# Vector Regression Functions for Texture Compression

YING SONG
Zhejiang Sci-Tech University and Chinese Academy of Sciences
JIAPING WANG
Aiur
LI-YI WEI
Dragoniac
and
WENCHEN WANG
Chinese Academy of Sciences

Raster images are the standard format for texture mapping, but they suffer from limited resolution. Vector graphics are resolution-independent but are less general and more difficult to implement on a GPU. We propose a hybrid representation called vector regression functions (VRFs), which compactly approximate any point-sampled image and support GPU texture mapping, including random access and filtering operations. Unlike standard GPU texture compression, VRFs provide a variable-rate encoding in which piecewise smooth regions compress to the square root of the original size. Our key idea is to represent images using the *multilayer perceptron*, allowing general encoding via regression and efficient decoding via a simple GPU pixel shader. We also propose a content-aware spatial partitioning scheme to reduce the complexity of the neural network model. We demonstrate benefits of our method including its quality, size, and runtime speed.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Color, shading, shadowing, and texture*; I.4.2 [**Image Processing and Computer Vision**]: Compression (Coding)—*Approximate methods*

General Terms: Algorithms, Theory, Experimentation

Additional Key Words and Phrases: Texture compression, vector graphics, neural network, machine learning, real-time rendering, graphics hardware

**ACM Reference Format:**

Ying Song, Jiaping Wang, Li-Yi Wei, and Wenchen Wang. 2015. Vector regression functions for texture compression. ACM Trans. Graph. 35, 1, Article 5 (December 2015), 10 pages.
DOI: http://dx.doi.org/10.1145/2818996

Corresponding author: Y. Song; email: yingsong@outlook.com.

0730-0301/2015/12-ART5 $15.00
DOI: http://dx.doi.org/10.1145/2818996

## 1. INTRODUCTION

Texture mapping is a core component for image synthesis [Heckbert 1986]. The standard format is a raster image which is general enough to support any content and simple enough for random access and fast filtering. However, the detail supported by a raster image is determined by its resolution. Obtaining enough detail everywhere might require extremely large images which compete for space, especially in limited GPU memory. Vector graphics provide edge detail at any magnification but are less general and more complex to encode, evaluate, and filter. Despite recent efforts [Qin et al. 2008; Nehab and Hoppe 2008; Sun et al. 2012; Ganacim et al. 2014], raster images remain the standard for texture mapping, especially in GPU implementations.

We seek a texture map representation that combines the advantages of raster images and vector graphics in order to yield a compact encoding for general content with fast runtime decoding and filtering.

As a step towards this goal, we propose *vector regression functions* (VRFs) which map 2D texture coordinates to 3D RGB colors. VRFs represent any point-sampled input and support GPU texture mapping, including random access and fast filtering. Standard GPU texture compression yields a constant compression ratio (defined as $\gamma = |\mathcal{I}|/|\phi|$, the ratio of original to compressed data size); VRFs are variable rate and can achieve greater compression. In particular, they obtain $\gamma \propto \sqrt{|\mathcal{I}|}$ for inputs consisting of piecewise smooth regions as in vector graphics. On the other hand, VRFs do not offer "infinite zoom-in" as with vector graphics, since they cannot preserve sharp edges with infinite magnification in most cases. However, VRFs still suffice in practical situations in which infinite zoom-in is rarely required.

Our key idea is to represent VRFs as neural networks, allowing general encoding via machine learning and efficient decoding via simple GPU pixel shaders. The benefits of this regression methodology have been demonstrated in prior graphics applications such as animation [Grzeszczuk et al. 1998], visibility [Nowrouzezahrai et al. 2009; Dachsbacher 2011], and global illumination [Ren et al. 2013]. For piecewise smooth inputs, our method detects the region boundaries and encodes the output with proportional complexity, leading to the square-root compression ratio.

Given an input 2D image $\mathcal{I}$, we seek a VRF $\Phi$ to approximate $\mathcal{I}$ based on a set of sample locations $\mathcal{S}$ on $\mathcal{I}$. We assume $\mathcal{I}$ is well represented by specifying its values on $\mathcal{S}$, but otherwise arbitrary. The function $\Phi$ supports efficient random access and filtering for runtime evaluation at any sample location. We investigate the multilayer perceptron (MLP) [Hinton 1989] as the function family to

Fig. 1. Results of texturing over curved surfaces with vector regression functions.

represent $\Phi$ and describe the corresponding encoding/training and decoding/rendering methods. We also propose a content-aware spatial partitioning scheme to reduce the complexity of $\Phi$ for inputs having complex features.

VRFs are best suited for piecewise smooth imagery, as common in vector graphics art and some natural photographs, and less suited for images having high-frequency patterns. For the former, our method offers square-root compression ratio, and for the latter, gracefully degrades to a constant compression ratio similar to the current block-based texture compression standard. Encoding and decoding our representation are slower than current GPU texture compression methods but still fast enough for real-time applications that demand high-quality MIPMAP and anisotropic filtering. See Figure 1 for an example of texturing over curved surfaces with VRFs.

## 2. PREVIOUS WORK

Texture mapping can be achieved by raster images or vector graphics. We brief prior art most relevant to our work.

*Raster Textures.* Raster images are the standard format for texture mapping. To ensure high quality as well as low memory and bandwidth consumption, significant efforts have been spent on compression and filtering for GPU texture mapping.

The current industry standard for texture compression is block-based, a simple fixed-rate compression scheme that is random-accessible and amenable for implementation as hardware units [Iourcha et al. 1999; Ström and Akenine-Möller 2005; OpenGL ARB 2010; Nystad et al. 2012]. Variable bit-rate compression can offer higher quality-to-size ratio but tends to be more complex to decode; thus, existing methods such as that of Olano et al. [2011] are more suitable for batch loading than random access.

The current industry standard for texture filtering includes bilinear, MIPMAP [Williams 1983] and anisotropic [McCormack et al. 1999; Mavridis and Papaioannou 2011] schemes. These are more effective for minification, as magnification can still manifest visible pixelization effects.

Even though compression and filtering can help increase effective quality and resolution under the same size, they do not change the fundamental resolution limits of raster textures.

VRF can be considered as a form of variable-rate texture compression but is simple to implement and supports MIPMAP and anisotropic filtering for minification similar to fixed-rate texture compression. However, it offers crisp resolution under magnification similar to vector graphics and offers much better compression ratio, usually square root instead of constant fixed rate compression for piecewise smooth inputs.

