UNDERSTANDING HARDWARE-ACCELERATED 2D VECTOR GRAPHICS

A Thesis
by
SPENCER C. IMBLEAU

Submitted to the Graduate School
Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

May 2022
Department of Computer Science

UNDERSTANDING HARDWARE-ACCELERATED 2D VECTOR GRAPHICS

A Thesis
by
SPENCER C. IMBLEAU
May 2022

APPROVED BY:

R. Mitchell Parry, Ph.D
Chairperson, Thesis Committee

Rahman Tashakkori, Ph.D
Member, Thesis Committee

James B. Fenwick, Jr., Ph.D
Member, Thesis Committee

Rahman Tashakkori, Ph.D
Chairperson, Department of Computer Science

Marie Hoepfl, Ed.D.
Interim Dean, Cratis D. Williams School of Graduate Studies

Abstract

UNDERSTANDING HARDWARE-ACCELERATED
2D VECTOR GRAPHICS

Spencer C. Imbleau
B.S., Western Carolina University
M.S., Appalachian State University

Chairperson: R. Mitchell Parry, Ph.D

With the rising support of compute kernels and low-level GPU architecture access over the past few years, friction with general-purpose GPU computing is fading. With new accessibility, new analytics methods for hardware-accelerated vector rasterization are being tried with new leverage. There are compelling reasons to optimize performance given the resolution-independent imaging model and inherent benefits. However, there is a noticeable lack of comparison between algorithms, techniques, and libraries which gauge the modern rendering capability. Analyzing the performance of vector graphics on the GPU is challenging, primarily when various technologies may compete for differing scarce computer resources. This thesis examines the contention found with modern vector graphic rendering and expands on analysis techniques used to deobfuscate efficacy by providing an analytic benchmarking framework for hardware-accelerated renderers.

Acknowledgements

I would like to thank Dr. Raph Levien and Dr. Mitch Parry, whose expertise have sharpened my thinking and brought my work to a higher level. Their involvement has helped guide this research and contribute to a larger community.

I would also like to thank Appalachian State University, for sponsoring my research with architecture and funding. Without their help, many of my results would not be possible.

Preface

This research aims to survey modern 2D vector graphic rendering contention and provide an analysis thereof. The subject matter is themed around modern rendering techniques and detailing the architecture and design of a benchmarking framework, *vgpu-bench*, engineered to provide the tooling for CPU and GPU-centric benchmarking. Parts of this work serve to provide code snippets, data artifacts, and theories supported by *vgpu-bench*. This research assumes an intermediate understanding of computer graphics and graduate knowledge of computer science.

Contents

List of Tables

List of Figures

List of Equations

List of Code Examples

Chapter 1

Introduction

1.1 Problem Statement

The plumbing of video-card architecture has been historically optimized for triangle arithmetic and data flow. This specificity has led to rigid render pipelines and difficulty with generalized parallel computation. Hence, this ingrained rigidity is why vector graphics are considered GPU-hostile. Given that vector images are formatted implicitly as "equations" rather than discrete pixel rows of color data, a different approach is necessary for GPU rendering; flexibility is required to parallelize the rasterization processing of vector graphics on the GPU. Moreover, processing implicit data adds a level of indirection, prompting a substantial overhead for rendering not similarly experienced in raster graphics.

With the rise of support for compute kernels and low-level GPU architecture access over the past few years, friction with general-purpose GPU computing is fading. With this new access to low-level hardware features comes experimentation. The field of hardware-accelerated vector graphics seems optimistic, with new attempts to leverage these features. However, there is a noticeable lack of comparison between techniques and libraries which gauge the modern rendering capability. This lack of comparison is partly due to the highly complex strategy required to precisely sample GPU metrics. Therefore, relative comparisons, time metrics, and *Big-O* is typically provided as a decent proxy.

Analyzing the performance of vector graphics on the GPU is *hard*. Various renderers and approaches are tuned for fonts, mobile power consumption, or other scarce computer resources. Given new technologies attempting to solve these issues, it is an appropriate step to respond with an analysis of the model and how to measure it. We can provide optics, encourage further research, and de-obfuscate the field by providing an analytic framework to measure hardware-accelerated vector graphics.

1.2 Research Outline

This research thesis will begin with required background information in Chapter 2 and a literature review of prior techniques relative to vector rendering in Chapter 3, provided for

comprehension. Afterward, we consolidate considerations of vector graphic analytics with synthesized theories. These theories accentuate a methodology and design section for an analytic framework we build to evaluate vector graphic rendering efficacy. We begin by introducing functional and non-function requirements for our analytic framework, which constitutes the basis for the methodology and architecture of our framework. We then defend our design choices, supplemented by diagrams and thorough reasoning. Finally, we provide results to prove our product through trial in a test case.

Ultimately, our product is theory and an analytic framework, *vgpu-bench*, which orchestrates sequential execution of small, independent test containers, augmented with atomically synchronized monitors to collect measurements in partial satisfaction of our requirements. Furthermore, our product is an extensible, open-source benchmarking tool, befitting the rapidly changing field of hardware-accelerated vector graphics.

Chapter 2

Background

Vector graphics are a unique image model, ideal for simple graphics that can be resolution independent, lightweight, and dynamic. This section will overview history, contention, and how to benefit from the image model.

2.1 Image Models

Contrary to vector graphics, *raster* images are established and used eagerly among computers today; raster graphics are likely what comes to mind when we think of images. Raster images are rendered by reading pixels or data fragments containing color and tonal information, typically stored in rows. These images are stored explicitly, inherently requiring no additional arithmetic to copy and display to a screen buffer during *rasterization*. Explicit storage makes the memory model of raster graphics exemplary for performant, elementary graphics. The first implementation of raster graphics was published in March of 1971 by Michael A. Noll in his publication *Scanned-Display Computer Graphics* [1]. The philosophy remains simple: store images in memory as discrete pixels, pre-computed such that rendering requires no additional computational overhead.

On the contrary, vector images are formatted and stored as geometric primitives in an implicit form. Generally speaking, vector images store points, lines, and equations rather than pixels. During the rasterization stage of vector graphic rendering, *varyings*, such as scale, are applied to the data to produce a discrete image. The first successful implementation of this concept was noticeably earlier than raster, presented by Turing award laureate Ivan Sutherland [2] in his seminal work Sketchpad [3] (1963).

2.2 Contention in Vector Graphics

Analytic vector graphic rendering brings hardship. This section will attempt to summarize friction encountered with vector graphics.

2.2.1 Image Encoding

Vector image encoding has many well-known implementations, such as *pdf* or *ai* by *Adobe Inc.* Open source standards for vector image encoding also exist, namely The World Wide Web Consortium's (W3C) *svg*, or *Scalable Vector Graphics*, established as a standard for the web.

W3C designed *SVG* particularly to target static image content at first. Unfortunately, to this day, it is a highly complex specification that is slow to establish rendering support. Most modern web browsers have support for rendering *svg* files, although a full implementation is not guaranteed and is comparatively rare to find.

