Revision Summary and Replies to Reviews

The authors are grateful for the associate editor's and reviewers' helpful comments which are very valuable for improving this paper. We have revised the paper according to the comments and summarized the major changes in this revision summary.

Summary of the revision:

Based on the suggestions from the associate editor and reviewers, we have made the follow major change: First, we have followed the recommendation of reviewers to clearly address our main novelty lying in real-time rendering 3D resolution independent animations. Accordingly, we have focused our main technical contributions on real-time resolution independent coloring with GPU-based and CPU-based rendering efficiency, texturing memory usage and transmission efficiency and effectiveness, and easiness to incorporate with commercial softwares such as Blender and Maya. Based on these, we have adjusted the discussion and contribution claims in **Introduction**, and further discussed our limitations and future works in **Conclusion and Future Works**. Second, high resolution texturing requires a large amount of memory usage and transmission bandwidth, and our algorithm can largely reduce their usage which fit the topics of TCSVT. Accordingly, we have added a few sentences in **Introduction** to make this link clear. Second, we have adjusted the discussion in **Related Works** to [Loop et al. 2005], [Nehab et al. 2008], and [Qin et al. 2008] in order to clarify our contributions over these algorithms and also toned town the critique to these algorithms for those we can prove in our demonstration. Furthermore, we have also added two related works, [Jeschke et al. 2009] and [Santina et al. 2006] in **Related Works**. Third, we have removed the repetitive discussion among our article and figure captions to condense our article. Fourth, we have added the relationship with discontinuity mesh in **Coloring Field and Texture Construction**. Since we have derived our coloring field computation from [Yao et al. 2011], we have added it for reference and given more implementation details in **Coloring Field and Texture Construction** of Section IV.B. Fifth, we have added quality and timing comparison to raster texturing while using textures of different resolution in **Results and Discussion**. We have added quality comparison to Loop's algorithm [Loop et al. 2005] along with several screenshots in **Results and Discussion**. We have also added quality and memory usage comparison to Nehab's algorithm [Nehab et al. 2008] along with several screenshots in **Results and Discussion**. Finally, we have added an extra future work for application our algorithm for displacement information to create dynamic geometric effects. The following table summarizes these changes and their locations. Those important changes are highlighted with blue.

Location(s)	Brief descriptions of changes	Revised content type(s)
Fig. 1	Add to illustrating our motivation for real-time resolution free 3D animations.	Add Figure

Fig. 3	Adjust the caption to remove repetitive discussion and condense our article.	Text
Fig. 5	Adjust the caption to detail how to split curves to ensure at most one segment in a triangle and remove repetitive discussion to condense our article.	Text
Fig. 8	Adjust the caption to condense our discussion and make our article more compact.	Text
Fig. 9	Adjust the caption to detail how to use our designed geometry shader to draw outlines and remove repetitive discussion to condense our article.	Text
Fig. 10	Add a column to show the detail viewed results of Loop's algorithm	Figure
Fig. 12	Add to show the rendering results of our algorithm and Loop's method.	Add Figure
Fig. 13	Add to show the rendering results of our algorithm and Nehab's method.	Add Figure
Table III	Add to give error analysis of our algorithm and raster texturing with a texture of different resolutions.	Add Table
Table IV	Add and adjust table contents and captions to give rendering frame rates for our algorithm, Loop's algorithm and raster texturing with different texturing resolutions.	Table and Caption
Table V	Add and adjust table contents and captions to give error analysis and memory usage of our algorithm and Nehab's algorithm.	Table and Caption
Abstract	Add a few sentences to highlight our novelties.	Text
Sect. 1 Para. 1	Add description to motivate our algorithm and the relationship to the journal.	Text
Sect. 1 Para. 2	Adjust the description to better describe our algorithm over other relative algorithms	Text
Sect. 1 Para. 3	Adjust our contributions.	Text
Sect. 2 Para. 3	Add the discussion about [Jeschke et al. 2009]	Text
Sect. 2 Para. 4	Adjust the description about our advantages over [Nehab et al. 2008]	Text
Sect. 2 Para. 4	Add the discussion about [Santina et al. 2006] and adjust the description about our advantage over	Text

	[Loop et al. 2005]	
Sect. 3 Para. 1	Adjust to address the focus of our algorithm	Text
Sect. 3. Para. 2	Adjust the overview to condense our article.	Text
Sect. 4.B Para. 1	Add two related works, [Heckbert et al.1992][Lischinski et al. 1992] and add more implementation details about coloring field computation with a reference to [Yao et al. 2011] for more details.	Text
Sect. 4.D. Para. 1	Adjust to address the focus of our algorithm	Text
Sect 5. Para. 1	Add descriptions about how we achieve interactive authoring.	Text
Sect. 5 Para. 2	Add descriptions about error analysis and more comparisons to raster texturing with a texture of different resolutions	Text
Sect. 5 Para. 3	Add descriptions about error analysis about Loop's algorithm.	Text
Sect. 5 Para. 4	Add descriptions about memory usage and error analysis to Nehab's algorithm.	Text
Sect. 6	Adjust to address the focus of our algorithm and add a future work for extending our algorithm for dynamic geometric editing.	Text

Reply to Associate Editor

Comment 1:

Please clarify the novelty of the paper.

Reply: Thanks for AE's reminding. We have followed the recommendation of reviewers to clearly address our main novelty lying in real-time 3D resolution independent animations. Accordingly, we have focused our main technical contributions on real-time resolution independent coloring with GPU-based and CPU-based rendering efficiency, texturing memory usage and transmission efficiency and effectiveness, and easiness to incorporate with commercial softwares such as Blender and Maya. We have decomposed coloring paths into unifying coloring paths and primitives into unifying coloring units to reduce memory usage, remove overdrawing, and shading complexity of [Loop et al. 2005] and remove overdrawing of [Nehab et al. 2008], and [Qin et al. 2008]. We have created coloring field and a coloring texture to get correct color computation for gradient coloring field and simplify shading instructions of [Loop et al. 2005] and simplify shading programming of [Nehab et al. 2008], and [Qin et al. 2008]. All two aspects incorporating with information embedment allow us to real-time render 3D animation with high quality results when comparing to raster texturing. The complete discussion in our article is as follows:

> *In the past, when texturing objects with vector images, it is general practice to convert them into raster images which can be accelerated by GPU shaders but the compactness and antialiasing abilities are lost after conversion. Furthermore, GPU-based renderers have limitation in allowable texture resolution, and it is really costly to render high-resolution textured objects with 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. Nehab et al. [Nehab et al. 2008], and Qin et al. [Qin et al. 2008] pack coloring information in an acceleration structure for shading, but packing induces overdrawing and deteriorate their rendering performance. Furthermore, their algorithms have complex shading and indirect memory accessing schemes which may bring barriers to plug them into real-time applications and off-line commercial animation softwares. Loop et al. [Loop et al. 2005] embed distance testing into meshes for shading, but multiple overlapping meshes induce serious overdrawing issues, require large amount of extra memory usage, and need complex composition and shading instructions for avoiding z-fighting issues. Furthermore, the algorithm does not provide any solution for gradient coloring operations and stroking. In order to overcome these issues, our algorithm decomposes these overlapping coloring regions into **unifying-coloring** units which are coloring regions using the same set of path-based coloring operations. Furthermore, to avoid indirect memory accessing, 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 easily incorporate with the rendering pipeline of commercial animation softwares 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 for rendering. Thus, our system embeds stroking paths into the object as control vertices and designs a geometry shader to place point sprites with a varying radius computed with the stroke width, viewing information, and embedded triangle transformation. At the end, we have embedded vector images into 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 against raster texturing to show that our algorithm is more memory effective with better rendering quality. While comparing to Loop's method [Loop et al.], our algorithm is more memory effective with better rendering efficiency and precise color reproduction. Furthermore, it is also more shading efficient and programming simpler than Nehab's method [Nehab et al.2008].

Furthermore, we have adjusted the claims in **Introduction** as follows:

This work 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:

- *Rendering 3D animated objects with high-resolution textures are 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 work 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 curve distance and coloring information. Later, we make use of our designed 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 texture for rendering and transmission efficiency, and we can easily concatenate all coloring textures with linear mapping on coloring coordinates for integration with games. Overall speaking, our algorithm enables high-quality real-time GPU-based and efficient CUP-based 3D animation rendering through our efficient SVG-embedded rendering pipeline.*
- *It is necessary to use a large number of high-quality textures for real-time 3D applications and movie production, and thus, to efficiently reduce texture memory usage plays an important role for high-quality results. This work minimizes memory usage by minimizing embedded information, localizing vertex-embedded shading data, and ensuring fixed-length shading instructions for data compactness. In other words, our algorithm*

maintain high-quality results without aliasing while only using a little amount of memory. As demonstrated in the results, our vector-embedded shading framework can real-time render complex 3D objects in high quality using very few and simple shading instruction without aliasing whiling limiting the texture memory usage and data transferring bandwidth.

Comment 2:

The topic of this paper is a bit unusual for TCSVT. However, vector image can be broadly considered as image and the AE felt it is ok to include this paper in the journal.

Reply: Thanks for AE's clarification. We have submitted the article to TCSVT due to the same reason. High resolution textures are important for high-quality production in gaming and movie industries, but they generally take up a large amount of texture memory and transmission bandwidth. Since vector images can largely increase the resolution of raster images without increasing memory usage in the image processing field, and thus, we have focused our main technical contributions on real-time resolution independent coloring with GPU-based and CPU-based rendering efficiency, texturing memory usage and transmission efficiency and effectiveness, and easiness to incorporate with commercial softwares such as Blender and Maya. We have added a motivation example and adjusted our description in Introduction as

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 with a texture of 4096x4096, 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 deteriorate rendering efficiency largely. Hence it is a trend to increase perceived quality by other means than just increasing the number of texels which can be described as another type of data compression problems. 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 work aims at developing a resolution independent vector-embedded shading method for 3D animated objects for compactness and infinite resolution with real-time animation efficiency and easiness in plugging into commercial animation softwares.

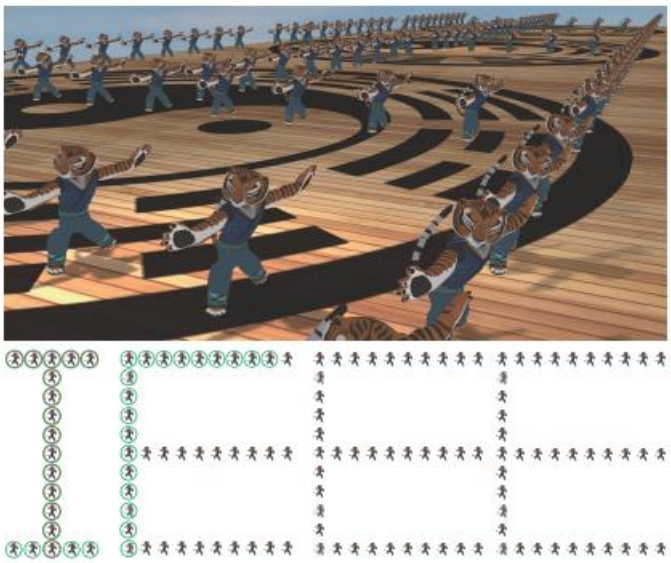

Fig. 1. The top shows the rendering result of a complex scenes consisting of 133 Tigress as distinct characters of independent meshes and textures using our algorithm to illustrate our motivation to reduce the amount of texture memory usage, maintain the rendering quality, and enhance rendering efficiency with GPU-based coloring. The bottom shows the composition of them as "IEEE". While using a computer with a $1GB$ graphics card, we can only load and shade 15 Tigress marked with red rings using raster texturing with a texture resolution of 4096×4096 which is the maximally available resolution of OpenGL. While using a computer with a $16GB$ memory, we can only load and shade 38 Tigress in Blender marked with green rings using raster texturing with a texture resolution of 8192×8192 for matching our rendering quality. To shade 15 Tigress with raster texturing on GPUs takes $2.7e-3$ seconds in NVidia Titan with OpenGL, to shade 133 Tigress in Intel I7-4960K and $64G$ memory with Blender takes 3 seconds, and to shade 133 Tigress in NVidia Titan with our algorithm takes $1.52e-2$ seconds.

Reply to review 1

Comment 1:

Clearly clarify the novelty.

Reply: Thanks for reviewer's reminding. We have followed the recommendation of reviewers to clearly address our main novelty lying in real-time 3D resolution independent animations. Accordingly, we have focused our main technical contributions on real-time resolution independent coloring with GPU-based and CPU-based rendering efficiency, texturing memory usage and transmission efficiency and effectiveness, and easiness to incorporate with commercial softwares such as Blender and Maya. We have decomposed coloring paths into unifying coloring paths and primitives into unifying coloring units to reduce memory usage, remove overdrawing, and shading complexity of [Loop et al. 2005] and remove overdrawing of [Nehab et al. 2008], and [Qin et al. 2008]. We have created coloring field and a coloring texture to get correct color computation for gradient coloring field and simplify shading instructions of [Loop et al. 2005] and simplify shading programming of [Nehab et al. 2008], and [Qin et al. 2008]. All two aspects incorporating with information embedment allow us to real-time render 3D animation with high quality results when comparing to raster texturing. The complete discussion in our article is as follows:

> *In the past, when texturing objects with vector images, it is general practice to convert them into raster images which can be accelerated by GPU shaders but the compactness and antialiasing abilities are lost after conversion. Furthermore, GPU-based renderers have limitation in allowable texture resolution, and it is really costly to render high-resolution textured objects with 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. Nehab et al. [Nehab et al. 2008], and Qin et al. [Qin et al. 2008] pack coloring information in an acceleration structure for shading, but packing induces overdrawing and deteriorate their rendering performance. Furthermore, their algorithms have complex shading and indirect memory accessing schemes which may bring barriers to plug them into real-time applications and off-line commercial animation softwares. Loop et al. [Loop et al. 2005] embed distance testing into meshes for shading, but multiple overlapping meshes induce serious overdrawing issues, require large amount of extra memory usage, and need complex composition and shading instructions for avoiding z-fighting issues. Furthermore, the algorithm does not provide any solution for gradient coloring operations and stroking. In order to overcome these issues, our algorithm decomposes these overlapping coloring regions into **unifying-coloring** units which are coloring regions using the same set of path-based coloring operations. Furthermore, to avoid indirect memory accessing, 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 easily incorporate with the rendering pipeline of commercial animation softwares 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 for rendering. Thus, our system embeds stroking paths into the object as control vertices and designs a geometry shader to place point sprites with a varying radius computed with the stroke width, viewing information, and embedded triangle transformation. At the end, we have embedded vector images into 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 against raster texturing to show that our algorithm is more memory effective with better rendering quality. While comparing to Loop's method [Loop et al.], our algorithm is more memory effective with better rendering efficiency and precise color reproduction. Furthermore, it is also more shading efficient and programming simpler than Nehab's method [Nehab et al.2008].

Furthermore, we have adjusted the claims in **Introduction** as follows:

This work 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:

- *Rendering 3D animated objects with high-resolution textures are 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 work 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 curve distance and coloring information. Later, we make use of our designed 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 texture for rendering and transmission efficiency, and we can easily concatenate all coloring textures with linear mapping on coloring coordinates for integration with games. Overall speaking, our algorithm enables high-quality real-time GPU-based and efficient CUP-based 3D animation rendering through our efficient SVG-embedded rendering pipeline.*
- *It is necessary to use a large number of high-quality textures for real-time 3D applications and movie production, and thus, to efficiently reduce texture memory usage plays an important role for high-quality results. This work minimizes memory usage by minimizing embedded information, localizing vertex-embedded shading data, and ensuring fixed-length shading instructions for data compactness. In other words, our algorithm*

maintain high-quality results without aliasing while only using a little amount of memory. As demonstrated in the results, our vector-embedded shading framework can real-time render complex 3D objects in high quality using very few and simple shading instruction without aliasing whiling limiting the texture memory usage and data transferring bandwidth.

Comment 2:

Another problem might be the long time it takes to embed a vector graphics on an object. This precludes interactive authoring and dynamic effects as shown in the work of Jeschke et al. mentioned further above.

Reply: Thanks for reviewer's reminding. Authoring basically focuses on editing the texture and align the texture onto the 3D object. We can easily achieve interactive authoring through general raster texturing by first rasterizing vector images into raster images for interactive authoring to avoid expensive retriangulation and path segment. We have addressed this in our discussion in **Results** as follows.

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 into objects for real-time and resolution independent shading.

[Jeschke et al. 2009] can apply diffusion curve images to dynamically create displacement maps for interesting geometric effects. It provides an interesting application direction and we will add this function in the future. We have added this as one of our future works as follows.

Fifth, Jeschke et al. [Jeschke et al. 2009] 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.

Comment 3:

Another potential problem is the lengthy description with several repetitions of the general goals and properties. A more compact, concentrated writeup might strengthen the paper.

Reply: Thanks for reviewer's reminding. We have removed repetitive discussion and condensed our articles.

Comment 4:

It seems unclear how well the paper topic fits to this journal.

Reply: Thanks for the reviewer's reminding. As classified by Associate Editor, we have submitted the article to TCSVT due to the same reason. High resolution textures are important for high-quality production in gaming and movie industries, but they generally take up a large amount of texture memory and transmission bandwidth. Since vector images can largely increase the resolution of raster images without increasing memory usage in the image processing field, and thus, we have focused our main technical contributions on real-time resolution independent coloring with GPU-based and CPU-based rendering efficiency, texturing memory usage and transmission efficiency and effectiveness, and easiness to incorporate with commercial softwares such as Blender and Maya. We have added a motivation example and adjusted our description in Introduction as

> *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 with a texture of 4096x4096, 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 deteriorate rendering efficiency largely. Hence it is a trend to increase perceived quality by other means than just increasing the number of texels which can be described as another type of data compression problems. 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 work aims at developing a resolution independent vector-embedded shading method for 3D animated objects for compactness and infinite resolution with real-time animation efficiency and easiness in plugging into commercial animation softwares.*

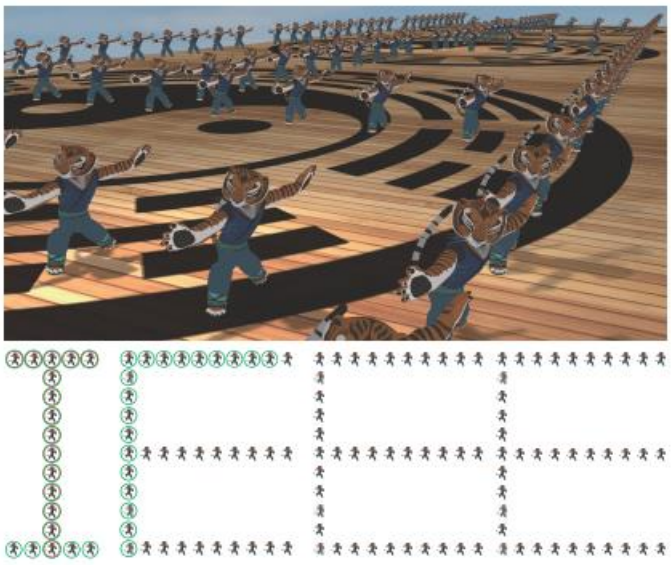

Fig. 1. The top shows the rendering result of a complex scenes consisting of 133 Tigress as distinct characters of independent meshes and textures using our algorithm to illustrate our motivation to reduce the amount of texture memory usage, maintain the rendering quality, and enhance rendering efficiency with GPU-based coloring. The bottom shows the composition of them as "IEEE". While using a computer with a $1GB$ graphics card, we can only load and shade 15 Tigress marked with red rings using raster texturing with a texture resolution of 4096×4096 which is the maximally available resolution of OpenGL. While using a computer with a $16GB$ memory, we can only load and shade 38 Tigress in Blender marked with green rings using raster texturing with a texture resolution of 8192×8192 for matching our rendering quality. To shade 15 Tigress with raster texturing on GPUs takes $2.7e-3$ seconds in NVidia Titan with OpenGL, to shade 133 Tigress in Intel I7-4960K and $64G$ memory with Blender takes 3 seconds, and to shade 133 Tigress in NVidia Titan with our algorithm takes $1.52e-2$ seconds.