*Vector graphics.* A major benefit of vector graphics is resolution independence—infinite resolution upon magnification under the same data size. Gradient mesh [Lecot and Levy 2006; Sun et al. 2007; Lai et al. 2009; Xia et al. 2009] and diffusion curve [Orzan et al. 2008; Finch et al. 2011; Ilbery et al. 2013; Sun et al. 2014; Xie et al. 2014] have emerged as two popular forms of recent vector graphics development. These offer different trade-offs: the former tends to be denser, easier to render, and more suitable for automatic conversion from raster images, while the latter tends to be sparser and more suitable for manual editing. Boyé et al. [2012] combine the advantages of both camps by using meshes as the internal representation for rendering and curves as the external interface for authoring. These representations were originally intended for 2D instead of 3D graphics. In particular, rendering diffusion curves involves solving a system of equations, and even though some of these methods can be accelerated by GPU rasterization, they are not suitable for general 3D texture mapping due to the lack of random accessibility and filtering support.

Direct support of random access and filtering for vector graphics appears to be a very challenging problem. One possible solution is to carefully design a rasterizing method, as in Jeschke et al. [2009], but this is not exact random-accessible or resolution-independent. Few methods offer closed-form formula for random access and area filtering, such as Sun et al. [2012], but tend to have limited scopes (e.g., accurate only for closed diffusion curves).

Hybrid representations that register vector primitives into acceleration data structures (e.g., grid or quad-tree) appear to be the dominant methodology to support vector texture mapping. This can be achieved by storing either a fixed (e.g., [Sen 2004; Tumblin and Choudhury 2004; Tarini and Cignoni 2005; Qin et al. 2008; Parilov and Zorin 2008]) or variable (e.g., [Nehab and Hoppe 2008]) number of primitives per grid cell and/or using an adaptive structure like quad-tree [Ganacim et al. 2014]. Unlike these methods, VRF does not offer infinite resolution but can handle images that are not vector graphics, such as real photographs with smooth regions.

*Neural networks for image compression and superresolution.* There are prior works that apply neural networks for image compression; Jiang [1999] provides a good survey. These methods are based on a blockwise scheme, which might not be efficient for high resolution images and filtering. Our method is also based on neural networks but not a block-based scheme. It is random-accessible and supports filtering.

Neural networks can also be used in image superresolution to learn an end-to-end mapping between the low- and high-resolution images [Dong et al. 2015]. This is a different problem from image compression, as the high-resolution image is usually unknown and the decoding does not have to be in real time. In contrast to

superresolution, our method has access to the original high-resolution image and requires real-time decoding.

## 3. OUR METHOD

Here we present details of our vector regression function (VRF), including representation (Section 3.1), training/encoding (Section 3.2), and rendering/decoding (Section 3.4). Orthogonal to those, we also describe how to scale up VRF for complex inputs via a spatial partition scheme (Section 3.3).

For clarity, our presentation focuses on 2D RGB images ($R^2 \rightarrow R^3$), even though our method applies to different input and output dimensions (e.g., volume textures or monochrome images).

### 3.1 Representation

The approximation of an input image $\mathcal{I}$ by a VRF $\Phi$ can be treated as a regression problem. Given an input 2D image $\mathcal{I}$, we seek a VRF $\Phi$ to minimize the following energy function based on a set of sample locations $\mathcal{S} = \{(x, y)\}$:

$$E = \sum_{(x,y)\in\mathcal{S}} \|\mathcal{I}(x, y) - \Phi(x, y)\|^2 . \quad (1)$$

Our training data consists of a set of N input-output pairs that sample $\mathcal{I}$. The $i$th pair comprises $(\mathbf{s}^i, \mathbf{t}^i)$, where $\mathbf{s}^i = (x_i, y_i)$ is the input sample position and $\mathbf{t}^i = \mathcal{I}(x_i, y_i)$ is the output value. Compression is achieved by designing the function $\Phi$ with a multilayer perceptron (MLP) [Hinton 1989] regression model, featured with the following benefits.

—*GPU-friendly* MLPs are random-accessible, have compact sizes, and can be evaluated efficiently.

—*Expressiveness.* Because of the nonlinear nature of MLPs, they are suitable and effective for capturing nonlinearities described by primitives of vector graphics.

An MLP is a weighted and directed graph whose nodes are organized in layers, as illustrated in Figure 2. Nodes in adjacent layers are fully connected by weighted edges. The weights of the edges constitute the components of the weight vector $\mathbf{w}$. We use an acyclic feed-forward network consisting of one input layer with two nodes for $(x, y)$, one output layer with three nodes for RGB, and several hidden layers with adjustable number of nodes $m$. In our implementation, we choose two hidden layers because they can approximate nonlinear structures in vector graphics, specifically, continuous or square-integrable functions on finite domains, with arbitrary small errors [Hinton 1989]. We keep the same number of nodes for both hidden layers, as experimentally, we have found this works best for the images we tested.

Each node in a hidden layer takes inputs from the preceding layer and computes an output based on the weight vector $\mathbf{w}$. Consider node $j$ in the $i$th layer, with $n^i_j$ denoting its output and $w^i_{j0}$ as its bias weight. For $i$th hidden layer, $n^i_j$ is calculated from the outputs of all nodes in the $(i-1)$th layer as follows:

$$z^i_j = w^i_{j0} + \sum_{k=1}^{m} w^i_{jk} n^{i-1}_k, \quad (2)$$

$$n^i_j = \sigma(z^i_j), \quad (3)$$

where $\sigma$ is the hyperbolic tangent function $tanh(z) = 2/(1+e^{-2z}) - 1$. This formulation takes a weighted sum $z^i_j$ of all outputs from the preceding layer and produces the response with a nonlinear mapping. The nonlinear nature of MLP is embodied by $\sigma$. For nodes in the output layer, the final output $n^i_j$ is simply the sum $z^i_j$ from nodes in the last hidden layer without $\sigma$.

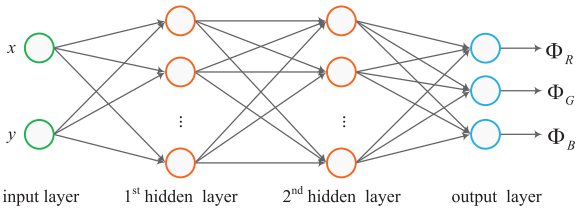
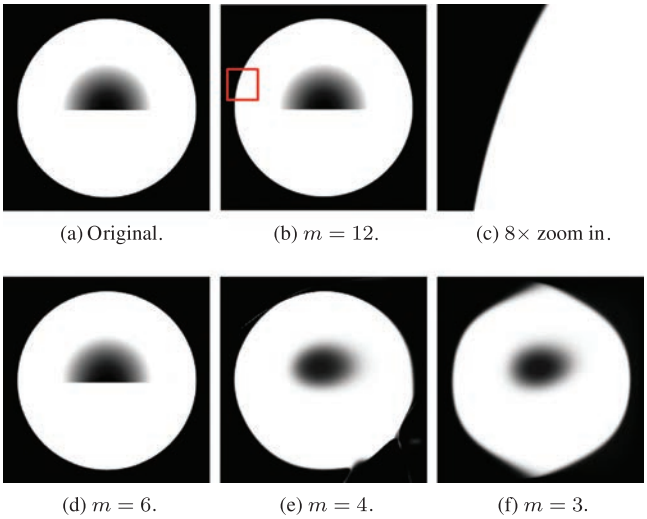



Fig. 2. Modeling the VRF by an acyclic feed-forward neural network. This defines a mapping from the input sample position $(x, y)$ to the output RGB values.



