Cornell University Computing and Information Science Technical Report: 1084465990, May 2004

Feature-Based Textures

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

[1] Department of Computer Science [2] Program of Computer Graphics

Abstract

This paper introduces feature-based textures, a new image representation that combines features and texture samples for high-quality texture mapping. Features identify boundaries within a texture where samples change discontinuously. They can be extracted from vector graphics representations, or explicity 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 Graphcs 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 the texture has the best image 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 perhaps greatly increased texture memory usage.

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

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

Figure 1 illustrates how FBTs are created and used. The top row shows how an input texture map and features for the map are combined into the FBT. Unusable samples from the input texture map are automatically discarded. Each FBT texel stores features and texture samples. Features are represented as line segments or, for higher quality, splines.

The middle row shows how FBTs are rendered. As in standard texture mapping, the texture value of a pixel is bilinearly interpolated from nearby texture samples. However, in FBTs, interpolated samples always lie on the same side of all features. Thus interpolation reproduces smooth gradients while maintaining sharp features.

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 bilinearly interpolated raster image output 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 3 gives an overview of feature-based textures and how they are used for rendering. Section 4 describes in more detail how the FBT representation is constructed to support efficient texture lookup. Section 5 describes how the FBT is

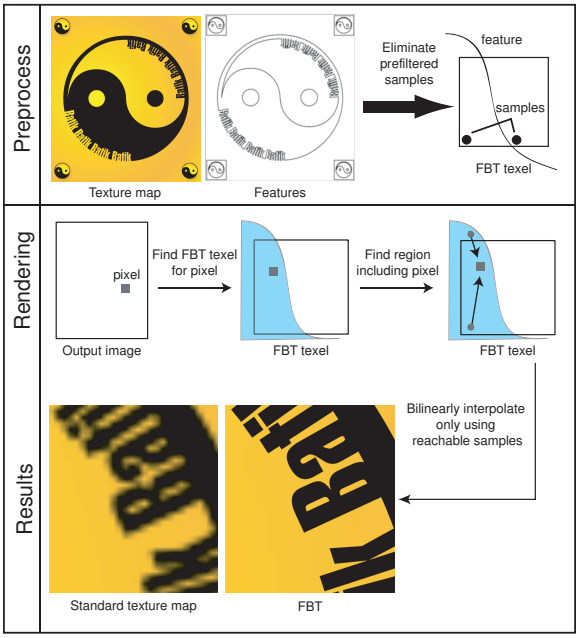

Figure 1: *Feature-based textures. Top row: FBT combines features and texture samples. Middle row: Pixel is rendered by interpolating reachable samples from adjacent FBT texels. Bottom row: FBT captures sharp features unlike standard bilinear interpolation.*

used in rendering textures. Section 6 presents results from a software implementation of FBTs, including a comparison of image quality and performance for FBT rendering versus texture mapping. Finally, we conclude and discuss future work in Sections 7 and 8.

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) [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. Raster images can be integrated in these formats for more visual complexity but then 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 accurately 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. We 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.

There is also a lot of relevant work in characterizing image features. 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. Our goal is different; we explicitly use features to improve the quality of the rendered result. The technique we present is perhaps most closely 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], but the focus there is on compression and fundamental image representations, not a mechanism for point 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 representation with piecewise linear discontinuities for resolution-independent pen-and-ink rendering. Our FBT representation captures both edges and curved features and is demonstrated for both vector formats and raster images. Because we focus on texture mapping, we demonstrate support for fast point queries and bilinear interpolation. and our technique must be more careful with reconstruction errors (which could be masked in an NPR setting).

3. Feature-Based Textures

Like a standard texture map, a feature-based texture 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 the leftmost texel in the figure. Sampling from empty texels is no more expensive than an ordinary texture lookup.

