Eurographics Symposium on Rendering (2004)
H. W. Jensen, A. Keller (Editors)

Feature-Based Textures

G. Ramanarayanan, K. Bala,[1†] and B. Walter[2‡]

[1] Department of Computer Science [2] Program of Computer Graphics
Cornell University, Ithaca, NY, USA

Abstract

This paper introduces feature-based textures, a new image representation that combines features and samples for high-quality texture mapping. Features identify boundaries within an image where samples change discontinuously. They can be extracted from vector graphics representations, or explicitly added to raster images to improve sharpness. Texture lookups are then interpolated from samples while respecting these boundaries. We present results from a software implementation of this technique demonstrating quality, efficiency and low memory overhead.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Three-Dimensional Graphics and Realism]: Color, shading, shadowing, and texture

1. Introduction

Texture mapping is a popular and inexpensive technique for conveying the illusion of scene complexity and increasing perceived image quality in graphics applications. Texture maps are fast, simple to use, and remarkably general. However, they have limited resolution, and thus there is an optimal viewing distance at which textures have the best quality. Viewing textures from distances farther than optimal creates aliasing artifacts; MIP-maps [Wil83] are often used to solve this problem. However, when textures are viewed at closer than the optimal distance, artifacts still arise due to inadequate sampling of the original scene. Interpolation alleviates this problem somewhat but causes excessive blurring. Increasing the original texture resolution also removes artifacts but at the cost of increased texture memory usage.

This paper presents *feature-based textures* (FBT) — an alternative image representation that explicitly combines features and samples. Features are resolution-independent representations of high-contrast changes in the texture map. They enable sharp, high-quality texturing at close viewing distances, while samples maintain the flexibility of traditional texture maps.

Figure 1 illustrates how FBTs are created and used. The top row shows how an input image and its features are combined to form the FBT. Unusable samples from the input are automatically discarded. Each FBT texel stores features and samples. Features are represented as line segments and, for higher quality, curves. The middle row shows how FBTs are rendered. As in standard texture mapping, the texture value at a point p is reconstructed using bilinear interpolation of nearby texture samples. However, in FBTs, only *reachable* samples are used - that is, those on the same side of all features as p.

The bottom row of Figure 1 compares FBT rendering with standard texture mapping using bilinear interpolation. The FBT captures sharp features of the text and subtle shading gradations. The output from standard texture mapping is blurry by comparison. For this example, an FBT of resolution 230×256 (416KB) is contrasted against a texture map of resolution 460×512 (690KB). To achieve image quality comparable to this FBT, the texture map would require 41MB of memory.

The rest of the paper is organized as follows. Section 2 discusses related work, and Section 3 gives an overview of FBTs. Sections 4 and 5 describe in detail how the FBT is created and used in rendering. Section 6 presents results, which are discussed in Section 7. Finally, we make some concluding remarks in Section 8.

† {graman, kb}@cs.cornell.edu
‡ bjw@graphics.cornell.edu

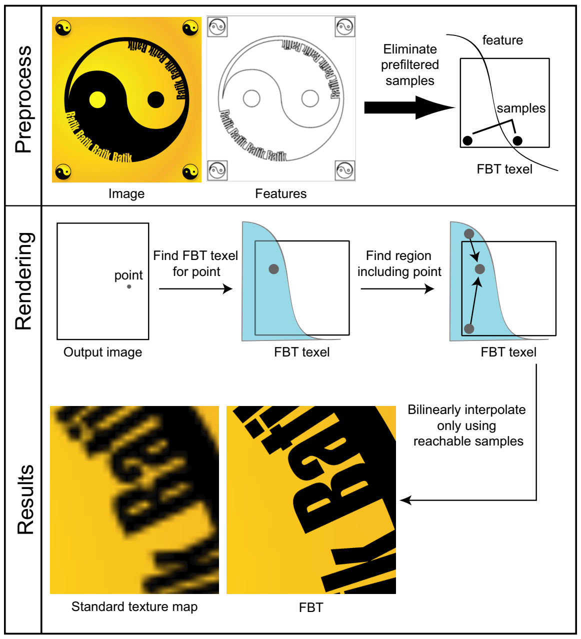

Figure 1: *Feature-based textures. Top row: FBT combines features and texture samples. Middle row: Sample is reconstructed by interpolating reachable samples from adjacent FBT texels. Bottom row: FBT captures sharp features unlike standard texture mapping.*

2. Related Work

The idea of using arbitrary resolution functions to model graphics is not new. Vector-based image representations such as Scalable Vector Graphics [SVG] and PostScript are resolution-independent, and therefore they are heavily used for printing and illustrations. However, they are not amenable to point sampling and cannot be used in arbitrary rendering contexts. Additionally, pure vector-based techniques are somewhat limited in the visual complexity they can produce. While these formats can include raster images, in doing so they are again subject to the resolution dependence of the raster representation.

Procedural textures [EMP*94] completely specify a resolution-independent texture function that can be directly sampled and manipulated; these textures are often generated using mathematical simulations or random noise. While useful for natural phonemena, traditional procedural techniques are unable to enhance existing images with plausible high-resolution information, making them unsuitable for image-based texture mapping.

Image superresolution [HT84, EF97, BS98] aims to generate a high resolution image from a series of low resolution inputs that capture the same scene from different viewing locations. FBTs introduce sharpness by annotating a single image, but it would be interesting to look at annotation of multiple images to create a higher quality result.

