvature and we apply texture images over them that emphasize or accentuate that curvature. **Figure 1** shows two sets of connected lines. The bottom set has had curvature-driven textures applied to it. There are two basic steps to the process.

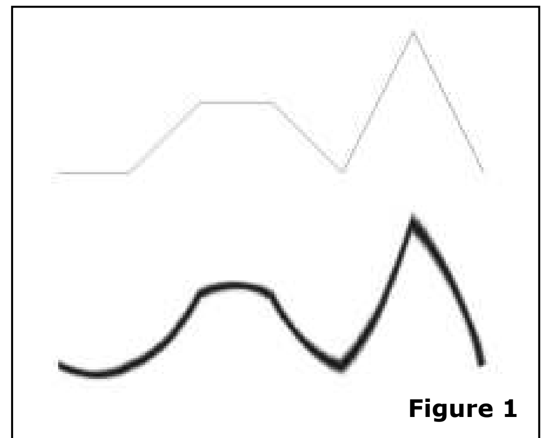

Figure 1

First, we must generate some surface area upon which to apply the textures. Lines have no width and cannot be textured (for the sake of argument), so we must first create polygons of appropriate shape, size, and orientation. It is worth noting that this step is fundamental to the concept of textured lines in general, curvature or no curvature. Curvature-driven textured lines are really just a subset of textured lines, and if we can resolve many of the CDT line issues, we will have laid the framework for doing general textured lines. If we have a robust procedure for texturing lines, we open the door for many stylistic inking techniques, which this discussion does not address.

At any rate, the second basic step in doing CDT lines is to determine the appropriate curvature that should be represented by the texture. For a given edge segment, this is accomplished by first finding edge segments that have common endpoints with the edge in question. Next, this small set of connected segments is projected into screen space (clip coordinates if you prefer). The angle between these projected segments determines the curvature of that portion of the silhouette (curvature relative to the current viewpoint), and thus determines the particular texture that should be applied to the edge in question.

3. Generating Quads

3.1 Orientation

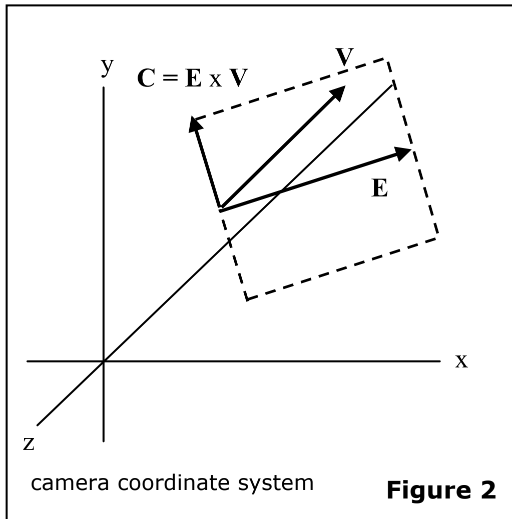

Figure 2

First, let's discuss how to go about creating the quadrilaterals that will get textured for each edge. The orientation of the quad is the first thing to consider. The quad needs to have its length parallel to the edge, and thus its width perpendicular to the edge. If the width is perpendicular to the edge, it is also parallel to the edge's normal. Note that any edge has an infinite number of normals, all coplanar. The trick is to use the right one. The right one is the normal that is also perpendicular to the view vector, i.e. it has no Z component (in camera space). This normal is obtained by taking the cross product of the view vector and the edge vector. In **Figure 2**, the edge vector **E** is crossed with the view vector **V** to obtain vector **C**.

If **V** is defined to be along the negative Z-axis in camera space, then the resulting vector **C** will be in the XY plane in camera space. This vector represents the orientation of the quadrilateral we're creating. We use this orientation because it maximizes the visibility of the quad, by ensuring that any perspective foreshortening of the width is due solely to the depth slope of the edge itself. Ideally, we would eliminate foreshortening entirely by doing a parallel projection of the quad and rendering it in 2D, but doing so prevents correct visibility testing with the rest of the polygons in the scene (model or otherwise). This issue is discussed in more detail later.

3.2 Shape and Size

Having determined the orientation of the quad, the next step is to determine its shape and size. The size is specified as a fixed width value that applies to all of the quads we will generate for a given model. The shape, on the other hand, is a much more complicated part of the procedure.