3.1. Features

Features characterize high-contrast changes in the input textures. A feature is represented by a connected chain of line segments or splines (Bezier curves in our implementation). We refer to each line segment or spline in a feature as a *subfeature*. Let us assume for now that the features and FBT resolution are both specified.

3.2. Texel features and regions

One of our goals is to have a representation general enough to reproduce textures with any configuration of features. For this reason, care is taken to accurately store all interactions between features and individual texels. We use the term *texel feature* to refer to a part of a feature intersecting a single texel. A feature that enters and exits a texel multiple times is split into multiple texel features so that each one enters and exits the texel at most once.

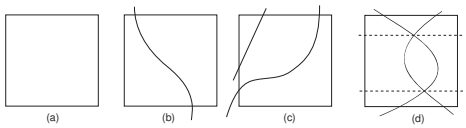

Figure 2: *Example texels, texel features and texel regions.*

As shown in Figure 2, a texel is divided into disjoint *regions* by the various features that cross it. Each of these regions contains exactly one sample, which can be used when interpolating texture values. For compactness, the location of the sample within the texel is not recorded; it is assumed to be at the bottom left of the region.

3.3. 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 nearby samples in region r and in 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 feature-based textures. 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 requires quickly locating the region that includes a given point. A simple implementation of this operation is to test which side of each feature the point lies on. However, this does not provide enough information in the (atypical) case where features intersect each other within a texel. For example, Figure 2-(d) illustrates two intersecting splines that split a texel into five regions. To handle such a texel, each intersecting feature is split at the point of intersection, and the adjacent regions on each side are divided into smaller regions by horizontal lines. This subdivision is combined with the simpler test to determine the containing region. See Section 4.5 for details.

Finding samples for interpolation

Step 4 requires 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-(b)).

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 can be used only if the location of the sample is not separated from the current point by a feature; otherwise, possibly erroneous interpolation would occur across this feature. Section 4.6 further explains how reachable samples are identified and interpolated.

4. Creating Feature-Based Textures

An FBT is a compact data structure that supports efficient, high-quality rendering. Some preprocessing is needed to make this possible, and this preprocessing depends to some extent on the kind of input being used to generate the FBT.

4.1. Input Specification

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

Finding features

Different types of input are amenable to different types of feature extraction. Features are found either through automated techniques or user input, as discussed below.

Vector-based representations. Vector-based representations can be queried directly to return all features. Tracing

programs [Sel] transform bitmaps into vector-based representations and could be used with manual intervention.

Manual specification. A user manually marks features to match the high contrast changes in the input texture. These features could be line segments or splines. The output of edge detection algorithms [Can87] could also be used to assist in this process. This user interaction is needed only once per input texture, and a library of FBTs can be reused by applications.

Finally, feature-aware procedural texturing algorithms could also directly output both features and samples.

FBT resolution

Because the FBT represents features explicitly, there is greater flexibility in choosing its 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 very simple solid-color SVG input only needs a few texels.

4.2. Feature processing

Each intersecting sub-feature (and its corresponding feature) is split at points of intersection. Line segments can be intersected trivially against each other. Intersection of splines and lines is also relatively straightfoward, requiring the use of a cubic solver. Robust spline-spline 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 for the entire image. It stores sub-features indexed on their positions. For example, each texel uses the kd-tree to find all of its texel features. The kd-tree is also used to accelerate the determination of feature intersections.

4.3. Invalidating prefiltered samples

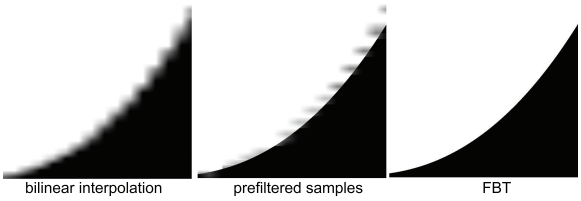

Figure 3: *Effect of prefiltered samples. Left: image produced by bilinearly interpolating texture samples from a raster texture map. Middle: using prefiltered samples causes artifacts. Right: eliminating prefiltered samples produces correct, high-quality rendering.*

When constructing an FBT from a raster image, we treat most texture samples as plausible point samples because they are in smooth regions. However, texture samples that lie close to features are often 'prefiltered' by the imaging device used to capture them. For example, most cameras have some transfer function that filters all incoming light through a pixel (and nearby pixels). If these prefiltered samples remain in the FBT, it will violate the invariant that texture lookup is only done from samples on the same side of features, causing rendering artifacts (as in Figure 3, middle).

