Resolution Independent Real-Time Vector-Embedded Mesh for Animation

Chih-Yuan Yao, *Member, IEEE*, Kuang-Yi Chen, Hong-Nian Guo, Cheng-Chi Li, and Yu-Chi Lai, *Member, IEEE*

***Abstract*— High-resolution textures are determinant of not only high rendering quality in gaming and movie industries, but also of burdens in memory usage, data transmission bandwidth, and rendering efficiency. Therefore, it is desirable to shade 3D objects with vector images such as scalable vector graphics (SVG) for compactness and resolution independence. However, complicated geometry and high rendering cost limit the rendering effectiveness and efficiency of vector texturing techniques. In order to overcome these limitations, this paper proposes a real-time resolution-independent vector-embedded shading method for 3D animated objects. Our system first decomposes a vector image consisting of layered close coloring regions into unifying-coloring units for mesh retriangulation and 1D coloring texture construction, where coloring denotes color determination for a point based on an intermediate medium such as a raster/vector image, unifying denotes the usage of the same set of operations, and unifying coloring denotes coloring with the same-color computation operations. We then embed the coloring information and distances to enclosed unit boundaries in retriangulated vertices to minimize embedded information, localize vertex-embedded shading data, remove overdrawing inefficiency, and ensure fixed-length shading instructions for data compactness and avoidance of indirect memory accessing and complex programming structures when using other shading and texturing schemes. Furthermore, stroking is the process of laying down a fixed-width pen-centered element along connected curves, and our system also decomposes these curves into segments using their curve–mesh intersections and embeds their control vertices as well as their widths in the intersected triangles to avoid expensive distance computation. Overall, our algorithm enables high-quality real-time Graphics Processing Unit (GPU)-based coloring for real-time 3D animation rendering through our efficient SVG-embedded rendering pipeline while using a small amount of texture memory and transmission bandwidth.**

***Index Terms*— Antialiasing, resolution-independent shading, vector embedded, vector representations.**

Fig. 1. Top: Rendering result of a complex scene consisting of 133 distinct Tigresses (dots marked with red, green, and blue) of distinct meshes and textures using our algorithm to illustrate our motivation to reduce the amount of texture memory usage, maintain the rendering quality, and enhance rendering efficiency with GPU-based coloring, where coloring denotes color determination for a point based on an intermediate medium such as a raster/vector image. Bottom: Their composition as IEEE'. While using a computer with a 1-GB graphics card, we can load and shade only 15 Tigresses (dots marked with red) using raster texture with a resolution of 4096×4096, which is the maximal available resolution of OpenGL. While using a computer with a 16-GB memory, we can load and shade only 38 Tigresses in Blender (dots marked with red and green) rings using raster texturing with a texture resolution of 8192×8192 to match our rendering quality. Shading 15 Tigresses with raster texturing on GPUs takes 0.0027 s in NVidia Titan with OpenGL, shading 133 Tigresses in Intel I7-4960K and 64 GB memory with Blender takes 3 s, and shading 133 Tigresses in NVidia Titan with our algorithm takes 0.0152 s.

I. Introduction

HIGH-RESOLUTION textures are *so determinant to* achieve high-quality visual results for gaming and video and movie production, but it generally requires high GPU memory storage and large data transmission for proper operations. Fig. 1 gives a motivated example: when raster-textured Tigress has a texture of 4096×4096, the maximal distinct model number for a 1 G graphics card is 15, but generally fantastic scenes require a number larger than 15. Furthermore, even when using a commercial animation software such as Blender, high-resolution textures not only require huge memory to limit the number of models but also deteriorate rendering efficiency due to lack of GPU-based coloring acceleration. Hence, it is a trend to increase perceived quality by other means rather than just increasing the number of texels, which can be described as another type of data compression problem. Vector images such as scalable vector graphics (SVG) images, which are traditionally used to represent logos, symbols, icons, and cartoons, may be an alternative to raster images to overcome resolution limitations. Therefore, this paper aims at developing

Manuscript received August 15, 2015; revised November 6, 2015 and February 14, 2016; accepted March 22, 2016. Date of publication April 21, 2016; date of current version September 5, 2017. This work was supported by the Research Grant under Grant MOST-104-2221-E-011-029-MY3, Grant MOST-103-2221-E-011-114-MY2, Grant MOST-104-2218-E-011-006, Grant MOST-103-2218-E-011-014, Grant MOST-104-2221-E-011-092, and Grant MOST-103-2221-E-011-076, Taiwan. This paper was recommended by Associate Editor C. Zhang. *(Chih-Yuan Yao and Kuang-Yi Chen contributed equally to this work.) (Corresponding author: Yu-Chi Lai.)*

The authors are with the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei 10607, Taiwan (e-mail: cheeryuchi@gmail.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSVT.2016.2555738

a resolution-independent vector-embedded shading method for 3D animated objects for compactness and infinite resolution with real-time animation efficiency and ease of plugging into commercial animation software.

In the past, when texturing objects with vector images, it was a general practice to convert them into raster images, which could be accelerated by GPU shaders, but the compactness and antialiasing abilities were lost after conversion. Furthermore, GPU-based renderers have a limitation in allowable texture resolution, and it is really costly to render high-resolution textured objects with Central Processing Unit (CPU)-based renderers. Therefore, it is desirable to directly use vector images for real-time shading and color computation to have minimal amount of data transmission and avoid sampling artifacts.

This paper follows SVG definitions to describe our vector images and aims at implementing most coloring instructions given in the format, where coloring denotes color determination for a point based on layered composite primitives. Generally, coloring primitives are regions enclosed by a path of a sequence of trajectories and contours including line segments, Bézier curves, and partial elliptical arcs, and each path has an associated set of numeric parameters known as control points, which are 2D points denoted by $\overline{p} = (x, y)$ in the vector space and can be embedded in triangle vertices for rasterization. As a result, coloring or path filling determines the set of primitives a point lies inside, i.e., the set of paths a point logically lies inside and their layered order for color computation. Algorithmically, coloring requires computation of distance to the enclosed paths for path/primitive interior/exterior determination. (We denote this process as inside/outside test.) Furthermore, SVG also equips with gradient coloring, which determines a color based on the distance to a line or point. Our system generally decomposes a primitive into one or more unifying-coloring units and a path into one or more unifying-coloring path segments, where unifying coloring denotes coloring using the same set of color computation operations, unifying-coloring units are regions using the same set of coloring operations, and unifying-coloring path segments are path segments whose both sides have the same coloring operations. Furthermore, a stroke is another important element, which is the region swept out by a fixed-width pen-centered element on the stroking path, and stroking is the process of drawing strokes. Nehab and Hoppe [1] and Qin *et al.* [2] pack SVG-based coloring and stroking information in an acceleration structure for shading, but packing induces overdrawing and deteriorates their rendering performance. Their algorithms also have complex shading and indirect memory accessing schemes which may bring barriers to plug them into real-time applications and off-line commercial animation software. Loop and Blinn [3] embed signed distance testing in meshes for shading, but multiple overlapping meshes induce serious overdrawing issues, require a large amount of extra memory usage, and need complex composition and shading instructions for avoiding z-fighting issues. In addition, the algorithm does not provide any solution to gradient coloring operations and stroking.

In order to overcome these issues, our algorithm decomposes these overlapping coloring primitives into unifying-coloring units. Furthermore, to avoid indirect memory access, simplify coloring instructions, and enhance rendering and memory efficiency, our system computes a coloring texture based on vector coloring instructions and estimated gradient coloring fields. These also allow our algorithm to be easily incorporated into the rendering pipeline of commercial animation software including Maya and Blender for enhancing their rendering efficiency. On the other hand, stroking requires expensive signed distance evaluation, and it is a critical bottleneck in rendering. Thus, our system embeds stroking paths in the object as control vertices and designs a geometry shader to place point sprites with a varying radius computed with the stroke width and view-triangle transformation. In the end, we have embedded vector images in several meshes, and the results show that our vector-embedded scheme can run in real time with minimum amount of memory usage. We conduct a comparison with raster texturing to show that our algorithm is more memory effective with better rendering quality. Compared with Loop's method [3], our algorithm is more memory effective with better rendering efficiency and precise color reproduction. Furthermore, it is also more shading efficient and requires simpler programming than Nehab's method [1]. This paper proposes a vector-based embedment shading scheme to render 3D animated objects with compact and resolution-independent color computation for real-time applications and makes the following two major contributions.