3.2.1 Rectangles

The quick and dirty approach for the shape is to simply use a rectangle. The length of the rectangle is the length of the edge segment, the width is whatever value is specified as line width, and the orientation used is the one described above. In short, just put a box on the line. This works, but its shortcomings are glaring. **Figure 3** shows an example set of line segments with quads generated via this technique.

Using this technique, we get the surface area we want, allowing for the application of textures that emphasize curvature, but there are prohibitive problems. First, notice that any nonzero angle causes two things to happen to adjacent quads. On one side, there is overlap, and on the other there is a crack. The overlap can cause Z-fighting, and the crack produces discontinuities in the textured stroke. The Z-fighting that actually occurs is proportional to the extent to which the adjacent quads are

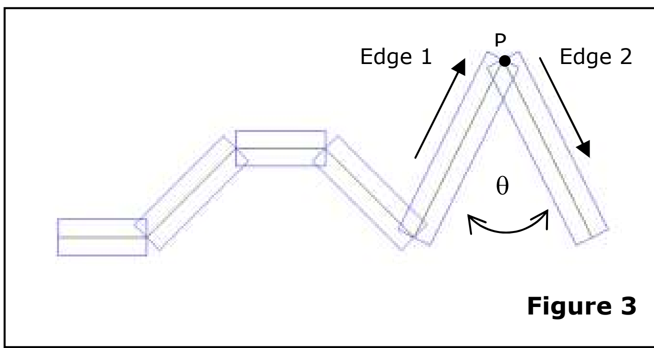

Figure 3

coplanar. In **Figure 3**, the line segments are all coplanar, so the Z-fighting would be severe in the overlapped regions. The overlap is obviously also proportional to the width of the quads and inversely proportional to the angle between the connected line segments.

At this point, it is worth introducing the issue of alpha blending. If you'll refer back to **Figure 1**, you'll notice that the textured strokes blend into the white background. The quadrilaterals used in that image are the same width as those shown in **Figure 3**, and any portion that isn't black is actually a part of the texture which has an alpha value that allows the background to show through.

So, one approach to the problems of overlapping and cracking would be to simply draw the textures so that the stroke part never reached the end of the texture. Thus, the overlapped and cracked areas would be completely transparent regions and you wouldn't be able to see them. However, that would only work for angles that weren't too acute. Notice in **Figure 3** how the last angle on the right (θ) is so acute that the overlapping quads actually intersect portions of the line segments themselves. To rely on transparency to completely avoid any visual artifacts, the stroke portion of the texture would have to terminate well before the end of the quad. To do so means sacrificing continuity in the textured curve. In fact, for short segments, the end result might begin to look more like a stipple than a curved outline. The only way to maintain continuity in the textures is to avoid the overlap and the crack altogether. The only way to do that is to seam the rectangles.

3.2.2 Seamed Rectangles

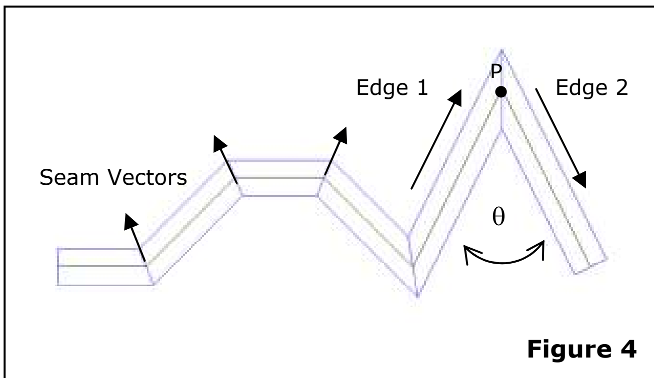

Figure 4

The next logical evolution to the quad-generating process is to seam the vertices. Comparing **Figure 4** with **Figure 3** will convey the idea. Instead of using the scaled normal for a given segment to generate the 4 vertices for the quad, it is necessary to generate 2 seam vectors. These seam vectors evenly split the angle between two segments, and are obtained simply by adding the normals of each of the two line segments.

It's simple, but there's a catch. How far away from the line should those vertices be? I.e., how long should the seam vector be? It shouldn't simply be the specified width of the quads. If we used that value uniformly, then the only quads that would actually have that width would be those for two segments that were essentially colinear ($\theta=180°$).