(a) Original. (b) $m = 12$. (c) $8\times$ zoom in.

(d) $m = 6$. (e) $m = 4$. (f) $m = 3$.

Fig. 3. The effects of $m$, the number of nodes per hidden layer. (a) The original input with resolution $256 \times 256$, while (b), (d), (e), and (f) are our results with different $m$. (c) $8\times$ magnification of the enclosed area in (b).

With this MLP representation, the VRF $\Phi(\mathbf{s}, \mathbf{w})$, where $\mathbf{w}$ is the weight vector of the MLP, can be determined by minimizing the energy function $E(\mathbf{w})$ as

$$\mathbf{w} = \arg\min E(\mathbf{w}) = \arg\min \sum_i \|\mathbf{t}^i - \Phi(\mathbf{s}^i, \mathbf{w})\|^2. \quad (4)$$

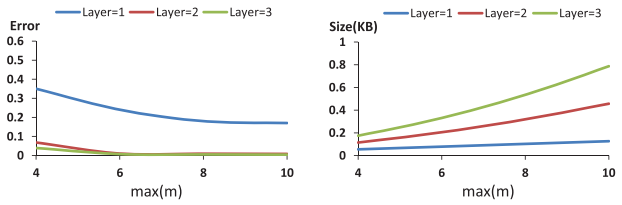
To find $\Phi$, we need to determine the structure of the neural network as well as its weight vector $\mathbf{w}$ by minimizing $E(\mathbf{w})$.

### 3.2 Training

*Discretization.* As described in Section 3.1, the input to our training phase consists of discrete input-output sample pairs $(\mathbf{s}^i, \mathbf{t}^i)$. These pairs can be obtained from sampling a vector graphics image or directly from a raster image. Either way, the input to our training phase is a sampled $\mathcal{I}$, and sufficient sampling resolution is required for the VRF $\Phi$ to be able to faithfully reconstruct $\mathcal{I}$. We ensure the resolution sufficiency by computing the ratio $\mu$ of *salient pixels* with significant local gradients $g(\mathbf{s}) > g_\epsilon$:

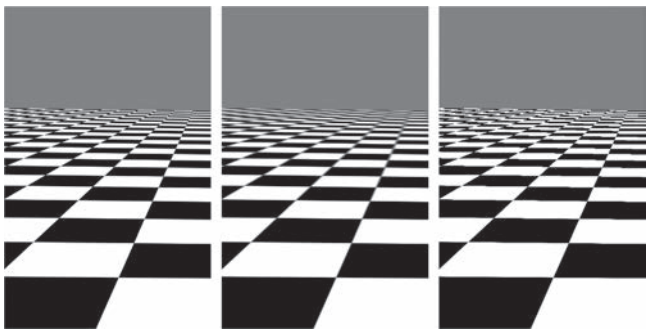
$$\mu = \frac{|\{\mathbf{s}\,|g(\mathbf{s}) > g_\epsilon\}|}{|\{\mathbf{s}\}|}, \quad (5)$$

where $g_\epsilon = 0.01 L_{avg}$ in our implementation, with $L_{avg}$ being the average luminance value of the rasterized $\mathcal{I}$. We define the local gradient $g(\mathbf{s})$ as the maximum absolute gradient towards all eight




(a) Training error. (b) Data size.

Fig. 4. The analysis of $m$ under different number of hidden layers. The results are computed from the same example in Figure 3.



(a) MIPMAP + aniso. (b) MIPMAP only. (c) No MIPMAP.

Fig. 5. Filtering with VRF. (a) MIPMAP + 7× anisotropic filtering rendered at 180Hz, (b) MIPMAP only at 1120Hz, (c) bilinear only without MIPMAP at 1540Hz.

(a) Original. (b) Salient pixels. (c) Regions.

Fig. 6. Partitioning. (a) Original input. (b) Locations of detected salient pixels across all resolutions. (c) Visualization of regions of the underlying k-d tree.

neighborhood pixels $\mathbf{n}_8$:

$$g(s) = \max_{\mathbf{s}' \in \mathbf{n}_8} \|\mathcal{I}(\mathbf{s}) - \mathcal{I}(\mathbf{s}')\|. \quad (6)$$

For a given vector input, we ensure its rasterized image has $\mu < 5\%$ by exponentially growing the resolution from a rough initial guess. For a given raster input, we use it directly. If the resolution is insufficient, our method may produce less satisfactory results as analyzed in Figures 8 and 14.

*Adaptive sampling*. The discretized input $\mathcal{I}$ may contain a large number of pixels which are time-consuming to train directly. We ameliorate this issue by only selecting pixels that are important for describing the image. Our basic idea is to build a Gaussian pyramid of $\mathcal{I}$ and select the set of salient pixels. Specifically, starting from the rasterized image with the resolution determined by the sufficiency criterion $\mu_0 < 5\%$ as $\mathcal{I}^0 = \mathcal{I}$, we accumulate salient pixel positions from each down-sized rasterized image $\mathcal{I}^i$ down to the image

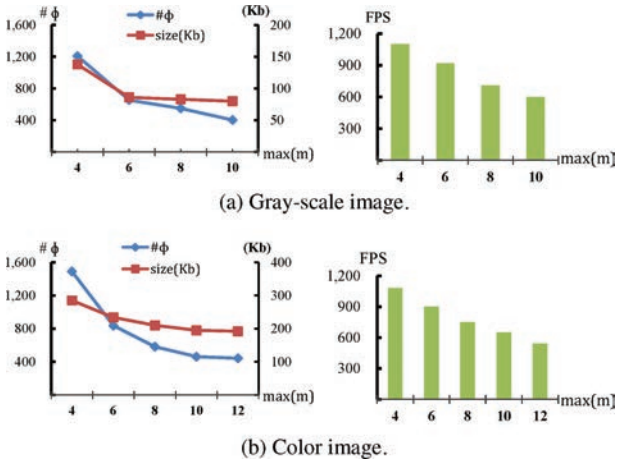



Fig. 7. Analysis of KD tree partition for both gray-scale (a) and color (b) cases. As $m_{max}$ increases, the number of Φs, VRF sizes, and rendering speed decrease.