Feature finding and analysis [DH72, Can87, MBLS01] is often used in computer vision for a variety of applications, including stereopsis, shape recognition, and object tracking. This has been extended to 3D point-based models as well [PKG03]. Our goal is different; we explicitly use features to improve the quality of the rendered result. The technique we present is related to anisotropic diffusion [PM90], which blurs grainy parts of an image but maintains sharpness in discontinuous regions. There is also similar work in image reconstruction [CGG91, Car88, HC00], but the focus there is on compression and fundamental image representations, not a mechanism for sampling in a rendering context.

Autotrace and Potrace [Sel] are excellent tools for tracing features in images and extracting vector representations. We have used Potrace to find features in some textures.

There is a substantial body of work in computer graphics on the explicit use of discontinuities in high quality reconstruction, such as radiosity discontinuity meshing [Hec92, LTG92], illumination functions [SLD92], and silhouette clipping [SGHS00]. Recently, there has been interest in new image representations that capture discontinuities for interactive global illumination [BWG03] and hardware-based shadowing techniques [SCH03].

Our work is most closely related to that of Salisbury et. al. [SALS96], who use a hybrid image representation with piecewise linear discontinuities for resolution-independent pen-and-ink rendering. The FBT representation captures both lines and curves, and is demonstrated for both vector graphics and raster images. Because our focus is texture mapping, we demonstrate support fast point queries and bilinear interpolation. Also, our technique does not use NPR rendering styles to mask artifacts.

3. FBT Overview

Like a standard texture map, an FBT is a two-dimensional array of texels. However, FBT texels store both *features* and *samples*. Features are discontinuity boundaries that intersect the texel; samples are values of the function being represented by the texture. Figure 2 shows some of the ways a texel can be intersected by features. In the FBTs shown in this paper, most texels are empty, like Figure 2-(a). Sampling from empty texels is no more expensive than a standard texture lookup.

3.1. Features and regions

Features characterize high-contrast changes in the input image; they are represented by connected chains of splines. In our implementation, we support Bezier curves, ranging from lines to cubics. We refer to individual splines in a feature as *sub-features*.

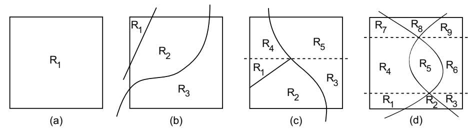

Figure 2: *Example texels, features, and regions (labeled as $\{R_i\}$). When features come to a T junction (c), or intersect completely (d), the texel is split (dashed lines) at the intersection points, forming horizontal bands that contain no intersections.*

Let us assume for now that the features and FBT resolution are both specified (as described in Section 4.1). As shown in Figure 2, texels are divided into various disjoint *regions* by their intersecting features. For compactness, every region contains exactly one sample, which we assume is located at the bottom left of the region. To sample properly, we need to correctly identify the regions different points are contained in. Usually this is very easy (Figure 2-(ab)) but when features intersect the problem is a little harder (Figure 2-(cd)). We will revisit this issue shortly.

3.2. Rendering an FBT

Texture maps can be queried in various ways. The most accurate and expensive technique is to map the input pixel's area into texture space and filter the area to return an antialiased texture value. We use an alternative, cheaper technique: map a point visible from the pixel into the texture, and do a lookup using bilinear interpolation. Supersampling is used to handle antialiasing. Thus FBT texture lookups involve the following operations:

1. Transform the point into texture space point p.
2. Find the FBT texel T that includes p.
3. Find the region R in T that includes p.
4. Look up the sample in R and samples from reachable regions in adjacent texels.
5. Return the bilinearly interpolated texture value.

Steps 1, 2 and 5 are straightforward and similar to standard texture map operations, whereas steps 3 and 4 are specific to FBTs. Therefore, the FBT must store just enough information to do steps 3 and 4 efficiently. Section 4 fully describes the FBT preprocess that accomplishes this.

Locating the region containing a point

Step 3 involves quickly locating the region that contains a given point p. A simple test accomplishing this is to see which side of each feature p lies on. This will work for any texel that has no intersections, but it may fail in the case where features intersect each other. For example, in Figure 2-(d), R_2 and R_8 are distinct, but they are on the same side of both features. To handle such situations, the texel is split horizontally at each feature-feature intersection (indicated by the dashed lines). This forms a series of *bands* that do not contain any intersections. Band subdivision is combined with the sidedness test above to determine p's region. See Section 4.5 for details.

Finding samples for interpolation

Step 4 involves identifying samples that can be used to compute texture values for a given point in the texture, using bilinear interpolation. For an empty texel (which contains exactly one region), bilinear interpolation is performed in the usual fashion using the single sample of that texel, along with samples from three adjacent texels. Because the sample is taken from the lower left corner, the three texels to the right, above, and diagonally above to the right must contain usable samples (see Figure 6-(a)).

For points that lie in nonempty texels, bilinear interpolation is performed using samples from the current texel and possibly also from regions in adjacent texels. A sample from an adjacent texel is used only if it is reachable from the current point; otherwise, possibly erroneous interpolation would occur across a blocking feature. Section 4.6 explains how reachable samples are identified and interpolated.

4. Creating FBTs

Some preprocessing is required to prepare the FBT data structure for use in rendering. The exact nature of the preprocessing depends on the kind of input being used to generate the FBT.

4.1. Input Specification

The input to the FBT preprocess consists of an image, a set of features, and a user-selected FBT resolution. This information is then combined to create a finished FBT.

Finding features: Different types of input are amenable to different types of feature extraction. Features are identified either through automated extraction or manual specification, as discussed below.