Yevhenii Reizner (*RazrFalcon*[1]), created a test suite poised to test *svg* compliance and edge cases while developing their own *svg* renderer named *resvg*. Yevhenii's tests encompass common web browsers and renderers, which quantify the lack of the spec's implementation [4]. As of March 2022, the results of their test suite cover more than 1400 edge cases and are shown in Figure 2.1 below.

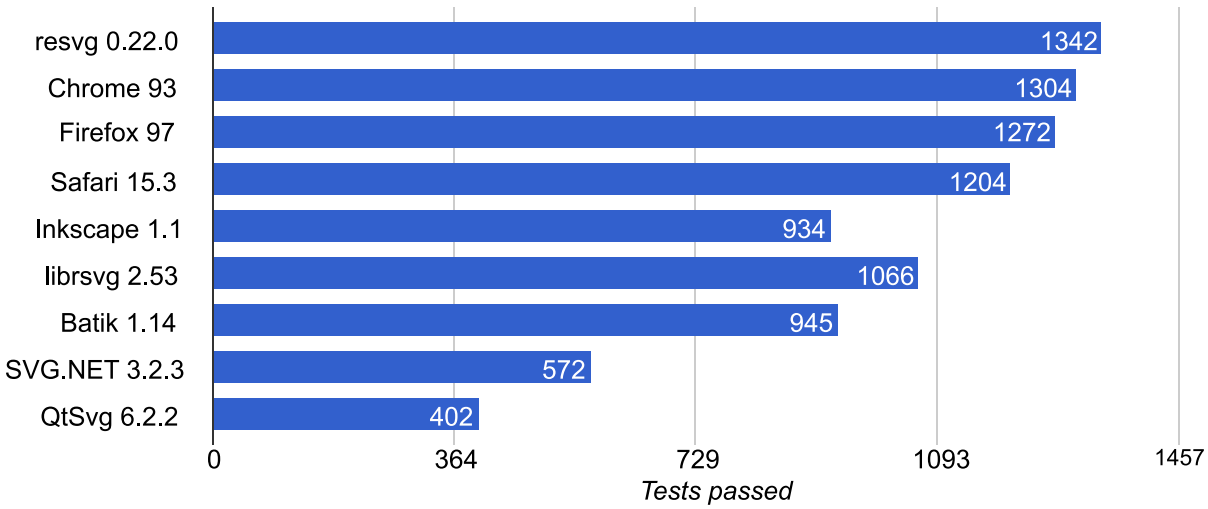


Figure 2.1: *SVG* specification adherence test results compiled in resvg.[2]

2.2.2 Locality

Vector images suffer from a locality issue. Unlike raster graphics with color and fill information encoded explicitly, determining the fill of a pixel fragment in a vector image model requires knowledge of the entire image. Every pixel requires a calculated *winding number*, or how many turns a curve takes around a point (pixel). After computing winding numbers, the image requires a presentation attribute called a *fill-rule* which

[1] see: https://github.com/RazrFalcon

[2] attribution: By Yevhenii Reizner, modifications by Spencer C. Imbleau, MPLv2.0

determines if a winding number is interpreted as *inside* or *outside* of a shape. In simple terms, a renderer requires information about all paths to determine any given pixel's fill.

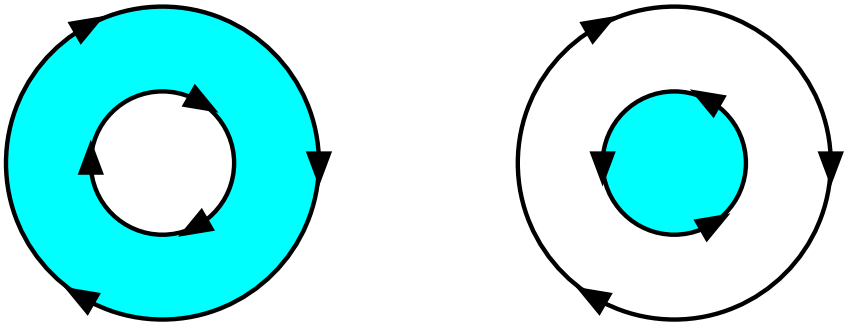


Figure 2.2: Two differing fill-rules.[3]

The locality issue negates certain advantages in the classic GPU parallel rendering structure. Moreover, this issue implies that rendering vector graphics might be a sequentially solved problem, difficult to parallelize.

2.2.3 The Bézier Curve

Efficient parallelism of path tracing is difficult. While the concept of the universal curve was engineered with relative simplicity, handling a system of curves non-atomically is complex.

The curve concept is intuitive, being that a curve is simply a linear interpolation between control points. The basics begin with De Casteljau's algorithm [5], given in Equation 2.1. De Casteljau algorithm defines the shape of a Bézier curve B to be within $t \in [0, 1]$ of an arbitrary degree n, where n is the number of control points $\beta_0, \ldots, \beta_n$.

$$B(t) = \sum_{i=0}^{n} \beta_i b_{i,n}(t) \tag{2.1}$$

where b is a Bernstein basis polynomial.

$$b_{i,n}(t) = \binom{n}{i}(1-t)^{n-i}t^i.$$

Tracing a curve's pixels is as simple as solving this equation in small increments, or *steps*, and connecting the dots. Increments should be small enough to minimize visual error during rasterization for a given display. Interpolating a quadratic ($N = 3$) curve from 10 segments is shown below in Figure 2.3.