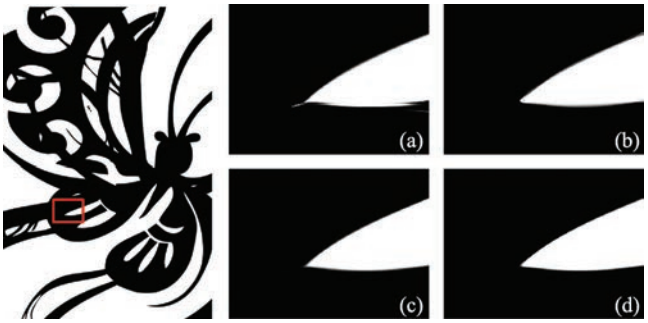



Fig. 8. Training resolution. Training with (a) $\mu = 15.8\%$, (b) $\mu = 8.8\%$, (c) $\mu = 4.9\%$, (d) $\mu = 2.6\%$.

Table I. Statistics

| Case | resolution | $\mu$ | $m_{max}$ | $d_{max}$ | #Φ |
|---|---|---|---|---|---|
| Butterfly | 1600 × 1536 | 4.9% | 6 | 16 | 633 |
| Car | 3192 × 1954 | 4.74% | 10 | 16 | 1270 |
| Zephyr | 1462 × 1462 | 4.72% | 10 | 16 | 948 |
| Flower | 1440 × 900* | 16.5% | 10 | 16 | 585 |

*Note*: Resolution for training, ratio of salient pixels $\mu$, maximum number of hidden nodes $m_{max}$ across all Φ functions, maximum depth of the partition tree $d_{max}$, and number of leaf nodes for k-d tree. The resolution of the flower case is the original resolution of the raster image.

$\mathcal{I}^f$ with $\mu_f > 30\%$. The final sample set $\tau = \bigcup_{k=0}^{f} (\mathbf{s}^i, \mathcal{I}^k(\mathbf{s}^i))$ includes accumulated salient pixels and all pixels in $\mathcal{I}^f$, whose positions $\{\mathbf{s}^i\}$ are uplifted to the original resolution of $\mathcal{I}$ with duplications removed.

*Optimization*. Given the number of nodes $m$ in each hidden layer and the training set, we follow Ren et al. [2013] to find the the weight $\mathbf{w}$ by applying the Levenberg-Marquardt algorithm [Hagan and Menhaj 1994] with a Jacobian matrix calculated based on back-propagation [Hinton 1989].

Table II. Comparison among ASTC, BC7, and VRF

| Case | size (KB) | | | RMS (%) | | | encoding time (min) | | | FPS | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | ASTC | BC7 | VRF | ASTC | BC7 | VRF | ASTC | BC7 | VRF | BC7 | VRF |
| Butterfly | 269 | 3277 | 86 | 1.80 | 0.16 | 0.48 | 2.3 | 0.9 | 60 | 1300 | 690 |
| Car | 678 | 8329 | 340 | 0.81 | 0.26 | 0.46 | 23 | 11 | 222 | 1240 | 360 |
| Zephyr | 233 | 5592 | 207 | 1.10 | 0.25 | 0.49 | 11 | 4 | 61 | 1290 | 420 |
| Flower | 169 | 1730 | 168 | 0.80 | 0.42 | 0.90 | 8 | 8 | 150 | 1300 | 340 |

*Note*: We could not measure ASTC framerate as it is not yet part of the official GPU standard.

The value of $m$ determines the capability of $\Phi$ for capturing the complexity of spatial variations in the image: the larger the $m$, the more complex image features can be represented. One the other hand, we need to minimize $m$ because of the quadratic cost increase and the risk of over-fitting [Vapnik 1995]. To find the right balance, we determine $m$ iteratively, starting from $m = 2$ and incrementing it by one until the training error $\epsilon$ is below a threshold $\Xi$. In particular,

$$\epsilon = \sqrt{\frac{\sum_i \|\mathbf{t}^i - \Phi(\mathbf{s}^i)\|^2}{\sum_i \|\mathbf{t}^i\|^2}}. \tag{7}$$

We set $\Xi$ between $0.5\%-1.0\%$. Figure 3 illustrates the visual effects of $m$. Figure 4 analyzes the expressiveness of various numbers of hidden layers and $m$. As shown, two hidden layers provide a good balance between training error and storage size.

*MIPMAP*. To build a MIPMAP version of $\Phi$, we add an extra level dimension $l$ to our energy formulation:

$$\sum_l \sum_{\mathbf{s}} \|\mathcal{I}(\mathbf{s}, l) - \Phi(\mathbf{s}, l)\|^2. \tag{8}$$

In particular, an additional input node is introduced for $l$. Similar to input nodes for $x$ and $y$, it is also fully connected to all nodes in the first hidden layer.

One option is to build a Gaussian pyramid from the rasterized $\mathcal{I}$ and train each output level as a separate function $\Phi(.,.,l_i)$ from the corresponding input level $\mathcal{I}(.,.,l_i)$ as previously described. However, through experiments, we have found it better to train a single output $\Phi$ for the entire pyramid, which yields a smaller model and faster evaluation—one direct fetch $\Phi$ with fractional level number other than two fetches of pyramid levels followed by linear interpolation.

Here is our training process. We first determine the set of training locations $\{\mathbf{s}^j\}$ based on the salient pixels of $\mathcal{I}$ as previously mentioned. We then build a Gaussian stack [Lefebvre and Hoppe 2005] from $\mathcal{I}$ and collect the training pairs from all levels using the same sample positions $\{\mathbf{s}^j\}$. In particular, the total training pairs are $\{(\mathbf{s}^j, l_i, \mathbf{t}^j_{l_i})\}$, where $\mathbf{t}^j_{l_i} = \mathcal{I}(\mathbf{s}^i, l_i)$. The training is conducted to optimize the following MIPMAP version of our energy function:

$$E(\mathbf{w}) = \sum_i \sum_j \|\mathbf{t}^j_{l_i} - \Phi(\mathbf{s}^j, l_i, \mathbf{w})\|^2. \tag{9}$$

Interestingly, even though we only train discrete levels, linear interpolation between output levels naturally works because of the coherence between successive input levels, as shown in Figure 5(b).

## 3.3 Partition

Using a single output $\Phi$ to represent a complex input $\mathcal{I}$ may require a large number of internal nodes. This can incur excessive computation time for both training and rendering, as the number of weights in $\mathbf{w}$ grows quadratically with the number of hidden nodes.