- **Automatic extraction.** Vector-based representations can be queried directly to return all features. Raster image features can be obtained either by using tracing programs [Sel], or by applying feature detection algorithms [Can87].
- **Manual specification.** A user can manually draw features to match the high contrast changes in the image. The output of automated extraction techniques can also be used to assist in this process. This user interaction is needed only once per image, and a library of FBTs can be reused by applications.

Selecting FBT resolution: Because FBTs represent features explicitly, there is some flexibility in choosing texel resolution. A natural tradeoff exists between texture quality/efficiency and compactness; different applications have different demands. For example, an input with gradients should use more texels to accurately capture shading variations, while a simple solid-color SVG input only needs a few texels.

4.2. Feature processing

One of our goals is to have a representation general enough to reproduce textures with any configuration of features. For this reason, we compute all feature-texel and feature-feature intersections, because they all affect the region determination process. Line intersection is trivial; line-curve intersection is also relatively straightforward, requiring the use of a cubic solver. Robust curve-curve intersection is possible using techniques such as interval-based intersection [Tup01] or Bezier clipping [SN90].

To accelerate computation involving features, a kd-tree is constructed over texture space. It can be queried to return all sub-features in a given bounding box, which accelerates intersection tests.

4.3. Invalidating prefiltered samples

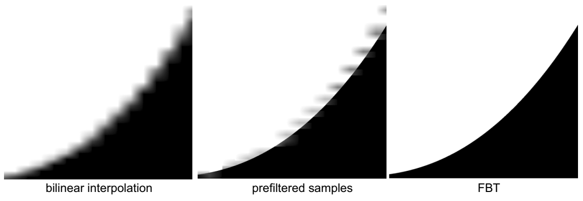

Figure 3: *Effect of prefiltered samples. Left: image produced by bilinearly interpolating texture samples from a raster texture map. Middle: using prefiltered samples in the FBT causes artifacts. Right: eliminating prefiltered samples in the FBT produces accurate output.*

When constructing an FBT from an image, we treat most samples as plausible point samples because they are in smooth regions. However, samples that lie close to features are often 'prefiltered' by the device used to capture them. For example, most cameras have some transfer function that filters all incoming light through a pixel (and nearby pixels). These prefiltered samples cannot act as point samples, so they may cause rendering artifacts (as in Figure 3, middle).

If the properties of the imaging device are known, sample invalidation can be decided using a metric similar to that of [IBG03]. Often, however, the imaging device is not known, so the user can explicitly specify an invalidation distance from features. Typically a (∞-norm) distance of 1 pixel unit in the original image suffices; this also applies to artist-drawn images, where antialiasing typically happens on a pixel level. Eliminating filtered samples improves reconstruction during rendering (Figure 3, right)

4.4. Filling holes

The invalidation process described above can create *holes* - regions in the texture with no sample. These holes are filled using information from nearby reachable samples.

In general, since features are composed of chains of splines, texel regions can have complicated boundaries. To fill holes we need a way to partition texture space. The constrained Delauney triangulation used in [SALS96] is limited to line segments; to handle curves, we use a trapezoid decomposition variant [O'R93]. For each texel, we record the y-coordinates of all feature-texel intersections, feature-feature intersections, and sub-feature maximum and minimum y-values. These coordinates correspond to horizontal lines that split the texel into simple 4-sided *sub-regions*. Each sub-region has a flat upper and lower boundary, and its right and left boundaries are either splines or sides of the texel. Some care must be taken to handle sub-features that are horizontal lines. Figure 4 shows the sub-regions $\{L_i\}$ computed for the texel on the right.

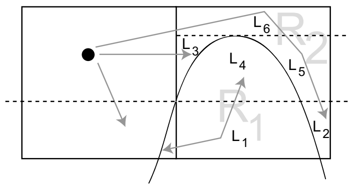

Figure 4: *Intermediate representation for reachability and hole filling. To form the sub-regions $\{L_i\}$, a horizontal line is drawn at the curve's maximum y value and its intersection with the middle texel boundary, splitting the right texel into three bands. The sample in the left texel is copied to the sub-regions in R_2, but it cannot reach anything in R_1.*

Once the sub-regions are constructed, we build a reachability graph where the sub-regions are vertices, and adjacent sub-regions are connected by an edge if the boundary between them is not blocked by a feature. Holes are then filled by searching for and copying the closest reachable sample. We will need the reachability graph later, so we will save it; however, at this point, we can eliminate sub-regions from the FBT representation. All sub-regions are collapsed and their samples merged through area-weighted averaging. Bands remain to handle feature intersections; all other texel divisions along the y-axis are eliminated. For example, in Figure 4, we merge L_1 and L_4 into R_1, and L_2, L_3, L_5, and L_6 into R_2.

4.5. Region testing

To perform efficient texture lookup during rendering, a fast test is needed to determine which texel region a point p lies in. We have handled feature intersections by forming horizontal bands, which leaves us with rectangular bounding areas divided by multiple nonintersecting features. Define the term *simple feature* to refer to a portion of a feature that splits a rectangular bounding area into two disjoint regions, which we arbitrarily term 'inside' and 'outside'. A simple feature is therefore either a closed loop, or a portion of a feature that enters and exits the bounding area exactly once.

With closed loops, the traditional method to distinguish 'inside' and 'outside' is to use an intersection *parity* test: cast a ray from p, and check the parity of the number of intersections with the loop. Odd parity means 'inside', and even parity means 'outside'. It is natural to shoot a ray in one of the four directions $(\hat{x}, -\hat{x}, \hat{y}, -\hat{y})$ because of computational convenience. To make a region determination test for a simple feature, we could imagine 'completing' the feature by outlining one of the two regions it delimits, forming a closed loop (bolded in Figure 5), but we would need to keep track of the extra boundary edges.