[3]attribution: By Spencer C. Imbleau, MIT/Apache 2.0

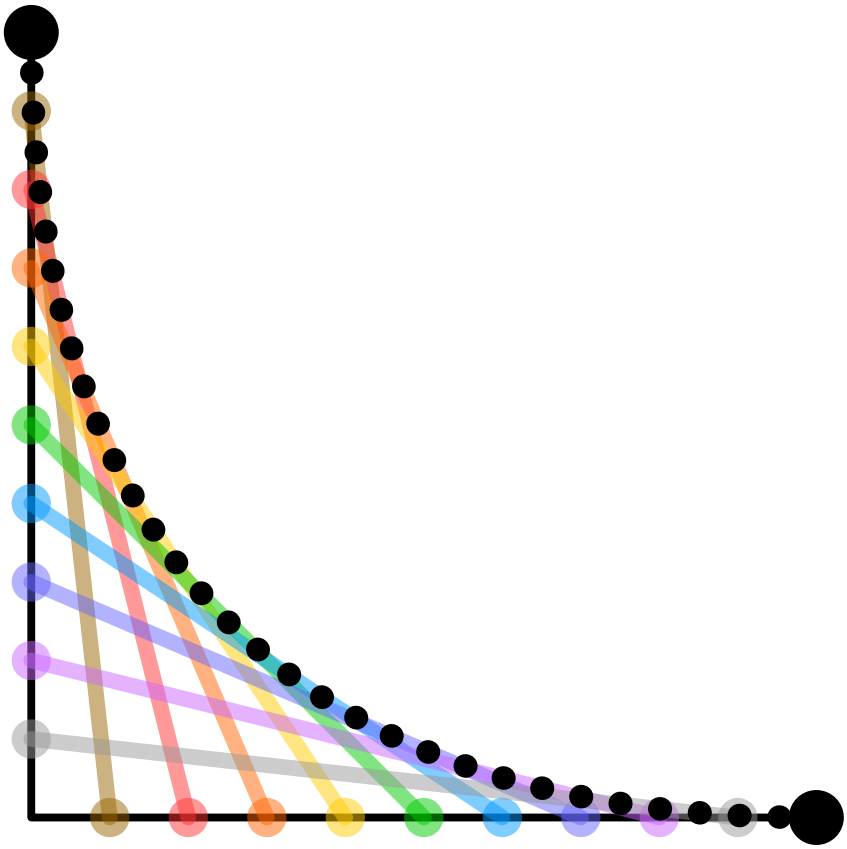


Figure 2.3: Visualizing linear interpolation of a quadratic Bézier curve.[4]

The concept of Bézier curves has been static for decades, and the lack of GPU features and flexibility in the pipeline have barred most experimentation. Complexity further increases with image processing, such as stroking, compositing, blending, or styling shapes, which are conventional necessities to the image model.

2.3 Benefits of Vector Graphics

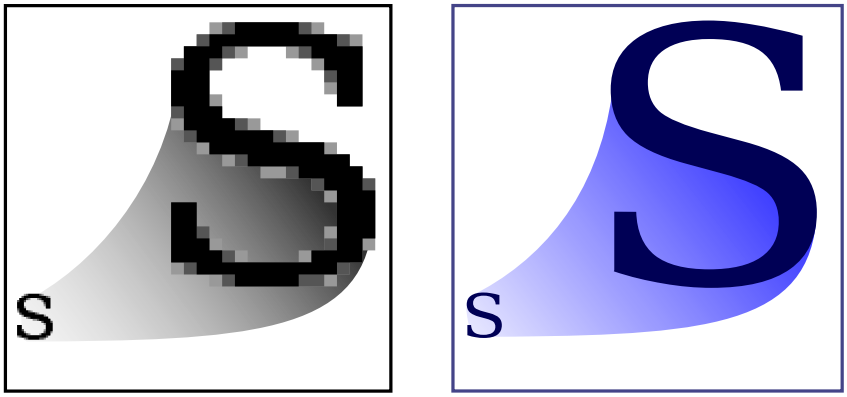
One should be aware of the implications of vector graphics and hence why we choose to examine them today. Given the effortless performance and tailored pipeline of raster graphics, it is a reasonable response to wonder why or how we can improve the imaging model with vector graphics. In the following sections, we will discuss the benefits of vector graphics.

2.3.1 Lossless Graphic Fidelity

Phones, televisions, and desktops have various resolutions and pixel densities, creating the need for resolution-independent graphics. We can solve this problem and show the benefit of lossless graphic fidelity with scaling in Figure 2.4 below. Vector graphics

[4]attribution: By Cmglee, modifications by Spencer C. Imbleau, CC-BY-SA-3.0

retain infinitesimal graphic fidelity at any scale or resolution, which implicates resolution independence. Although this is not a zero-cost abstraction for rendering, vector graphics are more portable across devices.



Raster
GIF, JPEG, PNG

Vector
SVG

Figure 2.4: Scaling comparison between vector and raster types.[5]

2.3.2 Storage Savings

Given that raster images are encoded pixel data, up-scaling raster images will grow the file size increasingly. On the contrary, vector graphics do not intrinsically encode concrete dimensions, and thus, file size is constant.

To prove this, we present a graphic of impossible cubes in Figure 2.5 and corresponding storage bloat in Table 2.1 below. The graphic file is canonically encoded in *svg* format, a common vector format. It is then scaled and encoded as a lossless raster format, *png*. While *svg* can grow and shrink without adjustments to file data, *png* can not. As such, we grow the *svg* to larger sizes and measure how the storage footprint changes for the *png* format.

[5]attribution: By Yug, modifications by Cfaerber et al., CC BY-SA 2.5

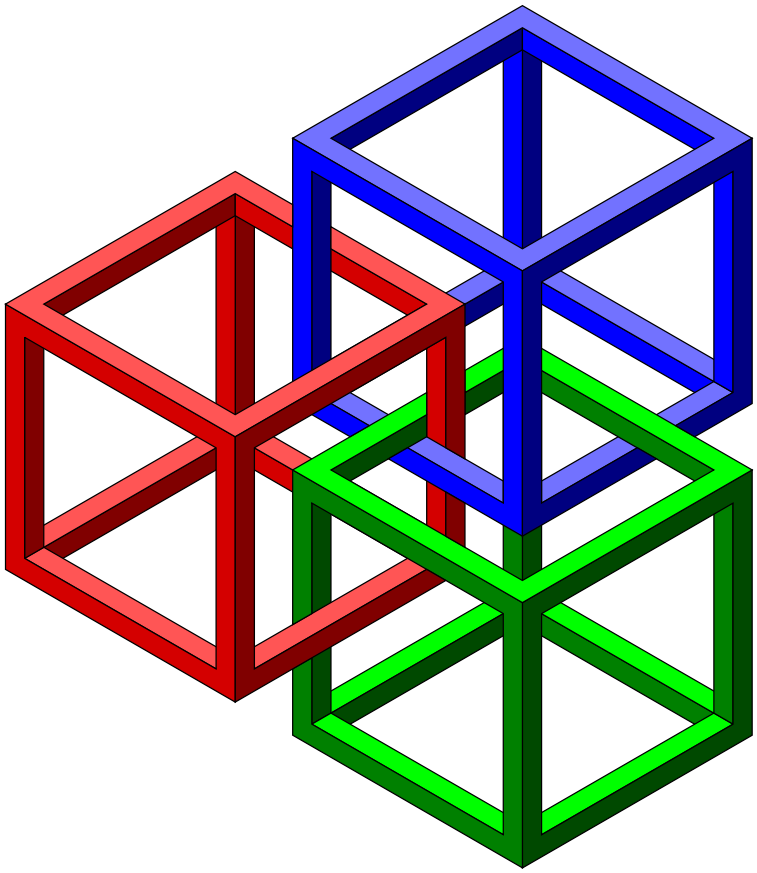


Figure 2.5: *Impossible cubes.*[6]

SVG vs PNG File Storage				
	SVG	PNG @ 1x	PNG @3x	PNG @6x
Size (KB)	9.4	61	210	474
Bloat		649%	2234%	5042%

Table 2.1: PNG file bloat from Figure 2.5.

The methodology for Table 2.1 is explained in Appendix A. The results show that vector types possess distinguished encoding supremacy resilient to scaling. Storage footprint has significant benefits when a file size incurs empirical consequences, such as latency incurred over network loading (e.g., web pages). It is also worth briefly mentioning *svg* is an *xml* format, which characteristically has significant amounts of repeated data. Compression algorithms, such as *svgz*, can make these results *better*.