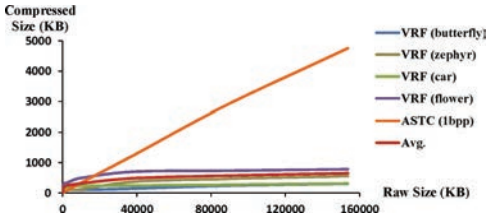



Fig. 9. Original versus compressed data sizes. ASTC has a constant compression ratio regardless of the input texture content and size/resolution. In contrast, VRF has a variable compression ratio roughly proportional to the square root of input data size.

We address this issue by partitioning the input $\mathcal{I}$ into multiple spatial regions and fitting a separate $\Phi$ for each region. We use a k-d tree as the hierarchical data structure, beginning with the entire $\mathcal{I}$ as the root node, and store a separate $\Phi$ at each leaf node. Within this context, the input $\mathcal{I}$ is represented by a VRF set, which consists of multiple $\Phi$ functions organized by the k-d tree. Each $\Phi$ is then trained individually with the subset of training pairs falling into its k-d tree region following Equation (4). We choose a k-d tree over a quad-tree due to fewer and more flexible partitions. The traversal of a k-d tree plus the evaluation of the corresponding $\Phi$ is usually much faster than evaluating a single big $\Phi$ without partition.

Next, we describe the construction of a k-d tree and how we maintain pattern continuity across tree cells.

*Construction*. For an input $\mathcal{I}$, we denote its k-d tree as $\Omega$. The $\Omega$ is constructed in a top-down manner to cover all training samples selected based on importance, as described in Section 3.2. From the root node, which covers the whole image, we recursively split each node if the training error $\epsilon$ of its corresponding $\Phi$ exceeds a threshold ($\Xi = 0.5\%$ as in Section 3.2) or its number of nodes per hidden layer $m$ exceeds a threshold $m_{max}$. In our current implementation, we set $m_{max} = 6$ for gray-scale textures and 10 for color textures, as detailed in the analysis part (Section 4.1). In each split, we always choose the longer dimension to divide in order to maintain good aspect ratios of the subdivided regions. The dividing point is optimized for best load balance between the two divided subregions so that they have similar numbers of salient pixels. We use binary search to find the point that minimizes the difference of the numbers of salient pixels in the two subregions. An example is shown in Figure 6.

*Continuity*. To ensure continuity across adjacent regions, we include additional training samples from neighboring regions if the content change is sufficiently smooth. In our experiments, we have found it sufficient to incorporate additional samples within a 2-pixel-wide border $\Psi$ along each dimension of the current region if the corresponding variation $\mathcal{I}_L$ of $\Psi$ is below a threshold 0.5. We can further reduce the discontinuity across the boundary of adjacent regions by creating a small transition zone for linear interpolation, even though we have not found it necessary.

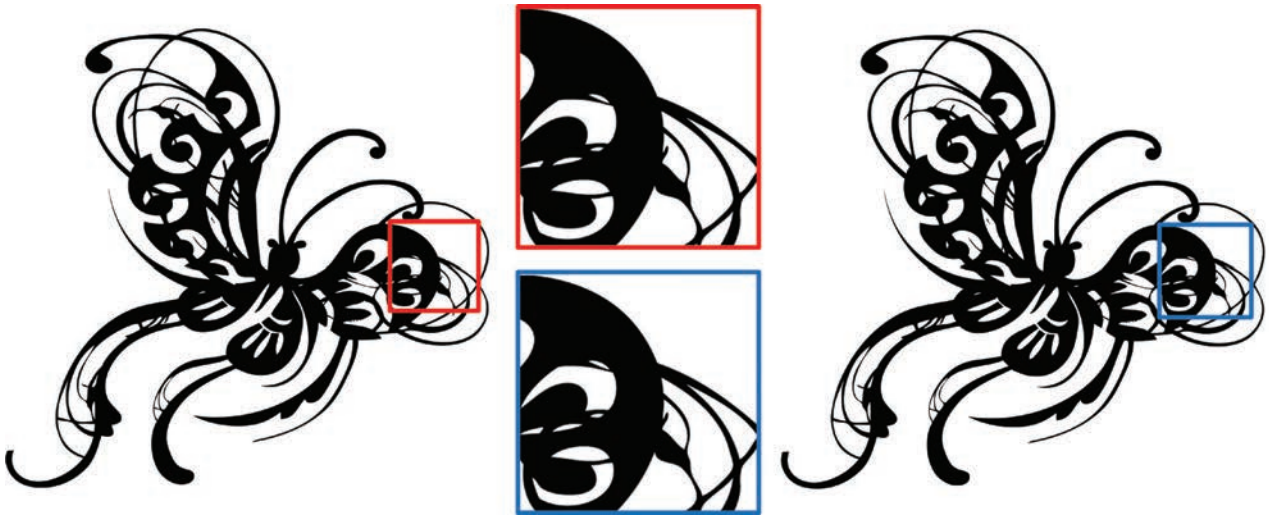


Fig. 10. VRF from a vector graphics. The left and right portions are the original input and our reconstructed result. The middle portion shows the 4× magnification of the corresponding regions for comparison.


Fig. 11. VRF from Sun et al. [2012]. (b) Reconstructed from a 1024 × 1024 image shown in (a). (c) Magnifications of the regions marked in (b). The first row in (c) is the reconstructed result of Sun et al. [2012] with resolution 4× zoomed in, while the second row is the reconstructed result from VRF with the same maginification.

### 3.4 Rendering on a GPU

Rendering with VRFs can be done purely in a pixel shader. For each query texture coordinate $(x, y)$, it first traverses the partition tree to locate the particular $\Phi$, and then evaluates the result as $\Phi(x, y)$.

Support for MIPMAP is straightforward by including the texture LOD parameter $l$ in the input for VRF evaluation, as formulated in Equation (8). Anisotropic filtering is done by fetching and combining multiple bilinearly filtered samples as in standard methods [McCormack et al. 1999; Mavridis and Papaioannou 2011], requiring multiple evaluation of the VRF. The quality impact due to MIPMAP and anisotropic filtering is shown in Figure 5.

We flatten the partition tree $\Omega$ as a linear array for storage. Each array element stores either offsets to child nodes (for internal nodes) or index to the corresponding $\Phi$ (for leaf nodes). We pack the weight vectors $\mathbf{w}$ of all $\Phi$ functions in a single floating-point texture. Due to different complexity of the image features in different partitions, the $\Phi$ functions can have different $m$ values. For better data layout and easier indexing, we pack dummy nodes with zero weights into each $\Phi$ so that they all have the same $m$ in storage. We observe speedup rendering performance at the cost of typically doubling the storage size because the memory alignment improves the efficiency of texture fetch.

