Chapter 2

Improved Alpha-Tested Magnification for Vector Textures and Special Effects

Chris Green[2]

(a) 64x64 texture, alpha-blended

(b) 64x64 texture, alpha tested

(c) 64x64 texture using our technique

Figure 1. *Vector art encoded in a 64x64 texture using (a) simple bilinear filtering (b) alpha testing and (c) our distance field technique*

2.1 Abstract

A simple and efficient method is presented which allows improved rendering of glyphs composed of curved and linear elements. A distance field is generated from a high resolution image, and then stored into a channel of a lower-resolution texture. In the simplest case, this texture can then be rendered simply by using the alpha-testing and alpha-thresholding feature of modern GPUs, without a custom shader. This allows the technique to be used on even the lowest-end 3D graphics hardware.

[2] email: cgreen@valvesoftware.com

With the use of programmable shading, the technique is extended to perform various special effect renderings, including soft edges, outlining, drop shadows, multi-colored images, and sharp corners.

2.2 Introduction

For high quality real-time 3D rendering, it is critical that the limited amount of memory available for the storage of texture maps be used efficiently. In interactive applications such as computer games, the user is often able to view texture mapped objects at a high level of magnification, requiring that texture maps be stored at a high resolution so as to not become unpleasantly blurry, as shown in Figure 1a, when viewed from such perspectives.

When the texture maps are used to represent "line-art" images, such as text, signs and UI elements, this can require the use of very high resolution texture maps in order to look acceptable, particularly at high resolutions.

In addition to text and UI elements, this problem is also common in alpha-tested image-based impostors for complicated objects such as foliage. When textures with alpha channels derived from coverage are magnified with hardware bilinear filtering, unpleasant "wiggles" as seen in Figure 1b appear because the coverage function is not linear.

In this chapter, we present a simple method to generate and render alpha-tested texture maps in order to minimize the artifacts that arise when such textures are heavily magnified. We will demonstrate several usage scenarios in a computer game context, both for 3D renderings and also user-interface elements. Our technique is capable of generating high quality vector art renderings as shown in Figure 1c.

2.3 Related Work

Many techniques have been developed to accurately render vector graphics using texture-mapping graphics hardware. In [FPR+00], distance fields were used to represent both 2-dimensional glyphs and 3-dimensional solid geometry. Quadtrees and octrees were used to adaptively control the resolution of the distance field based upon local variations. While GPU rendering of such objects was not discussed, recent advances in the generality of GPU programming models would allow this method to be implemented using DirectX10 [Blythe06].

In [Sen04] and [TC04], texture maps are augmented with additional data to control interpolation between texel samples so as to add sharp edges in a controllable fashion. Both line-art images and photographic textures containing hard edges were rendered directly on the GPU using their representation. In [LB05], implicit cubic curves were used to model the boundaries of glyphs, with the GPU used to render vector textures with smooth resolution-independent curves. In [QMK06], a distance based representation is

used, with a precomputed set of “features” influencing each Voronoi region. Given these features, a pixel shader is used to analytically compute exact distance values.

Alpha-testing, in which the alpha value output from the pixel shader is thresholded so as to yield a binary on/off result, is widely used in games to provide sharp edges in reconstructed textures. Unfortunately, because the images that are generally used as sources for this contain “coverage” information which is not properly reconstructed at the sub-texel level by bilinear interpolation, unpleasant artifacts