1) Rendering 3D animated objects with high-resolution textures is important for gaming and video and movie production, but it is a tough task for GPU-based and CPU-based renderers with raster texturing because of huge requirement of memory bandwidth and usage. This paper designs a unified and simple algorithm to decompose a vector image into unifying-coloring structures for construction of a coloring texture and retriangulation of a mesh to embed coloring information and distances to coloring paths. Later, we make use of our efficient and parallel GPU-based pixel coloring shaders and stroking geometry shader to simplify shading instructions and avoid redundant pixel coloring and costly distance computation, respectively. Furthermore, gaming requires tiling textures for rendering and transmission efficiency, and we can easily concatenate all coloring textures with linear mapping on coloring coordinates for integration with games. Overall, our algorithm enables real-time 3D animation rendering with resolution independence using modern GPU hardware through our efficient SVG-embedded rendering pipeline.
2) It is necessary to use a large number of high-quality textures for real-time 3D applications and movie production, and thus, efficiently reducing texture memory usage plays an important role in achieving high-quality results. This paper minimizes memory usage by minimizing embedded information, localizing vertex-embedded shading data, and ensuring

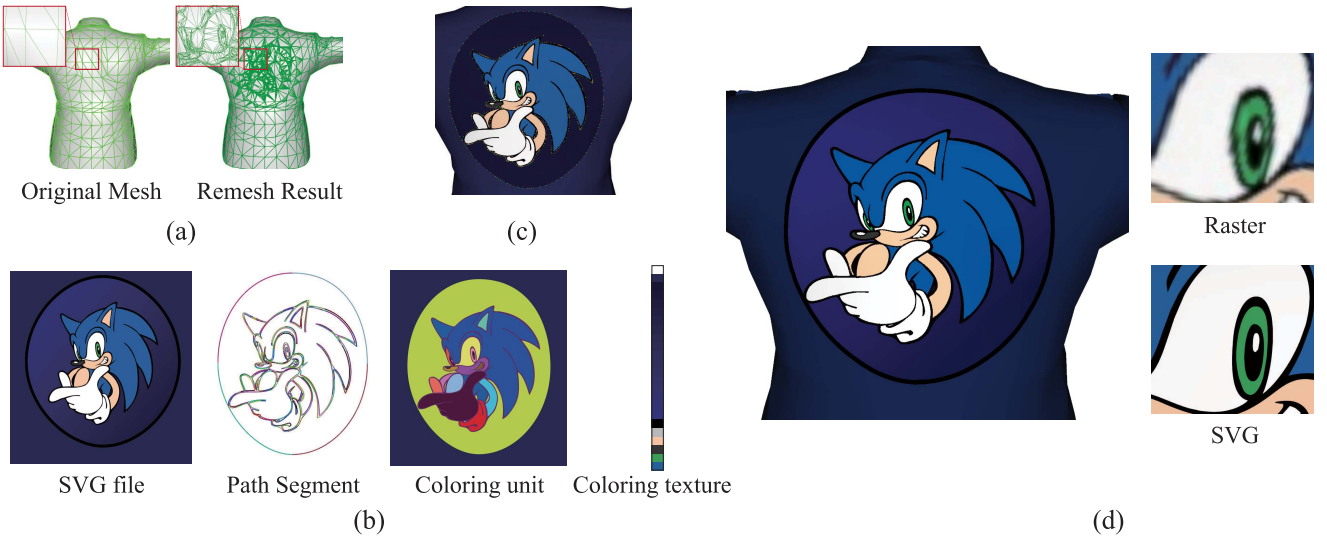


Fig. 2. This figure demonstrates the result of applying our vector-embedded shading framework on a jacket model. (a) Original mesh and retriangulated mesh of the embedded object in wireframe. Retriangulation adds triangles for coloring information embedment. (b) Input vector image, unifying-coloring path segments, unifying-coloring units, and 1D coloring texture. We mark these path segments and color units with different colors. The 1D coloring texture stores sampling colors from all possible coloring sets in this vector image. (c) Embedded stroke control vertices as dots of different colors. (d) Shading results of our vector-embedded mesh and the raster texturing mesh. The right top and bottom show the detailed-view results of our algorithm and raster texturing, respectively.

fixed-length shading instructions for data compactness. In other words, our algorithm maintains high-quality results without aliasing while using only a small amount of memory.

As demonstrated by the results, our vector-embedded shading framework can render complex 3D objects in real time with high quality using very few and simple shading instructions without aliasing while limiting the texture memory usage and data transmission bandwidth.

II. Related Work

Vector graphics have been the research topic for decades because of their popularity in rendering texts, icons, graphs, charts, and symbols. This paper focuses on shading 3D objects with vector images using most coloring and stroking functions defined in SVG, and hence, the following discusses only those works directly related to this research.

A. 2D Path Renderers and GPU Acceleration

Traditional path renderers use CPU-based scan line rasterization techniques by converting paths into a set of edges and intersecting each edge with scan lines to update pixel colors inside the path-bounded region. Although these renderers are work efficient and cache friendly, they are CPU intensive and sequential. Therefore, several research aim at pipelining all these tasks and exploiting GPU parallelism. Direct2D [4] transforms paths with a CPU and then tessellates them with trapezoids with GPUs. Path transformation is a performance bottleneck, and it does not properly handle aliasing problems. Ramanarayanan *et al.* [5] accelerate path rendering by computing a coverage mask of paths with a CPU-based rasterizer and rendering antialiasing coverage with GPUs. This hybrid scheme often has bottlenecks in dynamic texture updates required for every rendered path. Therefore, several researchers [6]–[8] recast CPU-based scan line schemes as GPU-based methods. Kokojima *et al.* [9] rasterize each glyph into the stencil buffer using the Loop–Blinn scheme and determine the winding number of TrueType glyph outlines to shade pixels. However, their method cannot rasterize all vector images because there is no convexity guarantee for path shading. Rueda *et al.* [10] decompose cubic Bézier hulls using the simplicial concept to guarantee convexity, but it requires an overly expensive discarding fragment shader based on Bézier normalization. Kilgard and Bolz [11] decouple the traditional two-pass pipeline into two steps, stencil and cover, and unify path rendering with OpenGL functions. All these methods require the use of stencil buffers and work only for 2D and 3D planes but not complex 3D surfaces. Our work embeds coloring units decomposed from vector images in a mesh for effectively shading a 3D model textured with vector graphics which is infinite resolution 3D object.

B. Hybrid Raster- and Vector-Based Texturing

Texturing is a very important technique for visual quality of real-time and off-line applications, but high-quality raster images require a high memory bandwidth for proper operations. Therefore, several previous research enhanced the details of raster images with new interpolation rules using cell-embedded features such as a few line segments [5], [12]–[16], an implicit bilinear curve [17], [18], a parametric cubic curve [19], two quadratic segments [20], or a fixed number of corner features [21], [22]. To incorporate regions with a high density of details requires fine lattices for the entire image, which globally increases storage cost. Additionally, discontinuities may occur when crossing cells to induce some curve shading artifacts. Line and curve representations may

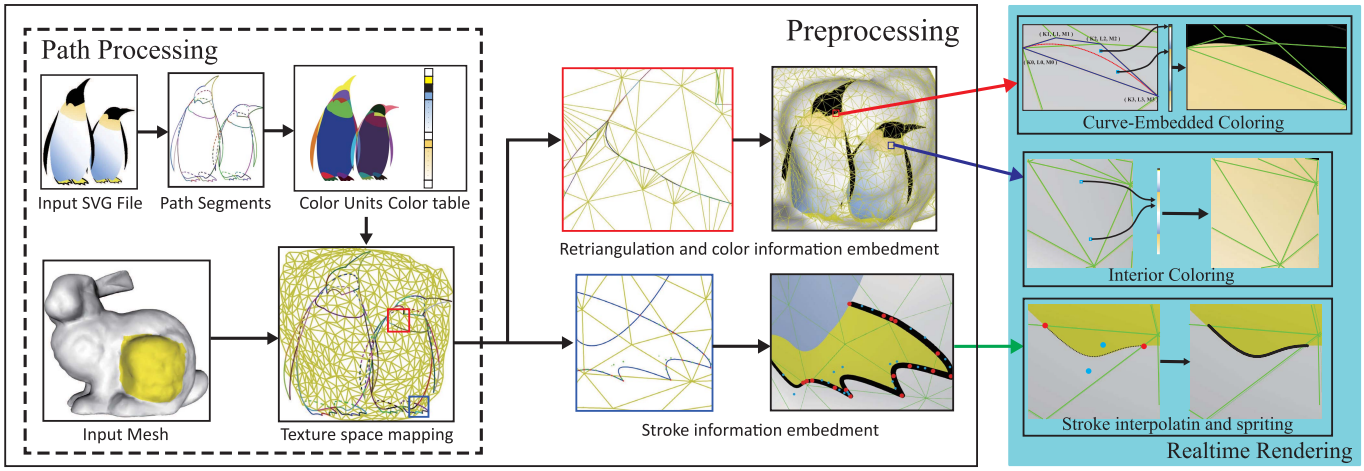


Fig. 3. The preprocessing step consists of the following stages: path segmentation and unifying-coloring unit formation, coloring field computation and coloring texture construction, segment-constrained retriangulation, and coloring and stroking information embedment. During rendering, our system shades the embedded 3D object with two coloring pixel shaders designed by us and the stroking geometry shader for resolution independence.

still miss certain sharp features to cause blurring artifacts. Jeschke *et al.* [23] use view-dependent warping and dynamic feature embedment to approximate diffusion curve images for effective texturing quality enhancement. However, view-dependent warping induces distortions, and feature embedment has similar limitations discussed previously. Our retriangulation scheme adjusts the density of triangles locally to ensure at most one feature per triangle for computation and memory efficiency while maintaining the sharpness of features.

C. Vector-Based Texturing

Nehab and Hoppe [1] and Qin *et al.* [2] pack path contents cleverly into GPU textures and then use a shader to decode the path contents. Nehab's method uses prefiltering and supersampling within a single pixel shader for antialiasing but supports only vector images containing lines and quadratic segments. Qin's algorithm computes exact distance using iterative binary normal searching for antialiasing. Although they can directly texture 3D geometry with path-based contents, their algorithms handle regions containing two or more overlapping coloring operations by composition based on the designed drawing order, and this induces overdrawing and deteriorates their rendering performance. Furthermore, their algorithms consist of complex loop and branch programming structures with hundreds of shading instructions and use complex indirect memory accessing due to variable-length packing information. These may act as barriers to integrating with real-time applications and off-line commercial software. Since their acceleration structures are image dependent, they would use the same packing density for tiling vector images with different details for gaming to induce extra memory and rendering cost. Our system removes overdrawing inefficiency, simplifies shading instructions, and removes indirect memory accessing by computing unifying-coloring units and boundary curves for retriangulation and embedding path information into vertices in the preprocessing step and computing only the necessary information during the shading step. Our coloring textures are simple for tiling, and our coloring schemes only need a few shading instructions for easy integration into commercial animation and gaming software.