Therefore, all samples that lie within some distance of any feature should be eliminated. If the imaging device producing the texture map is known, the invalidation distance can be set based on the characteristics of the device. Often, however, the camera is not known, so the user can specify this value explicitly. Typically a (∞-norm) distance of 1 pixel unit in the original texture image suffices. Eliminating filtered samples improves reconstruction during rendering (Figure 3)

4.4. Filling holes

The invalidation process described above can create holes in the texture with no sample. To fill these holes, the remaining samples are copied to nearby regions as needed. The invariant maintained is that a sample can only fill a hole if there is no feature blocking the sample from reaching the hole; i.e., the sample and hole are on the same side of all features.

In general, since features are composed of splines and line segments, texel regions can have complex boundaries. To fill holes we need a way to partition space in the texture. The constrained Delauney triangluation used in [SALS96] is limited to line segments; we use a different technique that handles curves as well. All splines are split into monotonic segments by finding critical points, and a variant of the trapezoid decomposition [O'R93] is used to divide each individual texel into thin horizontal strips of simple 4-sided subregions. The upper and lower boundaries of the strips coincide with all the start and end points of monotonic spline segments in the texel; thus, 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 robustly handle sub-features that are exactly horizontal. Figure 4 shows the sub-regions computed for the texel on the right. While this representation is not compact, it is only temporary; once used, it is discarded in favor of the FBT representation.

Once the sub-regions are constructed, they are used in hole-filling as follows. Each sample propagates in breadth-first order outward from its sub-region. Features block propagation of samples across them. Each sample is permitted to propagate for some maximum distance (2 texels in our implementation). This operation typically fills all holes; the cutoff distance prevents samples from flooding distant parts of the texture that are visually unrelated. If further holes remain, they are filled by searching from the hole (using a

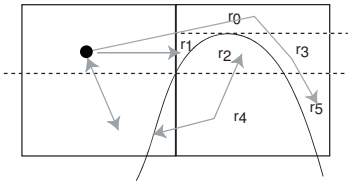

Figure 4: *Intermediate representation for hole filling. A dashed line is drawn at the curve's critical point, and its intersection with the middle texel boundary, splitting the right texel into three strips. Samples propagate in breadth-first order through sub-regions that are not blocked by features. The sample in the left texel propagates to the top part of the right texel as indicated by the arrows. The part under the spline in the right texel does not receive the sample.*

similar breadth first search) till a reachable sample is found. This second search happens rarely, only where there are very close features in the original image. After these two operations are done, all holes are filled.

4.5. Region testing

To perform efficient texture lookup during rendering, a fast test is needed to determine which texel region a pixel lies in. Let us consider how a texel feature affects the regions in a texel. (Note that we are assuming that feature intersections have already been handled by forming horizontal bands.) If the feature does not intersect any other feature, it divides the band into two regions, which can be arbitrarily called "inside" and "outside". Figure 5 shows various features and the regions they create. In the figure, each feature intersects the left boundary and one of the other four boundaries.

