Vector Textures Implementation in Photorealistic Rendering System on GPU

Vadim Sanzharov [1,2], Vladimir Frolov [1,2], Vladimir Galaktionov [2]

[1] Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia
[2] Keldysh Institute of Applied Mathematics RAS, Miusskaya pl., 4, Moscow, 125047, Russia

Abstract

Vector images provide a resolution independent representation that can be efficiently scaled to any resolution. However, there is no ready-to-use solution for supporting vector texturing in GPU based photorealistic rendering systems. In this work we present the experience of implementing support for vector textures in such rendering system using two different approaches – one based on signed distance fields and another based on rasterization. We describe implementation details and analyze the quality and performance of these methods.

Keywords

Photorealistic rendering, vector graphics, GPU rendering, texturing.

1. Introduction

Vector graphics is a method to represent graphical information which relies on storing instructions for drawing instead if actual pixel values. These instructions are based on the mathematics of analytic geometry and define geometric shapes on a Cartesian plane - points, circles, lines, splines, polygons and others. This method of storing graphics information provides certain advantages:

- scalability - vector data can be easily magnified or minified to a desired resolution without loss of information or introducing any artifacts;
- file size is based on complexity rather than resolution or color depth - vector images are often very compact compared to raster images, especially in the case of high resolution images;
- text-based description of geometric shapes can be easily generated by a computer program;
- individual geometric shapes and other objects can be manipulated independently;
- precision - objects placement and description is based on Cartesian coordinates and shape equations rather than discrete raster grid with fixed resolution in case of raster images.

Disadvantages of vector graphics are mainly associated with the complexity of the representation:

- it is hard (if not impossible) to represent detailed real-life objects with complex coloring and form using simple geometric primitives;
- World Wide Web Consortium (W3C) standard for vector graphics – scalable vector graphics (SVG) [1] specifies a lot of different features which leads to difficulty fully supporting all of them;
- finding the color of an individual pixel within a vector graphics object requires traversing all of its primitives while raster images offer efficient random-access at any point which is also implemented in graphics hardware.

From the combination of these advantages and drawbacks follows the application of vector graphics in domains with the requirements for geometric precision and the ability to represent information as a set of simple geometric primitives, - typography, chemical formulas, graphic design, computer-aided design, web design, user interfaces and others.

GraphiCon 2022: 32nd International Conference on Computer Graphics and Vision, September 19-22, 2022,
Ryazan State Radio Engineering University named after V.F. Utkin, Ryazan, Russia
EMAIL: vadim.sanzharov@graphics.cs.msu.ru (V. Sanzharov); vfrolov@graphics.cs.msu.ru (V. Frolov); vlgal@gin.keldysh.ru (V. Galaktionov)
ORCID: 0000-0001-6455-6444 (V. Sanzharov); 0000-0001-8829-9884 (V. Frolov); 000-0001-6460-7539 (V. Galaktionov)

In 3d computer graphics vector images are widely used for user interfaces [2], mainly in videogames and other real-time graphics software. Other applications include rendering text and specific textures such as decals [3].

In this work, we share our experience implementing vector textures support in a photorealistic GPU rendering system using different methods - an approach adapted from real-time graphics applications and naive rasterization. We describe software architecture of developed solution and provide experimental evaluation of implemented methods.

2. Related work

2.1. Vector graphics in photorealistic rendering

In photorealistic rendering vector graphics are not used often. One reason for lack of vector graphics usage in photorealistic rendering is the difficulty creating vector-based representation of naturally occurring real-world textures. However, with recent advances in differentiable rendering [4] allow for automated synthesis of vector graphics description from photographs and other raster images. And also realistic 3d scenes can still include text, ornamentations and symbolic or geometric patterns with sharp boundaries which are well-represented by vector textures (Figure 1).

Figure 1: A 3d scene rendered using proposed solution for vector textures. All textures applied to meshes in this scene are vector images in SVG format

If a vector texture needs to be used in a 3d scene, in a majority of photorealistic renderers user first needs to manually rasterize it in some arbitrary high resolution using 3rd party software and then use it as an ordinary raster image.

However, photorealistic rendering is used in domains such as printing, optical systems modeling, visual effects and animated films which require very high-resolution output rendered images. This requires textures used in the scene to be in even higher resolution to maintain visual quality. Also, many photorealistic rendering systems switched to GPU implementation motivated by hardware accelerated ray-tracing technologies [5]. These two facts limit the use of the naive approach of just rasterizing the vector image, because it incurs a large storage and bandwidth penalty.

2.2. Existing solutions

It is necessary to convert vector graphics data to a raster form to display it on raster devices or to evalute texture color during the rendering of a 3d scene. Therefore, to support vector textures we need to incorporate one of such vector-to-raster conversion methods into the 3d rendering system.