Comment 5:

Include and discuss relevant literature.

1. Stefan Jeschke, David Cline, and Peter Wonka. 2009. Rendering surface details with diffusion curves. In ACM SIGGRAPH Asia 2009 papers. ACM press, Article 117, 8 pages.
2. HECKBERT, P. 1992. Discontinuity meshing for radiosity. In Third Eurographics Workshop on Rendering, 203-226.
3. LISCHINSKI, D., TAMPIERI, F., AND GREENBERG, D. P. 1992. Discontinuity meshing for accurate radiosity. IEEE Comput. Graph. Appl. 12, 6, 25-39.

Reply: Thanks for reviewer's reminding. We have added the following

1. [Jeschke et al. 2009] 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 the following limitations: To incorporate regions with a high density of details requires fine lattices for the entire image which globally increase storage cost. Additionally, discontinuities may occur

when crossing cells to induce some curve shading artifacts. Line and curve representations may still miss certain sharp features to cause blurring artifacts. We have added this discussion in our related works as follows:

Jeschke et al. [Jeschke et al. 2009] 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.

2. [Heckbert et al. 1992] and [Lischinski et al. 1992] refine their estimation by using fine elements along the discontinuity(D0, D1, and D2). Generally, these discontinuities are estimated based on geometry information existing in the world, and our algorithm focuses on finding the discontinuity at intersections of paths and intersection of gradient fields and having a better interpolation mechanic for precise color computation. Therefore, we have added this part into our discussion in Section IV. B. "Coloring Field and Texture Construction" as follows.

Discontinuity mesh [Heckbert et al. 1992][Lischinski et al. 1992] refines radiosity elements along possibly 0-th, 1-st, and 2-nd order radiosity-estimation discontinuity for more precise estimation, and our algorithm bears the same spirit to split mesh along the possible discontinuity along the coloring computation.

Comment 6:

As a minor remark, Section II makes strong statements about reduced rendering speed (due to potential overdraw) and high memory requirements of existing techniques, but the paper finally misses to make a strong point that the proposed method is that much faster and needs a lot less memory. How fare memory requirements compared to Nehabs and Hoppes method, for example? It seems that all given examples (showing not very high complexity) could also be rendered with existing techniques. So I propose to tune down the critique on existing techniques a little.

Reply: Thanks for reviewer's reminding. We have toned town the critique for those we can proven and the details are as follows.

***Vector-based texturing**: Nehab et al. [Nehab et al. 2008] and Qin et al. [Qin et al. 2008] 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. 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 indirection memory accessing due to the variable-length packing information. These may bring barriers to integrate with real-time applications and off-line commercial softwares. 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 only computing 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 softwares.

Comment 7:

Add a memory-footprint comparison to Nehab and Hoppe.

Reply: Thanks for reviewer's reminding. We have added a memory-footprint in Table VI and relative discussion in **Results.** We have adapted them out in the following for clarity.

Our algorithm can perform 1.5 to 4 times faster than Nehab's algorithm~\cite{Nehab2008} when using 1.4 to 1.9 more memory. 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 Internet, we cannot 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 can only find the visual 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.

	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

TABLE V

This shows the timing statistics in FPS when comparing our algorithm with Nehab's algorithm [1] using the data set and code posted in their paper. All data is collected in the resolution of 1920×1080 with turning on the GPU antialiasing function. No. and Zo. are the view of having the entire object fitting inside the screen and zooming view of the details. Err. denotes the quality performance and Me. denotes the memory footprint for the method measured by GPU-Z [30]. 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.

Fig. 13. (a) This shows the rendering result of Butterfly with our algorithm in the left and Nehab's algorithm in the right. (b) This shows the rendering result of Lion with our algorithm in the left and Nehab's algorithm in the right.

Reply to review 2

Comment 1:

Related works are well covered but include and discuss “Resolution Independent NURBS Curves Rendering using Programmable Graphics Pipeline”

Reply: Thanks for reviewer’s reminding. We have added the paper into our discussion and we have adapted the discussion from our articles.

> *Santina et al. [Santina2011 et al.] extend Loop's idea for boundaries defined with Non-uniform Rational B-Splines (NURBS) by transforming them into low-order quadratic forms for embedment. Their algorithm has similar limitations as Loop's method.*

Comment 2:

Give more details about "region's colors are computed based on unifying coloring instructions set by designers." in Section IV. B.

Reply: Thanks for reviewer’s reminding. Fundamentally, our harmonious coloring field computation is adapted from [Yao et al. 2011] and we have added it into the reference. We have also added more details in the section as follows:

> *Discontinuity mesh [Heckbert et al. 1992][Lischinski et al. 1992] refines radiosity elements along possibly 0-th, 1-st, and 2-nd order radiosity-estimation discontinuity for more precise estimation, and our algorithm bears the same spirit to split the mesh along the possible discontinuity along the coloring 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. Our system further decomposes the gradient coloring regions based on the constraints of their color and gradient direction, and later, we use the decomposed results as constraints to compute the color index map through the harmonic function composition method proposed by Yao et al.[Yao et al. 2011]. In order to have simplified and fixed-count shading instructions, we design a precomputed coloring texture to record all possible colors generated by all sets of coloring operations. We first denote each coloring unit as $\boldsymbol{CU_i}$, and label them as a solid or gradient unit based on whether their coloring instructions generate a solid or gradient color inside the unit.*
>
> $$\mathbf{CU}_i = \begin{cases} S, \text{ if } \mathbf{CU}_i(\overline{p}) = \mathbf{T}_s \\ G, \text{ if } \mathbf{CU}_i(\overline{p}) = \\ \quad \mathbf{G}(\mathbf{C}^b_{i,0}, \mathbf{C}^e_{i,0}, \tilde{C}_{i,0}) \; op \; \mathbf{G}(\mathbf{C}^b_{i,1}, \mathbf{C}^e_{i,1}, \tilde{C}_{i,1}) \cdots \end{cases} \quad (1)$$
>
> *where S/G denotes the label of the unit, $\boldsymbol{T_s}$ is a solid color and recorded as a single texel in our 1D coloring texture, $\boldsymbol{G(C_b, C_e, \widetilde{C})}$ is the gradient coloring operation based on the coloring constraints, $\boldsymbol{C_b}$ and $\boldsymbol{C_e}$, and geometric constraint, $\boldsymbol{\widetilde{C}}$, which may be a line or a point*

and op is a composition operation inside the unit. Both $\boldsymbol{G}()$ *and op define a set of gradient coloring operations set by designers on* $\boldsymbol{C_b}$ *and* $\boldsymbol{C_e}$ *and* $\widetilde{\boldsymbol{C}}$ *and our system sets up a coloring field as* $\boldsymbol{\nabla CU_i}$*. 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 labelled as a solid color, it occupies 1 texel in the 1D coloring texture; otherwise, a gradient coloring set is formulated as a harmonic function [Yao et al. 2011] to get a 1D scalar field for each triangle vertex,* $\boldsymbol{G(u,v) = \Delta CU_i}$*, where (u,v) is the texture coordinate of the vertex estimated by exponential mapping [Schmidt et al. 2006]. 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. (c) gives an exemplar coloring texture. Later, Section IV-D gives the details about how to embed the shading information into vertices.*