D. Vector-Embedded Mesh Rendering

Loop and Blinn [3] texture 3D objects with vector images of two solid colors defined with paths and bounded regions for inside/outside testing. When applying texturing to multilayered vector images, their method requests multiple overlapping meshes to have correct coloring, and this requests complex intersection tests and composition operations to relieve multiple overlapping intersection problems and z-fighting issues. These multiple overlapping meshes request extra storage and transmission cost and induce overdrawing inefficiency because it shades and composites separate meshes retriangulated based on each set of coloring primitives. Furthermore, their algorithm does not handle gradient coloring operations and stroking. Santina [24] extends Loop's idea for boundaries defined with nonuniform rational B-splines by transforming them into low-order quadratic forms for embedment. Their algorithm has similar limitations as Loop's method. Our vector-embedded shading framework decomposes vector images into units of unifying vector operations for removal of overdrawing inefficiency and complex composition and intersection shading programming. Furthermore, our coloring field computation and coloring texture construction enable the computation of gradient coloring operations. Both mechanics make it easily integrate with GPU-based and CPU-based renderers. Additionally, we also provide a stroking geometry shader for dynamically placing point sprites for stroking effects.

III. Overview

Vector graphics are compact with infinite resolution. This paper aims at embedding a vector image in a 3D object for real-time resolution-independent shading without complex computation and programming structures such as loops and branches while using a small amount of memory and transmission bandwidth. Fig. 2 illustrates the intermediate results and final rendering of our algorithm on texturing a

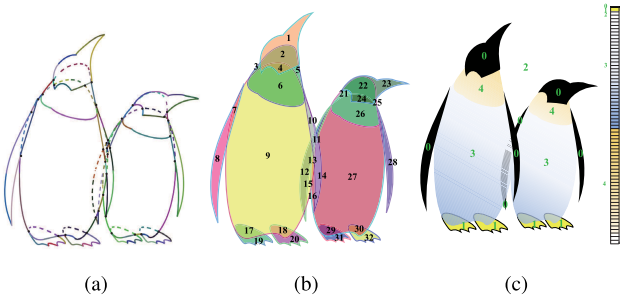
3D cloth model with a Tiger vector image, and Fig. 3 gives an overview of our algorithm. The input is an applied mesh with selected vector images of distinct coloring primitives. Our system first decomposes primitives in the vector image into unifying-coloring units by determining intersections among paths, decomposing the paths into segments, and then traces these path segments and intersections to form close units. We then label and tabulate the corresponding coloring operations of all units, compute coloring fields for gradient coloring sets, and construct a coloring texture using the coloring operations in all units with one sample for a solid coloring set and 100 samples for a harmonious coloring segment derived from gradient coloring fields. We retriangulate the mesh to ensure that all triangles have at most one path passing through for correct point-to-path distance computations and shading operations, and our system embeds corresponding coloring and path information in these triangles. Additionally, our system transforms stroking paths into mesh-based control vertices along with their color and width. During the rendering phase in real-time applications, we develop a two-pass shading algorithm to render the object. The first pass renders the coloring mesh with two coloring pixel shaders, and the second pass renders the stroking vertices with the stroking geometry shader. Our embedded method can color a 3D object using an infinite-resolution vector texture with only little penalty to its efficiency.

IV. Vector-Space Retriangulation and Stroke Embedment

In order to shade 3D objects independent of resolution in real time, our system preprocesses the shading object to embed coloring and stroking information. This section describes the details of these preprocessing steps.

A. Unifying-Coloring Unit Construction

Since elliptical arcs and parametric splines may become loops to induce coloring vagueness, we have decomposed these self-looped curves into two or more segments before any operation. In order to properly embed the shading and curve information in triangles, all path segments must be unifying coloring, i.e., the exterior and interior sides of path segments must, respectively, use the same coloring operations. However, SVG uses close primitives enclosed by a close path to determine a pixel color, and this only guarantees that the interior of a curve uses the same set of coloring instructions. When observing SVG coloring results, the exterior of a curve may have several different shading instructions when the enclosed primitive is above or below two or more other enclosed primitives. For example, the left foot of the left penguin in Fig. 4(b) is above the background and below the body, and the exterior of the top is the body coloring operation and the bottom is the background coloring operation. Furthermore, the overlapping boundaries are also the exterior color changing borders, and we can determine these overlapping boundaries by curve intersections. Thus, we must first find the intersections of these boundaries for segmenting the unifying-coloring units. In order to avoid redundant computation, we first compute the axis-aligning bounding box of lines and curves. For those lines and curves whose bounding boxes overlap each other, our system determines their intersections using Lou's algorithm [25]. Fig. 4(a) shows an exemplar result of path intersection. Due to length limitation, readers can refer to the supplemental Website[1] for more details.



(a) (b) (c)

Fig. 4. (a) Result of path intersection and segmentation for a penguin vector image. The black dots mark path intersections, and there are totally 37 intersections. Each path segment marked with a unique color has distinct coloring instructions in its exterior and interior sides. In total, there are 76 path segments. (b) Result of coloring unit formation. Each region enclosed by one or more path segments is a unifying-coloring unit. There are totally 32 coloring units. (c) Exemplar coloring texture of the penguins' image based on its 4 distinct coloring sets. In this example, we choose 15 samples for each gradient coloring field, and thus, the resolution of the texture is 33. Our system sets the ID of two gradient coloring sets as 3 and 4, respectively, and assigns their starting coordinates as 0.09735 and 0.53125.

After finding intersections, we segment paths into small segments with a unified interior/exterior coloring operations which involve in enclosing two unifying-coloring units. In order to properly enclose these units, our system duplicates each segment into two using both end intersections as their start and end intersections, respectively, and sorts all intersections from a path into a list in counterclockwise order to determine the direction of all segments from the path. Then, we first select a segment from the list, mark its flag, and determine its start and end intersections in the counterclockwise direction. We start from the end intersection and pick its first neighboring clockwise segment to the current segment as the next tracing segment. The process continues until a unifying coloring unit is complete. We can then pick up another unlabeled segment and continue to form another coloring unit. Fig. 4(b) shows an exemplar result of coloring unit construction.

B. Coloring Field and Texture Construction

Discontinuity mesh [26], [27] refines radiosity elements along possibly zeroth-, first-, and second-order radiosity-estimation discontinuity for more precise estimation, and our algorithm bears the same spirit to split the mesh along the possible discontinuity while doing color computation. Therefore, we decompose our selected vector images into solid/gradient shading regions whose colors are computed based on unifying-coloring instructions set by designers. In order to have simplified and fixed-count shading instructions, we design a precomputed coloring

[1]Website: http://graphics.csie.ntust.edu.tw/pub/EmbedVGbyRemesh/main.html.

texture to record all possible colors generated by all sets of coloring operations. We first denote each coloring unit by $\mathbf{CU}_i$, and label it as a solid or gradient unit based on whether its coloring instructions generate a solid or gradient color inside the unit

$$\mathbf{CU}_i = \begin{cases} S, & \text{if } \mathbf{CU}_i = \mathbf{T}_s \\ G, & \text{if } \nabla(\mathbf{CU}_i(\overline{p_j})) = \mathbf{G}\left(\mathbf{C}_i^b, \mathbf{C}_i^e, \omega_j\right) = \mathbf{T}_g^j \end{cases} \quad (1)$$

where S/G denotes the label of the unit, $\mathbf{T}_s$ is a solid color and recorded as a single texel in our 1D coloring texture, $\mathbf{G}(\mathbf{C}^b, \mathbf{C}^e, \omega_j)$ is the gradient coloring operation based on the coloring constraints, $\mathbf{C}^b$ and $\mathbf{C}^e$, and a corresponding harmonious scalar, ω_j. We can compute ω based on a discrete harmonic function, $h(\omega)$, through a uniform image grid [28]. Accordingly, we denote each grid point by $\overline{v}_i$ and choose the $h(\omega)$ as

$$h(\omega_i) = \omega_i - \sum_{j \in \mathbf{N}(\overline{v}_i)} c_{ij}\omega_i = 0 \quad (2)$$

where $\mathbf{N}(\overline{v}_i)$ is the one-ring grid of $\overline{v}_i$, and c_{ij} is the cotangent weight [29]. We can get the corresponding texel, $\mathbf{T}_g^j$, by interpolating the scalar value, ω_j. Furthermore, we represent a gradient coloring set as a series of discrete color values, and our system controls its color sampling density base on a scalar value, ω. After determining our gradient coloring representation, we then tabulate the coloring instructions of all coloring units and run the shading operations of each valid coloring set to construct the 1D coloring map as follows: if a unifying-coloring set is labeled as a solid color, it occupies 1 texel in the 1D coloring texture; otherwise, a gradient coloring set is formulated as a harmonic function [28] to get a 1D scalar value for each triangle vertex, $\mathbf{s}(u, v) = \nabla(CU_i(\overline{p_j}))$, where (u, v) is the texture coordinate of the vertex estimated by exponential mapping [30]. According to the triangle distribution, we can get different color samplings as a color set to form the gradient unit in the 1D texture. After collecting all texel samples, we can compute the valid texture coordinate according to the number of texels. Fig. 4(c) gives an exemplar coloring texture. Later, Section IV-D gives the details about how to embed the shading information in vertices.