It is possible to pick a ray direction such that the test result of the feature alone is the same as the test result of this closed loop. Figure 5-(ab) illustrates this principle. In each diagram, applying the given test to points in shaded/unshaded areas returns odd/even parity, respectively. In (a), notice how the closed loop shares a portion of the left boundary. Therefore, if one considers the parity test against only the simple feature in the direction $-\hat{x}$, the area in the middle will have reversed parity because the ray-boundary intersection was ignored ((a)-top). However, if we pick a direction that can never intersect that boundary, the parity test result against the simple feature will be sufficient; thus $\hat{x}$ produces the correct result ((a)-bottom). The example in (b) is similar. In general, the ray cannot be cast towards a boundary that intersects the simple feature; any other direction can be chosen.

We can now create a test to distinguish the $n+1$ regions created by n simple features. The semantics of this test will correspond to a linear search of a sorted array. We examine the simple features $\{f_i\}$ in order; if the point is inside (less than) f_j, it is in region R_j, and if it is outside (greater than) all n features, it is in region R_{n+1}. Figure 5-(c) shows how this works.

We are assuming that all features in the texel or band are simple. Any partial feature that does not cut the whole area into two regions either terminates at an intersection point (in which case bands handle it) or 'floats' inside the bounding area, in which case it is ignored.

4.6. Texture lookup with interpolation

As described in 3.2, we would like to use bilinear interpolation to capture smooth texture shading. In a standard texture lookup, we bilinearly interpolate the four samples nearest to the point. Let the texture sample at p be denoted by s_p, and let the four nearby samples be $\{s_i\}$, with bilinear interpolation weights $\{w_i\}$. Then, $s_p = \sum_i w_i s_i$.

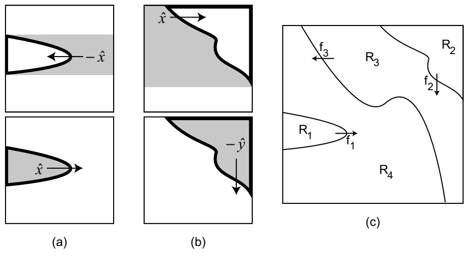

Figure 5: *Region determination with simple features. The ray casting direction for each feature is indicated by the arrows. (ab) Top: Testing intersection parity (odd/even = shaded/unshaded) against the feature alone is not sufficient if the ray points towards a boundary that the feature intersects. Bottom: Any of the other directions is correct. (c) The complete region determination test with sorted simple feature array $\{f_i\}$.*

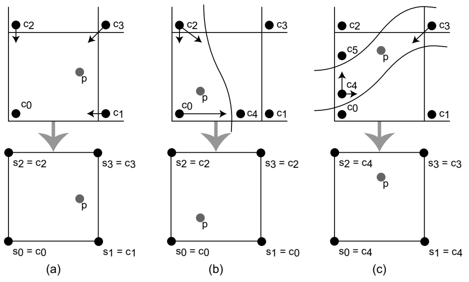

Figure 6: *Bilinear interpolation using neighboring reachable samples. Texel representative samples are in the lower left corners of the texels. If all 4 samples are not available, existing samples are used instead. (a) Standard texture mapping. (b) The right side samples are blocked, so the ones on the left are copied, preserving the gradient. (c) Only the upper right sample can be reached.*

Standard bilinear interpolation is fine for smooth regions, but given the complications of variable numbers and shapes of regions, it is unclear how to explicitly define a general reconstruction function that is quickly computable, both in terms of picking appropriate reachable samples and calculating accurate interpolation weights. We could store four samples at the corners of each region of a texel, but this would cause roughly a four-fold increase in memory usage. Our goal is to interpolate texture values while still storing one sample per region, like standard texture maps.

We have adopted a practical solution to this problem that is adequate in many situations. An FBT stores only one sample per texel region; this sample is associated with the region's lower left corner ($\{c_i\}$ in Figure 6). The sample that is in the lower left corner region of the texel is the *represen-*

tative sample (c_0, c_1, c_2, c_3 in Figure 6). To perform bilinear interpolation, we use the single sample in the region p maps to, and all reachable representative samples from neighboring texels, for a total of 4 possible samples. These samples are placed at the corners of an imaginary texel and interpolated using the resulting weights $\{w_i\}$. If we don't have all 4 samples, we fill the empty spots by reusing the closest (distance-wise) of the ones we have. See Figure 6 for examples. In the final FBT, reachability information for each region is computed using the previously computed reachability graph and stored in a 1 byte *sample availability mask* (2 bits to encode which of the 4 possible samples to use in each corner).

4.7. FBT Memory usage

To store features, each FBT maintains a global list of 2D points. Each feature is defined by an array of indices into this point list, with an index for each sub-feature; each index uses 2 bytes. Splines are represented by 2 to 4 control points each.

Each texel stores an array of horizontal bands, which each store an ordered list of simple features. Each simple feature stores the following: feature number (2 bytes), start sub-feature index (2 bytes), end sub-feature index (2 bytes), start parameter value (1 float) and end parameter value (1 float). The start and end parameter values are the spline parameter values indicating when the start/end sub-features enter/exit the texel. Together, this information is sufficient to find the chain of sub-features comprising the simple feature. Additionally, each simple feature uses 2 bits to indicate which ray direction to use with the feature during intersection parity tests. In total, each simple feature uses 15 bytes. Additionally, each sample associated with a region stores 4 bytes (3 bytes for color, and 1 byte encoding the neighboring sample availability). Given k texel features in a horizontal band, $k \times 15 + (k+1) \times 4$ bytes of data are stored.