Consider a point "inside" the feature. Depending on how the feature intersects the boundary of the texel, a ray is shot from the point in one of four directions: $(\hat{x}, -\hat{x}, \hat{y}, -\hat{y})$. The parity of the number of intersections (odd or even) made by the ray from the point with the feature determines if the point is inside the region or not. For example, the features shown in Figure 5 are tested in the directions shown. The rule in determining the direction for testing a feature is that the direction should not match the boundaries that intersect the feature. Note that we are assuming that all features enter and exit a texel. Partial features either terminate at an intersection point (in which case they are handled) or 'float' inside the texel, in which case they do not partition the texel into two regions, and are ignored. Closed loops still separate a band into an "inside" and "outside", so they are marked as loops and are otherwise unaffected by the algorithm.

4.6. Texture lookup with interpolation

As described in 3.3, we would like to use bilinear interpolation to capture smooth texture shading. In a traditional texture lookup, the four samples nearest to a point are bilinearly

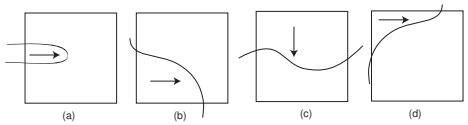

Figure 5: *Region determination within a horizontal band.*

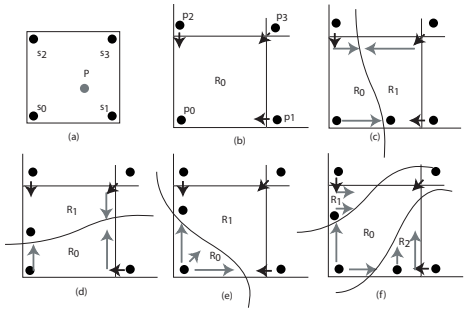

Figure 6: *Bilinear interpolation using neighboring reachable samples.*

interpolated. Let the texture sample at the point p be denoted by t_p and let the four nearby samples be s_i, with the weights from bilinear interpolation w_i. Then, $t_P = \sum_i w_i s_i$.

To achieve a visual effect similar to bilinear interpolation we could store four samples at the corners of each region of a texel, as in Figure 6-(a). However, this would cause roughly a four-fold increase in memory usage. Our goal is to interpolate texture values but only store one sample per region, like regular texture maps.

Another complication that arises is that regions in the FBT are irregularly shaped and accurate computation of interpolation weights could be prohibitively expensive. Instead our reconstruction is performed as if the four samples for any region lie at the four corners of the texel.

In choosing samples for interpolation, we would like to use the four nearest samples. To avoid creating artifacts, we must restrict ourselves to reachable samples only. Figure 6 shows some of the possible configurations of texels and the samples used for bilinear interpolation in each configuration.

An FBT stores only one sample per texel region; this sample is associated with the bottom left corner of the region (s_0 in Figure 6-(a)). As depicted in the figure, the remaining three samples are computed from adjacent texels while maintaining the invariant that they lie on the same side of the feature. For simplicity, the only nearby samples used are samples found at the lower-left corner of adjacent texels.

The reconstruction algorithm tries to preserve gradients across texel boundaries that have no features intersecting them; for example, the left and right boundaries in Figure 6-(c). For each region, the samples used in the locations of s_0, s_1, s_2, and s_3 are determined from the samples stored in the

region p_0 and the samples from the 3 neighboring texels: p_1 from the right texel, p_2 from the top texel, and p_3 from the top-right texel.

In Figure 6-(c), consider a point in region R_0. Bilinear interpolation uses the bottom-left sample of the region and the sample of the top pixel; i.e. $s_0 = p_0$ and $s_2 = p_2$. Each of these samples is also "copied" to the other two locations (shown along the grey arrows) to get the four "corner" samples that are bilinearly interpolated ($s_1 = p_0$ and $s_3 = p_2$). This rule preserves the gradient across the vertical left edge of the texel. A similar rule is followed to preserve the gradient along the horizontal edges in Figure 6-(d).

In the final FBT data, the choice of interpolant samples for each texel region is encoded in 1 byte: 2 bits for each mapped value. Once all holes are filled and reachability computed, sub-regions are collapsed and their samples merged through averaging. Horizontal bands remain wherever features intersect; all other bands along the y-axis are eliminated. For example, in Figure 4, r_0, r_1, r_3, and r_5 are merged into one region in the texel on the right. Similarly r_2 and r_4 are merged in the final representation.