C. Retriangulation Based on Coloring Segments

In order to shade efficiently with fixed-count shading instructions, each triangle of the mesh must have at most one intersected path segment. Unifying-coloring-operation segments are generally still too long to fit into a single triangle and thus, we decompose them into smaller segments using the steps: First, our system first embed all triangles in the texture space with transformation described in Fig. 5 using exponential mapping [30], and when a segment intersects with the edge in the texture space, the segment is split at the intersection. Second, when a triangle contains multiple segments, we build the convex hull of these segments for overlapping evaluation. If two hulls overlap each other, we split the corresponding curves into two small segments correspondingly using the de Casteljau algorithm. Fig. 6 shows

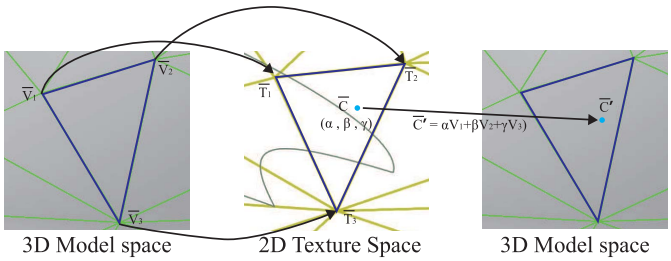


Fig. 5. Mutual transformation between the texture and model space. From model to texture, we embed all triangles in the texture space using their three vertex texture coordinates generated with exponential mapping [30]. From texture to model, we barycentrically interpolate the vertex locations of the texel-enclosed triangle in the model space to determine the corresponding 3D location of the texel.

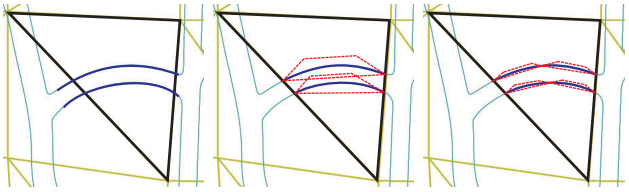


Fig. 6. Exemplar path-decomposition process. (a) Two path segments are decomposed into multiple shorter path segments using edge–segment intersections. (b) There are two path segments inside this exemplar triangle, and we build their convex hulls for overlap evaluation. (c) Since their convex hulls overlap, we split the two segments into four smaller segments using the de Casteljau algorithm.

an exemplar process. The decomposition process terminates provided that there is no overlapping among all convex hulls of segments inside a triangle. In order to compute shading colors correctly, we use path line segments and convex hull edges as constraints to retriangulate the entire mesh. In addition to having correct inside/outside testing inside triangles, we would also like to have correct gradient coloring computation inside each triangle. Since we compute a color of a gradient coloring set by sampling scalar values, ω_i, we would split a triangle into three using its centroid if one of the mutual differences of scalar values at three vertices is larger than T_{scalar}, where we choose 0.05 for the entire work.

D. Embed Path Implicitization and Coloring Information

Our system implicitizes unifying-coloring path segments for coloring computation with inside/outside testing of texture-coordinate-based distance interpolation [3]. We can describe all coloring paths as integral curves in the form of $f(\overline{p}) = \tilde{k}(\overline{p})^3 - \tilde{l}(\overline{p})\tilde{m}(\overline{p})$ where $\tilde{k}$, $\tilde{l}$, and $\tilde{m}$ are lines passing through corresponding inflection points. We compute the inflection points and corresponding $\tilde{k}$, $\tilde{l}$, and $\tilde{m}$ using those equations described by Loop and Blinn [3]. Then, we can use $f(\overline{p})$ and $\tilde{k}(\overline{p})$ to evaluate whether a point is inside/outside the path. Furthermore, when a point $\overline{V}$ is inside the convex hull of a cubic Bézier curve, we express its location as barycentric interpolation of its four control points, $\{\overline{V}_0, \overline{V}_1, \overline{V}_2, \overline{V}_3\}$. Similarly, its distance to a line, $\tilde{L} = ax + by + c$, can also barycentrically interpolate the line distance of the four control points. Therefore, we can compute and store the distance to $\tilde{k}$, $\tilde{l}$, and $\tilde{m}$ as $\mathbf{K}$, $\mathbf{L}$ and $\mathbf{M}$ for all control points and use barycentric interpolation of these values for shading a pixel.

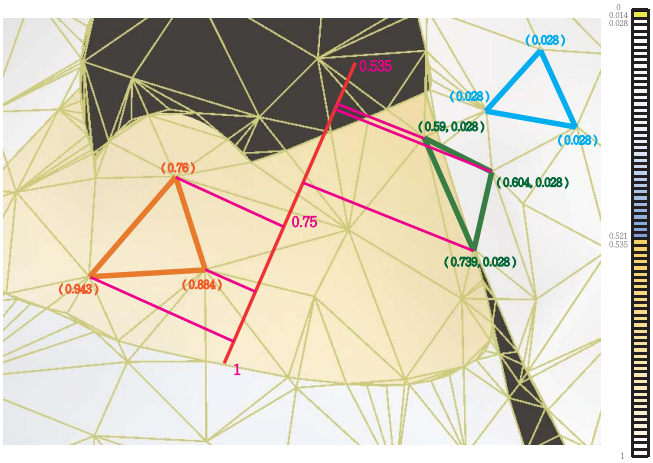


Fig. 7. Examples of embedding coloring information in triangle vertices by having x component for its interior color and y component for its exterior color. There are three triangles marked in different colors: the blue one is totally in the background, and its coloring index to the white is recorded in x of the texture coordinate; the orange one lies inside the line gradient coloring region, and we compute the gradient coloring index based on its estimated harmonious value and record it in x of the coloring texture; the green one lies between the line gradient coloring region and background and we compute the gradient coloring index based on its estimated harmonious value and record it in x and record its background coloring index in y.

Due to length limitation, readers can refer to the supplemental Website (see footnote 1) for more details. After applying edge-constrained triangulation, our system classifies those triangles intersecting with a path segment as curve-embedded triangles and the others as interior triangles and computes and embeds their path-evaluating **k**, **l**, and **m** in vertices of curve-embedded triangles. Then, our system also determines the interior and exterior coloring operations as discussed in Section IV-B and computes and records their coloring indices as a 2D texture coordinate in the curve-embedded triangle vertices as follows: If the coloring set is solid, the texel coordinate for the set is the coloring index. Otherwise, our system computes the coloring harmonious value for a vertex as described in Section IV-B and projects it to the coloring map for its coloring index. For example, for a single linear gradient coloring whose color is determined based on the distance to a designed line segment, we can describe its coloring field as the distance to the reference line segment; we then project the SVG texture coordinate of a vertex onto the reference line segment and use the projection as its coloring index. Fig. 7 shows three exemplar cases. Similarly, we compute and embed coloring index in the x component of the vertex texture coordinates for all interior triangles. Later, Section V gives the details of rendering objects with embedded information.

E. Stroking Vertex Construction

Strokes are also important elements for SVG, and Qin et al. [2] [21] draw them based on costly distance computation which makes them improper for gaming and real-time applications. Stroking generally depicts the solid feature outlines and can be viewed as sweeping point sprites of a fixed radius along stroking paths. When shading, sprite radii should vary with their assigned width and the triangle-viewing transformation. We solve this issue by defining a stroke vector as the tangent vector which is perpendicular to the path segment and parallel to the triangle plane with a length of the stroke width and it can be used as the stroke width factor. After projecting onto the screen coordinate, its projected length is the effective stroke width on the screen space. Therefore, our system encodes segment-triangle-based stroking information for efficiently rendering these radius-varied stroke sprites as follows: First, we transform all triangles into the texture coordinate to intersect and decompose stroke paths into smaller segments. Second, we locate segment control points and transform them to the model space as described in Fig. 5. Third, we compute the tangents at control points using Bézier definition and set the normals as the embedded triangle normals. Finally, we compute and embeds the stroke vectors whose direction is parallel to the cross of the tangent and normal with a length equal to the assigned stroke width along with their locations into the triangle. Later, Section V-B describes how our designed stroking geometry shader uses the information to render strokes on the surface of the object. In our current implementation, we dynamically adjust the point sprite density to avoid aliasing artifacts.

V. Rendering

After retriangulating objects and embedding coloring and stroking information, we design a two-pass rendering method for real-time applications. The following gives shading details.

A. Coloring

Our system directly uses similar vertex shaders as those used for rendering static meshes, skin meshes, and shape morphing meshes for incorporation with existing deformation techniques. Two coloring pixel shaders, interior and curve-embedded coloring pixel shaders, shade interior and curve-embedded triangles, respectively, for real-time resolution-independent rendering.

1) Interior Triangles: After vertex transformation and tessellation, our system uses the interior coloring pixel shader listed in the following to determine the color of a pixel directly from the 1D coloring texture built in Section IV-B.

```
in float texCoord;
out vec4 FragColor;
uniform sampler1d tex;
void main()
{
  FragColor = texture(tex, texCoord);
}
```

Accordingly, each vertex must also record the following information:

```
Interior_Vertex_data
{
  Position : vec3
  Normal   : vec3
  ColorID  : float
}
```

Our algorithm determines a pixel color using barycentric texture coordinate interpolation and 1D coloring texture

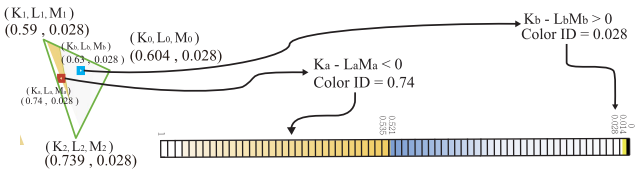


Fig. 8. Two exemplar cases of shading pixels with our curve-embedded pixel shaders. The red pixel is inside the curve, and the blue pixel is outside the curve based on the barycentric interpolation of **K**, **L**, and **M**. Our system computes their color indices by barycentrically interpolating embedded xs and ys, respectively, for their colors.