If the seam vector has a fixed size, then as the angle between two segments becomes more and more acute (as $\theta \rightarrow 0$), the effective width of those segments' quads decreases. To emphasize the relationship, consider two edges that are almost colinear, but not quite. The seam vector, which splits the difference between them, will therefore also be almost colinear with the two edges. The seamed vertices for the quad are obtained by positioning the seam vector in both its positive and negative direction at the shared endpoint (**P** in **Figures 3**,**4**). The distance of these vertices from

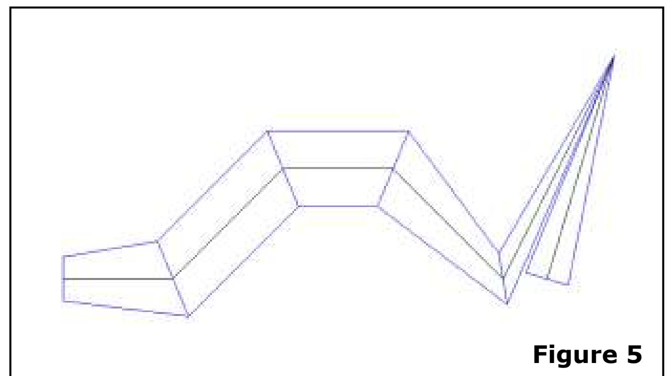
Figure 5

the edges is proportional to the sine of the angle between the edges and the seam vector. Thus as that angle approaches zero, so does the distance of the vertices from the edges and thus the width of the quad. To demonstrate, this policy was used to generate the quads shown in **Figure 5**.

So, in order to help maintain a consistent width to all of the generated quads, we need to use a different metric for the length of the seam vector. Instead of using a fixed value for the magnitude of the seam vector, we can allow it to scale up or down to accommodate the angle of the two connected segments. For colinear segments (θ=180°), the magnitude would indeed start out as the specified width. As the angle varies (180°<θ<0°), the magnitude would need to increase. This is demonstrated in **Figure 6**. Notice on the right most two segments how instead of squishing the quads, the vertices have moved far enough away from the shared endpoint to allow the quads to maintain their width.

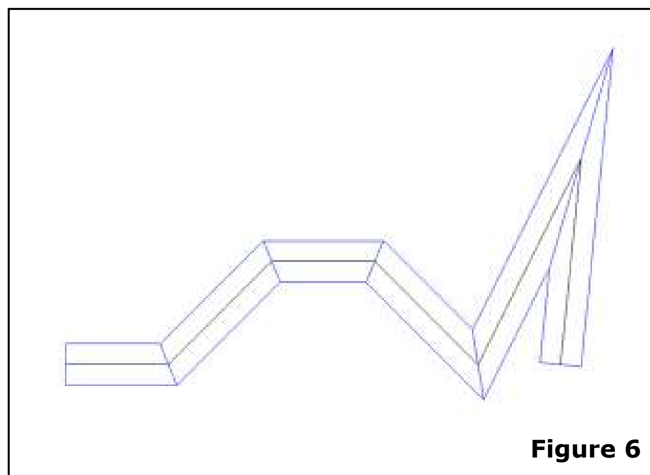

Figure 6

What should be immediately apparent in **Figure 6** is the problem that arises for very acute angles. For small angles, the two seamed vertices would be pushed an extreme distance from the endpoint, resulting in an extreme skew to the two quads. In fact, taken to the limits, as θ approaches zero the two vertices would approach an infinite distance from the shared endpoint. Extremely acute angles tend to be fairly common, since we are measuring the angles between projected line segments. Thus, this method for determing the length of the seam vector is not completely acceptable either.

3.2.3 Elbows

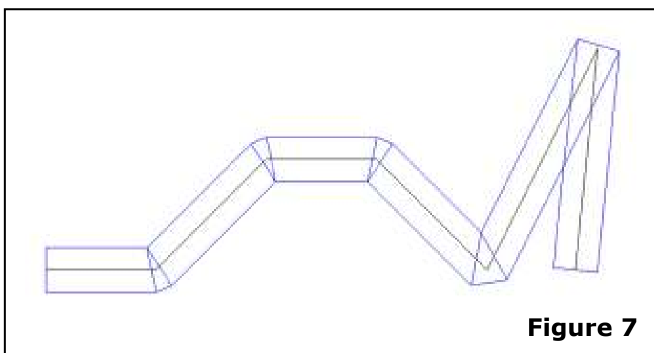

Figure 7

One possible solution to the infinite skewing problem described above is to introduce slightly more complex geometry. Instead of using two quads for a pair of segments, we will try using two quads and a triangle. The triangle will function as an *elbow* for the joint between the two segments. **Figure 7** illustrates the new approach. This looks pretty good. Unfortunately, even this approach doesn't completely resolve the problem of highly acute angles, and it definitely complicates the texturing process.