[6]attribution: OpenClipart, SVG ID: 33931 , Public Domain

2.3.3 Powerful Primitives

Vector images amalgamate several primitives, such as points, lines, and Bézier curves. Bézier curves will be of particular interest, engineered as a "universal curve." The primitive's inherent malleability attributes this moniker; curves may be mutated directly with many abstract geometric transformations and through the control points, such as shown in Subsection §2.3.3.

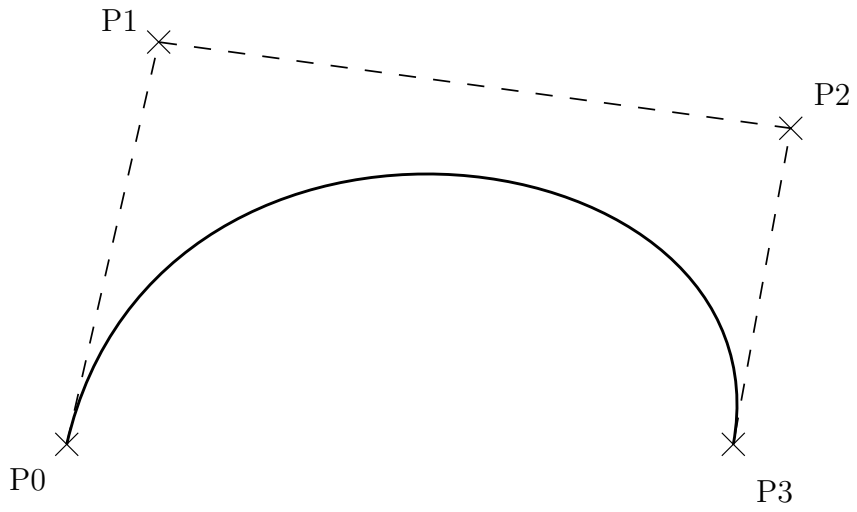


Figure 2.6: A cubic Bézier curve with four control points: $P0$, $P1$, $P2$, and $P3$.[7]

All curves can deform via affine transformations, such as translation and rotation. Curves also support sophisticated operations such as warping. Vector graphics also typically have support for complex set operations, z-ordering, and rich styling [6], shown in Subsection §2.3.3, although implementation support varies. Curves are the crux of vector graphics because of their complex features and mathematical properties, such as being able to be recursively subdivided into piecewise Bézier curves.

[7] attribution: Wikimedia Commons, Public domain

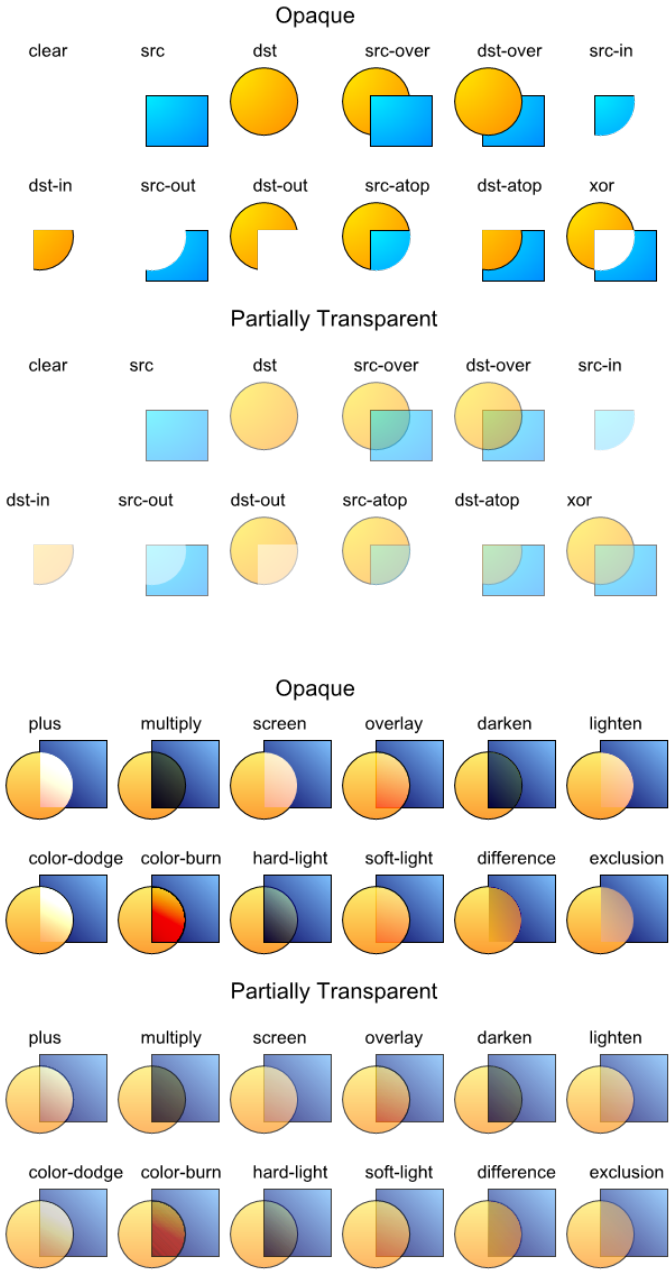


Figure 2.7: Compositing examples in vector graphics.[8]

Moreover, the ability to bend and deform Bézier curves losslessly leads to exciting implications for physics and animation. Finally, because vectors are independent of scale, we benefit from infinitesimally-precise data, valuable for scientific visualization and modeling.

[8]attribution: SVG Compositing Specification by W3C©, W3C©License

2.4 Disadvantages of Vector Graphics

While the vector imaging model can benefit us, there are cons to the model which impact a user's decision to adopt it.

2.4.1 Indirection Costs

It is generally much slower to process the vector imaging model. The raster model's performance is a symptom of image data stored in a readily accessible map of color data, called a *bitmap*. Unlike a bitmap, vector image data is stored implicitly as path data. This path data then must undergo processing in addition to rasterization. This processing expense is unique to vector types. Since the model is not stored in an immediately readable format, it is not easy to compete with the performance of raster graphics.

This cost is negatively compounded by the locality issue discussed in Subsection §2.2.2, requiring recalculation of the entire shape for minor alterations. Expensive redraws are indeed an obstacle for any dynamic, real-time applications which attempt to minimize input latency. While computation caching may help processing speed, this comes at the expense of additional memory.

2.4.2 Realism Storage Bloat

Contradicting the results of Subsection §2.3.2's *Impossible Cubes*, vector graphics struggle to reach a graphic fidelity comparable to photo-realism without file bloat, branding vector graphics hostile for realistic visualization. The file storage savings observed in Table 2.1 can be misleading due to the lack of complexity in the image. *SVG* is poor at compressing complexity. Algorithms used to convert raster formats (*jpg, png, etc.*) to vector formats (*svg*) produce high-volume output, depending on the degree of color discontinuity. Conversion occurs by joining similarly colored pixels and approximating areas into shapes, reducing information. The number of color discontinuities found in the input may produce many paths, even approaching the number of pixels, depending on the level of the output detail requested. However, due to poor vector image encoding standards cited in Subsection §2.2.1, conversions are magnitudes larger than the original raster image. Despite larger storage requirements, information is never gained and ironically lost typically, making the resulting vector file less useful than expected.

