1. Introduction and previous work

Traditional texture mapping uses sampled images of limited resolution to represent patterns on surfaces. Sampling inherently assumes that the underlying pattern is band limited, which in plain language means that it must not contain any crisp edges. Unfortunately, most real world patterns violate that constraint, and the result is that texture mapped surfaces appear either artificially pixelated or blurred when viewed up close. Thresholding after interpolation can reduce some defects, but the fundamental problem persists: sampled images are not well suited to reproduce curved contours with crisp edges.

We present a new take on previously presented principles, using an approximate but very hardware-friendly representation of contours which we call level set contour textures. Conceptually, the representation is a discretized, lossy encoding of a distance field and its gradient which can encode patterns with curved contours and a high but limited complexity. The contours can be rendered with analytically anti-aliased crisp edges in real time using only a small amount of the processing power of a modern GPU. Furthermore, we present a complete and free software toolchain for automated creation of texture data from vector-based artwork or bitmapped images, making this method freely available and immediately useful for content creators.

The proposed texturing method is approximate, but good enough for many purposes and very fast. It can be used even in low-end modern GPUs to greatly improve the quality of text, line drawings, inlays, tilings, alpha masks, bump maps and other textures where crisp edges and smooth curves are important. The distance field approach makes it easy to use the data for special effects like glows, outlines of arbitrary width and embossing.

Several approaches have been suggested previously for real time rendering of contour-based representations of patterns with crisp edges. Early attempts by Tumblin and Choudhury [1], Ramarayanan et al [2] and Sen [3] were restricted by limitations in then-current hardware. An approach by Loop and Blinn [4] required fine-grained pattern-dependent triangulation of the textured plane and was not a drop-in replacement for texture mapping. Ray et al [5] presented a method aimed at font rendering, but the shader was quite complex, and the texture creation was cumbersome and involved undisclosed software tools. A simple, fast but restricted method was proposed by Gustavson [6]. Recent approaches by Nehab and Hoppe [7], Parilov and Zorin [8] and Qin et al [9, 10] have all aimed for accuracy and generality, but suffered from shader complexity and unsatisfactory rendering speeds on current hardware. Texture creation is generally complicated and uses undisclosed tools.

The only approach which has found use in mass market content creation is a simplistic approach presented by Green [11]. Like [11] and [8], we use a sampled distance field as suggested by Frisken et al [12]. Like [11], our method renders quickly on a wide range of hardware, but with higher quality and precision. Our detail quality is lower than [8], but rendering is orders of magnitude faster.

2. Contour representation

In our approach, contours are encoded by a vectorized level set, augmenting a signed distance field $D(u,v)$ with explicit gradient information. Each texel encodes a local linear approximation of $D(u,v)$ as $F(u,v) = g_{ux} u + g_{vy} v + D_i$. The coefficients $\{g_{ux}, g_{vy}, D_i\}$ for each texel can be stored directly in a 3-channel texture. See Figure 2.

![Figure 2. Eight texels, each encoding a local gradient $(g_{ux}, g_{vy})$ (orange) and signed distance $D_i$ to the nearest edge (green/red).](image)

During rendering, $\{g_{ux}, g_{vy}, D_i\}$ are interpolated between neighboring texels, and a function value $F(u,v)$ is computed at the current $(u,v)$ position using the interpolated polynomial coefficients. The result is an approximation of the original distance field $D(u,v)$. $F(u,v)$ is smooth and has near unit slope everywhere. It is an implicit contour representation, with $F(u,v)=0$ along a curve which closely approximates the original edge $D(u,v)=0$. The value of $F$ is the signed distance to the nearest edge.

$F$ is well defined over the entire texture plane and can be evaluated anywhere, not just around edges. Analytic antialiasing works well even for minification, and the signed distance field makes it possible to render outlines, bevels, halos, distorted edges and numerous other special effects using the signed distance field. See Figure 3.

3. Texture creation from artwork

3.1 Preparing the input image

To create a level set contour texture, artwork is rendered to an anti-aliased high resolution image. See figure 4.