To deal graciously with highly acute angles, a quick fix is to simply establish a threshold angle, and to abort any attempt to seam two adjacent quads if their angle is less than this threshold. Thus, for angles less than the threshold, the quads would revert back to the original rectangle method for that particular endpoint. This is demonstrated in **Figure 8**. Further complicating the issue is the fact that a single threshold angle isn't appropriate for all cases. The shorter an edge segment is, the smaller the threshold angle would need to be, to prevent orphaning vertices of the quad. This is what occurs in **Figure 9**, where the angle is actually greater than the threshold value, so an attempt is made to seam, but the line segment is too short, relative to the

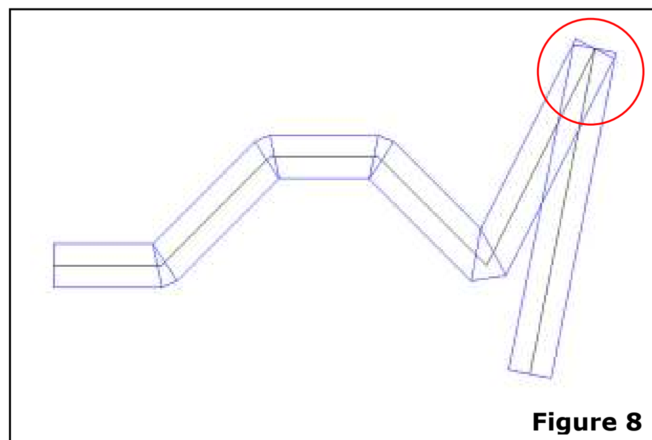

Figure 8

width of the quads. This problem can probably be avoided by calculating a specific threshold angle for each edge segment. However, even doing that would not completely resolve the problem of acute angles. The reason is that we are currently not making any of the geometry information persistent.

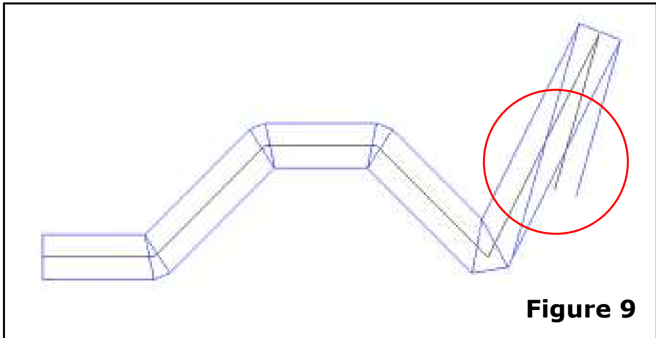
Figure 9

The shape of each quad depends only upon the angles formed at each end of the edge (if there are connecting segments). In other words, each edge finds its connected segments, determines the two angles, and generates four vertices for its quad. None of this information is persistent, resulting in very low memory requirments for the procedure. In order for two quads to seam, each edge must have come to the same conclusion regarding where those two vertices should go. The disadvantage is that there are circumstances where an edge needs to know about an angle which it doesn't specifically help form, i.e. an angle between one of its neighbor segments and the neighbor of its neighbor segment. This problem is not dealt with currently.

3.2.4 Sharing Endpoints

Another problem that has been swept under the rug is when more than two edge segments share an endpoint. For now, we are ignoring this situation, and simply use the first connected edge segment that we find as the neighbor. Consider three lines (**A**, **B**, **C**) sharing an endpoint. When processing line **A**, we may encounter line **B** first and use it and the resulting angle between **A** and **B** for the quad associated with line **A**. Later on, we process line **B** and encounter line **C** first instead of **A**. Thus, we use line **C** and the angle between **B** and **C** to form **B**'s quad. Finally, when processing **C**, we can wind up treating either **A** or **B** as the neighboring segment. Obviously, the likelihood of the resulting textured geometry looking appropriate, let alone seaming, is extremely low. This situation is shown in **Figure 10**. This is another example of where a more complex, persistant structure would allow for correct steps to be taken under special circumstances.