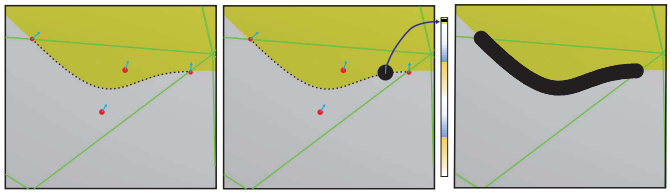


Fig. 9. Our system uses the embedded four control vertices to determine the location and stroke vector of uniform samples and places point sprites with a radius of the projected stroke vector length using the recorded stroke color.

look-up as follows: First, when it is a solid coloring set, all three vertices have the same 1D texture coordinate pointing to the coloring texel. Barycentric interpolation of the texture coordinate still points to the same coloring texel. Second, when it is a gradient coloring set, each vertex stores its corresponding coloring texture coordinate estimated based on its gradient harmonious value. For a pixel inside the triangle, we estimate its coloring texture coordinate by barycentrically interpolating vertices' coloring texture coordinates. For example, when the set of coloring instructions is simple linear gradient coloring, the harmonious value of each vertex reflects its distance to the reference line. Since the distance of a pixel to the reference line is estimated by barycentric interpolation of vertices' distance, i.e., its harmonious value can also barycentrically interpolate all vertices' harmonious values stored as the texture coordinates. Therefore, we can directly gain colors for gradient coloring operations by barycentric interpolating the texture coordinates to indexing the 1D coloring texture.

2) Curve-Embedded Triangles: For a curve-embedded triangle, a pixel color depends on two aspects: The first is whether it is inside or outside the embedded path segment, and the second is the interior/exterior coloring operations. Therefore, we design our curve-embedded pixel shader listed as follows:

```
in vec2 texCoord;
in vec4 klm;
out vec4 FragColor;
uniform sampler1d tex;
void main()
{
 if ( pow(klm.x,3)-klm.y*klm.z < $0$ )
   FragColor = texture(tex, texCoord.x);
 else
   FragColor = texture(tex, texCoord.y);
}
```

The shader must first determines a pixel color by first running the inside/outside test with the embedded **K**, **L**, and **M** coefficients to check whether $K(\overline{p})^3 - L(\overline{p})M(\overline{p})$ is over zero or not. When it is larger than zero, we barycentrically interpolate the x texture coordinate stored in three vertices in a similar manner as the interior shading for determining its color. Otherwise, we barycentrically interpolate the y texture coordinates for determining its color. Fig. 8 shows two examples. Accordingly, in addition to position, normal, interior color texture coordinate, we must store extra exterior coloring texture coordinate and K, L, and M as follows:

```
Curve_Embedded _Vertex_data
{
  Position     : vec3
  Normal       : vec3
  inoutColorID : vec2
  KLM          : vec3
}
```

Furthermore, for programming simplicity, we can shade all triangles with the curve-embedded coloring pixel shaders, but the shader would require interior triangles to store extra **K**, **L**, and **M** coefficients and an extra texture coordinate. Furthermore, our system must also shade them with extra inside/outside testing. It is not efficient for rendering and not effective for memory usage. Therefore, we decide to add a little programming complexity for efficiency and effectiveness.

B. Stroking

Our system renders strokes as sweeping point sprites along the stroke paths recorded as sets of control points embedded in the intersected triangles as shown in Fig. 9. Our system uses the same vertex shader as shading coloring meshes to transform the control vertices and corresponding stroke vectors to its screen-space location based on the embedded triangle. Then, our system then places point sprites with our designed geometry shader listed in the following:

```
in float stroke_size[];
in nPoint;
void main()
{
 if ( stroke_size[0] > 0)
 {
   for ( int i = 0; i < nPoint; i++ )
  {
    vec3 pos = getCubicPoint
   (gl_in[0].gl_Position.xyz,
    gl_in[1].gl_Position.xyz,
    gl_in[2].gl_Position.xyz,
    gl_in[3].gl_Position.xyz);
    gl_Position = modelViewProjection
    Matrix * vec4(pos, 1.0);
    gl_PointSize = stroke_size[0];
    EmitVertex();
  }
 }
}
```

TABLE I

TIMING STATISTICS WHEN USING OUR ALGORITHM ON SPIDERMAN, SPIDERMAN USING STROKING PATHS, STANFORD BUNNY, SONIC OF JACKET, LEGO, TIGER CLOTH, ZEBRA, AND TIGRESS.

	Int.(ms)	Imp.(ms)	Remesh(ms)	Tex.(ms)	Emb.(ms)
Spider	3150	0.30	22230	0.14	0.02
Spider(S)	755	0.25	2483	3.62	0.01
Bunny	1268	0.71	27261	0.14	0.03
Jacket	192	0.45	1053	0.15	0.01
Lego	76	0.46	2670	0.11	0.01
Cloth	13984	0.99	117761	0.13	0.07
Zebra	7964	0.48	87831	0.10	0.04
Tigress	4464	0.45	44268	0.16	0.02

INT. DENOTES THE STAGE OF PATH INTERSECTION AND SEGMENT DECOMPOSITION, IMP. DENOTES THE STAGE OF PATH IMPLICITIZATION, REMESH DENOTES THE STAGE OF CONSTRAINED RETRIANGULATION, TEX. DENOTES THE STAGE OF COLORING TEXTURE CONSTRUCTION, AND EMB. DENOTES THE STAGE OF EMBEDDING COLORING INFORMATION.

TABLE II

QUALITATIVE PERFORMANCE STATISTICS WHEN USING OUR ALGORITHM, LOOP'S METHOD, AND RASTER TEXTURING WITH TEXTURES OF DIFFERENT RESOLUTIONS FOR SPIDERMAN, STANFORD BUNNY, SONIC OF JACKET, LEGO, TIGER CLOTH, ZEBRA, AND TIGRESS.

	Ours	Loop's	R.T.	R(2k)	R(4k)	R(8k)	R(16k)
Spider	1.37	1.37	125	46.2	19.0	6.18	1.12
Bunny	0.823	142	69.4	32.6	14.9	6.20	1.83
Jacket	6.10	20.0	649	1079	45.2	15.5	2.99
Lego	0.420	0.420	220	26.7	11.6	4.48	1.23
Cloth	2.63	6.24	155.7	55.1	21.7	6.49	1.17
Zebra	1.10	1.10	60.5	25.2	10.8	3.95	1.02
Tigress	1.39	27.3	78.6	25.6	10.9	3.89	0.93

OURS IS FOR OUR ALGORITHM, LOOP'S IS FOR LOOP'S ALGORITHM, R.T. IS FOR RASTER TEXTURING USING A MEMORY-MATCHED RASTERIZED TEXTURE, R(MK) IS FOR RASTER TEXTURING USING A MEMORY-MATCHED RASTERIZED TEXTURE OF A RESOLUTION WHOSE WIDTH AND HEIGHT ARE EQUAL TO mK. THE NUMBER IS MSE OF THE DETAIL-VIEWED FRAME IN THE VIDEO SEQUENCE SHOWN IN OUR SUPPLEMENTAL WEB SITE[1].

Our geometry shader first takes the four control vertices for generation of uniform samples along the stroking path and computes a stroke vector $\vec{T}$ by Bézier interpolation of the projected control stroke vectors. Then, after projecting onto the screen coordinate, the projected stroke vector length is effective stroke width on the screen space as the sprite radius. Finally, we transfer the coloring information to the pixel shader for shading. Fig. 9 shows an exemplar case. Accordingly, our system stores all embedded control vertices in the line adjacency format as

```
Stroke_Vertex_data
{
 Position     : vec3
 Normal       : vec3
 strokeVector : vec3
 ColorID      : float
}
```

VI. RESULTS AND DISCUSSION

We have used C++ and Window's Form to create our CPU-based framework and GLSL to implement the coloring pixel shaders and stroking geometry shader. Later, we have measured most timing statistics presented in this section based on a computer with Nvidia GTX 560 Ti, Intel i7 3770, and 16 GB main memory by turning ON the GPU antialiasing function. Figs. 2 and 10 show various rendering results when using our algorithm. Table I shows the timing statistics for each preprocessing stage when applying our algorithm on Spiderman, Spiderman using stroking paths, Stanford bunny, Sonic of Jacket, Lego, Tiger Cloth, Zebra, and Tigress. Generally, retriangulation takes the largest portion of the preprocessing time, and the amount depends on the path segment number and original mesh size. Path decomposition takes the second largest portion, and the amount depends on the curve number in the vector image. These two stages make interactive editing and authoring impossible. Therefore, our system applies rasterized textures of vector images to shade objects for interactive editing and authoring, and later, vectorizes and embeds vector images in objects for real-time and resolution-independent shading.

Raster texturing is well accepted and popular, and therefore, we have conducted comparisons between our algorithm and raster texturing. In order to visually and perceptually compare rendering quality, we have selected a rasterized texture with the criterion of using a comparative amount of GPU memory in both methods when rendering the object. Table III shows the amount of memory usage for each model and their corresponding resolution in each rasterized texture. As shown in the seventh and eighth columns of Fig. 10 for the detailed view of rendering results, our system can render the object with infinite-resolution coloring details for color sharpness, but raster texturing blurs the coloring details. Furthermore, we would like to understand our qualitative performance when comparing with raster texturing. After selecting the memory-matching resolution listed in Table III, we have rasterized ground truth (GT) textures whose width and height are $64K$ for shading objects for GT rendering results. We can then compute the mean square error (MSE) for qualitative analysis. Our qualitative performance outperforms raster texturing in terms of MSE. Additionally, we would also like to know the error performance while shading the objects with different rasterized resolutions. We list all qualitative analysis data in Table II, and our algorithm can render objects with a quality better than $8k$ rasterized textures. Additionally, Table III shows the comparison of the triangle count of the original and retriangulated mesh. The complexity of a vector image affects the ratio between these two counts. When complexity increases, the ratio increases. In the same table, we also show the memory usage. Table IV shows the frame rate of our algorithm and raster texturing with different resolutions. Although our rendering efficiency is sometimes a little worse than raster texturing, the qualitative performance is much better than raster texturing. Additionally, the frame rate of raster texturing drops when resolution increases, but our frame rate maintains the same and has more cases outperforming high-resolution raster texturing. Furthermore, when it is over the limit of OpenGL, CPU-based rendering has very low shading efficiency. Fig. 1 shows a complex scene consisting of 133 Tigresses acting as distinct characters with independent meshes and textures for demonstrating our real-time high-quality shading ability.