There are existing full-featured libraries for working with vector images, such as Skia [6], Blend2d [7], resvg [8] and others. They support a variety of features, exhibit good performance and offer different backends (in case of Skia). But obviously it is impossible to just call a library function from a GPU renderer kernel.

OpenVG [9] is an API for low-level hardware accelerated vector graphics rendering. It has implementations in mobile hardware (for example, [10]), on top of OpenGL (such as [11]) and software implementations. OpenVG targets 2d graphics applications such as map viewers, e-book readers and others.

OpenGL extension NV_path_rendering [12] provides support for hardware-accelerated of 2d vector graphics represented as paths in 3d graphics applications using OpenGL. The main disadvantage is that this solution is a vendor-specific extension and therefore is not cross-platform. Also, OpenGL API does not provide access for hardware-accelerated ray-tracing which is also a significant downside for photorealistic rendering systems.

In [13] authors propose a method of exact rendering of parametric curves. The method is based on tessellation of curve segments and rendering resulting triangles with conventional graphics pipeline. The main disadvantage of the method is the fact that it produces high quantity of additional geometry. In photorealistic raytracing-based rendering it would be needed to be incorporated into the acceleration structure. And every time textures are changed in the scene, the acceleration structure would need to be rebuilt/updated. Also, if the vector objects are small in size several of the outline curve segments intersect in a single pixel, method produces artifacts.

Approach proposed in [14] uses new representation for vector images and rendering algorithm for it implemented on top of conventional graphics pipeline. The downsides include the requirement for implementing conversion tool for vector textures to entirely new representation and the use of graphics pipeline specific features, such as discard fragment instruction.

Signed distance field (SDF) based approach was first introduced in [15]. Method proposed by authors computes distance fields for parametric curves representing vector graphics objects. Computed SDFs are then used to determine which object should be rendered. In [3] author proposes a simpler approach to SDF computation and combines it with alpha-blending to develop a vector texturing system for a game engine. SDF-based methods struggle with correctly rendering sharp corners making them appear rounded and require high resolution SDFs to handle complex detailed shapes.

An improvement to SDF-based solution was proposed in [16]. Approached proposed by the author, called multi-channel SDF (MSDF) uses several distance fields to represent different edges of a shape which are stored in different color channels of a texture. Disadvantages of SDF-based methods [3, 15, 16] include possible artifacts for complex shapes with sharp corners and the fact that these methods cannot represent several curves intersecting in a single pixel and consider only a case of separate non-intersecting shapes.

Work [17] also uses distance-based approximation for rendering vector graphics. However, in contrast to other works, authors of [17] define vector image as layers of filled and stroked primitives and implement compositing techniques to support correct rendering of different intersecting vector shapes. However, proposed algorithm is very complex and includes ray tracing in texture space and also uses graphics pipeline built-in *ddx* and *ddy* operations to perform mapping from texture space to screen space. Implementing equivalent operations is possible with ray differentials. But that approach produces strictly correct results only for primary rays and can generate errors with recursive textured reflections [18]. It is unclear, how the approximations used by the ray differentials method would interact within the framework of the vector texturing proposed in [17].

In [19, 20] approaches specific to text rendering are proposed. Applying ideas presented in these works to more general vector graphics can be a topic for future research.

2.3. Summary

All of the existing solutions have some drawbacks limiting their use. Some solutions depend on specific hardware [9, 12], others use graphics pipeline features [14, 17] unavailable in the context of compute pipelines often used in photorealistic rendering. Approaches based on SDF computation such as [3, 16] are limited to non-intersecting vector shapes and can produce artefacts in some cases (such as sharp corners). Full-featured 2d rendering implementations such as [6-8] can only be used as preliminary step to 3d rendering to rasterize vector images.

Thus, to implement vector textures support in a photorealistic 3d rendering system we will need to combine different existing solutions to cover common use-cases.

3. Proposed solution

Overall, in our solution we combine MSDF-based method [16] with the composition of different vector shapes similar to [17] albeit in a simpler way. To cover the cases where SDF-based methods produce unsatisfactory results, we also incorporate a rasterization mechanism.

The general scheme of the proposed solution is presented on Figure 2. SVG loader, MSDF computation and SVG rasterizer components can be replaced with different implementations. For example, instead of computing MSDF, one can use a simpler SDF computation algorithm.