(a) (b) (c)

Fig. 4. (a) This shows the result of path intersection and segmentation for a penguins 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. Totally, there are 76 path segments. (b) This shows the result of coloring unit formation. Each region enclosed by one or more path segments is a unifying coloring unit. Totally, there are 32 coloring units. (c) This shows an 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.

Comment 3:

Perform some experiments comparing timing and qualities between the rendering results of raster texturing using different resolutions and vector texturing.

Reply: Thanks for reviewer's reminding. We have added experiments to comparing timing and qualities between the rendering results of raster texturing using different resolutions and vector texturing in **Results**. We have summarized the performance in Table II, III, and IV. The following is the discussion we added into our articles in **Results**.

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 on 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 to 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 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 do. 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 sometimes is 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 Tigress acting as distinct characters with independent meshes and textures for demonstrating our real-time high-quality shading ability.

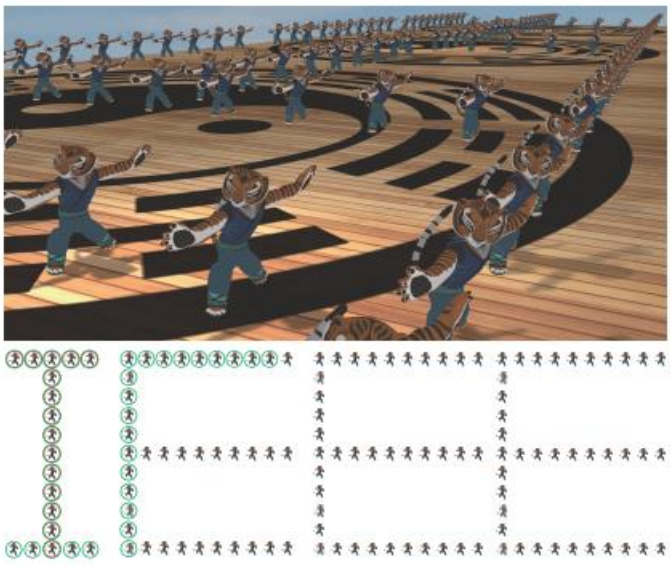

Fig. 1. The top shows the rendering result of a complex scenes consisting of 133 Tigress as distinct characters of independent meshes and textures using our algorithm to illustrate our motivation to reduce the amount of texture memory usage, maintain the rendering quality, and enhance rendering efficiency with GPU-based coloring. The bottom shows the composition of them as "IEEE". While using a computer with a $1GB$ graphics card, we can only load and shade 15 Tigress marked with red rings using raster texturing with a texture resolution of 4096×4096 which is the maximally available resolution of OpenGL. While using a computer with a $16GB$ memory, we can only load and shade 38 Tigress in Blender marked with green rings using raster texturing with a texture resolution of 8192×8192 for matching our rendering quality. To shade 15 Tigress with raster texturing on GPUs takes $2.7e-3$ seconds in NVidia Titan with OpenGL, to shade 133 Tigress in Intel I7-4960K and $64G$ memory with Blender takes 3 seconds, and to shade 133 Tigress in NVidia Titan with our algorithm takes $1.52e-2$ seconds.

	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

TABLE II

This shows the 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 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].

	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)

TABLE III

This shows the 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 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.

	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

TABLE IV

This shows the timing statistics for Spiderman, Spiderman using stroking paths, Stanford bunny, Sonic of Jacket, Lego, Tiger Cloth, Zebra, and Tigress. 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.

Comment 4:

Show some detailed comparison with existing methods (Loop and Nehab) zooming in some parts of the textured mesh and comparing the level of detail and error when the texture is rendered over the retriangulated model.

Reply: Thanks for reviewer's reminding. We have shown the detail-viewed results of our algorithm and Loop's algorithm in Fig. 10 and 12. We have also added error analysis of our algorithm and Loop's method. The complete description in our article is as follows:

> *We would like to understand our performance when comparing to 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 results 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 applying to different models in Table IV. Because different viewing distances show different amount 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 and thus, 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 concept, we have also created a stroke-based spiderman SVG to shading the spiderman model. As shown in the first and second rows in Table IV, the performance of coloring-based embedment is better than stroking-based embedment. Similarly, we have also analyzed their qualitative performance. When a vector image contain 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.