## 4. EXPERIMENTAL RESULTS

We have implemented the training and partitioning algorithms on a CPU and the rendering algorithm on a GPU via OpenGL APIs and GLSL shading language. All results and performance measures shown here are conducted on a PC with 3.4GHz i7-4770 CPU, 8GB memory, and NVIDIA GTX 760 graphics card.

(a) Original. (b) Reconstruction.

Fig. 12. VRF from a complex vector graphics with rich details.

### 4.1 Parameters

*Partition*. There is a trade-off between the number of $\Phi$ functions and the complexity of each $\Phi$ within each VRF, as determined by the k-d tree partition. A simple k-d tree with a small number of $\Phi$ functions will be faster to traverse, but each $\Phi$ might contain more nodes and thus take longer to evaluate. To determine the best balance point, we have analyzed the partition granularity as shown in Figure 7. Even though different inputs may have different optimal configurations, we have found it a good rule of thumb to choose $m_{max} = 6$ for gray-scale textures and 8–10 for color textures.

*Resolution*. Our method needs sufficient input resolution to produce good output quality, as discussed in Section 3.2. Otherwise, discontinuity or high-frequency artifacts may show up for certain inputs under significant magnification. In Figure 8, we test VRF training result with a variety of input resolutions with different corresponding saliency ratio $\mu$. As shown, $\mu < 5\%$ produces good results.

### 4.2 Comparisons and Demonstrations

We have tested our method on inputs with different characteristics and complexities, including both vector graphics and raster images, as shown in Figures 10 to 13. In general, our method can successfully reproduce the original inputs and is compact and efficient for real-time rendering applications (hundreds KB storage and hundreds FPS). Table I shows detailed statistics of our method. Table II compares our method with two block compression methods: ASTC [Nystad et al. 2012] with state-of-the-art compression ratio but not yet part of official GPU standard, and BC7 [Microsoft Corporation 2013], a commercial standard method for GPU implementation (with 8 bpp bit rate). As shown, our method is more compact and maintains lower error for vector graphics inputs while remaining competitive for natural images. Our biggest disadvantage is encoding, as training VRF is more time-consuming than ASTC and BC7, whose block-based scheme is very simple to train.

Figure 9 illustrates the relationship between the storage sizes of VRF/ASTC and the resolutions of the input images. The training error threshold $\Xi$ is set to 0.5%. As shown, VRF offers a square-root compression ratio in contrast to the constant compression ratio of ASTC and the constant total size of typical vector graphics schemes (not shown in the figure).

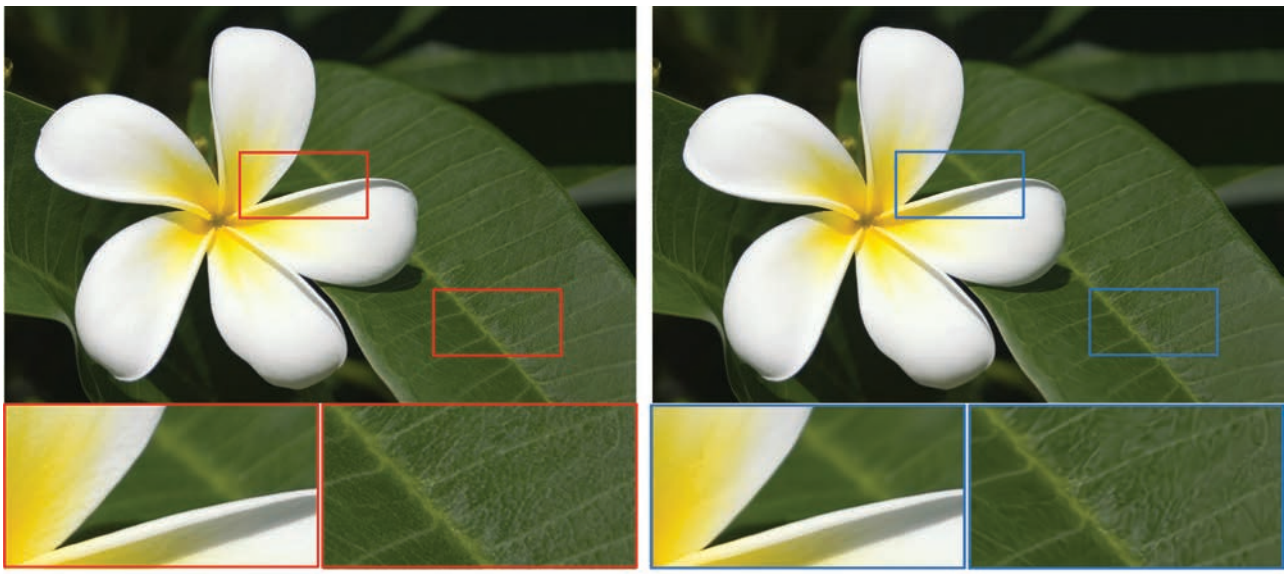
Figure 10 shows a result of vector graphics with sharp boundaries and thin curved features. The original vector graphics is represented with Bézier curves and solid interiors. Figure 11 demonstrates a diffusion curve input which exhibits complex nonlinear gradient regions with sharp boundaries. This case is demonstrated in Sun et al. [2012] with rendering speed at 27FPS. Our method, in contrast, achieves orders of magnitude speedup. Figure 12 is another complex vector graphics with both smooth and sharp features, such as the headlight and specular highlights in the car body, which our method can still successfully capture.

We also push the limit of our method to test on a nature image with both smooth and sharp features in Figure 13. As shown, our method can faithfully reconstruct the smooth flower petals and leaf textures as well as sharp petal boundaries and leaf veins. Figure 14 provides further tests on natural images from the standard "Kodak" image set [Franzen 1999]. As shown, our method can provide higher resolutions than existing methods. Similar to Figure 8, insufficient input resolution can cause our method to produce artifacts, in particular, sharp features appearing as smooth gradients upon sufficient zoom-in, as in Figure 14(e). However, these appear to be more visually pleasing than the pixelization artifacts produced by exiting methods.