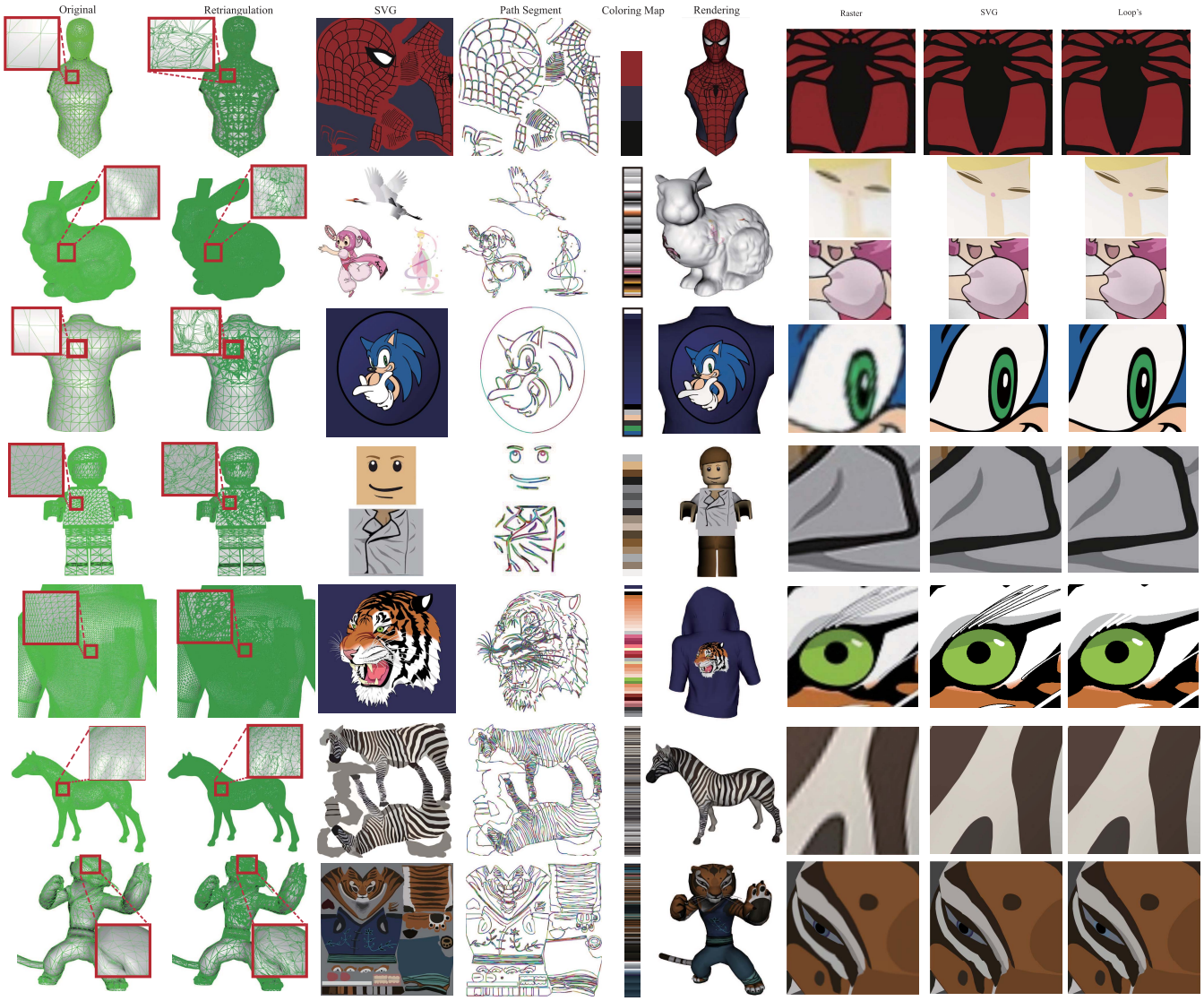


Fig. 10. Results of applying our vector-embedded shading framework on Spiderman, Stanford bunny, Jacket of Sonic, Lego, Tiger Cloth, Zebra, and Tigress. The first and second columns show the original and retriangulated meshes of embedded objects in wireframe. The third, fourth, and fifth columns show the original SVG images, path segment results, and coloring maps. The sixth column shows the rendered results of embedded objects. The seventh, eighth, and ninth columns show the detailed view of several regions rendered with raster texturing, our algorithm, and Loop's algorithm, respectively.

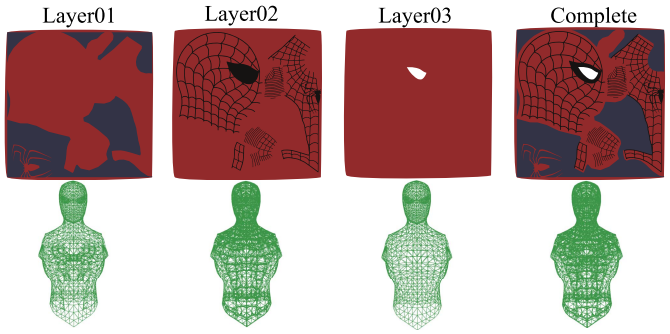


Fig. 11. Simple vector image containing a few layers of coloring operations and Loop's algorithm [3] must retriangulate the original mesh into several different independent meshes that hurt the shading and storage performance.

We would like to understand our performance when comparing with Loop's method. We have separated a vector image into layers of coloring operations based on their drawing order and then used coloring curves of each layer along with the background color to retriangulate the original mesh into a new mesh. Fig. 11 shows an exemplar remeshed result when using our algorithm and Loop's method. As shown in Table III, the triangle count increases with the number of layers. Furthermore, we have also compared the shading performance of our implemented Loop's method with our shading algorithm by measuring the timing statistics when applied to different models in Table IV. Different viewing distances show different amounts of details and require different numbers of coloring and stroking shading operations. Generally, to view the full object, let the object occupy only parts of the screen for less shading computation; hence, the rendering time is shorter. When viewing the details of an object, the time is longer. Overall, our algorithm can achieve real-time rendering efficiency. Furthermore, stroking requires the geometry shader to dynamically place point sprites and is generally more expensive than coloring. In order to verify this

TABLE III

MEMORY USAGE STATISTICS FOR RASTER TEXTURING, LOOP'S VECTORIZATION METHOD [3], AND VECTOR-BASED EMBEDMENT FOR SPIDERMAN, SPIDERMAN USING STROKING PATHS, STANFORD BUNNY, SONIC OF JACKET, LEGO, TIGER CLOTH, ZEBRA, AND TIGRESS.

	Raster			Loop's			Ours	
	# Tris.	Resolution	Mem. (MB)	# Tris.	# Layers	Mem. (MB)	# Tris.	Mem. (MB)
Spider	2584	820×820	2.10 (0.0832,2.02)	38774	3	4.75 (0.93, 3.82)	33236	2.02 (0.922, 1.10, 0)
Spider(S)	2584	406×406	0.577 (0.0832,0.494)	13328	3	1.18 (0.15, 1.03)	7884	0.577 (0.140, 0.251, 0.184)
Bunny	69630	$353 \times 353 \times 3$	3.07 (1.95,1.12)	1621953	23	136 (0.56, 135)	88667	3.07 (0.500, 2.49, 0.081)
Jacket	2168	357×255	0.350 (0.0768,0.273)	19188	7	1.66 (0.13, 1.53)	9951	0.346 (0.142, 0.192,0.0107)
Lego	19034	$160 \times 160 \times 2$	0.686 (0.533,0.153)	98130	5	8.27 (0.08, 8.19)	29853	0.685 (0.0824, 0.603,0)
Cloth	175118	833×833	7.06 (4.98,2.08)	3763921	21	317 (1.73, 315)	216455	7.06 (0.0846, 6.01,0.203)
Zebra	40310	926×926	3.54 (0.968,2.57)	207695	3	17.8 (0.99,16.8)	89861	3.53 (0.621, 2.91, 0)
Tigress	5688	725×725	1.74 (0.162,1.58)	98018	10	8.65 (1.14, 7.51)	34781	1.73 (0.821, 0.907, 0.00632)