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 4 control points, while bounded line segments are represented by 2 points.

Each texel stores an array of horizontal bands, which stores an ordered list of texel features. Each texel 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 are the values of the distance along a sub-feature when it enters or exits the texel (the t value for a spline). Together, this information is sufficient to find the chain of sub-features comprising the texel feature. Additionally, each texel feature uses 2 bits to denote which direction a ray should be intersected with the feature during region determination. In total, each texel 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 Feature-Based Textures

The previous section discussed the preprocessing necessary to construct the FBT. We now discuss how FBTs support efficient rendering, focusing on the two steps from Section 3.3 that differ from ordinary texture maps. Given a point that is mapped to some texture coordinates, the first step is to identify the region it falls into, and the second step is to find samples reachable from that point without crossing any features.

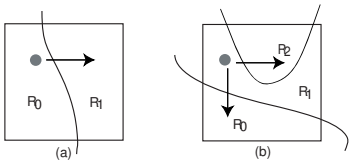

Figure 7: *Finding the region including a point in a texel.*

5.1. Finding the FBT region for a point (Step 3)

Given a 2D point p that lies in a texel t, we want to efficiently find the region of the texel the point lies in. Within each texel the y-coordinate of p determines the horizontal band to search. Each band stores an ordered list of texel features against which p is tested. The parity of the number of intersections (odd or even) determines if p lies inside or outside the region. If the point is determined to be outside the region delimited by a feature, the point location is tested against the next texel feature in the list.

Figure 7 depicts this test for some texels. In Figure 7-(a), the point is tested against the stored feature by tracing a ray along $\hat{x}$. If p is in R_0, one intersection will be found; otherwise no intersection will be found. In Figure 7-(b) an ordered list of two features is stored in the texel, creating 3 regions. The test associated with the first feature is along $-\hat{y}$. If there is no intersection, the point is determined to be in R_0. If there is an intersection, the second feature must be tested. Now a ray is shot from p along $\hat{x}$. If there are 0 or 2 intersections, p is in R_1, otherwise it is in R_2.

Intersecting a ray with features is fast. For a line segment, the test is straightforward. For splines, the intersection can be directly computed by solving the cubic, which is slow. To eliminate unnecessary cubic solving we first test the intersection of the ray with the spline convex hull. If the point is inside, the cubic solver is invoked. If the point is outside, the even-odd test for the spline can be deduced from the test for the convex hull, so we are done.

5.2. Finding reachable samples (Step 4)

Once the region has been identified, its 1-byte sample availability mask encodes which neighboring samples to use for bilinear interpolation. These four samples are then bilinearly interpolated as described in Section 4.6.

6. Results

In this section we present results comparing FBTs to regular 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 raytracer context, using 4 point samples per pixel. During rendering the use of FBTs imposes no noticeable performance overhead over ordinary texture maps.

Figure 8: *Example inputs and their corresponding features.*

Two types of input textures were used for this evaluation: SVGs and raster images. Figure 8 shows some example textures 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 texture samples and for feature extraction; the yin yang example is also from Batik. For the flower, stained-glass, and wizard skin raster texture maps, we manually annotated the image with edges. The banana example was annotated with splines obtained using Potrace.

The wizard skin example is used primarily to illustrate potential applications in games; it is no different from the other raster images, and thus it does not appear in any of the memory / performance tables.

6.1. Memory Comparisons

As mentioned earlier, the user can choose the appropriate FBT resolution for each texture map. To make comparisons fair, ordinary texture maps are used that consume strictly more memory than the FBT being compared against. Table 1 shows the memory usage for the two SVG examples.

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

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

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 9-(a) would require approximately 41 MB. Similarly, the zoomed-in stop sign in Figure 10 could be rendered at the same quality as the FBT output if the stop sign texture map used 3MB.