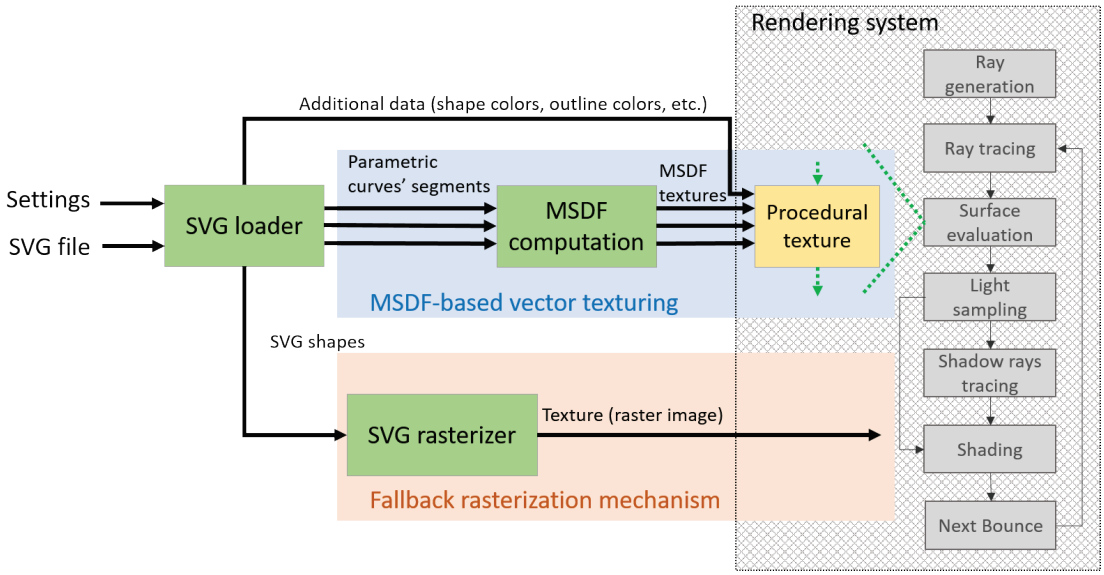

Figure 2: Architecture of the proposed solution. Procedural texture (yellow block) is executed on GPU as a part of rendering process. The code for procedural texture is generated based on the input settings. SVG loader, SVG rasterizer and MSDF computation blocks (shown in green) are executed on CPU as a part of additional software layer which ties together the rendering system with client applications (such as 3d editors). These 3 blocks can be relatively easily replaced with different implementations – a GPU accelerated rasterizer or a loader supporting newer SVG standard

3.1. Implementation details

We choose SVG format [1] as one of the most common for image data. SVG format is very complex and can contain a wide variety of features such as text objects, embedded raster images, animations, embedded code (JavaScript) and others. These objects can also have many different properties and modifiers applied to them such as color and gradient filling, outlines with different stroke types, filters

(such as blur), etc. We limit the supported primitives to paths as any other geometric shape or text can be converted to path representation with conventional vector editors (such as InkScape [21]).

First of all, we load and parse SVG file and extract path objects together with their additional properties (such as color, transformation and others). Each path object is represented as a sequence of parametric curves' segments. Next, each path object is passed to a multi-threaded CPU implementation of the MSDF algorithm [16]. This way for each separate path object (corresponding to separate vector shapes in a source image) we obtain a separate MSDF in the form of an ordinary texture.

As the next step we pass an array of resulting MSDF textures together with arrays of corresponding paths' properties to a procedural texture. In its code we implement the logic for rendering and composition of paths. In the simple case of a single MSDF texture we essentially perform alpha blending of the shape color with the background color (for example, diffuse color of the material using the vector texture) with the distance used as a weight for blending:

$$\begin{gathered} dist = median(msdf) \\ \alpha = smoothstep(T - S, T + S, dist) \\ color = bgColor * (1 - \alpha) + baseColor * \alpha \end{gathered} \quad (1)$$

where:

- *median(a)* - is a function that takes a vector and computes median value of its components;
- *msdf* is a 3-component vector containing multi-channel SDF values;
- *smoothstep(a, b, x)* is a function that performs smooth Hermite interpolation between 0 and 1 when $a < x < b$;
- *T* is constant which sets the value of the distance (from 0 to 1) representing the border of the shape, for objects without a stroked outline, the value of 0.5 should be used as it results in the accurate sizing of the rendered shape;
- *S* is a distance smoothing constant, the larger the value, the smoother the transition at the edge of the shape will be;
- *bgColor* is a background color;
- *baseColor* is a shape fill color.

Function *smoothstep* and *smoothFac* constant serve as an anti-aliasing mechanism to soften up the harsh edges instead of per-pixel screen space derivatives computation available in graphics pipeline.