5. Rendering FBTs

We now discuss how FBTs support efficient rendering, focusing on the two steps from Section 3.2 that differ from standard texture maps. The first step is to identify the region p falls in, and the second step is to find samples reachable from p without crossing any features.

Finding the FBT region for a point (Step 3): To find which region p is in, we examine its y-coordinate to identify the horizontal band to search. As described earlier, each band stores an ordered list of simple features against which p is tested sequentially (Figure 5-c).

Intersecting a ray with sub-features is fast. For a line segment, the test is straightforward. For curves, the intersection can be directly computed by solving a cubic, which could be slow. To eliminate unnecessary cubic solving we first test the intersection of the ray with the curve convex hull. If the point is inside, the cubic solver is invoked. If the point is outside, the intersection parity test for the curve can be deduced from the convex hull test.

Finding reachable samples (Step 4): Once the region containing p has been identified, its sample availability mask encodes which neighboring samples to use for bilinear interpolation. These four samples are then interpolated as described in Section 4.6.

6. Results

In this section we present results comparing FBTs to standard texture mapping, focusing on image quality, memory usage, and performance issues. The FBT system is implemented in Java, and all results were obtained on a dual 3.06 GHz Pentium Xeon machine with 2 GB RAM. Constructing an FBT from features and samples as a preprocessing step runs in time proportional mainly to the number of FBT texels; for the examples we show, this is typically under 30 seconds, and at worst one minute. Unless indicated otherwise, all images are generated in a raytracing context, using 4 point samples per pixel. During rendering the use of FBTs imposes no noticeable performance overhead over standard texture maps.

Two types of inputs were used for this evaluation: SVGs and raster images. Figure 7 shows some example inputs along with their associated features. For the stop sign and yin yang SVGs, the Batik open-source SVG framework (http://xml.apache.org/batik/) was used to acquire the input samples and extract features. For the flower, stained-glass, and wizard skin, we manually annotated the image with line segments. The banana example was annotated with splines obtained using Potrace.

The wizard skin example is included primarily to illustrate potential applications in games; it has only been partially annotated, so its potentally skewed memory / performance results are not included in the tables.

6.1. Memory Comparisons

As mentioned earlier, the user can choose the appropriate FBT resolution for each texture map. To make comparisons fair, we use standard texture maps that consume strictly more memory than the corresponding FBT. Table 1 shows the memory usage for the two SVG examples.

As a point of comparison, a texture map that could achieve the same quality as the FBT for the zoomed-in viewpoint shown in Figure 8-(a) would require approximately 41 MB. Similarly, the zoomed-in stop sign in Figure 9 could be rendered at the same quality as the FBT output if the stop sign texture map used 3MB.

In the raster image examples, our goal was to annotate

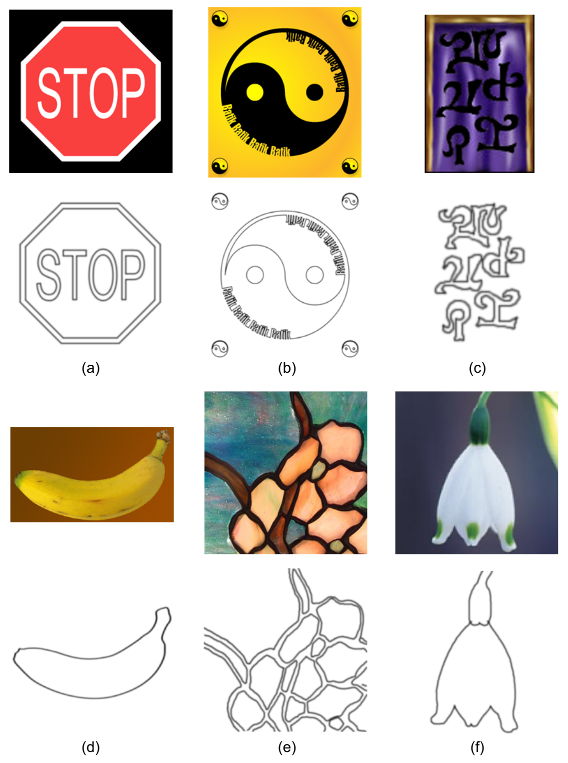

Figure 7: *Example inputs and their corresponding features. (a) and (b) are SVGs; (c), (e), and (f) are raster images annotated by hand using line segments; (d) is a raster image annotated by Potrace using splines.*

Example	FBT Res.	FBT Size	Raster Res.	Raster Size
Stop sign	16×16	9KB	64×64	12KB
Yin yang	230×256	416KB	460×512	690KB

Table 1: *Comparison of resolution and storage size of FBT vs. standard texture map (stored as packed RGB).*

an existing image with extra sharpness and detail, providing higher quality during magnification. To retain all of the information in the source images, we constructed an FBT with the same dimensions. In our experience, the overall size of an annotated FBT constructed in this way is about twice the size of the original raster image.

6.2. Quality Comparisons

Figure 8 compares several image reconstruction methods. The highest-quality rendering, shown in Figure 8-(a), is the SVG rendering of the image. Figure 8-(b) shows results produced using the FBT. It can be seen that the FBT correctly captures the sharp detail and subtle gradients of the SVG, whereas standard texture mapping (bottom) generates output of lower quality. Given the poor output of nearest neighbor sampling (c), for the rest of the results we only compare FBTs with bilinear interpolation (d).

Figure 9 shows the stop sign comparison. At high magnification, the FBT faithfully reconstructs the image, while the standard texture map exhibits significant artifacts.