In the raster texture map examples, our goal was to annotate an existing image with extra sharpness and detail, providing higher quality at increased magnification. To retain all of the information in the source texture maps, 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.

6.2. Quality Comparisons

Figure 9 compares several image reconstruction methods. The highest-quality rendering, shown in Figure 9-(a), is the SVG rendering of the image. Figure 9-(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 regular texture maps (right side) generate output of lower quality. Given the poor output of point sampling (c), for the rest of the results we only compare FBTs with bilinearly interpolated texture maps (d).

Figure 10 shows the stop sign comparison. At high magnification, the FBT faithfully reconstructs the image; the raster texture map exhibits significant artifacts.

Figure 11 compares results of FBT rendering vs. bilinear interpolation from raster texture maps. In each case, features have been added to increase the sharpness and contrast between different regions of the image. While our system supports splines fully (as shown by the raster banana texture map and SVG results), one can see from the flower and stained glass images that considerable sharpness can be added simply by using line segments alone, which is advantageous when considering a GPU implementation of this representation.

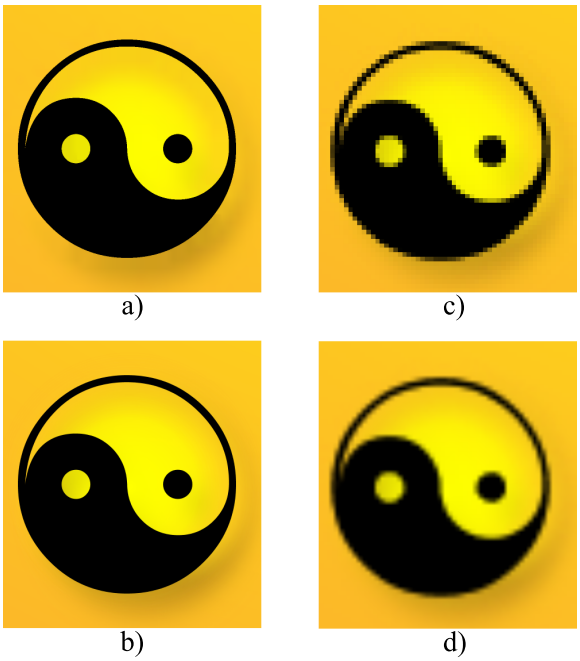

Figure 9: *Reconstruction of lower left corner of Yinyang image using (a) Vector-based SVG rendering; (b) 230×256 FBT; (c) 460×512 texture map with point sampling; (d) 460×512 texture map with bilinear sampling.*

Figure 10: *Stop sign quality comparison. Left: with FBT; Right: with raster texture map.*

Texture mapping a 3D model: To demonstrate our results on a 3D model we acquired a skinned, low-polygon-count wizard model from the game Warcraft 3 (Figure 12). This is a particularly appropriate example because the game displays this model both in closeup and at a distance. The skin was annotated with features along the runes of the cloak and hood, resulting in significantly improved sharpness. The efficacy of FBTs is indicated by the quality contrast between the runes, where we added features, and the hem of the cloak, where we did not.