To prove this point, we present a vectorized image experiencing bloat, "*Landscape with the Castle of Massa di Carrara.*" This image, shown in Figure 2.8, displays a raster variant (left) and vectorized format (right). The original raster dimensions are 791x600 pixels and formatted as a *png*. Storage sizes of the variants are found in Table 2.2.

Figure 2.8: A raster (left) and vectorization (right) of famous art.[9]

Realism in PNG vs SVG File Storage			
	PNG	SVG	SVGZ
Size (KB)	979.1	7266.4	2563.1
Bloat		742.15%	261.77%

Table 2.2: Vectorization file bloat from Figure 2.8.

2.5 Tessellation

Tessellation, also called triangulation, may perhaps be the most famous, straightforward, and naive solution for 2D rendering. Tessellation is the conversion of complex paths into discrete triangles for use in a traditional rendering engine. This computation flattens curve primitives into line segments to connect all vertices into triangles, which is often a significant computation. Tessellation facilitates easy integration with any GPU graphics engine and requires few GPU features, making it an attractive option. Generally speaking, the complexity of paths is abstracted away. A tessellator ingests complex shapes as input and generates triangle geometry easily consumed by graphics APIs such as OpenGL, Vulkan or Direct3D.

[9]attribution: By Leo von Klenze, 1827

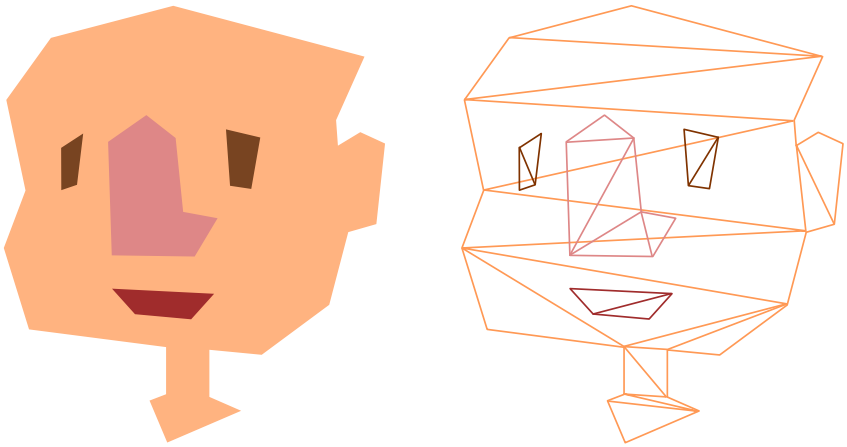


Figure 2.9: A visualization of tessellation.[10]

Vector tessellators do have special needs, however, because they must operate on curves. To allow this, libraries like *Lyon* perform curve flattening, which uses a linear approximation to generate line segments [7].

Figure 2.10: Curve flattening of a cubic Bézier curve.[11]

Curve flattening is a function of *tolerance*, the maximum distance between a curve and its linear approximation. Tolerance directly affects precision, and hence, a smaller tolerance provides higher precision and more segments. This variable is usually chosen in conjunction as a function of zoom level.

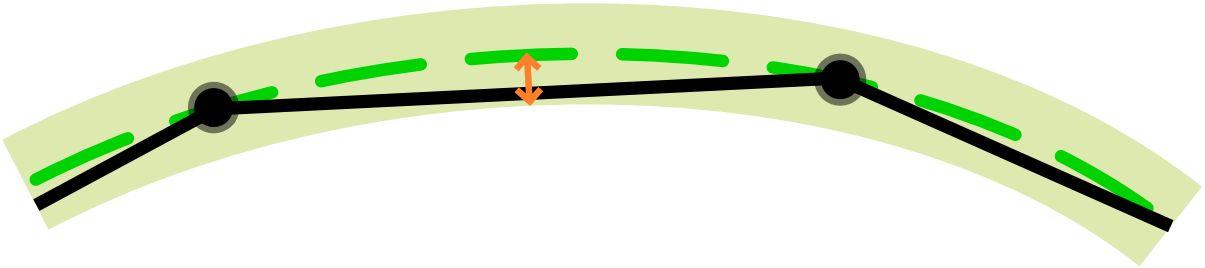


Figure 2.11: Permitted approximation error (tolerance) in curve flattening.[12]

[10] attribution: Lyon, MIT/Apache 2.0
[11] attribution: Lyon, MIT/Apache 2.0

Using tessellation as a fulcrum or supplement in vector pipelines is quite common. For example, one can find parts of Microsoft's Direct2D API [8] leverage a tessellation-based approach.

2.6 Conclusion

Vector graphics promise extraordinary benefits over raster graphics, such as lossless graphic fidelity, storage savings, and powerful primitives. These benefits make vector graphics a flexible image model worth further analysis.

[12]attribution: Lyon, MIT/Apache 2.0

Chapter 3

Literature review

Modern 2D GPU vector graphic rendering on the GPU is a culmination of impressive research. Insights include tessellation triangle-batching [7], stencil-buffer curve rendering [9], random-access vector graphics [10], a massively parallel pipeline [11], novel scan-line algorithms [12], and GPU architecture leveraging [13]. These findings have been integrated into many technologies, both individuals and entities. This section aims to survey notable vector rendering methods that expose field advancements to the modern-day. We attempt to qualify significance at a high level.

3.1 Technologies

Listed below are projects of significance due to popularity, performance, or variance in methodology. These technologies would justify first-class support in our analytic tool.

3.1.1 Skia

Skia[1] is the most widely used C++ 2D hardware-accelerated graphics library with support for vector graphics. The library has had commercial support from Google since 2005 while being open source [14]. *Skia* is used for rendering in Mozilla Firefox and Google Chrome web browsers, making it one of the most established graphic libraries.

The greatest difficulty with *Skia* is complexity. *Skia* is very feature-rich, supporting CPU and GPU rendering, multiple input and output formats, filters, color spaces, and color types. The project is over 370,000 lines of code, excluding dependencies. With dependencies, the project amasses over 7,000,000 lines of code and requires 8 gigabytes of disk space to be built. In addition, the final binary is 3-8 megabytes, depending on enabled features, causing contention for those optimizing bundle size. In addition, *Skia* can only be built with *clang*[2] and requires an obscure build system called *gn*[3] which uses *Python 2*. The complex library is old and complicated to work with, while most contributions originate from Google engineers and affiliates directly, rather than interested

[1] see: https://skia.org
[2] see: https://clang.llvm.org/
[3] see: https://gn.googlesource.com/gn/

volunteers. The renderer technology features a vector logo, shown below in Figure 3.1.

