Rendering Vector Art on the GPU

Team 11

AMANDEEP KAUR and DIPANSHU AGGARWAL

1 INTRODUCTION

The representation of computer graphics in vector form has gained popularity over the last few decades. In vector graphics, image outlines are defined using primitive paths. These paths are made up of straight lines or curves of varying degrees controlled by a set of vertices. This is in contrast to storing RGB values for each pixel in a 2D matrix as in raster graphics. Vector graphics is known to be a better alternative because 1) storing guidelines for paths take much less space and 2) using mathematical representation of the paths in an image and calculating pixel color values at the time of viewing makes the image resolution independent and artifact-free. However, rendering vector graphics involve solving the point-in-shape problem: determining the points that belong in the interior of the provided shape. Performing the above task on a screen with high resolution demands efficient algorithms along with GPU parallelization.

2 LITERATURE REVIEW

The various techniques that have been used in the available literature can be briefly divided into three main categories: 1) Scanline Filling Methods 2) Stencil, then Cover and 3) use of alternate representations of vector graphics data. [1]

2.1 Scanline Filling

In Scanline filling method, one breaks the interior of given vector shapes into horizontal spans. Various scanline approaches can be found in popular rendering packages like Skia [3] and Cairo [7]. This approach, however, is mostly used with CPU-based algorithms and is considered GPU hostile. [5] was the first paper to use a scanline approach on the GPU.

2.2 Stencil, and Cover

Stencil, and cover follows a two passes per-path method. It generates a mask/stencil on a path, determining and marking which pixel is inside a patch in the first step, and performs the shading of marked pixels is the second step. An efficient approach of implicitization of parametric curves to determine the side of any sample w.r.t. a primitive curve was introduced by [6]. Though efficient, this approach required costly tessellation on CPU which was hard to parallelize. Newer approaches support multiple curves per primitive and use hierarchical rasterization to gain efficiency, but still lag behind as the multiple passes may alter samples not used in the resultant stencil. [4][5]

2.3 Alternative Representations

Various alternate representations that have been used to store vector graphic data for parallel rendering are include shortcut trees [2], feature curves [8], Cpatches[1],etc. Although these approaches are able to achieve parallel execution on a GPU, they are very compute-intensive as they involve creating auxiliary data structures or new representations from primitive vector data before rendering.

Authors' address: Amandeep Kaur, amandeep18014@iiitd.ac.in; Dipanshu Aggarwal, dipanshu18139@iiitd.ac.in.

2.4 Our Approach

Our approach involves the re-implementation of [6] with added advantage of tessellation done on GPU[9]. For simplicity, we will be working with single layered vector input files with no overlapping objects. A brief description of the steps involved in this vector rendering approach are as follows:

- Reading input from a vector graphic file
- Converting all curves to quadratic/cubic bezier curves
- Performing constrained delaunay triangulation on curves to get triangles
- Marking triangles as lying inside or outside the object and rendering inner triangles
- Implicitization of parametric curve for outer triangles
- Rendering border triangle with implicit functions using custom shader

We will implement these steps on CPU as well as GPU and compare the performance.

3 MILESTONES

S. No.	**Milestone**	**Member**
	Mid evaluation	
1	Converting vector input to bezier curves ✓	Dipanshu, Amandeep
2	Implementing constrained Delaunay Algorithm on CPU ✓	Amandeep
	Final evaluation	
3	Implementing constrained Delaunay Algorithm on GPU	Dipanshu
4	Rendering inner triangles with OpenGL	Amandeep
5	Implicitization of Parametric curves on CPU and GPU	Dipanshu
6	Implicitization of Parametric curves on CPU and GPU	Amandeep
7	Rendering final output	Dipanshu
8	Performance Analysis	Dipanshu, Amandeep

4 APPROACH

We start by reading the given SVG file using a function from the library called NanoSVG. The function returns a cubic bezier curve(2 end points and 2 control points) corresponding to each stroke in the SVG. The following task of triangulation is divided into two parts. 1) Rendering inner triangles on the object using custom object contour and 2) Rendering parts of outer quadrilaterals using curve implicitization. The former is done after analysing each bezier curve and finding the control points that should be used to form constrained edges. Next, we pass all edge points and chosen control points to a Delaunay Triangulation function provided in the library poly2tri, which returns coordinates to all triangles. The above library uses sweep-line algorithm, the points are sorted and an advancing front moves from -inf to +inf, as the front touches a new point, the point is projected on the front and a new triangle is formed. If the new triangle fulfils the empty circle property(given by Delaunay) with its neighbouring triangles, it is added to the front, else the common edge is swapped to achieve another edge, the property is checked recursively, this is called legalization.

5 RESULTS

We were able to extract all points, pick out points that formed contour for inner triangles and triangulate them.

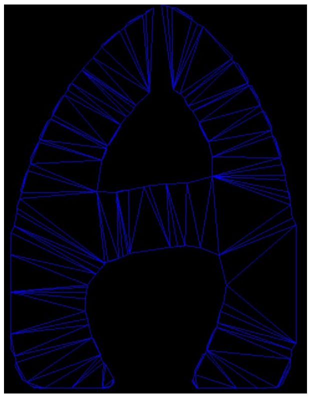

Fig. 1. An image of how the triangulation looks for alphabet A

REFERENCES

[1] Mark Dokter, Jozef Hladký, Mathias Parger, Dieter Schmalstieg, Hans-Peter Seidel, and Markus Steinberger. 2019. Hierarchical Rasterization of Curved Primitives for GPU Vector Graphics Rendering. *Computer Graphics Forum* 38, 2 (2019), 93–103. https://doi.org/10.1111/cgf.13622

[2] Francisco Ganacim, Rodolfo S. Lima, Luiz Henrique de Figueiredo, and Diego Nehab. 2014. Massively-Parallel Vector Graphics. *ACM Trans. Graph.* 33, 6, Article 229 (Nov. 2014), 14 pages. https://doi.org/10.1145/2661229.2661274

[3] Google. [n.d.]. Skia Graphics Library. https://skia.org/

[4] Mark J. Kilgard and Jeff Bolz. 2012. GPU-Accelerated Path Rendering. 31, 6, Article 172 (Nov. 2012), 10 pages. https://doi.org/10.1145/2366145.2366191

[5] Rui Li, Qiming Hou, and Kun Zhou. 2016. Efficient GPU Path Rendering Using Scanline Rasterization. *ACM Trans. Graph.* 35, 6, Article 228 (Nov. 2016), 12 pages. https://doi.org/10.1145/2980179.2982434

[6] Charles Loop and Jim Blinn. 2005. Resolution Independent Curve Rendering Using Programmable Graphics Hardware. *ACM Trans. Graph.* 24, 3 (July 2005), 1000–1009. https://doi.org/10.1145/1073204.1073303

[7] ESFAHBOD B. PACKARD K., WORTH C. [n.d.]. cairographics.org. https://www.cairographics.org/

[8] Evgueni Parilov and Denis Zorin. 2008. Real-Time Rendering of Textures with Feature Curves. 27, 1, Article 3 (March 2008), 15 pages. https://doi.org/10.1145/1330511.1330514

[9] Meng Qi, Thanh-Tung Cao, and Tiow-Seng Tan. 2012. Computing 2D Constrained Delaunay Triangulation Using the GPU. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (Costa Mesa, California) *(I3D '12)*. Association for Computing Machinery, New York, NY, USA, 39–46. https://doi.org/10.1145/2159616.2159623