In SVG format shapes are specified in the order they are drawn - the second path in the file will be drawn on top of the first, etc. Thus to achieve correct composition of different paths we just need to preserve the order they were obtained during parsing through the whole computation process. So, to combine several shapes in the procedural texture code we iterate over the array of MSDF textures. At each iteration we perform alpha blending of the next shape with the previous them using the same mechanism as (1). The algorithm can be then described with following steps:

1. Set *color* to background color
2. for each shape:
    - set *baseColor* to this shape's color
    - sample value of MSDF texture for this shape and set *msdf* to this value
    - perform blending according to (1), but substituting *bgColor* with *color*
3. return *color*

Obviously, iterative sampling of a texture array as large as total shapes number in the source vector image is not very efficient. We provide an option to combine all of the generated MSDF into a single texture. Of course it produces correct results only if the shapes in the source image do not intersect as in the case of text-only textures. A possible further improvement would be to find non-intersecting subsets of all shapes and produce a separate MSDF texture for each subset.

In addition to solid color our solution supports solid stroked outlines. For this purpose, we generate MSDFs specifying the width of the range between the lowest and highest signed distance to be at least equal to the outline width. Then, during the blending stage, if the distance value *dist* falls into this range, we compute the *baseColor* value, used in (1) by blending between shape and outline colors (Figure 3):

$$\omega = \begin{cases} smoothstep(0, S, dist), dist < S \\ smoothstep(1, 1 - S, dist), dist \geq S \end{cases} \quad (2)$$
$$baseColor = shapeColor * (1 - \omega) + outlineColor * \omega$$

where:

- S is a distance smoothing constant, same as (1);
- *dist* is a distance value obtained from a MSDF texture, see (1);
- *shapeColor* is a shape fill color;
- *outlineColor* is a stroke fill color for an outline.

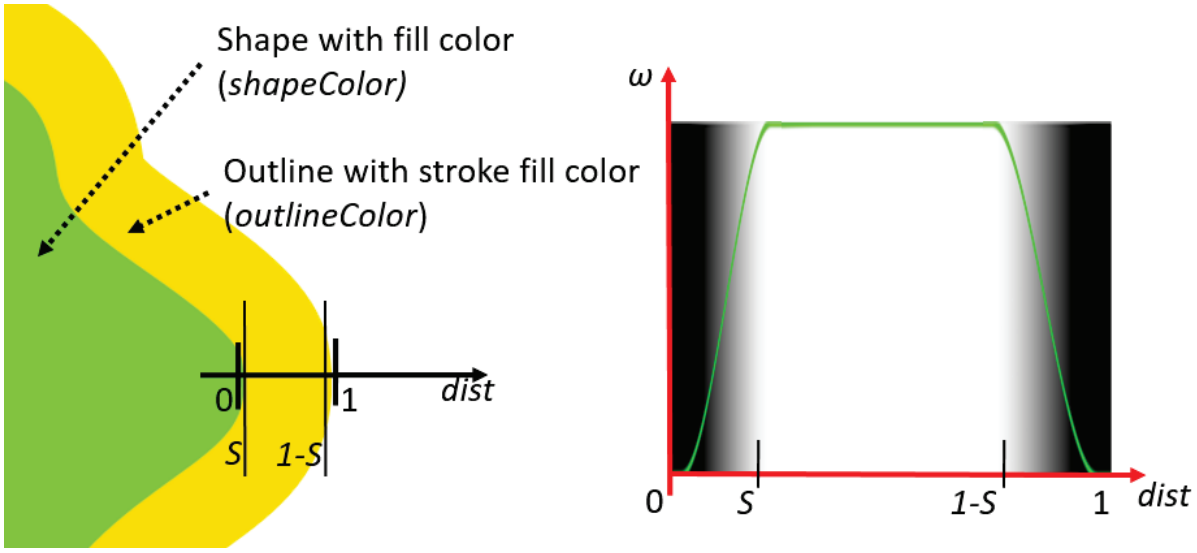

Figure 3: An illustration to outline and shape color blending. Smoothing constant S is a small value, on a scale of 1/256. The general idea is to blend shape color and outline color at *dist* values between 0 and S, and to blend outline color and background color between $1 - S$ and 1

Our photorealistic rendering system interacts with client applications (such as 3d scene editors) through an additional software layer which implements an API for 3d scene creation among other features. To support vector textures, we extend this API with an additional function, which takes a path to the vector texture file and a *struct* exposing a variety of settings as an input. We allow the end user to:

- switch between described MSDF-based algorithm and rasterization of the vector texture;
- change the smoothing constant;
- use texture matrices and different texture application modes (such as clamp and wrap);
- force merging of MSDF textures;
- turn outline rendering on or off;
- override shape and outline color;
- generate a mask from vector texture (for example, to use it for blending different materials).