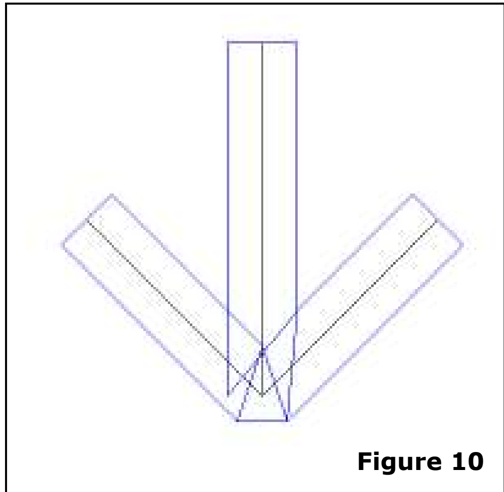
Figure 10

4. The Occlusion Problem

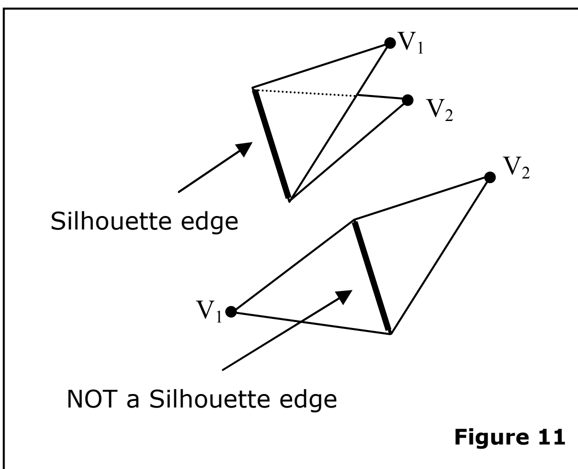

Figure 11

All of the previous effort dealt with correctly generating quadrilaterals for each edge, so that when they are finally textured, the desired effect is obtained. Unfortunately, a lot of that effort is moot when the process is actually applied to a 3D model. Remember that the edges with which we are concerned are the outline or silhouette edges (and occasionally other importanted edges). As you'll recall, a silhouette edge is any edge shared by a pair of polygons (triangles) of which one triangle is front-facing and the other is back-facing (see **Figure 11**). The problem is this: for two faces forming an edge, when a quadrilateral is created and positioned over the edge in 3D space, the quadrilateral is either 50% occluded, fully occluded, or not occluded at all.

To visualize why this happens, remember that the quad has a fixed orientation, intended to minimize the perspective foreshortening of the quad's width. The two faces that form the edge can have an arbitrary orientation about that edge; each face can have any degree of rotation about that edge. The determining factor is the third vertex of each triangle in the pair (vertices V_1 and V_2 in **Figure 11**). If both V_1 and V_2 are deeper in the scene than the edge, then the quad that is positioned over the edge will be completely visible. If V_1 and V_2 are both closer to the viewer than the edge, then the quad will be completely occluded – half of it occluded by each face. Lastly, if one vertex is closer to the viewer than the edge and the other vertex is further away, then the quad will be 50% occluded. This last case is what always occurs for silhouette edges.

As an aside, it is interesting to note that the rendering engine is already drawing lines as quads, oriented towards the viewer. The quads are just normally a few pixels wide. The engine can be asked to render lines of greater width, in which case the lines begin to assume the properties of filled polygons. The occlusion problem manifests when you attempt to draw wide lines as polygons on a model. With OpenGL, this problem is not particularly apparent, but not because the engine is avoiding or solving it. It is less noticeable for two reasons: first, the scale of the lines is typically very small relative to the model and secondly, the lines are filled with a single uniform color, as opposed to being textured. The point is that most of the major problems we have encountered when generating textured geometry for lines are not problems that have been solved, or that only occur when texturing lines, rather they are unresolved problems that have been ignored because they tend not to be noticeable.