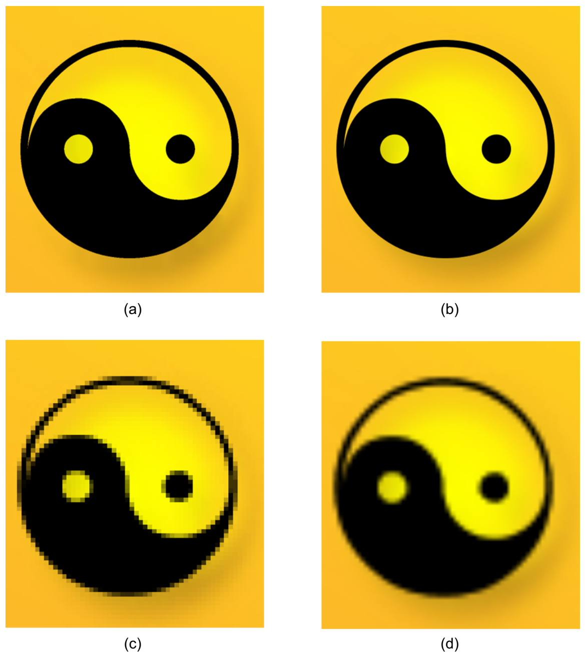

Figure 8: *Reconstruction of lower left corner of yin yang example using (a) vector-based SVG rendering; (b) 230×256 FBT; (c) 460×512 texture map with nearest neighbor sampling; (d) 460×512 texture map with bilinear interpolation.*

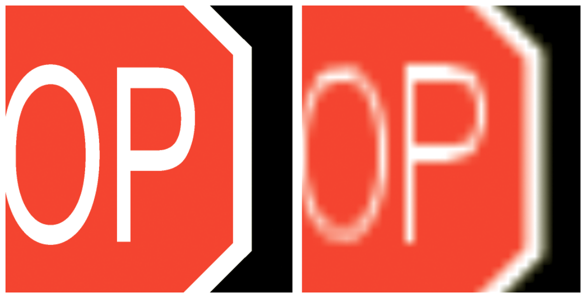

Figure 9: *Stop sign quality comparison. Left: FBT; Right: standard texture map.*

Figure 10 compares results of FBT rendering versus bilinear interpolation from standard texture maps. While our system supports curves fully (as shown by the banana and SVG examples), one can see from the flower and stained glass examples that considerable sharpness can be added simply by using line segments alone, which is advantageous when considering a GPU implementation of this technique.

Texture mapping a 3D model: In order to demonstrate results on an actual 3D model, we acquired a skinned, low-polygon-count wizard from the game *Warcraft III®: Reign of Chaos™* (Figure 11). This is a particularly appropriate example because although the game usually views the model from afar, the user can zoom in if he chooses, revealing the quality limitations of the texture. Compare the sharpness of the runes on the back of the cloak and hood, where we added features, to the blurriness of the hem of the hood, cloak, and sleeve, where we did not. Also compare the zoomed-in renderings of the runes.

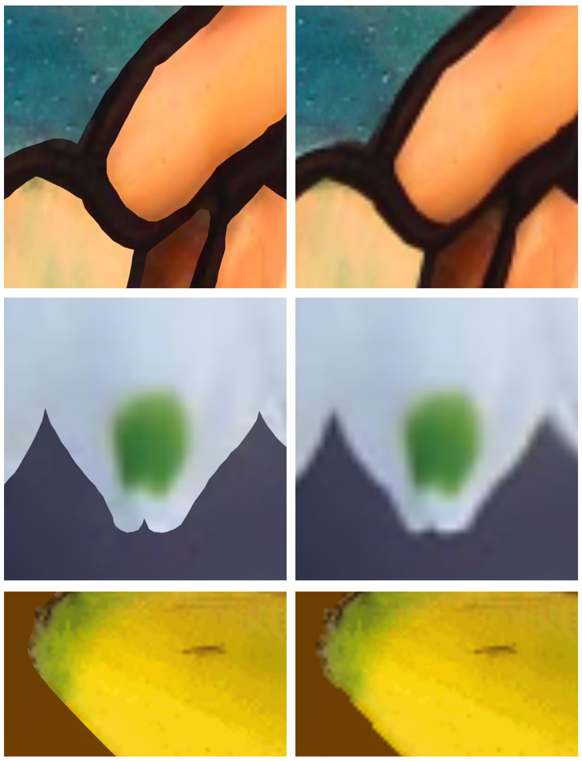

Figure 10: *Comparisons of the stained glass, flower, and banana. Left: FBT; Right: standard texture map. The stained glass and flower were annotated by hand strictly using line segments; the banana was annotated by higher order splines obtained from Potrace.*

6.3. Performance

The FBT representation is designed to mimic a standard texture map whenever possible, and to fall back on more expensive computations only near features. Thus, the work required for a single FBT query is proportional to the complexity of the target texel. Table 2 shows the breakdown of texel types in each of the FBT textures presented above, illustrating the tradeoff between FBT size and texel complexity (and therefore lookup speed). We see that more than 99% of texels have at most 2 regions (except for the artificially low-resolution stop sign texture).