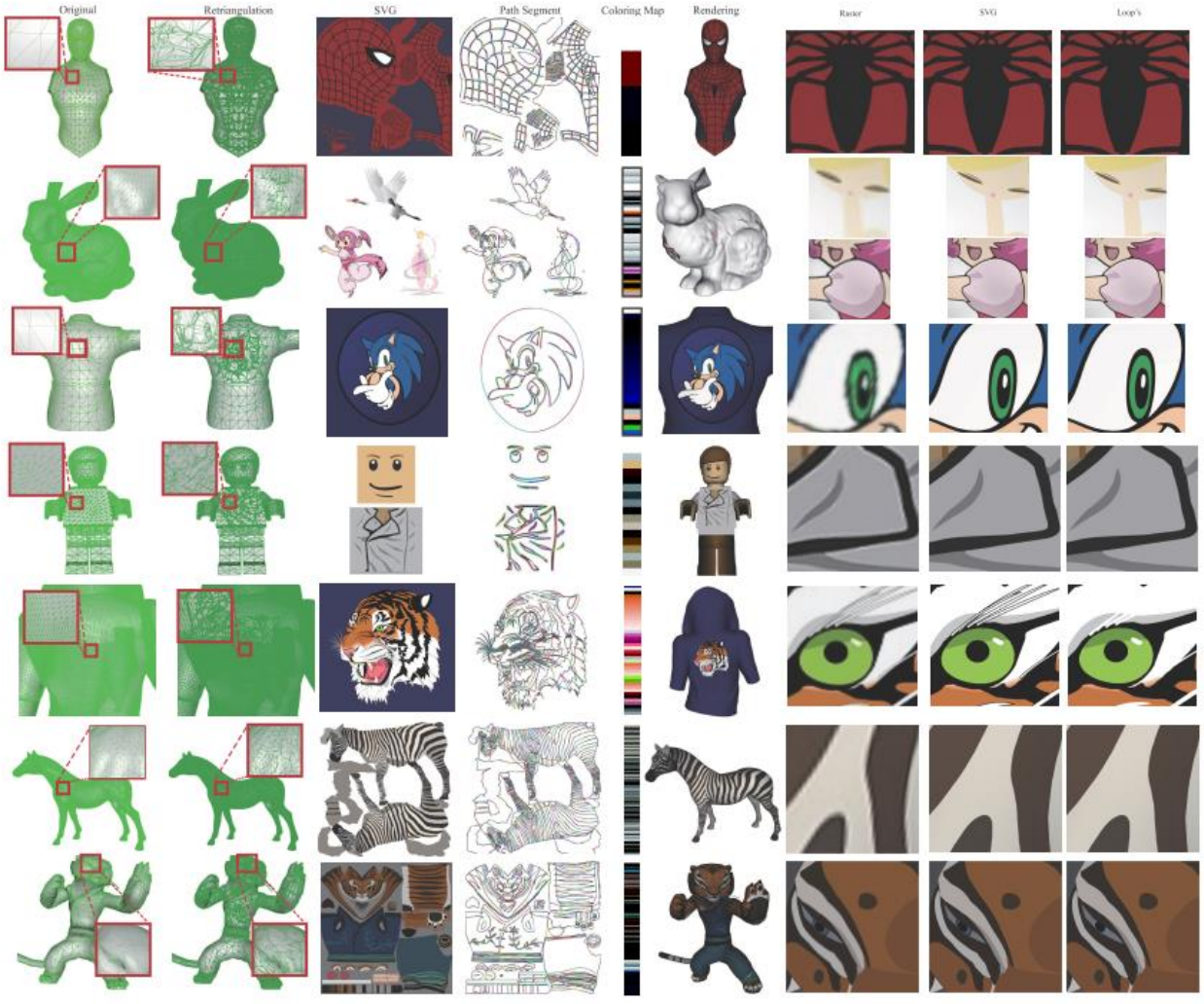

Fig. 10. This demonstrates the results when applying our vector-embedded shading framework on Spiderman, Stanford bunny, Jacket of Sonic, Lego, Tiger Cloth, Zebra, and Tigress. The first and second columns show the original and retriangulated mesh of embedded objects in wire frame. The third, fourth, 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.

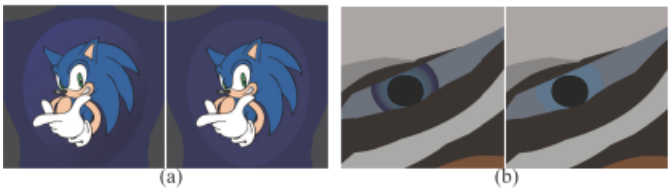

Fig. 12. (a) This shows the rendering result of Jacket with our algorithm in the left and Loop's algorithm in the right. (b) This shows the detail-viewed rendering result of Tigress with our algorithm in the left and Loop's algorithm in the right.

	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

TABLE II

This shows the 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 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].

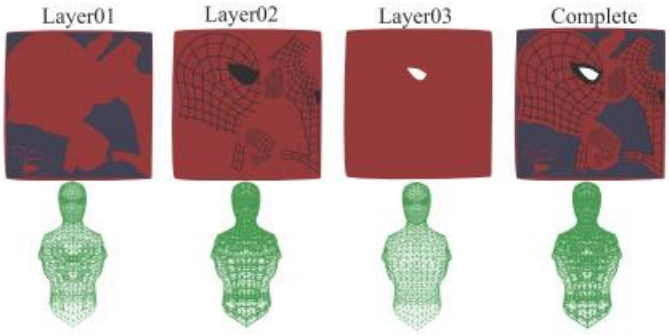

Fig. 11. This shows a simple vector image containing a few layers of coloring operations and Loop's algorithm [3] must retriangulate the original mesh into several different independent meshes which hurt the shading and storage performance.

We have added the snapshots of our algorithm and Nehab's algorithm in Fig. 13. We have also conducted error analysis for Dancer and Lion which can find visually matching SVG from Internet. The detailed discussion in our article is as follows.

Fig. 13. (a) This shows the rendering result of Butterfly with our algorithm in the left and Nehab's algorithm in the right. (b) This shows the rendering result of Lion with our algorithm in the left and Nehab's algorithm in the right.

	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

TABLE V

This shows the timing statistics in FPS when comparing our algorithm with Nehab's algorithm [1] using the data set and code posted in their paper. All data is collected in the resolution of 1920×1080 with turning on the GPU antialiasing function. No. and Zo. are the view of having the entire object fitting inside the screen and zooming view of the details. Err. denotes the quality performance and Me. denotes the memory footprint for the method measured by GPU-Z [30]. 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.

We have also compared our embedded method against the vector texturing algorithm [Nehab et al. 2008] in Table V. We have collected the model and execution code from Nehab's web [Nehab et al. 2008]or effective comparison and run all cases under the same computer with

turning on the GPU antialiasing function. Our algorithm can perform 1.5 to 4 times faster than Nehab's algorithm[Nehab et al. 2008] when using 1.4 to 1.9 more memory. 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 Internet, we cannot 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 can only find the visual 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.

Comment 5:

Fix typos and grammar errors.

1. Section VI: ("statistics collected in this section is measured")
2. Section IV. B "Later, SectionIV-D"

Reply: Thanks for reviewer's reminding. We have carefully proofread the paper and corrected all possible typos and grammar errors that we can find.

Reply to review 3

Comment 1:

Prepare more challenging examples to convince the importance of the proposed method in terms of time performance on strengthening its novelty.

Reply: Thanks for reviewer's reminding. we have followed the recommendation of reviewers to clearly address our main novelty lying in real-time rendering 3D resolution independent animations. Accordingly, we have focused our main technical contributions on real-time resolution independent coloring with GPU-based and CPU-based rendering efficiency, texturing memory usage and transmission efficiency and effectiveness, and easiness to incorporate with commercial softwares such as Blender and Maya. Therefore, we have designed a complex scene consisting of 133 Tigress acting as distinct characters of independent meshes and textures to simulate our ability to load in a large number of distinct characters and real-time shade them with high-quality results while high resolution textures limit the possibility of GPU-based shading and put a burden on the memory usage and rendering speed. We have also used this as our motivation and the complete sentence in our article is as follows:

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 with a texture of 4096x4096, 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 deteriorate rendering efficiency largely. Hence it is a trend to increase perceived quality by other means than just increasing the number of texels which can be described as another type of data compression problems. 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 work aims at developing a resolution independent vector-embedded shading method for 3D animated objects for compactness and infinite resolution with real-time animation efficiency and easiness in plugging into commercial animation softwares.