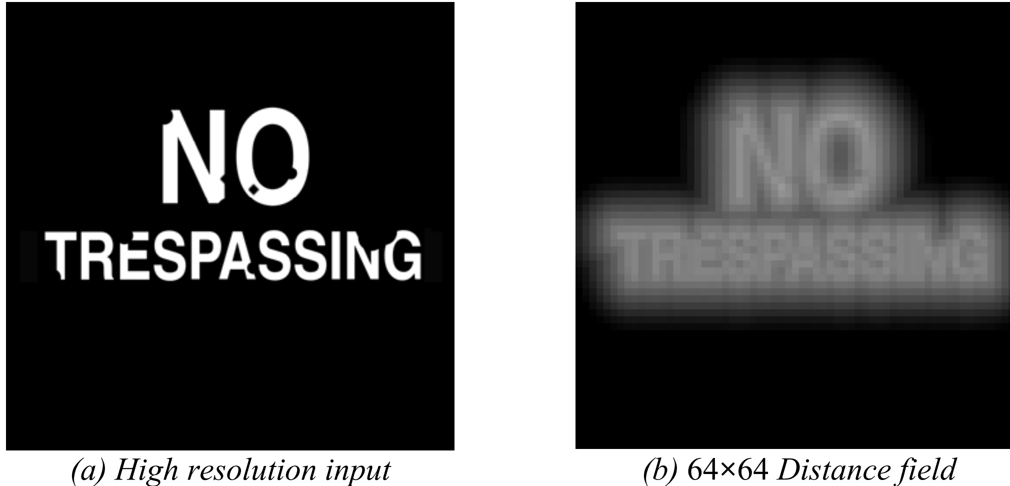
2.4 Representation and Generation

In order to overcome the artifacts of simple alpha testing while keeping storage increase to a minimum, we sought a method for displaying vector textures that could

- Work on all levels of graphics hardware, including systems lacking programmable shading
- Run as fast as, or nearly as fast as, standard texture mapping
- Take advantage of the bilinear interpolation present in all modern GPUs
- Function inside of a pre-existing complex shader system [MMG06] with few changes
- Add at most a few instructions to the pixel shader so that vector textures can be used in existing shaders without overflowing instruction limits
- Not require that input images be provided in a vector form
- Use existing low-precision 8-bit texture formats
- Be used as a direct replacement for alpha-tested impostor images

We chose to implement a simple uniformly-sampled signed distance field representation, with the distance function stored in an 8-bit channel. By doing so, we are able to take advantage of the native bilinear texture interpolation which is present in all modern GPUs in order to accurately reconstruct the distance between a sub-texel and a piecewise-linear approximation of the true high resolution image. While this representation is limited in terms of the topology of features which can be represented compared to other approaches, we felt that its high performance, simplicity, and ease of integration with our existing rendering system made it the right choice for Valve’s *Source* engine.

While it is possible to generate the appropriate distance data from vector-based representations of the underlying art, we choose instead to generate the low-resolution distance fields from high resolution source images. In a typical case, a 4096×4096 image will be used to generate a distance field texture with a resolution as low as 64×64, as shown in Figure 2.



(a) High resolution input *(b)* 64×64 *Distance field*

Figure 2. *(a) A high resolution (*4096×4096*) binary input is used to compute (b) a low resolution (*64×64*) distance field*

At texture-generation time, the generator takes as its input a high resolution binary texture where each texel is classified as either "in" or "out." The user specifies a target resolution, and also a "spread factor," which controls the range which is used to map the signed distance into the range of 0 to 1 for storage in an 8-bit texture channel. The spread factor also controls the domain of effect for such special rendering attributes as drop-shadows and outlines, which will be discussed in Section 2.5.2.

For each output texel, the distance field generator determines whether the corresponding pixel in the high resolution image is "in" or "out." Additionally, the generator computes 2D distance (in texels) to the nearest texel of the opposite state. This is done by examining the local neighborhood around a given texel. While there are more efficient and complex algorithms to compute the signed distance field than our simple "brute-force" search, because of the limited distance range which may be stored in an 8-bit alpha channel, only a small neighborhood must be searched. The execution time for this simple brute-force method is negligible.

Once this signed distance has been calculated, we map it into the range 0..1, with 0 representing the maximum possible negative distance and 1.0 representing the maximum possible positive distance. A texel value of 0.5 represents the exact position of the edge and, hence, 0.5 is generally used for the alpha threshold value.

2.5 Rendering

In the simplest case, the resulting distance field textures can be used as-is in any context where geometry is being rendered with alpha-testing. Under magnification, this will produce an image with high-resolution (albeit, aliased) linear edges, free of the false curved contours (see Figure 1b) common with alpha-tested textures generated by storing and filtering coverage rather than a distance field. With a distance field representation, we merely have to set the alpha test threshold to 0.5. Since it is fairly common to use alpha testing rather than alpha blending for certain classes of primitives

***Figure 3**. 128×128 "No trespassing" distance image applied to a surface in Team Fortress 2*

in order to avoid the costly sorting step, this technique can provide an immediate visual improvement with no performance penalty.

In Figure 3, we demonstrate a 128×128 distance field representation of a "No Trespassing" sign rendered as a decal over the surface of a wall in the game *Team Fortress 2*. The apparent resolution of this decal is incredibly high in world space and holds up well under any level of magnification that it will ever undergo in the game. We will refer to this particular decal example in the next section as we discuss other enhancements available to us when representing our vector art in this manner.