Figure 3.1: The *Skia* Logo.[4]

3.1.2 Pathfinder

Pathfinder[5] is a new, sophisticated 2D renderer designed for vector and font rendering. The renderer gains applause because it renders paths in a performant, analytic way. *Pathfinder* decomposes a very complex vector object into many smaller and simpler objects stored in a tiled lattice. Next, the library determines which tiles are occluded and enforces a culling policy on occluded shapes. These opaque tiles are submitted as a batch of instanced quads, minimizing redraw on pixels encountered in a traditional painter's algorithm. Quad batching allows more time to be spent on busier sections of the image and keeps the rest of the image inexpensive to draw [15].

To visualize how much overdraw a general example incurs with a traditional painter's algorithm, we present Figure 3.2 below with two versions of "*Ghostscript Tiger*." The left tiger paths are stripped of color value and replaced with a translucent white fill to visualize overlapping shapes easily. Hence, the whiter the pixel, the more times the pixel is drawn without occlusion culling. The methodology for generating this image is described in Appendix B.

[4]attribution: https://skia.org/, Fair use
[5]see: https://github.com/servo/pathfinder

Figure 3.2: "*Ghostscript Tiger*" shape overlap without occlusion culling (left) and original fill (right).[7]

Historically, *Pathfinder* was slighted to be used in the Servo[8] mission, which once shared code with Mozilla Firefox as an open-source embedded web engine. However, *Pathfinder* has lacked consistent development in its lifetime from the leading developer, Patrick Walton, and has suffered many backwards incompatible re-writes. Nevertheless, the project still remains useful for critique and analysis [16]. The renderer technology features a vector logo, shown below in Figure 3.3.

Figure 3.3: The *Pathfinder 3* logo.[9]

[7] attribution: By Nicolas Silva, modifications by Spencer C. Imbleau, MIT
[8] see: https://servo.org/
[9] attribution: Pathfinder, MIT/Apache 2.0

3.1.3 piet-gpu

piet-gpu[10] is an experimental prototype 2D GPU renderer currently in development, and relatively stable. The renderer features a novel compute-centric pipeline. The prototype is a retained-mode renderer, buffering scene-graph fragments on the GPU to accelerate static continuity. In addition, *piet-gpu* offers a portable runtime and compatibility fallback, making the renderer relatively general purpose. The research has contributed impressive results, namely with leveraging a sort-middle GPU architecture[11].

3.1.4 Spinel

Lastly, *Spinel*[12] is a perplexing renderer developed by Google, with little outside details. The future technology is self-described by Google as "a high-performance GPU-accelerated vector graphics, compositing and image processing pipeline" [17]. The technology currently exists as a graphics API in Google's new operating system, *Fuchsia*[13], but is likely to be integrated into *Skia* and follows a similar role. For now, the project is experimental and locked behind Google's operating system. Moreover, building and running Spinel is onerous, with little information to an end-user. For these reasons, working fluidly with Spinel may be oppressively difficult for the discernible future. However, promises Spinel makes are exciting, such as inexpensive redraw, extensibility for animation, entirely GPU-processed pipeline, and explicit support for paths, styling, and composition [18].

3.1.5 Lyon

Lyon[14] is not a vector renderer, instead it is a mature tessellator. However, Lyon is very popular because it abstracts away the difficulty of vector primitives via substitution, which integrates into a traditional raster pipeline with little to no GPU features [19]. Lyon implements an efficient sweep-line algorithm, traversing a shape from top to bottom with a knowledge of local geometry [20], although there are many methods. Such methods include constrained Delaunay triangulation [21] which may be hardware-accelerated [22] and ear clipping [23].

[10] see: https://github.com/linebender/piet-gpu
[11] see: https://raphlinus.github.io/rust/graphics/gpu/2020/06/12/sort-middle.html
[12] see: https://fuchsia.googlesource.com/fuchsia/+/refs/heads/main/src/graphics/lib/compute/spinel
[13] see: https://fuchsia.dev/
[14] see: https://github.com/nical/lyon

Figure 3.4: The logo for project Lyon.[15]

3.2 Research

Listed below are significant research advancements in arithmetic, theory, or results. This research is attractive to those wishing to join the field or seek more detail into the space.

3.2.1 Improved Alpha-Tested Magnification for Vector Textures and Special Effects

[24]

Originally a product of Valve, this research was presented at SIGGRAPH in 2007 as a novel encoding of raster images to improve the magnification of textures with low storage requirements efficiently. These encodings are signed distance fields, or *SDFs*. Rendering SDFs requires low hardware requirements and a trivial shader for the GPU. In addition, the model provides support for anti-aliasing and considerable up-scaling to traditional textures, making the research attractive to game developers. Given multiple channels, scaling can also be improved [25], shown in Figure 3.5. While SDFs are not a vector graphics model, the encoding is worth mentioning due to its popularity and similarity. It is also worth noting that generating an SDF requires a significant amount of computational resources and is typically done on the CPU, making it a pre-baked asset not suitable for dynamic rendering.

[15]attribution: Lyon, modifications by Spencer C. Imbleau, MIT/Apache 2.0

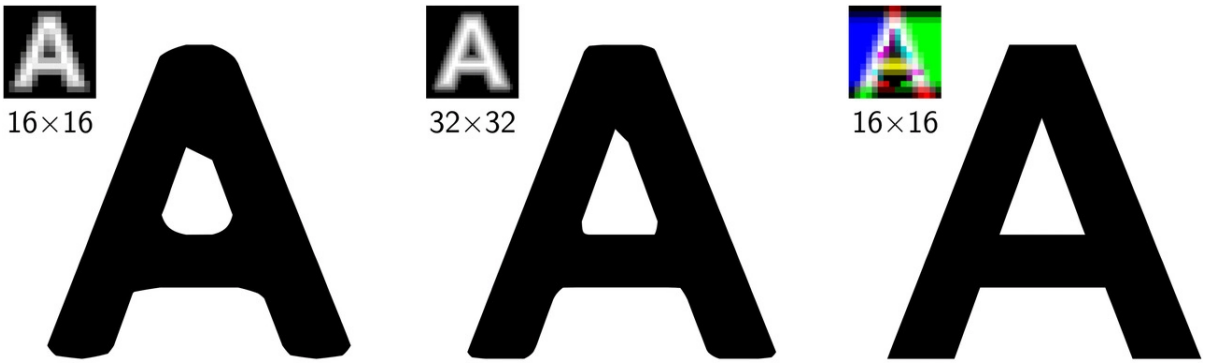


Figure 3.5: Low-resolution SDF upscaling (left), high-resolution SDF upscaling (middle), and multi-channel low-resolution SDF upscaling (right).[17]

3.2.2 Resolution Independent Curve Rendering Using Programmable Graphics Hardware

[9]