## 5. LIMITATIONS AND FUTURE WORK

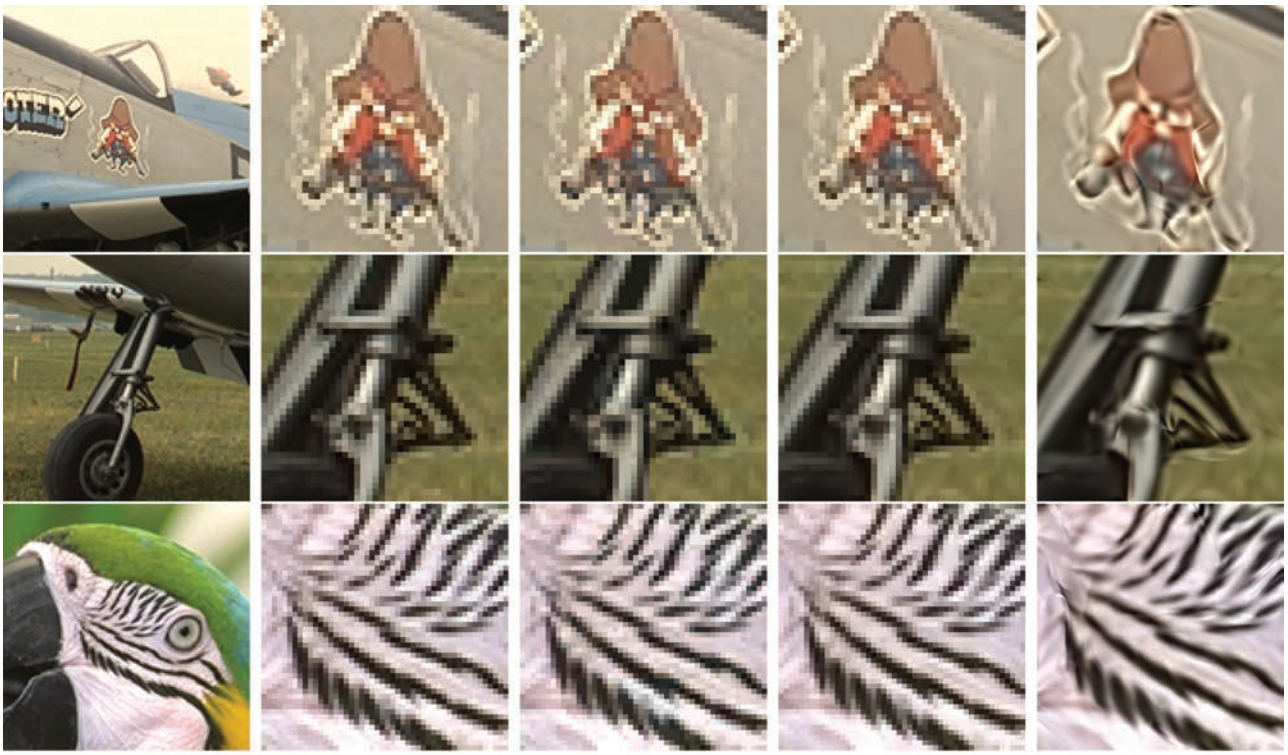
We have proposed a new representation for resolution-efficient texture mapping based on vector regression function (VRF). VRF is most suitable for images composed of piecewise smooth regions, offering square-root compression ratio and fast, random-access texturing for real-time rendering applications. We have also compared our method against ASTC and BC7, state of the art, standard methods for GPU texture mapping, and have demonstrated smaller storage size under similar rendering quality.

A main limitation of our method is that the output resolution is always limited in contrast to vector graphics which offers



(a) Original. (b) Reconstruction.

Fig. 13. VRF from a natural image. (b) Reconstructed from a 1440 × 900 photo shown in (a). The bottom row are 4× magnified results of the enclosed parts in the top row.



(a) Input. (b) 16 × 16 zoom-in of (a). (c) ASTC - 4bpp. (d) VRF - (a) resolution. (e) VRF - 16 × 16 resolution.

Fig. 14. LDR-RGB examples from Franzen [1999]. (a) Input images; (b) to (d) are 16 × 16 magnified results (via nearest sampling) with the same resolution as (a). The bit rate of ASTC is 4bpp. The output sizes of VRF in (d) and (e) are approximately the same as the sizes of ASTC in (c). (d) and (e) are evaluated from the same VRFs but at 1 × 1 and 16 × 16 the resolution of (a). Notice the artifacts in (e) caused by insufficient input resolution.

infinite resolution. With sufficient zoom-in, sharp features will appear as smooth gradients in our outputs. Furthermore, sufficient input resolution is required to capture small features. While this can be ensured for vector inputs by dense enough rasterization, we currently rely on the original resolution for raster inputs. A potential remedy is proper up-sampling [Fattal 2007; Shan et al. 2008] prior to training, which can be performed as an orthogonal pre-process to our method.

Our current implementation consumes long encoding time. We believe this can be significantly accelerated via GPU/CPU parallelization or alternative neural network training methods.

Traversing a tree partition incurs a variable number of steps. An alternative is spatial hashing [Lefebvre and Hoppe 2006], which allows constant and potentially faster traversal time. Since tree traversal costs less than 10% of the overall shader execution, we leave such optimization as future work. For images with repetitive details, storage size can be further reduced by sharing repetitive content across multiple $\Phi$ functions. Our method can be easily extended to high-dimensional texture (e.g., solid texture) and texture functions (e.g., svBRDFs and BTFs).

In addition to the level number for MIPMAP, it is also possible to directly train the Jacobian of texture coordinates for anisotropic filtering. Our method can also be applied for higher-dimensional textures such as 3D volumes. We leave these as potential future work.

## ACKNOWLEDGMENTS

We would like to thank Peiran Ren for his help on data training, John Snyder for writing improvements, and the anonymous reviewers for their valuable suggestions.

## REFERENCES

S. Boyé, P. Barla, and G. Guennebaud. 2012. A vectorial solver for free-form vector gradients. *ACM Trans. Graph.* 31, 6 (Nov.), 173:1–173:9.

C. Dachsbacher. 2011. Analyzing visibility configurations. *IEEE Trans. Visual. Comput. Graph.* 17, 4 (Apr.), 475– 486.

C. Dong, C. C. Loy, K. He, and X. Tang. 2015. Image super-resolution using deep convolutional networks. *CoRR abs/1501.00092*.

R. Fattal. 2007. Image upsampling via imposed edge statistics. In *Proceedings of the ACM SIGGRAPH Papers (SIGGRAPH'07*). ACM, New York, NY.

M. Finch, J. Snyder, and H. Hoppe. 2011. Freeform vector graphics with controlled thin-plate splines. In *Proceedings of the SIGGRAPH Asia Conference (SA'11).* ACM, New York, NY, 166:1–166:10.

R. Franzen. 1999. Kodak lossless true color image suite. http://r0k.us/graphics/kodak/.

F. Ganacim, R. S. Lima, L. H. de Figueiredo, and D. Nehab. 2014. Massively-parallel vector graphics. *ACM Trans. Graph.* 33, 6 (Nov.), 229:1–229:14.

R. Grzeszczuk, D. Terzopoulos, and G. Hinton. 1998. Neuroanimator: Fast neural network emulation and control of physics-based models. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'98).* 9–20.

M. T. Hagan and M. B. Menhaj. 1994. Training feedforward networks with the Marquardt algorithm. *Trans. Neur. Netw.* 5, 6 (Nov.), 989–993.