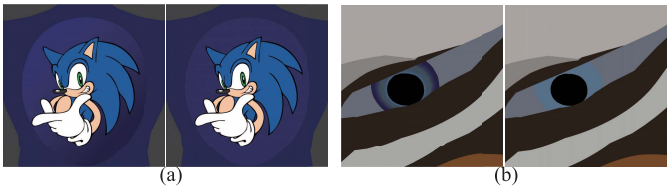
RASTER INDICATES THE RASTER TEXTURING TECHNIQUE, LOOP'S INDICATES LOOP'S VECTORIZATION METHOD [3], AND OURS INDICATES OUR VECTOR-BASED EMBEDMENT. # TRIS DENOTES THE NUMBER OF TRIANGLES FOR EACH METHOD RESPECTIVELY. RESOLUTION IS THE RESOLUTION OF THE MEMORY-MATCHING RASTERIZED TEXTURE. MEM. IS THE MEMORY USAGE FOR EACH METHOD RESPECTIVELY WHERE THE NUMBER IS THE TOTAL MEMORY USAGE, THE FIRST NUMBER INSIDE THE BRACKET IS THE MEMORY USAGE FOR MESH AND THE SECOND IS THE TEXTURE USAGE FOR RASTER TEXTURING, THE NUMBER IS THE TOTAL MEMORY USAGE, THE FIRST NUMBER INSIDE THE BRACKET IS THE MEMORY USAGE FOR THE CURVE-EMBEDDED MESHES AND THE SECOND IS FOR THE INTERIOR MESHES FOR LOOP'S METHOD, THE NUMBER IS THE TOTAL MEMORY USAGE, THE FIRST NUMBER INSIDE THE BRACKET IS THE MEMORY USAGE FOR THE CURVE-EMBEDDED MESHES, THE SECOND IS FOR THE INTERIOR MESHES, AND THE THIRD IS FOR THE STROKING INFORMATION FOR OUR METHOD. FOR RASTER TEXTURING, THE REQUIRED MEMORY USAGE CONTAINS THE TEXTURE IMAGES AND ORIGINAL MESH INCLUDING VERTEX POSITIONS, NORMALS, AND TEXTURE COORDINATES, AND FACE INDICES. FOR OUR ALGORITHM, THE REQUIRED MEMORY USAGE CONTAINS THE RETRIAGULATED MESH, EMBEDDED COLORING AND STROKING INFORMATION, AND COLORING TEXTURE IN GPU FOR THE MODEL.

TABLE IV

TIMING STATISTICS FOR SPIDERMAN, SPIDERMAN USING STROKING PATHS, STANFORD BUNNY, SONIC OF JACKET, LEGO, TIGER CLOTH, ZEBRA, AND TIGRESS.

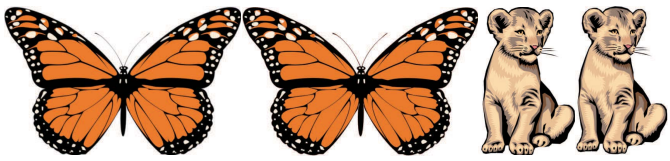
	Normal (FPS)					Zoom In (FPS)					Zoom Out (FPS)				
	Ours	Loop's	R.T.	R(4k)	R(8k)	Ours	Loop's	R.T.	R(4k)	R(8k)	Ours	Loop's	R.T.	R(4k)	R(8k)
Spider	448	99	409	205	2.04	84	48	194	84	1.92	756	308	964	683	6.66
Spider(S)	151	56	409	205	2.04	67	19	194	84	1.92	223	78	964	683	6.66
Bunny	149	19	248	185	1.05	73	7	191	78	1.02	174	37	445	263	1.25
Jacket	223	56	261	189	2.04	52	24	154	83	1.88	496	87	479	297	5.56
Lego	101	33	183	139	2.08	44	21	86	37	1.79	198	88	335	269	7.14
Cloth	56	9	137	126	0.74	34	6	75	69	0.69	73	11	251	235	0.87
Zebra	371	69	356	208	2.08	109	47	117	78	1.69	794	204	442	321	5.26
Tigress	307	35	165	134	2.5	70	12	120	112	1.58	469	83	339	223	5.56

NORMAL IS THE OVERALL FRAME RATE FOR SHADING AN OBJECT WHEN VIEWING IT FULLY, ZOOM IN IS THE OVERALL FRAME RATE FOR SHADING AN OBJECT WHEN ZOOMING IN TO VIEW PARTS OF THE MODEL AND ZOOM OUT IS THE OVERALL FRAME RATE FOR SHADING AN OBJECT WHEN ZOOMING OUT TO HAVE THE MODEL OCCUPY HALF OF THE WINDOW. OURS IS FOR OUR ALGORITHM WITH ANTIALIASING, LOOP'S IS FOR OUR IMPLEMENTED LOOP'S ALGORITHM [3] WITH ANTIALIASING, R.T. IS FOR RASTER TEXTURING USING A RASTERIZED TEXTURE MATCHING THE MEMORY USAGE OF OUR ALGORITHM LISTED IN TABLE III AND R(MK) IS FOR RASTER TEXTURING USING A RASTERIZED TEXTURE WITH ITS WIDTH AND HEIGHT EQUAL TO mk. PLEASE NOTICE THE FRAME RATE IS MEASURED WHEN TURNING ON THE GPU ANTIALIASING FUNCTION. FURTHERMORE, SINCE OPENGL HAS RESOLUTION LIMITATION OF 4096, AND THUS, THE FRAME RATE FOR R(8K) IS MEASURED WITH CPU-BASED RENDERER IMPLEMENTED IN CPU.



(a) (b)

Fig. 12. (a) Rendering result of Jacket with our algorithm on the left and Loop's algorithm on the right. (b)Detailed-view rendering result of Tigress with our algorithm on the left and Loop's algorithm on the right.

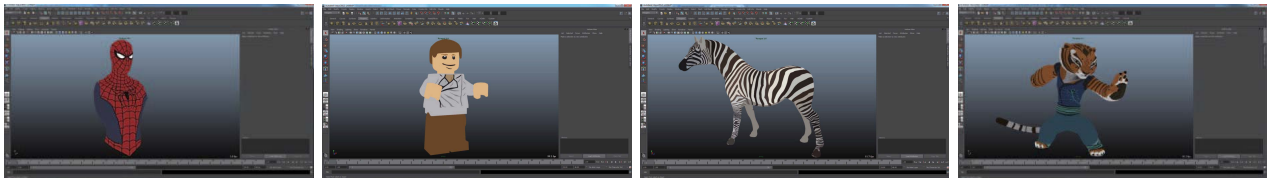


(a) (b)

Fig. 13. (a) Rendering result of Butterfly with our algorithm on the left and Nehab's algorithm on the right. (b) Rendering result of Lion with our algorithm on the left and Nehab's algorithm on the right.

concept, we have also created a stroke-based spiderman SVG for shading the spiderman model. As shown in the first and second rows in Table IV, the performance of coloring-based embedment is better than that of stroking-based embedment. Similarly, we have also analyzed their qualitative performance. When a vector image contains only solid shading units, since both our algorithm and Loop's algorithm use the same spirit, they have comparative quality performance as shown in Fig. 10. However, when containing gradient shading units, our coloring texture can truthfully maintain color variations inside these regions as shown in Fig. 12.

We have also compared our embedded method with the vector texturing algorithm [1] in Table V. We have collected the model and execution code from Nehab's Website [1] for



(a) (b) (c) (d)

Fig. 14. Snapshots of Spiderman (a), Lego (b), Zebra (c), and Tigress (d) in Maya after plugging our algorithm into it.

TABLE V

TIMING STATISTICS IN FPS WHEN COMPARING OUR ALGORITHM WITH NEHAB'S ALGORITHM [1] USING THE DATASET AND CODE POSTED IN THEIR PAPER. ALL DATA ARE COLLECTED AT A RESOLUTION OF 1920×1080 BY TURNING ON THE GPU ANTIALIASING FUNCTION. NO. AND ZO. ARE THE VIEW OF HAVING THE ENTIRE OBJECT FITTING INSIDE THE SCREEN AND ZOOMING-IN VIEW OF THE DETAILS.

	Nehab's			Ours		
	No.	Zo.	Me. (MB)	No.	Zo.	Me.(MB)
Butterfly	35	18	9	71	68	13
Dancer	65	17	6	86	69	11
Skater	52	18	8	87	70	12
Lion	56	18	8	73	66	11
Tiger	45	18	10	67	47	19

ERR. DENOTES THE QUALITY PERFORMANCE AND ME. DENOTES THE MEMORY FOOTPRINT FOR THE METHOD MEASURED BY GPU-Z [31]. OURS DENOTES THE USE OF OUR ALGORITHM AND NEHAB'S DENOTE THE USE OF THE ONE PROPOSED BY NEHAB *et al.* [1]. ANTIALIASING CAN LARGELY REDUCE THE FRAME RATE. FOR EXAMPLE, SHADING THE BUTTERFLY WITH NEHAB'S ALGORITHM ON PLANE HAS A FRAME RATE OF 250/35 WITHOUT/WITH ANTIALIASING.

effective comparison and run all cases on the same computer by turning ON the GPU antialiasing function. When our algorithm use 1.4 to 1.9 times more memory than Nehab's algorithm [1]. In order to gain better rendering efficiency and simpler shading implementation, we have made a trade-off of memory usage. Fig. 13 shows two results. Generally, we can generate GT results using the method described previously if we can get the source SVG images for generation of the rendering results. However, when searching through the Internet, we could not find any visually matching SVG images for Butterfly, Skater, and Tiger, where Butterfly has small differences on its wings, Skater has slight differences at its outlines, and Tiger has slight differences in the stroking style and coloring orders. We could only find the visually matching ones to Dancer and Lion for generating GT for error analysis, where Dancer has 0.0367 for ours and 1.25 for theirs and Lion has 5.44 for ours and 5.46 for theirs. We believe that both ours and Nehab's algorithms can render models' resolution independently, and thus, the qualitative performance is comparative.