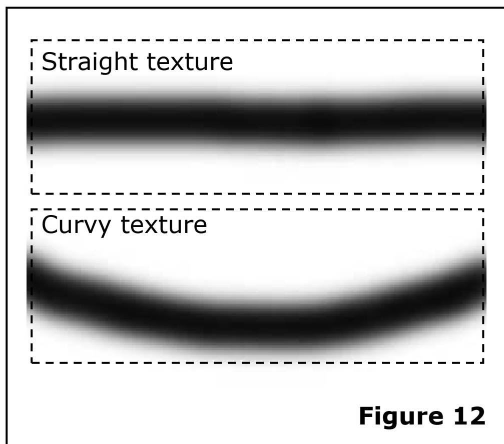

Figure 12

If we were just filling our quads with a color, as is done with the line primitive of the graphics engine, instead of using the quad to accommodate a texture with a curved stroke, then the occlusion problem would likely still be annoying, but not prohibitively so. When filling the quads with a color, the visual effect of the occlusion problem is that some lines look thicker than others. The thick lines are the ones which have quads that are not occluded. When using the textures, however, the occlusion has the effect of making it appear as though some of the edges simply aren't being drawn. Observe the texture examples in **Figure 12**. The important texture is the curvy texture, as it is the one that will provide the illusion of curvature. As you can see, if the bottom half of that texture happens to get occluded, virtually none of the stroke is seen. This is demonstrated in **Figure 13**. The gaps in the outline of the textured image on the right are instances

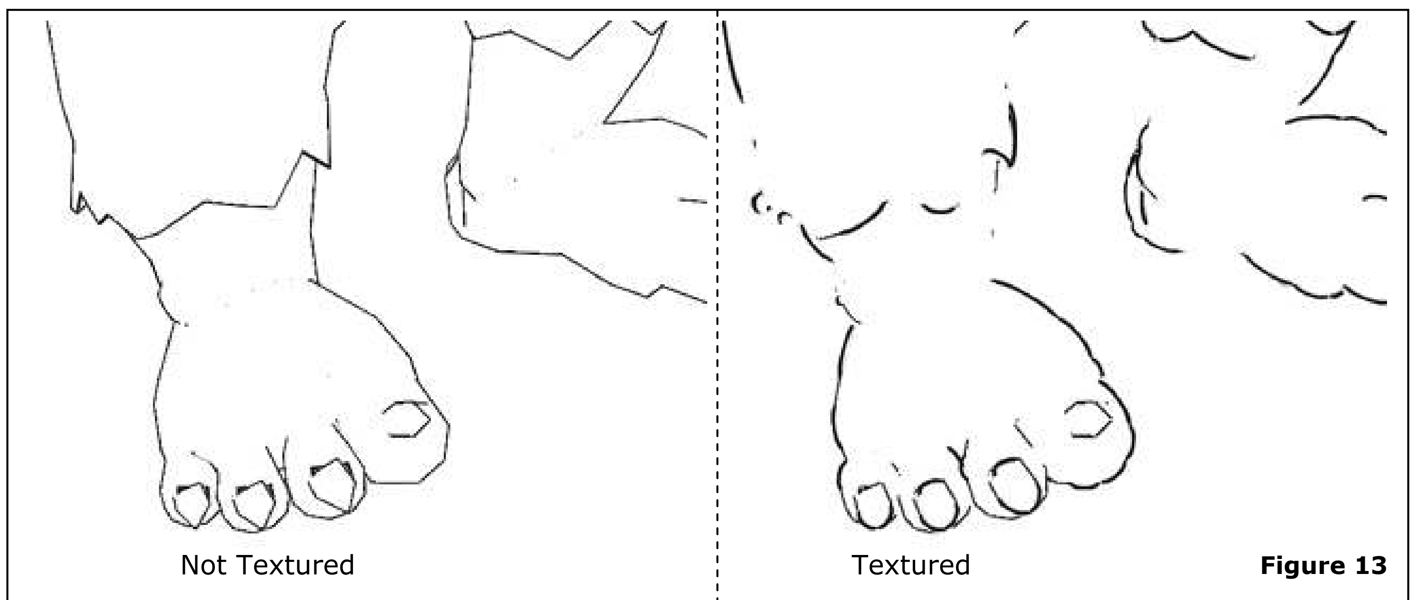

Figure 13

where the stroke portion of the curvy texture happens to be occluded by the triangles of the model itself. **Figure 13** is a good example both of how dramatic the CDT line effect can be and also of how many issues exist which prevent it from being broadly applicable.

The occlusion problem is a major one. We have not identified any feasible solutions to it at this time. One possible solution might be to position quads wholly on one side of an edge or the other, as opposed to centering them over the edge. In order to do this, we would need to know which side of an edge represented the 'outside' of the model and which the 'inside'. There is certainly enough spatial information inherant in the vertices of the neighboring faces to do this, but the calculations would add to the already steep requirements for each edge. Still, it's a possible solution.