Presented at SIGGRAPH Asia in 2005 and published in the ACM Transactions on Graphics (TOG), Microsoft researchers Charles Loop and James Blinn presented the first major analytic algorithm to render resolution-independent vector graphics using programmable graphics hardware. The method constructs vector images from mosaics of triangulated Bézier control points using a newly conceptualized *stencil buffer* data structure. The method worked in two passes. First, a hull of triangles constructed asserts the shape's fill using a stencil buffer. After fill is determined, a second pass is required to cut out the fragments with a shader. The shader is, similar to SDFs, trivial. The shader algorithm functions by assigning varyings $u, v)$ to the control points of a quadratic Bézier curve, discarding the fragment under a retention policy. The shader's retention policy is denoted below in Equation 3.1.

$$\begin{gathered}\text{Given control points } P_0,\ P_1, \text{ and } P_2,\\ \text{apply varyings } (u,v) = (0,0),(0.5,0),(1,1),\\ \\ u^2 - v \geq 0\text{: Discard fragment}\\ u^2 - v < 0\text{: Keep fragment}\end{gathered} \tag{3.1}$$

3.2.3 Random Access Vector Graphics

[10]

[17] attribution: Improved Corners with Multi-Channel Signed Distance Fields, Fair Use

Presented at SIGGRAPH Asia and published by the ACM ToG in 2008, Diego Nehab and Hugues Hoppe created a tiling approach for vector graphics based on a considerable upfront computation expense. This pre-computation model enhanced the image's interactivity by providing an approach to redraw mapped vector images on arbitrary objects inexpensively. This technique significantly extended the ability to render static vector graphics (with support for transformations) at interactive rates. The pre-computation method required considerable resources to cache, making the process impossible for interactive applications that may deform the vector texture. In practice, the approach encoded "*Ghostscript Tiger*" in 0.44 seconds [10], which is not a challenging render by modern standards.

Figure 3.6: Ghostscript Tiger[18]

3.3 High Performance Software Rasterization on GPUs

[26] Authors Samuli Laine and Tero Karras, researchers from NVIDIA, had their work published at the ACM SIGGRAPH Symposium in 2011. Their implementation "CUDA Raster" was easily extensible and featured a traditionally software-based graphics pipeline on a GPU, which obeyed ordering constraints from traditional rendering pipelines. Their performance improved the CPU-based equivalence by 2–8x, comparing the approach to a top-of-the-line GPU in 2011. This research did not focus on vector graphics but set up many theories behind compute-based parallel rendering.

3.3.1 GPU-Accelerated Path Rendering

[27] Presented at SIGGRAPH Asia and published by the ACM ToG in 2012, Mark J. Kilgard and Jeff Bolz released one of the first analytic rendering approaches to 2D

[18]attribution: Ghostscript authors, AGPL

vector graphics on the GPU. Their approach builds upon existing techniques for curve rendering, specifically the stencil buffer technique[19]. Kilgard and Bolz however explicitly decouple the stencil step to determine path fill and stroked coverage with parallelism.

3.3.2 Massively Parallel Vector Graphics

[11] Published in the ACM ToG and Proceedings of ACM SIGGRAPH Asia 2014, Ganacim et al. reached higher levels of parallelization in vector graphics rendering. This solution further builds on previous models by Diego Nehab[20], which applied deformations and warps to vector graphics on arbitrary surfaces but optimized the pipeline for dynamism. The rendering pipeline divides into a pre-processing component that builds a novel, the shortcut tree, and a rendering component that processes all samples and pixels in parallel. As a result, tree construction is efficient and parallel at the segment level, enabling dynamic vector graphics.

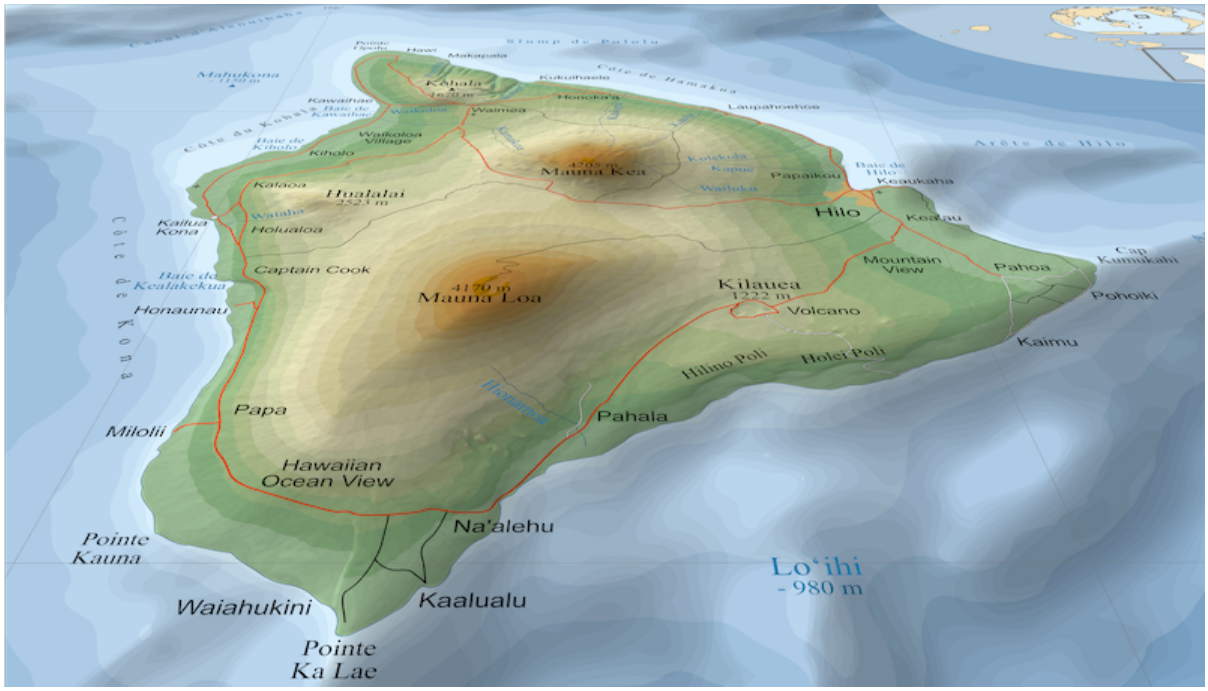


Figure 3.7: Massively parallel vector graphics rendered under a perspective warp.[21]

[19] see: Subsection §3.2.2
[20] see: Subsection §3.2.3
[21] attribution: Massively-Parallel Vector Graphics by Ganacim et al., Fair Use

3.3.3 Efficient GPU Path Rendering Using Scanline Rasterization

[12] Published in the ACM ToG and presented in SIGGRAPH Asia 2016, Li et al. released a significant milestone in vector graphics rendering. The solution is parallel, optimized, and supports dynamism inherently. The methods presented parallelize over boundary fragments (pixels intersecting the path boundary), and non-boundary pixels process in bulk, similar to CPU scanline rasterizers. This novel scanline algorithm significantly saves on the number of winding number computations. To this day, it remains one of the fastest methods for rasterization and GPU efficiency. Moreover, it supports animated input and outperforms many state-of-the-art alternatives.