Based on the specified settings, we generate the code for the procedural texture. Only single channel values are computed if settings specify mask generation. If outline rendering is turned off, all operations related to it are removed from the shader code. If there is only a single MSDF texture (for example, because settings specify merging), no additional information related to blending is loaded on the GPU and a texture array can be replaced with a single texture. This allows us to simplify the shader code as much as possible.

3.2. Image quality

To evaluate our solution, we render a 3d plane with applied vector image textures with different characteristics and compare the results to the reference 2d rendering in InkScape [21].

The rendered and reference images are presented on Figure4.

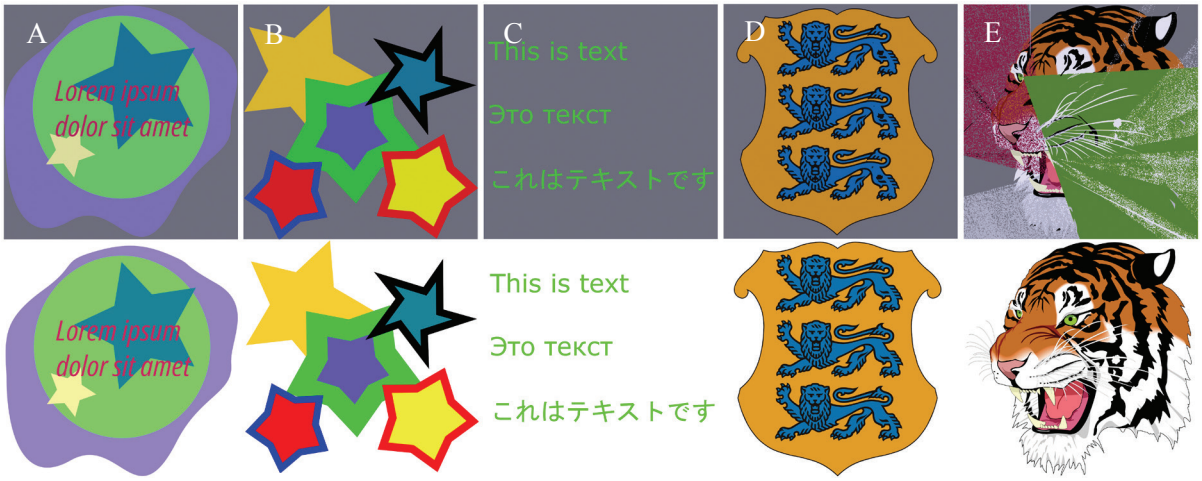

Figure 4: Top row – rendered 3d plane with applied vector image texture using MSDF-based method. Bottom row – original vector images rendered by InkScape

As we can see, MSDF-based method produces artifacts for complex vector drawings caused by extremely sharp features of the shapes. Figure 5 demonstrates examples of MSDFs causing these artifacts, as well as some of the problematic shapes in the source images. Most of such artifacts appear when a path in a source vector image has a sharp and narrow feature at an angle of this path. In cases like these we have to fallback to rasterizing vector images.

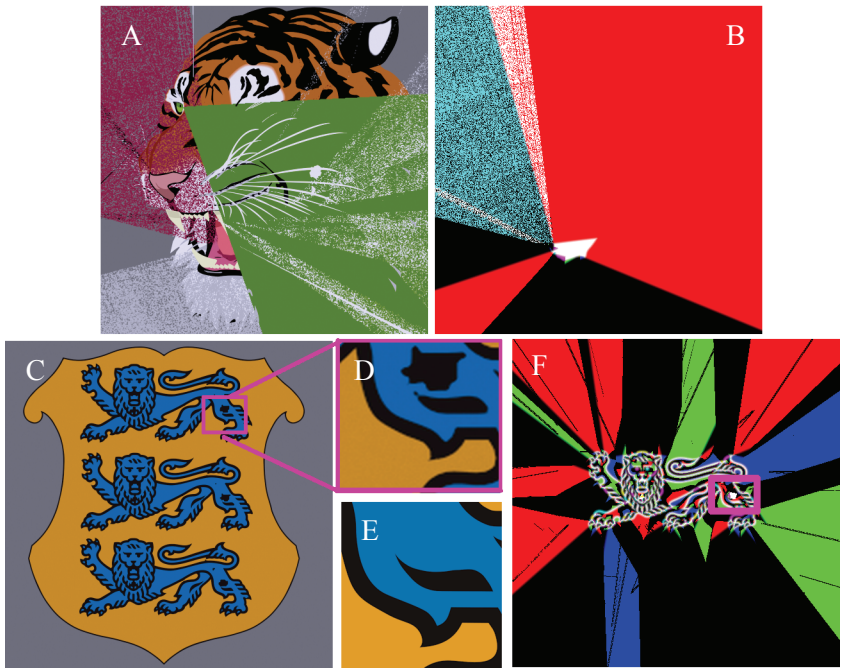

Figure 5: Visual artifacts produced by errors in MSDF computations for sharp features. A – rendered vector texture with significant errors; B – visualization of the MSDF computed for one of the path components of the original vector image. C – rendered vector texture with smaller artifacts; D – enlarged fragment with an artifact present, E - correct image of this fragment (obtained from InkScape), visible artifacts are caused by a stroke with a narrow and sharp end in the middle of the correct image; F - visualization of the MSDF computed for one of the path components of the original vector image with problematic area highlighted