There is a temptation to solve the problem by altering the test that is used for depth values, when rendering the textured quads. It is true that if you disable depth testing when rendering the quads, there is no chance of occlusion. Unfortunately, disabling the depth test prevents correct clipping of the edges caused by legitimate occlusion. This is not a viable solution. Another temptation is to use the polygon offset feature of OpenGL, which introduces small amounts of depth offset when rendering polygons. Normally, this feature is used to eliminate Z-fighting when drawing decals or polygons known to be coplanar. Unfortunately, it takes a large offset to accommodate the significant width of the quads that is necessary to get much use out of a curvy texture. Such a large offset introduces other visibility errors. The use of polygon offset has not proven to be a viable solution, although we have not significantly experimented with it.

5. Alpha Gaps and Dimples

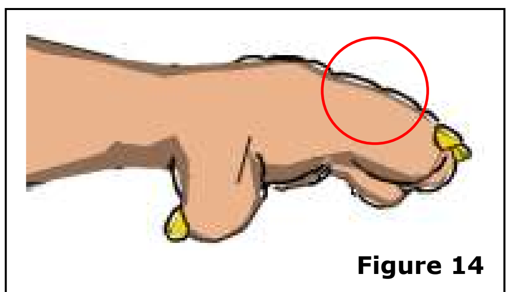

Figure 14

In all of the images thus far, we have only drawn the lines or the textures, and have not attempted to draw and fill the faces of a model. In other words, we have been drawing outlines and only outlines. When we actually render the faces of the model in addition to the CDT lines, we notice another significant issue – that of alpha gaps. Notice in **Figure 14** how there are gaps between the filled polygons of the model and the stroke portion of the line textures. This is another issue for which there is no immediately intuitive solution. Generating textures at runtime is a possibility, but that is not conducive to the use of artistic, hand-authored textures, not to mention the fact that it would be very slow. A more adaptive curvature would help, instead of using a single curve texture. This would also probably alleviate the dimpling that occurs when uniformly applying the single curve texture to each edge segment, regardless of segment length. This dimpling effect can be seen on the top of the finger in **Figure 14**, as well as on the big toe in **Figure 13**. We wouldn't necessary need a large set of textures, and might even be able to resolve the problem using a single texture by being more creative when generating texture coordinates in order to flatten out or exaggerate the curved stroke. This has not been tested, however.

6. Future Work

Most of the really troublesome obstacles to the CDT line technique stem from the fact that we are attempting to reproduce a two-dimensional effect within a scene that is in turn attempting to reproduce three-dimensional effects using three-dimensional data. We are introducing geometry into a 3D scene that then gets processed to create a 2D image. It seems that perhaps we are attempting to apply the algorithm in the wrong space. Theoretically, CDT lines would work best if we simply had a representation of only the *visible* portions of the silhouette edges, as projected onto the projection plane. In other words, allow the 3D rendering to take care of the edge detection and occlusion, and apply a CDT line technique as a post 3D process. In this regard, the stencil buffer capabilities of OpenGL stand out as potentially being tremendously useful. In fact, the ability to process frame buffer contents feels like a very powerful means of applying NPR techniques in general. Unfortunately,

working with the frame buffer and stencil buffer also appears to place a heavy burden on performance. Whether or not this approach is currently feasible for real-time 3D rendering remains to be seen.

7. Summary

Curvature-driven textured lines are capable of producing very compelling effects. The technique is decidely better suited for models with low to moderate polygon counts. At higher counts, models tend to already have significant amounts of curvature, even in perceived outlines. The amount of work necessary to do CDT lines scales with the number of detected edges for any given frame, and so for high count models, the price is higher and the effect definitely has diminishing returns. Moreso, at higher counts, the edge segments tend to get shorter, which exacerbates many of the visual artifacts that currently plague CDT lines.

There are some significant obstacles preventing CDT lines from being a truly high calibre NPR effect. Namely, these are texture occlusion, alpha gaps, and dimpling. Also, it does not feel as though we have found a truly robust method for generating the necessary geometry for the textures. Ultimately, applying this technique within three-dimensional space may not be the right approach. Exploiting the framebuffer and the stencil buffer could prove to be a much more appropriate means of applying not only CDT lines but also many other stylistic NPR techniques.