For the success of the automatic conversion, it is essential that the input image is a proper contour image, i.e. that it has clean, uniform areas of pure black and pure white, with gray values only at the anti-aliased edges. No features in the image should be thinner than one pixel, and the boundary between the foreground and background should be antialiased with a box filter the size of one pixel, such that the antialiased edge is exactly one pixel.
The modified EDT is run twice, once to create a distance field and an antialiased grayscale image with respect to the nearest edge pixel. However, near the contours where we want the greatest precision, this vector deviates significantly from the local gradient of the distance field. Therefore, a pair of convolution filters is run on the distance field $D$, to estimate the local gradient direction $[g_u, g_v]$ at each texel center. Our simple gradient estimation filters use fixed convolution kernels which are not ideally suited for gradient estimation around very thin features and at points where the gradient changes rapidly, so for accuracy reasons the distance transform and the gradient estimation are performed on an image with 4 times higher resolution than the final output texture, and the computed texture is subsampled before export. Better gradient estimation filters could reduce the need for extra data, speed up the conversion and slightly increase the contour quality in problematic regions.

Texture creation is an automatic preprocessing step. A 256x256 RGBA texture can be created in a few seconds with GNU Octave.

3.4 Texture file storage

The sampled gradient and distance can be stored directly as a three component feature vector $[g_u, g_v, D]$ for each texel in a floating point RGB texture image. However, while the range and precision of $D$ is crucial to the quality of the rendered output, the two gradient components have a limited range of $[-1,1]$ and moderate requirements on precision. We encode the data in an 8-bit RGBA texture using 8 bits each for $(g_u, g_v)$ and $8+8 = 16$ bits for $D$. This saves $1/2$ on texture memory, it makes the texture files viewable by standard image editing tools, and the shader requires no GPU support for floating point textures.

4. Rendering

In the vicinity of each texel center, a good estimate of the true value for $D(u,v)$ is the bilinear polynomial

$$F_i(u,v) = g_u(u-u_i) + g_v(v-v_i) + D_i,$$

where $[g_u, g_v, D_i]$ is the texture data at texel $i$, and $(u_i, v_i)$ is the center of texel $i$. Further away from the texel center, the linear polynomial is less accurate unless the distance field is a linear ramp in that region. A better estimate of the local feature vector $[g_u, g_v, D]$ can be computed by interpolating between texels. The standard bilinear interpolation mechanism for texture filtering could do the job, but to achieve the desirable property of second-order continuity of the contour, the texel interpolation is implemented in shader code using a cubic weighting function. The explicit texel interpolation requires some careful computation of local coordinates, see Figure 5.

The texture of size $w$ by $h$ texels is indexed without interpolation using integer coordinates in the range $([0, w-1], [0, h-1])$. The texel lookup and interpolation is performed as follows:
The computations required to evaluate \( F \) at one point amount to four texture lookups in a 2x2 texel neighborhood and roughly thirty multiplications and additions. The texel interpolation amounts to two thirds of these computations. All arithmetic operations are readily vectorized for efficient GPU computation. Despite the cubic interpolation function, only a 2x2 texel neighborhood is used, and the operation is only slightly more complicated than a bilinear interpolation.

The final interpolated function \( F(u,v) \) is a good approximation of \( D(u,v) \), except where the gradient undergoes a rapid local change and component-wise linear interpolation no longer yields a good estimate of the gradient direction between the sample points. Such errors distort the shape of the curve, but do not shift the contour more than a fraction of a texel from its true position.

Near thin details and sharp corners. However, the generated contours are reasonably accurate, and they remain crisp at any magnification, limited only by the floating point precision of the GPU. Analytic anti-aliasing is particularly simple because the gradient is computed explicitly for every fragment. The texture behaves well even under extreme minification. The absence of mipmaps makes the method particularly well suited for animated procedural effects.

5. Performance

Contrary to most methods presented previously, level set contour texturing shows very high performance. Rendering speed is around 1.5 G fragments per second on an ATI 4850 GPU (a 512x512 window renders at 6,000 FPS). About 1/3 of that was achieved on ATI 3650 and Nvidia 8600 GPUs. Overall, the method shows excellent performance for high-end and mid-range GPUs, and highly useful performance for GLSL-capable budget-level GPUs. The data structure and the shader are both very straightforward, and rendering speed is independent of texture size and pattern complexity. We conclude that our texturing method is useful on any modern GPU, even considering that many other tasks compete for resources in real applications.

6. Source code and demo

Complete source code for the texture creation process and the rendering shader, along with instructions for use and a multi-platform interactive demo, is located at this address: 
http://www.itn.liu.se/~stegu/Siggraph09/

References