However, with rasterization we need to use much higher resolution if we want to achieve the same image quality. Figures 6-7 show rendered images of the same scene with a texture plane with a distance from plane to camera to achieve texel to pixel ratio of 1 to 32. When using MSDF textures (Figure 6) there is no pixilation and the borders are sharp even at small MSDF resolution (25% of rendering resolution), but there are some artifacts at the corners. With resolution increased to 50% of rendering resolution, the artifacts are almost gone, and at 100% of rendering resolution there is no noticeable artifacts. While with the same texture rasterized (Figure 7) at 100% of rendering resolution, significant pixilation and blurring is present at borders because of filtering when the textured object is close and texel to pixel ration is low. To achieve relatively sharp borders, the rasterization resolution should be at least 400% of the rendering resolution in the provided example.

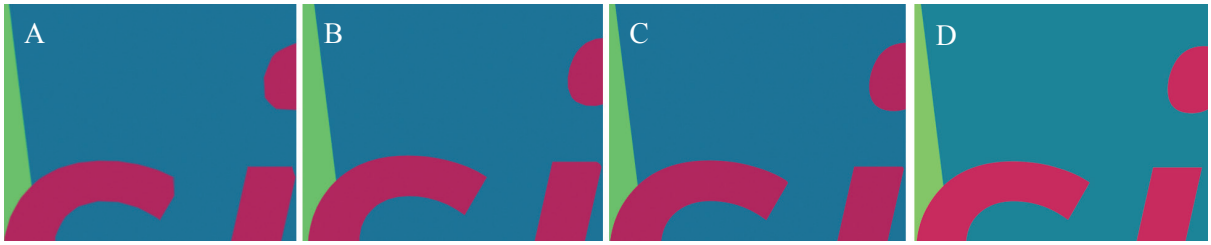

Figure 6: A fragment of a rendered 3d plane with applied vector texture using MSDF-based algorithm. Camera was placed near the 3d plane to achieve 1 to 32 texel to pixel ratio. The total rendering resolution was 1024 x 1024, the MSDF texture resolution: A – 256 x 256, B – 512 x 512, C – 1024 x 1024, D – reference from InkScape

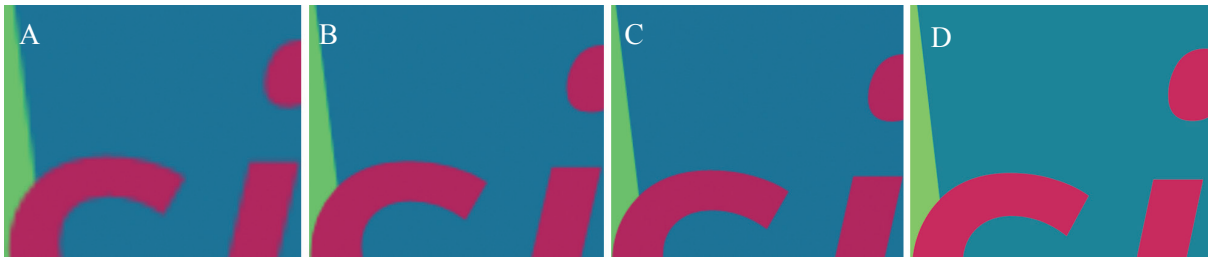

Figure 7: A fragment of a rendered 3d plane with applied rasterized vector. Camera was placed near the 3d plane to achieve 1 to 32 texel to pixel ratio. The total rendering resolution was 1024 x 1024, the rasterized texture resolution: A – 1024 x 1024, B – 2048 x 2048, C – 4096 x 4096, D – reference from InkScape

However, if the source vector texture contains intersecting shapes and we need to store a separate MSDF texture for each shape, memory used by this method will depend on the number of shapes. In the provided example the source vector texture (Figure 4A) contains 26 shapes (here we count each letter as a separate shape). So, 26 textures in 512x512 resolution without compression take up 26 Mb of memory. And an uncompressed 4096x4096 rasterized texture needs 64 Mb, which is 2.46 times more. If MSDF textures can be combined into a single texture (for example, in the case of text-only texture) the amount of saved memory is much more significant, since we need to store only a single MSDF texture.