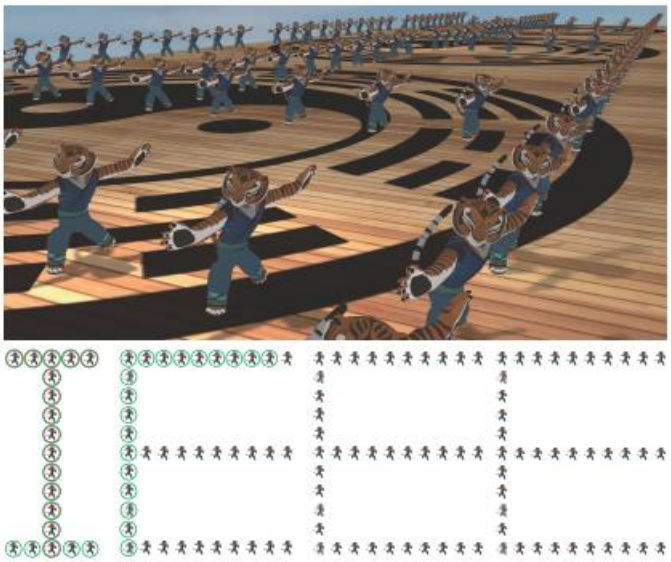

Fig. 1. The top shows the rendering result of a complex scenes consisting of 133 Tigress as distinct characters of independent meshes and textures using our algorithm to illustrate our motivation to reduce the amount of texture memory usage, maintain the rendering quality, and enhance rendering efficiency with GPU-based coloring. The bottom shows the composition of them as "IEEE". While using a computer with a $1GB$ graphics card, we can only load and shade 15 Tigress marked with red rings using raster texturing with a texture resolution of 4096×4096 which is the maximally available resolution of OpenGL. While using a computer with a $16GB$ memory, we can only load and shade 38 Tigress in Blender marked with green rings using raster texturing with a texture resolution of 8192×8192 for matching our rendering quality. To shade 15 Tigress with raster texturing on GPUs takes $2.7e-3$ seconds in NVidia Titan with OpenGL, to shade 133 Tigress in Intel I7-4960K and $64G$ memory with Blender takes 3 seconds, and to shade 133 Tigress in NVidia Titan with our algorithm takes $1.52e-2$ seconds.

Comment 2:

Give the comparison or analysis of the visual quality between the proposed methods and the state-of-the-art methods.

Reply: Thanks for reviewer's reminding. We have added experiments to comparing timing and qualities between the rendering results of raster texturing using different resolutions and vector texturing in **Results**. We have summarized the performance in Table III, IV, and V. The following is the discussion we added into our articles in **Results**.

> *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 on 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 to 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 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 do. 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 sometimes is 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 Tigress acting as distinct characters with independent meshes and textures for demonstrating our real-time high-quality shading ability.

	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)

TABLE III

This shows the 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 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.

	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

TABLE IV

This shows the timing statistics for Spiderman, Spiderman using stroking paths, Stanford bunny, Sonic of Jacket, Lego, Tiger Cloth, Zebra, and Tigress. 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.

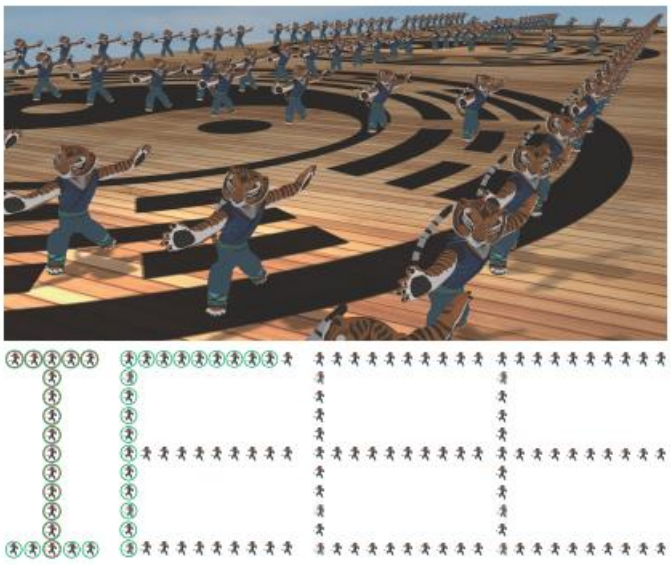

Fig. 1. The top shows the rendering result of a complex scenes consisting of 133 Tigress as distinct characters of independent meshes and textures using our algorithm to illustrate our motivation to reduce the amount of texture memory usage, maintain the rendering quality, and enhance rendering efficiency with GPU-based coloring. The bottom shows the composition of them as "IEEE". While using a computer with a $1GB$ graphics card, we can only load and shade 15 Tigress marked with red rings using raster texturing with a texture resolution of 4096×4096 which is the maximally available resolution of OpenGL. While using a computer with a $16GB$ memory, we can only load and shade 38 Tigress in Blender marked with green rings using raster texturing with a texture resolution of 8192×8192 for matching our rendering quality. To shade 15 Tigress with raster texturing on GPUs takes $2.7e-3$ seconds in NVidia Titan with OpenGL, to shade 133 Tigress in Intel I7-4960K and $64G$ memory with Blender takes 3 seconds, and to shade 133 Tigress in NVidia Titan with our algorithm takes $1.52e-2$ seconds.

	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

TABLE II

This shows the 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 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].

We have shown the detail-viewed results of our algorithm and Loop's algorithm in Fig. 10 and 12. We have also added error analysis of our algorithm and Loop's method. The detailed discussion about the comparison to Loop's method in our article is as follows:

> *We would like to understand our performance when comparing to 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 results 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 applying to different models in Table IV. Because different viewing distances show different amount 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 and thus, 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 concept, we have also created a stroke-based spiderman SVG to shading the spiderman model. As shown in the first and second rows in Table IV, the performance of coloring-based embedment is better than stroking-based embedment. Similarly, we have also analyzed their qualitative performance. When a vector image contain 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.