Additionally, Since GPU-based and CPU-based renderers consist of complex shading pipelines, it is important to have simple color evaluation mechanism for integration with these renderers. Since our algorithm is simple with only a few shading instructions, it is easy to incorporate into commercial software. In order to demonstrate the flexibility and easiness of our algorithm to integrate with other commercial animation tools, Fig. 14 shows snapshots of a popular commercial software Maya after plugging in our algorithm. Due to the length limitation, complete results and data are shown in supplemental Website.[1]

VII. CONCLUSION

This paper designs a mechanism to highly efficiently shade 3D objects with vector images for high-quality results with a small amount of memory usage and transmission bandwidth by decomposing them into unifying-coloring units, retriangulating objects with constraints built from coloring units along with embedment of coloring and stroking information, and shading with our coloring pixel shaders and stroking geometry shader. Our designed algorithm has the chance to improve the rendering quality, enhance rendering efficiency, and reduce the memory usage and transmission bandwidth of gaming and movie and animation production. As demonstrated, our framework is simple and easy to incorporate into other commercial gaming and animation software. However, our system is not without limitations, and there are a few future research directions. First, there are situations where convex hull splitting can result in a set of small triangles. For example, when two circles intersect at a point, the region around the intersection can result in a large number of splitting. We would like to develop different splitting mechanics for these situations to reduce triangle counts. Second, our system currently does stroking with radius-varied point sprites instead of distance computation, and strokes may cross the triangle boundaries. Furthermore, our system also supports antialiasing through GPU sampling instead of distance computation. In the future, we would like to develop an efficient distance computation method to improve stroking and antialiasing. Third, the retriangulating process may induce a large number of small triangles. We would like to reduce the triangle count by some mesh optimization mechanism. In addition, we would also like to develop a vector-based mesh design tool to both reduce the design barrier and improve rendering efficiency. Fourth, our current algorithm focuses only on SVG-like vector images generally used for nonphotorealistic logos and cliparts and it is hard to record photorealistic information. We would like to develop other types of vector-based texturing techniques for realistic texturing. Fifth, Jeschke *et al.* [23] create diffusion curve-based displacement maps for interesting dynamic geometric effects. This is an interesting application direction, and we would like to adjust our embedded framework for taking geometric vectorization with vector images on surfaces into consideration. Finally, we would like to properly modify the algorithm for modern game engines.

REFERENCES

[1] D. Nehab and H. Hoppe, "Random-access rendering of general vector graphics," *ACM Trans. Graph.*, vol. 27, no. 5, Dec. 2008, Art. no. 135.

[2] Z. Qin, M. D. McCool, and C. Kaplan, "Precise vector textures for real-time 3D rendering," in *Proc. Symp. Interact. 3D Graph. Games (I3D)*, 2008, pp. 199–206.

[3] C. Loop and J. Blinn, "Resolution independent curve rendering using programmable graphics hardware," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 1000–1009, 2005.

[4] K. Kerr, "Windows with C++–introducing Direct2D," *MSN Mag.*, vol. 24, no. 6, Jun. 2009. [Online]. Available: https://msdn.microsoft.com/en-us/magazine/dd861344.aspx

[5] G. Ramanarayanan, K. Bala, and B. Walter, "Feature-based textures," in *Proc. 15th Eurograph. Workshop Rendering Techn.*, 2004, pp. 1–8.

[6] Freescale, Multimedia Applications Division, "i.MX35 accelerated 2D graphics: Optimizing 2D graphics with OpenVG and i.MX35," Freescale Semicond., Inc., Austin, TX, USA, Appl. Note AN3975, 2010.

[7] R. Huang and S.-I. Chae, "Implementation of an OpenVG rasterizer with configurable anti-aliasing and multi-window scissoring," in *Proc. 6th IEEE Int. Conf. Comput. Inf. Technol. (CIT)*, Sep. 2006, p. 179.

[8] D. Kim, K. Cha, and S. I. Chae, "A high-performance OpenVG accelerator with dual-scanline filling rendering," *IEEE Trans. Consum. Electron.*, vol. 54, no. 3, pp. 1303–1311, Aug. 2008.

[9] Y. Kokojima, K. Sugita, T. Saito, and T. Takemoto, "Resolution independent rendering of deformable vector objects using graphics hardware," in *Proc. ACM SIGGRAPH Sketches (SIGGRAPH)*, 2006, Art. no. 118.

[10] A. J. Rueda, J. R. de Miras, and F. Feito, "GPU-based rendering of curved polygons using simplicial coverings," *Comput. Graph.*, vol. 32, no. 5, pp. 581–588, 2008.

[11] M. J. Kilgard and J. Bolz, "GPU-accelerated path rendering," *ACM Trans. Graph.*, vol. 31, no. 6, 2012, Art. no. 172.

[12] P. Sen, M. Cammarano, and P. Hanrahan, "Shadow silhouette maps," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 521–526, Jul. 2003.

[13] P. Sen, "Silhouette maps for improved texture magnification," in *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graph. Hardw. (HWWS)*, 2004, pp. 65–73.

[14] J. Tumblin and P. Choudhury, "Bixels: Picture samples with sharp embedded boundaries," in *Proc. 15th Eurograph. Conf. Rendering Techn. (EGSR)*, 2004, pp. 255–264.

[15] S. Lefebvre and H. Hoppe, "Perfect spatial hashing," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 579–588, Jul. 2006.

[16] S. Lefebvre and H. Hoppe, "Compressed random-access trees for spatially coherent data," in *Proc. 18th Eurograph. Conf. Rendering Techn. (EGSR)*, 2007, pp. 339–349.

[17] M. Tarini and P. Cignoni, "Pinchmaps: Textures with customizable discontinuities," *Comput. Graph. Forum*, vol. 24, no. 3, pp. 557–568, 2005.

[18] J. Loviscach, "Efficient magnification of bi-level textures," in *Proc. ACM SIGGRAPH Sketches*, 2005, Art. no. 131.

[19] N. Ray, T. Neiger, X. Cavin, and B. Lévy, "Vector texture maps on the GPU," Inst. ALICE (Algorithms, Comput., Geometry Image Dept. INRIA Nancy Grand-Est/Loria), Tech. Rep. ALICE-TR-05-003, 2005.

[20] E. Parilov and D. Zorin, "Real-time rendering of textures with feature curves," *ACM Trans. Graph.*, vol. 27, no. 1, 2008, Art. no. 3.

[21] Z. Qin, M. D. Mccool, and C. S. Kaplan, "Real-time texture-mapped vector glyphs," in *Proc. Symp. Interact. 3D Graph. Games*, 2006, pp. 125–132.

[22] C. Yuksel, J. Keyser, and D. H. House, "Mesh colors," *ACM Trans. Graph.*, vol. 29, no. 2, 2010, Art. no. 15. [Online]. Available: http://doi.acm.org/10.1145/1731047.1731053

[23] S. Jeschke, D. Cline, and P. Wonka, "Rendering surface details with diffusion curves," *ACM Trans. Graph.*, vol. 28, no. 5, 2009, Art. no. 117.

[24] R. Santina, "Resolution independent nurbs curves rendering using programmable graphics pipeline," in *Proc. 21st Int. Conf. Comput. Graph. Vis.*, 2011, pp. 1–4.

[25] Q. Lou and L. Liu, "Curve intersection using hybrid clipping," *Comput. Graph.*, vol. 36, no. 5, pp. 309–320, 2012.

[26] P. S. Heckbert, "Discontinuity meshing for radiosity," in *Proc. 3rd Eurograph. Workshop Rendering*, 1992, pp. 203–226.

[27] D. Lischinski, F. Tampieri, and D. P. Greenberg, "Discontinuity meshing for accurate radiosity," *IEEE Comput. Graph. Appl.*, vol. 12, no. 6, pp. 25–39, Nov. 1992.

[28] C.-Y. Yao, M.-T. Chi, T.-Y. Lee, and T. Ju, "Region-based line field design using harmonic functions," *IEEE Trans. Vis. Comput. Graphics*, vol. 18, no. 6, pp. 902–913, Jun. 2012.

[29] K. Hormann, B. Lévy, and A. Sheffer, "Mesh Parameterization: Theory and Practice," in *Proc. ACM SIGGRAPH Courses (SIGGRAPH)*, 2007, Art. no. 1. [Online]. Available: http://doi.acm.org/10.1145/1281500.1281510

[30] R. Schmidt, C. Grimm, and B. Wyvill, "Interactive decal compositing with discrete exponential maps," in *Proc. ACM SIGGRAPH Papers (SIGGRAPH)*, 2006, pp. 605–613.

[31] (2015). *GPU-Z*. [Online]. Available: https://www.techpowerup.com/gpuz/

Chih-Yuan Yao (M'15) received the M.S. and Ph.D. degrees in computer science and information engineering from National Cheng Kung University, Tainan, Taiwan, in 2003 and 2010, respectively.

He is currently an Assistant Professor with the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan. His current research interests include computer graphics, including mesh processing and modeling, and nonphotorealistic rendering.

Kuang-Yi Chen received the B.S. and M.S. degrees from the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, in 2011 and 2014, respectively.

He is currently with International Game System, Taipei. His current research interests include computer graphics and GPU shader.

Hong-Nian Guo received the B.S. degree from the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, in 2015, where he is currently pursuing the M.S. degree.

His current research interests include computer graphics, vision, and nonphotorealistic rendering.

Cheng-Chi Li received the B.S. degree from the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, in 2014, where he is currently pursuing the M.S. degree.

His current research interests include computer graphics, vision, and parallel computing.

Yu-Chi Lai (M'14) received the B.S. degree from the Electrical Engineering Department, National Taiwan University, Taipei, Taiwan, in 1996, the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Wisconsin, Madison, WI, USA, in 2003 and 2009, respectively, and the M.S. and Ph.D. degrees in computer science, in 2004 and 2010, respectively.

He is currently an Assistant Professor with the National Taiwan University of Science and Technology, Taipei. His current research interests include graphics, vision, and multimedia.