Another factor, which should be taken into account is the rendering resolution. If we increase the rendering resolution in the same example scene to 4096 x 4096, then rasterized texture resolution will be needed to be 16384 x 16384 to maintain quality, this texture without compression will take up 1 Gb of memory. But the resolution needed by a MSDF texture does not depend on the rendering resolution and depends on the detail present in it. So for the same scene, the memory consumption ratio between uncompressed rasterized vector texture and MSDF textures is 38.46 times (Figure 8).

If the rendering system supports compression, the benefits of using MSDF-based approach become much more significant because MSDF textures have very limited amount of colors and can potentially achieve high compression ratios.

Finally, it should be noted, that with proposed solution architecture (Figure 2) which incorporates the rasterization module, it is possible to treat vector textures as a case for “precomputed procedural textures” and use mechanism described in [22] to automatically determine rasterization resolution needed to achieve 1 to 1 texel to pixel ratio.

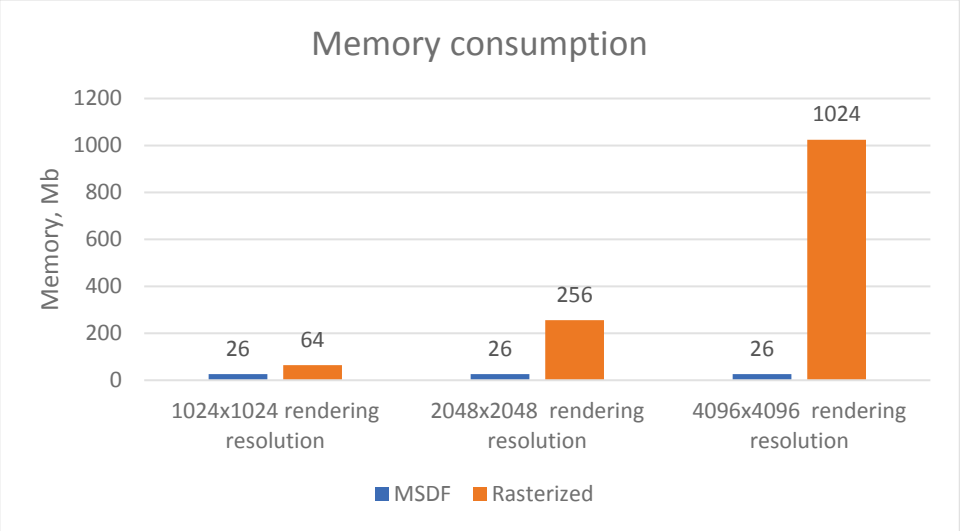

Figure 8: Memory taken by uncompressed MSDF textures for different rendering resolutions of the same scene (a plane with applied texture from Figure 4A facing camera) and by rasterized texture. To maintain the same visual quality, rasterized texture needs to increase the resolution when the rendering resolution is increased (in this comparison resolution of a rasterized texture is equal to rendering resolution), while quality of an image produced with MSDF textures is independent of the rendering resolution

3.3. Performance

The proposed method which uses procedural texture implementing MSDF-based approach has an impact on rendering performance (table 1).

Table 1
Texture

Texture	Number of shapes (paths) in the texture	MSDF, rendering performance, millions of samples per second	MSDF combined (if possible), rendering performance, millions of samples per second	Rasterized, rendering performance, millions of samples per second	MSDF, procedural texture kernel execution time, milliseconds
Figures and text (Figure4, A)	26	78	-	80	1.38
Star pattern with outline (Figure4, B)	5	80	80	80	1.18
Text (Figure.4, C)	27	76	80	80	1.49
Coat of arms (Figure4, D)	11	80	-	80	0.96
Tiger (Figure 4, E)	240	50	-	80	5.92

If separate MSDF textures are generated for every path object in the vector texture, performance degenerates dependent on the number of generated textures. This is caused by an increase of texture lookups and arithmetic operations needed to be performed on the fetched texel values. However, significant drops in the performance are observed only for vector textures with a lot of path objects, such as the Tiger texture (Figure 4E). And if all MSDF textures can be combined into one (like in the case of Figure 4 B and C), rendering performance is unaffected - essentially we sample a single texture and perform few simple operations with samples values.

However, as shown in the previous section on quality, complex vector drawings with a lot of complex shapes are likely to cause artifacts with MSDF-based approach and should be rasterized anyway.

Impact of using procedural texture implementing MSDF-based approach on rendering performance, all measurements were done on Nvidia RTX2070 Super. Third to fourth columns show total rendering performance (including acceleration structure traversal, shading, etc.) and the last column shows isolated performance of the procedural texture evaluation.