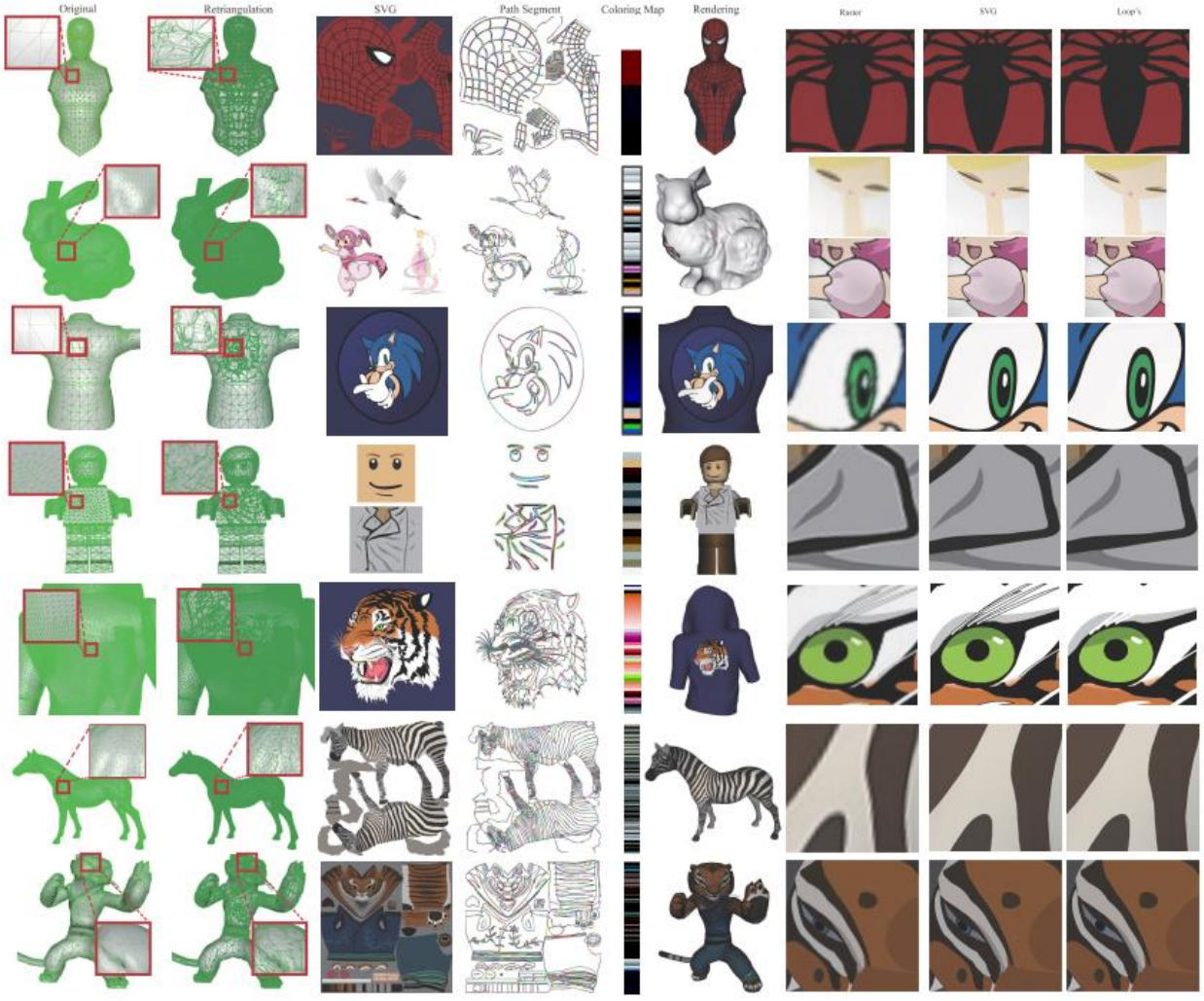

Fig. 10. This demonstrates the results when applying our vector-embedded shading framework on Spiderman, Stanford bunny, Jacket of Sonic, Lego, Tiger Cloth, Zebra, and Tigress. The first and second columns show the original and retriangulated mesh of embedded objects in wire frame. The third, fourth, 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.

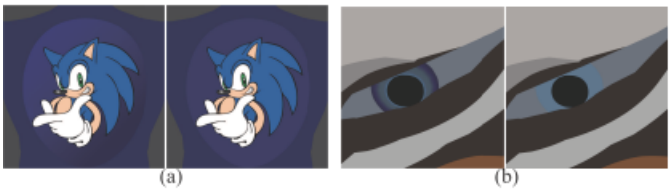

Fig. 12. (a) This shows the rendering result of Jacket with our algorithm in the left and Loop's algorithm in the right. (b) This shows the detail-viewed rendering result of Tigress with our algorithm in the left and Loop's algorithm in the right.

	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

TABLE II

This shows the 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 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].

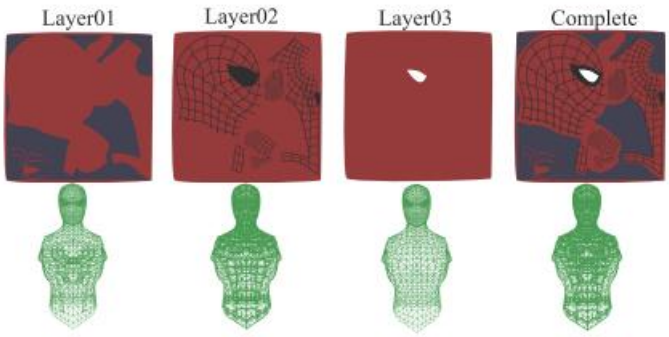

Fig. 11. This shows a simple vector image containing a few layers of coloring operations and Loop's algorithm [3] must retriangulate the original mesh into several different independent meshes which hurt the shading and storage performance.

We have added the snapshots of our algorithm and Nehab's algorithm in Fig. 13. We have also conducted error analysis for Dancer and Lion which can find visually matching SVG from Internet. The detailed comparison to Nehab's algorithm given in our article is as follows:

> *We have also compared our embedded method against the vector texturing algorithm [Nehab et al. 2008] in Table V. We have collected the model and execution code from Nehab's web [Nehab et al. 2008]or effective comparison and run all cases under the same computer with turning on the GPU antialiasing function. Our algorithm can perform 1.5 to 4 times faster than Nehab's algorithm[Nehab et al. 2008] when using 1.4 to 1.9 more memory. 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 Internet, we cannot 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 can only find the visual 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.

(a) (b)

Fig. 13. (a) This shows the rendering result of Butterfly with our algorithm in the left and Nehab's algorithm in the right. (b) This shows the rendering result of Lion with our algorithm in the left and Nehab's algorithm in the right.

	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

TABLE V

This shows the timing statistics in FPS when comparing our algorithm with Nehab's algorithm [1] using the data set and code posted in their paper. All data is collected in the resolution of 1920×1080 with turning on the GPU antialiasing function. No. and Zo. are the view of having the entire object fitting inside the screen and zooming view of the details. Err. denotes the quality performance and Me. denotes the memory footprint for the method measured by GPU-Z [30]. 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.

Comment 3:

Proofread the paper.

Reply: Thanks for reviewer's reminding. We have carefully proofread the paper.