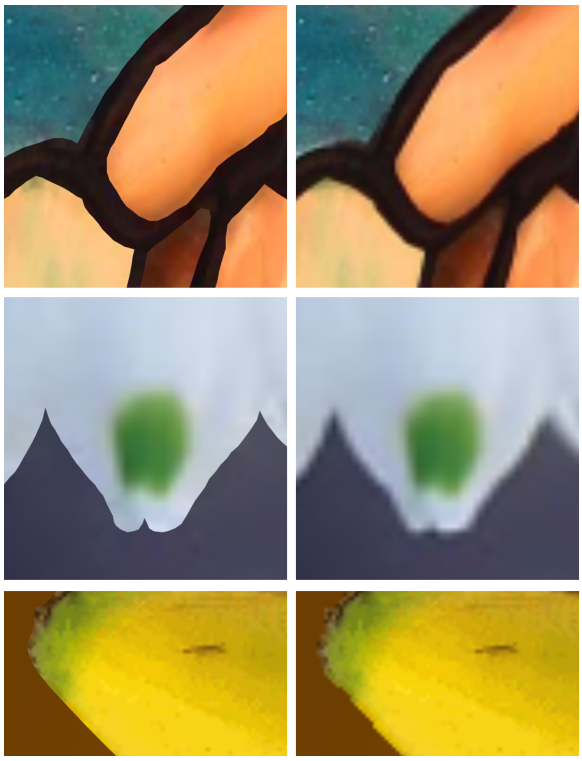

Figure 11: *Comparisons of the stained glass, flower, and banana. Left: with FBT; Right: with original texture map. The stained glass and flower were annotated by hand strictly using straight edges; the banana was annotated by smooth splines obtained from potrace.*

6.3. Performance

The FBT representation is designed to mimic a raster 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).

To analyze cost, we are interested in the number of ray-spline 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 spline'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 splines considered in

Figure 12: *Wizard model from the game Warcraft 3. Left: Antialiased rendering of model using an FBT skin. Right: zoomed-in comparison (Top: FBT; Bottom: original raster skin).*

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.*

each query, and f_{test} is the fraction of splines 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.

7. Discussion and Future Work

In this section we discuss issues that arise when using FBTs. One issue is that not all types of image discontinuities can be modeled accurately using sharp features. In vector graphics this is not a problem, but in raster images glaringly sharp boundaries may look flat or cut-out. This problem can potentially be alleviated by introducing different functions for discontinuity reconstruction.

FBTs currently support point queries as a basic mechanism for texture lookup; however, for anti-aliased quality, this requires supersampling the texture or using features with an edge-antialiasing rendering system such as [BWG03]. Exploring more sohphisticated anti-aliasing mechanisms would be interesting. Another issue is that of texture quality when zooming out of the texture. MIP-mapping of textures using features is an open question that could require investigation into multi-resolution feature representations. We believe this is an important area of future research that will provide insight into antialiasing techniques as well.

Each FBT texel region stores one representative sample. Therefore, it is not possible to respect two smooth gradients in a single region. This could create small blocking artifacts, but these are typically not noticeable using a large enough FBT. Some artifacts can arise during sample propagation because holes are filled with other samples. Even with a cut-off distance to prevent samples from filling remote regions, some smearing may be visible under magnification. We have not found this to be a major problem, but an alternative is to use pixel-based texture synthesis to fill holes.

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 on representing only edges and restricts the number of features per texel to a small number. Our results suggest that this is possible because most FBTs texels are empty. While the stop sign FBT is an exception because of its low resolution, for other textures, more than 99% of the texels have fewer than 3 regions.

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: *Spline test data for 500×500 renderings of the full example images and zoomed in images shown in the figures. The stained glass and flower are not included because they contain no spline features. Zooming in on complicated regions increases the number of cubic tests per query, but not significantly.*

8. Conclusions

This paper introduces a texture representation that combines features and texture samples for high-quality texture map-

ping. The FBT is a compact representation that permits efficient texture lookups. Samples are interpolated from the appropriate regions of FBT texels while accurately preserving features. We have demonstrated the use of FBTs for rendering a range of images with high image 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. The point-sampling interface supported by FBTs make them directly applicable to ray tracers and software scanline renderers. There is important research to be done in filtered sampling and minification with discontinity-based representations (an FBT analog to MIP mapping). Investigating a GPU-based implementation of FBTs also is an interesting area of future research. Finally, we would like to experiment with different characterizations of discontinuities so that we may more accurately reconstruct a wider class of images.

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, 4

[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-samples 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

[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

[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

[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, 4

[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. www.w3.org/TR/SVG11/. 2

[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