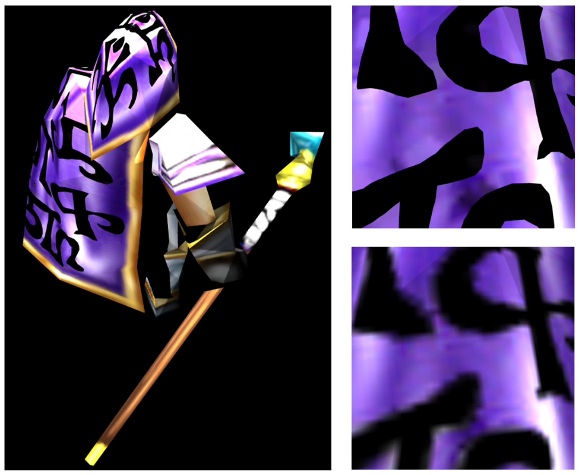

Figure 11: *Comparisons of wizard model. Left: Antialiased rendering of model using an FBT skin. Right: zoomed-in comparison (Top: FBT; Bottom: original raster skin). Artwork from Warcraft III®: Reign of Chaos™ provided courtesy of Blizzard Entertainment.*

Image	FBT Res.	Empty	2 regions	3+ regions
Stop sign	16×16	50.0%	24.6%	25.4%
Yinyang	230×256	92.9%	6.4%	0.7%
Stained glass	256×256	93.7%	6.3%	0.0%
Flower	128×128	97.1%	2.8%	0.1%
Banana	300×175	98.2%	1.8%	0.0%

Table 2: *Breakdown of texel occupancy. Empty texels have no texel features and sample lookups require no extra work compared to standard texture maps.*

To analyze cost, we are interested in the number of ray-curve intersection tests we have to do, because they are expensive compared to texel lookups and even convex hull tests (both of which can be coded very efficiently and are amenable to GPU implementation). Let c_{lookup} be the average cost to map a given point into the correct band for a region search, let c_{hull} be the cost to test against a curve's convex hull, and let c_{cubic} be the cost of a cubic intersection test. The average cost c_q of query q is approximately

$$c_q = c_{lookup} + s_{avg}(c_{hull} + f_{test} c_{cubic})$$

where s_{avg} is the average number of curves considered in each query, and f_{test} is the fraction of curves actually tested using the cubic solver, on average. In general, the majority of texels in an FBT have either one or two regions, so we expect that s_{avg} and f_{test} will be small; additionally, our convex hull test will reduce these even further. Table 3 consolidates this information for our set of inputs. The small values of s_{avg} and especially f_{test} demonstrate that performance is reasonable even if c_{cubic} is high.

FBTs are easily incorporated into the edge-and-point renderer (EPR) [BWG03], which was used to make Figure 11. The EPR creates high-quality renderings from sparse samples by treating discontinuities as first-class display primitives and using them to control interpolation. FBTs did not impact the EPR's interactive performance (8-14 fps).

Image	s_{avg}	f_{test}	cubic tests / query
Stop sign	1.051511	0.0051	0.0054
Stop sign zoom	1.571101	0.0078	0.0124
Yinyang	0.092352	0.0041	0.0004
Yinyang zoom	0.268469	0.0009	0.0024
Banana	0.018809	0.0028	< 0.0001
Banana zoom	0.023328	0.0066	0.0001

Table 3: *Higher order curve test data for 500×500 renderings of the full example images and zoomed in images shown in Figures 8, 9, and 10. The stained glass and flower are not included because they only contain line segments. Zooming in on complicated regions increases the number of cubic tests per query, but not significantly.*

7. Discussion and Future Work

There are some important issues that arise in the use of FBTs, which we discuss below. One issue is that not all types of image discontinuities can be modeled accurately using sharp features. With vector graphics inputs this is not a problem, but with raster image inputs, sharp boundaries may look flat or cut-out. This can potentially be alleviated by introducing different functions for discontinuity reconstruction.

FBTs currently use point queries as a basic mechanism for texture lookup; to antialias, we must supersample the FBT or use a discontinuity-based antialiasing rendering system such as [BWG03]. Exploring more sophisticated antialiasing mechanisms would be interesting. A related problem is that of texture quality when zooming out. MIP-mapping of textures using features is an open question that requires investigation of multi-resolution feature representations. As a temporary solution, one could simply revert to normal MIP-maps at a suitable distance from the FBT.

Each FBT texel region stores one representative sample. Therefore, it is not possible to respect two smooth gradients across a texel boundary. This could create small blocky artifacts, but they are typically not noticeable when using a large enough FBT. Solving this problem robustly is related to issues with antialiasing, MIP-mapping, and a more general reconstruction framework.

Some artifacts can also arise because holes are filled by copying nearby samples. Even using a local reconstruction filter, some smearing may be visible under magnification since we are using distance as a primary criteria in reconstructing data. Pixel-based texel synthesis can potentially solve this problem.

A GPU implementation of FBTs raises interesting challenges in terms of its representation of features because of our support for curves and variable numbers of features per texel. We are experimenting with a GPU implementation that focuses only on line segments and restricts the number of features per texel to a small number. Table 2 suggests that this is possible because at reasonable resolutions, most FBTs texels are either empty or only have a few features. We are also optimistic about FBTs on the latest architectures (such as the NV40) which support branching in the pixel shader. Concurrent with our work, [TC04, Sen04] present fixed-size image representations that include discontinuities; however, the goal of maintaining fixed sizes is achieved by sacrificing some reconstruction quality.

Users of our system commented that the presence of a few sharp features significantly improved the overall look of a texture. We believe this is because blurriness is most objectionable when jarring artifacts of bilinear interpolation (staircasing / feathering) are observed. If these are eliminated, the overall blurriness of the texture is less noticeable. User specification of features works particularly well in this regard. Detailed studies to evaluate the perceived improvement of texture quality would be useful.