4. Conclusion

We implemented support for vector images in limited subset of SVG format for use in photorealistic 3d rendering. We modified the base MSDF method [16] to support overlapping vector shapes and stroked outlines and integrated it together with a vector image rasterization solution into our middle-layer API for 3d scene creation to hide the complexity from the user. This way we achieve a flexible solution, capable of providing memory savings with no impact on rendering performance for specific use cases such as text and simple geometric patterns. And for complex vector drawing our solution provides fallback mechanism implementing rasterization. Incorporating a vector image rasterizer in our solution allows using additional mechanisms to automatically determine rasterization resolution to achieve 1 to 1 texel to pixel ratio.

5. References

[1] Scalable Vector Graphics (SVG) Full 1.2 Specification, 2005. URL: https://www.w3.org/TR/SVG12.

[2] Noesis GUI. User Interface middleware for videogames and realtime applications, 2022. URL: https://www.noesisengine.com.

[3] C. Green, Improved alpha-tested magnification for vector textures and special effects, in: ACM SIGGRAPH 2007 courses, 2007, pp. 9-18.

[4] T.M. Li, M. Lukáč, M. Gharbi, J. Ragan-Kelley Differentiable vector graphics rasterization for editing and learning. ACM Transactions on Graphics (TOG), 2020, 39(6), 1-15. doi:10.1145/3414685.3417871.

[5] V.V. Sanzharov, V.A. Frolov, V.A. Galaktionov (2020). Survey of Nvidia RTX technology. Programming and Computer Software, 46(4), 2020, 297-304. doi:10.1134/S0361768820030068.

[6] Skia: The 2D graphics library, 2022. URL: https://skia.org/

[7] Blend2d. 2D Vector graphics engine, 2022. URL: https://blend2d.com/

[8] resvg, SVG rendering library, 2022. URL: https://github.com/RazrFalcon/resvg

[9] OpenVG, the standard for vector graphics acceleration, 2022. URL: https://www.khronos.org/openvg/

[10] Mali-450 technical specifications, 2022. URL: https://developer.arm.com/Processors/Mali-450

[11] ShivaVG, an open-source LGPL ANSI C implementation of the Khronos Group OpenVG specification, 2010. URL: https://github.com/ileben/ShivaVG.

[12] M.J. Kilgard, J. Bolz GPU-accelerated path rendering. ACM Transactions on Graphics (TOG), 2012, 31(6), 1-10. doi:10.1145/2366145.2366191.

[13] C. Loop, J. Blinn Resolution independent curve rendering using programmable graphics hardware. In ACM SIGGRAPH 2005 Papers (pp. 1000-1009). doi:10.1145/1186822.1073303.

[14] N. Ray, X. Cavin, B. Lévy Vector texture maps on the GPU. Inst. ALICE (Algorithms, Comput., Geometry Image Dept. INRIA Nancy Grand-Est/Loria), 2005, Tech. Rep. ALICE-TR-05-003.
[15] Z. Qin, M.D. McCool, C.S. Kaplan Real-time texture-mapped vector glyphs. In Proceedings of the 2006 symposium on Interactive 3D graphics and games (pp. 125-132). doi:10.1145/1111411.1111433/
[16] V. Chlumsky Shape decomposition for multi-channel distance fields, Master's thesis, Czech Technical University, 2015. URL: https://dspace.cvut.cz/bitstream/handle/10467/62770/F8-DP-2015-Chlumsky-Viktor-thesis.pdf.
[17] D. Nehab, H. Hoppe Random-access rendering of general vector graphics. ACM Transactions on Graphics (TOG), 2008, 27(5), 1-10. doi:10.1145/1409060.1409088.
[18] T. Akenine-Möller, J. Nilsson, M. Andersson, C. Barré-Brisebois, R. Toth, T. Karras Texture level of detail strategies for real-time ray tracing. In Ray Tracing Gems (pp. 321-345). 2019, Apress, Berkeley, CA. doi:10.1007/978-1-4842-4427-2_20.
[19] E. Lengyel Gpu-centered font rendering directly from glyph outlines. Journal of Computer Graphics Techniques, 2017, Vol, 6(2).
[20] O. Alvin Rendering Resolution Independent Fonts in Games and 3D-Applications. Master's thesis, Lund University, 2020.
[21] InkScape, free and open-source vector graphics editor, 2022. URL: https://inkscape.org/
[22] V.V. Sanzharov, V.A. Frolov Level of detail for precomputed procedural textures. Programming and Computer Software, 2019, 45(4), 187-195. doi:10.1134/S0361768819040078.