2.5.1 Antialiasing

If alpha-blending is practical for a given application, the same distance field representation can be used to generate higher quality renderings than mere alpha testing, at the expense of requiring custom fragment shaders.

Figure 4 demonstrates a simple way to soften up the harsh aliased pixel edges. Two distance thresholds, $Dist_{min}$ and $Dist_{max}$, are defined and the shader maps the distance field value between these two values using the `smoothstep()` function. On graphics hardware which supports per-pixel screen-space derivatives, the derivatives of the distance field's texture coordinates can be used to vary the width of the soft region in order to properly anti-alias the edges of the vector art [QMK06]. When the texture is minified, widening of the soft region can be used to reduce aliasing artifacts. Additionally, when rendering alpha-tested foliage, the alpha threshold can be increased with distance, so that the foliage gradually disappears as it becomes farther away to avoid LOD popping.

Figure 4. *Zoom of 256×256 ``No Trespassing'' sign with hard (left) and softened edges (right)*

2.5.2 Enhanced Rendering

In addition to providing crisp high resolution anti-aliased vector art using raster hardware, we can apply additional manipulations using the distance field to achieve other effects such as outlining, glows and drop shadows. Of course, since all of these operations are functions of the distance field, they can be dynamically controlled with shader parameters.

2.5.2.1 Outlining

By changing the color of all texels which are between two user-specified distance values, a simple texture-space outlining can be applied by the pixel shader as shown in our decal example in Figure 5. The outline produced will have crisp high quality edges when magnified and, of course, the color and width of the outline can be varied dynamically merely by changing pixel shader constants.

2.5.2.2 Glows

When the alpha value is between the threshold value of 0.5 and 0, the smoothstep function can be used to substitute a “halo” whose color value comes from a pixel shader constant as shown in Figure 6. The dynamic nature of this effect is particularly powerful in a game, as designers may want to draw attention to a particular piece of vector art in the game world based on some game state by animating the glow parameters (blinking a health meter, emphasizing an exit sign etc).

Figure 5*. Outline added by pixel shader*

Figure 6*. Scary flashing "Outer glow" added by pixel shader*

Figure *7. Soft drop-shadow added by the pixel shader. The direction, size, opacity, and color of the shadow are dynamically controllable.*

2.5.2.3 Drop Shadows

In addition to effects which are simple functions of a single distance, we can use a second lookup into the distance field with a texture coordinate offset to produce drop shadows or other similar effects as shown in Figure 6. In addition to these simple 2D effects, there are surely other ways to reinterpret the distance field to give designers even more options.

2.5.3 Sharp Corners

As in all of the preceding examples, encoding edges using a single signed distance "rounds off" corners as the resolution of the distance field decreases [QMK06]. For example, the hard corners of the letter G in Figure 2a become more rounded off as illustrated in Figures 5, 6 and 7.

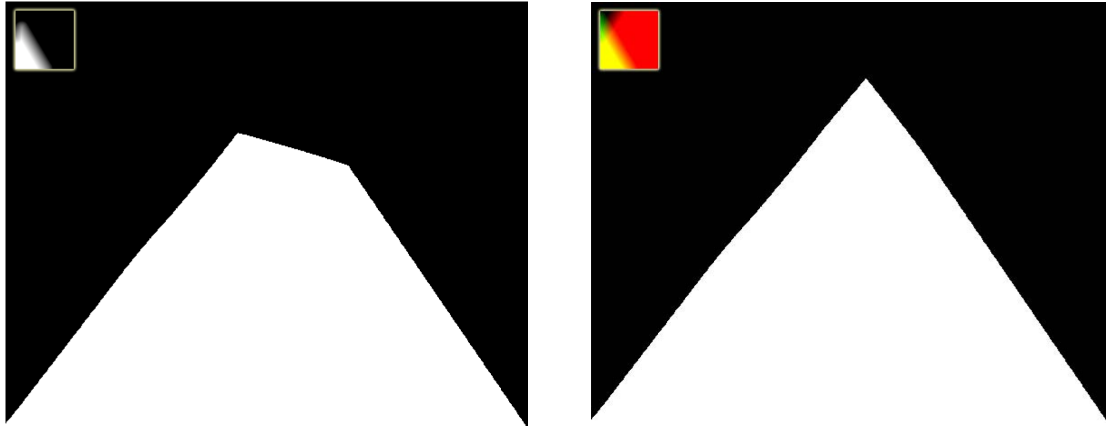


Figure 8*. Corner encoded at 64x64 using one distance field (left) and the AND of two distance fields (right)*

Sharp corners *can* be preserved, however, by using more than one channel of the texture to represent different edges intersecting within a texel. For instance, with two channels, the intersection of two edges can be accurately represented by performing a logical AND in the pixel shader. In Figure 8, we have stored these two edge distances in the red and green channels of a single texture, resulting in a well-preserved pointy corner. This same technique could also be performed on the "No Trespassing" sign if we wished to represent sharper corners on our text. As it stands, we like the rounded style of this text and have used a single distance field for this and other decals in *Team Fortress 2*.