8. Conclusions

This paper introduces feature-based textures, an image representation that combines features and samples for high-quality texture mapping. The FBT is a compact representation that permits efficient texture lookups while accurately preserving features. We have demonstrated the use of FBTs for rendering a range of images with high quality and a relatively low impact on rendering performance. FBTs have the potential to substantially improve image quality both in offline rendering applications and interactive applications, such as games. The point-sampling interface supported by FBTs makes them directly applicable to ray tracers and software scanline renderers. To further broaden the scope of FBTs, we would like to investigate a GPU-based implementation and a more general reconstruction framework.

References

[BS98] BORMAN S., STEVENSON R. L.: Super-resolution from image sequences - A review. In *Proceedings of the 1998 Midwest Symposium on Circuits and Systems* (Notre Dame, IN, 1998). 2

[BWG03] BALA K., WALTER B., GREENBERG D.: Combining edges and points for interactive high-quality rendering. In *SIGGRAPH '03* (July 2003), pp. 631–640. 2, 9

[Can87] CANNY J.: A computational approach to edge detection. In *RCV87* (1987), pp. 184–203. 2, 3

[Car88] CARLSSON S.: Sketch based coding of grey level images. *Signal Processing 15*, 1 (1988), 57–83. 2

[CGG91] CUMANI A., GRATTONI P., GUIDUCCI A.: An edge-based description of color images. *CVGIP: Graph. Models Image Process. 53*, 4 (1991), 313–323. 2

[DH72] DUDA R., HART P.: Use of the Hough transform to detect lines and curves in pictures. *CACM 15*, 1 (January 1972), 11–15. 2

[EF97] ELAD M., FEUER A.: Restoration of a single super-resolution image from several blurred, noisy, and down-sampled measured images. *IEEE Transactions on Image Processing 6*, 12 (1997), 1646–1658. 2

[EMP*94] EBERT D. S., MUSGRAVE F. K., PEACHEY D., PERLIN K., WORLEY S.: *Texturing and Modeling: a Procedural Approach*. Academic Press Professional, Inc., 1994. 2

[HC00] HUNTER A., COHEN J. D.: Uniform frequency images: adding geometry to images to produce space-efficient textures. In *Proceedings of the Conference on Visualization '00* (2000), pp. 243–250. 2

[Hec92] HECKBERT P.: Discontinuity meshing for radiosity. In *3rd Eurographics Workshop on Rendering* (1992), pp. 203–226. 2

[HT84] HUANG T. S., TSAY R. Y.: Multiple frame image restoration and registration. *Advances in Computer Vision and Image PRocessing 1* (1984), 317–339. 2

[IBG03] ISMERT R., BALA K., GREENBERG D.: Detail synthesis for image-based texturing. In *Symposium on Interactive 3D Graphics '03* (Apr. 2003), pp. 171–176. 4

[LTG92] LISCHINSKI D., TAMPIERI F., GREENBERG D. P.: Discontinuity meshing for accurate radiosity. *IEEE Comput. Graph. Appl. 12*, 6 (1992), 25–39. 2

[MBLS01] MALIK J., BELONGIE S., LEUNG T. K., SHI J.: Contour and texture analysis for image segmentation. *International Journal of Computer Vision 43*, 1 (2001), 7–27. 2

[O'R93] O'ROURKE J.: *Computational Geometry in C*. Cambridge University Press, 1993. 4

[PKG03] PAULY M., KEISER R., GROSS M.: Multi-scale feature extraction on point-sampled surfaces. In *14th Eurographics Workshop on Rendering* (2003), pp. 281–289. 2

[PM90] PERONA P., MALIK J.: Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence 12*, 7 (1990), 629–639. 2

[SALS96] SALISBURY M., ANDERSON C., LISCHINSKI D., SALESIN D. H.: Scale-dependent reproduction of pen-and-ink illustrations. In *SIGGRAPH '96* (July 1996), pp. 461–468. 2, 4

[SCH03] SEN P., CAMMARANO M., HANRAHAN P.: Silhouette shadow maps. In *SIGGRAPH '03* (July 2003), pp. 521–526. 2

[Sel] SELINGER P.: Potrace: a polygon based tracing algorithm. potrace.sourceforge.net/potrace.pdf. 2, 3

[Sen04] SEN P.: Silhouette maps for texture magnification. In *Graphics Hardware 2004 (to appear)* (2004). 9

[SGHS00] SANDER P. V., GORTLER S. J., HOPPE H., SNYDER J.: Silhouette clipping. In *SIGGRAPH '00* (Aug. 2000), pp. 327–334. 2

[SLD92] SALESIN D. H., LISCHINSKI D., DEROSE T.: Reconstructing illumination functions with selected discontinuities. In *3rd Eurographics Workshop on Rendering* (May 1992), pp. 99–112. 2

[SN90] SEDERBERG T. W., NISHITA T.: Curve intersection using Bezier clipping. In *Computer-Aided Design* (Nov. 1990), pp. 538–549. 4

[SVG] Scalable Vector Graphics 1.1. specification. http://www.w3.org/TR/SVG11/. 2

[TC04] TUMBLIN J., CHOUDHURY P.: Bixels: Picture samples with sharp embedded boundaries. In *15th Eurographics Workshop on Rendering (to appear)* (2004). 9

[Tup01] TUPPER J.: Reliable two-dimensional graphing methods for mathematical formulae with two free variables. In *SIGGRAPH '01* (2001), pp. 77–86. 4

[Wil83] WILLIAMS L.: Pyramidal parametrics. In *SIGGRAPH '83* (1983), pp. 1–11. 1