P. S. Heckbert. 1986. Survey of texture mapping. *IEEE Comput. Graph. Appl.* 6, 11 (Nov.), 56–67.

G. E. Hinton. 1989. Connectionist learning procedures. *Artif. Intell.* 40, 1–3 (Sept.), 185–234.

P. Ilbery, L. Kendall, C. Concolato, and M. McCosker. 2013. Biharmonic diffusion curve images from boundary elements. *ACM Trans. Graph.* 32, 6 (Nov.), 219:1–219:12.

K. Iourcha, K. Nayak, and Z. Hong. 1999. System and method for fixed-rate block-based image compression with inferred pixel values. U.S. Patent 5956431, filed October 2, 1997, and issued September 21, 1999.

S. Jeschke, D. Cline, and P. Wonka. 2009. Rendering surface details with diffusion curves. In *Proceedings of the ACM SIGGRAPH Asia Papers (SIGGRAPH Asia'09)*. 117:1–117:8.

J. Jiang. 1999. Image compression with neural networks – A survey. *Signal Process. Image Commun.* 14, 9 (July), 737–760.

Y.-K. Lai, S.-M. Hu, and R. R. Martin. 2009. Automatic and topology-preserving gradient mesh generation for image vectorization. In *Proceedings of the ACM SIGGRAPH Papers (SIGGRAPH'09)*. ACM, New York, NY, 85:1–85:8.

G. Lecot and B. Levy. 2006. Ardeco: Automatic region detection and conversion. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques (EGSR'06).* Eurographics Association, Aire-la-Ville, Switzerland, 349–360.

S. Lefebvre and H. Hoppe. 2005. Parallel controllable texture synthesis. In *Proceedings of the ACM SIGGRAPH Papers (SIGGRAPH'05).* 777–786.

S. Lefebvre and H. Hoppe. 2006. Perfect spatial hashing. In *Proceedings of the ACM SIGGRAPH Papers (SIGGRAPH'06).* ACM, New York, NY, 579–588.

P. Mavridis and G. Papaioannou. 2011. High quality elliptical texture filtering on GPU. In *Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D'11).* 23–30.

J. McCormack, R. Perry, K. I. Farkas, and N. P. Jouppi. 1999. Feline: Fast elliptical lines for anisotropic texture mapping. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'99).* 243–250.

Microsoft Corporation. 2013. Directx11 bc6h/bc7 directcompute encoder tool. https://code.msdn.microsoft.com/windowsdesktop/BC6HBC7-DirectCompute-35e8884a.

D. Nehab and H. Hoppe. 2008. Random-access rendering of general vector graphics. *ACM Trans. Graph.* 27, 5 (Dec.), 135:1–135:10.

D. Nowrouzezahrai, E. Kalogerakis, and E. Fiume. 2009. Shadowing dynamic scenes with arbitrary brdfs. *Comput. Graph. Forum* 28, 2, 249–258.

J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson. 2012. Adaptive scalable texture compression. In *Proceedings of the 4th ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics (EGGH-HPG'12).* Eurographics Association, Aire-la-Ville, Switzerland, 105–114.

M. Olano, D. Baker, W. Griffin, and J. Barczak. 2011. Variable bit rate GPU texture decompression. In *Proceedings of the 22nd Eurographics Conference on Rendering (EGSR'11).* 1299–1308.

Opengl Arb. 2010. Arb texture compression bptc. http://www.opengl.org/registry/specs/ARB/texture_compression_bptc.txt.

A. Orzan, A. Bousseau, H. Winnemöller, P. Barla, J. Thollot, and D. Salesin. 2008. Diffusion curves: A vector representation for smooth-shaded images. In *Proceedings of the ACM SIGGRAPH Papers (SIGGRAPH'08).* ACM, New York, NY, 92:1–92:8.

E. Parilov and D. Zorin. 2008. Real-time rendering of textures with feature curves. *ACM Trans. Graph.* 27, 1 (Mar.), 3:1–3:15.

Z. Qin, M. D. McCool, and C. Kaplan. 2008. Precise vector textures for real-time 3D rendering. In *Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D'08).* 199–206.

P. Ren, J. Wang, M. Gong, S. Lin, X. Tong, and B. Guo. 2013. Global illumination with radiance regression functions. *ACM Trans. Graph.* 32, 4 (July), 130:1–130:12.

P. Sen. 2004. Silhouette maps for improved texture magnification. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS'04)*. 65–73.

Q. Shan, Z. Li, J. Jia, and C.-K. Tang. 2008. Fast image/video upsampling. In *Proceedings of the ACM SIGGRAPH Asia Papers (SIGGRAPH Asia'08)*. 153:1–153:7.

J. Ström and T. Akenine-Möller. 2005. ipackman: High-quality, low-complexity texture compression for mobile phones. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS'05)*. ACM, New York, NY, 63–70.

J. Sun, L. Liang, F. Wen, and H.-Y. Shum. 2007. Vectorization using optimized gradient meshes. In *Proceedings of the ACM SIGGRAPH Papers (SIGGRAPH'07)*. ACM, New York, NY.

T. Sun, P. Thamjaroenporn, and C. Zheng. 2014. Fast multipole representation of diffusion curves and points. *ACM Trans. Graph.* 33, 4 (July), 53:1–53:12.

X. Sun, G. Xie, Y. Dong, S. Lin, W. Xu, W. Wang, X. Tong, and B. Guo. 2012. Diffusion curve textures for resolution independent texture mapping. *ACM Trans. Graph.* 31, 4 (July), 74:1–74:9.

M. Tarini and P. Cignoni. 2005. Pinchmaps: Textures with customizable discontinuities. *Comput. Graph. Forum* 24, 3, 557–568.

J. Tumblin and P. Choudhury. 2004. Bixels: Picture samples with sharp embedded boundaries. In *Proceedings of the 15th Eurographics Conference on Rendering Techniques (EGSR'04)*. 255–264.

V. N. Vapnik. 1995. *The Nature of Statistical Learning Theory*. Chapter IV. Springer-Verlag New York, Inc., New York, NY.

L. Williams. 1983. Pyramidal parametrics. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'83)*. 1–11.

T. Xia, B. Liao, and Y. Yu. 2009. Patch-based image vectorization with automatic curvilinear feature alignment. In *Proceedings of the ACM SIGGRAPH Asia Papers (SIGGRAPH Asia'09)*. ACM, New York, NY, 115:1–115:10.

G. Xie, X. Sun, X. Tong, and D. Nowrouzezahrai. 2014. Hierarchical diffusion curves for accurate automatic image vectorization. *ACM Trans. Graph.* 33, 6 (Nov.), 230:1–230:11.

Received September 2014; revised April, August 2015; accepted August 2015