2.7 Conclusion

In this chapter, we have demonstrated an efficient vector texture system which has been integrated into the *Source* game engine which has been previously used to develop games such as the *Half-Life® 2* series, *Counter-Strike: Source* and *Day of Defeat: Source*. This vector texture technology is used in the upcoming game *Team Fortress 2* with no significant performance degradation relative to conventional texture mapping. We were able to effectively use vector-encoded images both for textures mapped onto 3D geometry in our first person 3D view and also for 2D screen overlays. This capability has provided significant visual improvements and savings of texture memory.

2.8 References

[BLYTHE06] BLYTHE, D. 2006. The direct3d 10 system. In SIGGRAPH '06: ACM SIGGRAPH 2006 Papers, ACM Press, New York, NY, USA, pp. 724–734.

[FPR+00] FRISKEN, S. F., PERRY, R. N., ROCKWOOD, A. P., AND JONES, T. R. 2000. Adaptively sampled distance fields: a general representation of shape for computer graphics. In SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, pp. 249–254.

[LB05] LOOP, C., AND BLINN, J. 2005. Resolution independent curve rendering using programmable graphics hardware. In SIGGRAPH '05: ACM SIGGRAPH 2005 Papers, ACM Press, New York, NY, USA, pp. 1000–1009.

[MMG06] MITCHELL, J., MCTAGGART, G., AND GREEN, C. 2006. Shading in Valve's source engine. In SIGGRAPH '06: ACM SIGGRAPH 2006 Courses, ACM Press, New York, NY, USA, pp. 129–142.

[QMK06] QIN, Z., MCCOOL, M. D., AND KAPLAN, C. S. 2006. Real-time texture-mapped vector glyphs. In I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games, ACM Press, New York, NY, USA, pp. 125–132.

[RNC+05] RAY, N., NEIGER, T., CAVIN, X., AND LEVY, B. 2005. Vector texture maps. In Tech Report.

[SEN04] SEN, P. 2004. Silhouette maps for improved texture magnification. In HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, ACM Press, New York, NY, USA, pp. 65–73.

[TC04] TUMBLIN, J., AND CHOUDHURY, P. 2004. Bixels: Picture samples with sharp embedded boundaries. In Rendering Techniques, pp. 255–264.

```
float distAlphaMask = baseColor.a;

if ( OUTLINE &&
     ( distAlphaMask >= OUTLINE_MIN_VALUE0 ) &&
     ( distAlphaMask <= OUTLINE_MAX_VALUE1 ) )
{
  float oFactor=1.0;
  if ( distAlphaMask <= OUTLINE_MIN_VALUE1 )
  {
    oFactor=smoothstep( OUTLINE_MIN_VALUE0,
                        OUTLINE_MIN_VALUE1,
                        distAlphaMask );
  }
  else
  {
    oFactor=smoothstep( OUTLINE_MAX_VALUE1,
                        OUTLINE_MAX_VALUE0,
                        distAlphaMask );
  }
  baseColor = lerp( baseColor, OUTLINE_COLOR, oFactor );
}

if ( SOFT_EDGES )
{
  baseColor.a *= smoothstep( SOFT_EDGE_MIN,
                             SOFT_EDGE_MAX,
                             distAlphaMask );
}
else
{
  baseColor.a = distAlphaMask >= 0.5;
}

if ( OUTER_GLOW )
{
  float4 glowTexel =
        tex2D( BaseTextureSampler,
               i.baseTexCoord.xy+GLOW_UV_OFFSET );

  float4 glowc = OUTER_GLOW_COLOR * smoothstep(
                   OUTER_GLOW_MIN_DVALUE,
                   OUTER_GLOW_MAX_DVALUE,
                   glowTexel.a );
  baseColor = lerp( glowc, baseColor, mskUsed );
}
```

Listing 1. *HLSL source code for outline, glow/drop shadow, and edge softness.*