3.3.4 Bay Area Rust March 2017: GPU Rasterization

[28] Patrick Walton was given a feature panel in Air Mozilla's Bay Area Rust event in March 2017, where he discussed his project *Pathfinder*[22]. During his presentation, he exposed the implementation in an easily accessed format. He noted *Pathfinder* uses tessellation to split curved shape edges into small line fragments within an arbitrary tolerance (3 pixels) using tessellation shaders. In the fragment shader, *Pathfinder* calculates the area defined by the bound tessellation fragments and stores the area relative to those around it in a novel way called *delta coverage*. After computing the delta coverage, *Pathfinder* sweeps every column in parallel to calculate the coverage in a prefix sum which translates to the winding fill rule for every pixel.

3.3.5 Sort-Middle Architecture

[13] Dr. Raph Levien's research blog defined a new architecture merged into his renderer "*piet-gpu*", further mentioned in Subsection §3.1.3. As described in the blog post,

> *The architecture calls for sorting in the middle of the pipeline, so that in the early stage of the pipeline, (2D) triangles can be processed in arbitrary order to maximally exploit parallelism, but the output (2D) render still correctly applies the triangles in order.* [13]

This research has helped *piet-gpu* to be a modern, experimental solution to dynamic vector rendering with support for mass input and animation. The architecture explains that the motivation for this compute-centric pipeline is to maximize parallelism. Dr. Raph Levien accomplished this through a multi-stage compute-centric pipeline with a sorting procedure in the middle. The performance claims and results listed display *signifcant* results for *piet-gpu* on NVIDIA©hardware in Figure 3.8, and work is still ongoing[23]. Dr. Raph Levien's blogs[24] have been insightful and reputable as a source of vector graphic field study and advancement in recent years.

[22] see also: Subsection §3.1.2
[23] see: https://github.com/linebender/piet-gpu
[24] see: https://raphlinus.github.io/

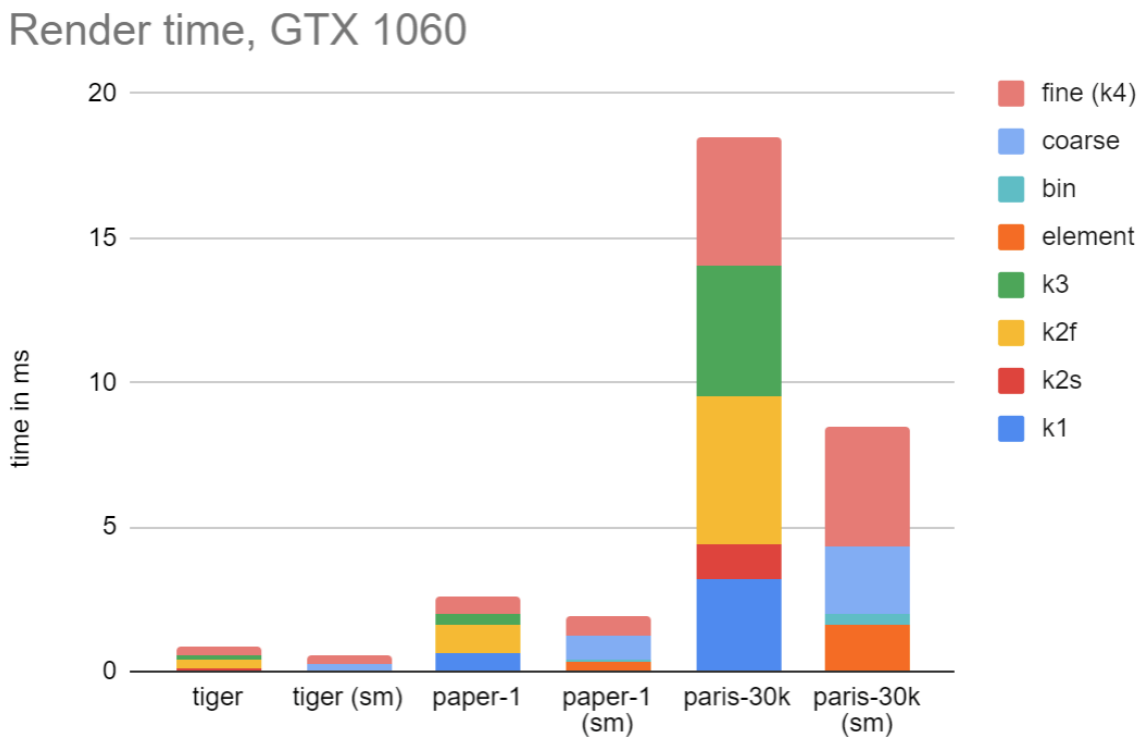


Figure 3.8: Sort-middle-architecture performance on NVIDIA©hardware[25]

[25]attribution: By Raph Levien, AGPL

Chapter 4

Theory

Presently, hardware-accelerated rendering of vector graphics is fairly onerous for those unfamiliar with the imaging model. This difficulty leads many to wonder if the 2D imaging model is nearing uselessness, or can we prove, with testable predictions, that 2D imaging can extend its usefulness? Due to the evolving nature and experimentation still ongoing, nothing has earned an established reputation or developed with mature documentation and resources.

The lack of mature resources has imposed a steep learning curve for developers. Moreover, developers question the certainty of adopting the image model with no mainstream attention. Hence, we find it to be an appropriate step to provide tooling for such concerns. The following sections explain patterns that guide our decision-making and methodology in the following section.

4.1 Diverse Optimization Goals

Analyzing performance for vector graphics on the GPU is complicated due to many optimization goals in varying contexts and dimensional spaces. While 3D graphics typically optimize for a level of graphic-richness without sacrificing an acceptable frame rate, 2D graphics have many different cultural applications. Optimization goals may be low latency, power consumption for mobile environments, the contention of scarce resources (CPU $\Leftrightarrow$ GPU bandwidth), or balancing several of these factors.

Conducting a hardware-accelerated performance analysis is a stark contrast to traditional time trials and discrete measurements such as *fps*. In typical cases, these metrics are usually enough, and *Big-O* is a decent proxy. However, GPU analysis tools should be more contextually agnostic, offer accurate instrumentation, and support hardware metric sampling to yield measurements that support varied optimization goals.

4.2 Rendering Models

Technologies and research examined in our literature review appear diverging and experimental, but there are some similarities between items. Below, we interpret vector ren-

dering classifications but admit there are no strict definitions or generalized approaches.

4.2.1 Pre-Computation Models

We define pre-computation rendering models as an umbrella term for rendering techniques that pay computer resources up-front at runtime for a GPU-friendly representation. These approaches typically leverage GPU caching and re-use of computed assets in volatile memory (RAM, GPU memory). Pre-computation models almost always optimize inexpensive re-draw of static vector graphics and may often be computed on the CPU before being uploaded to a storage buffer on the GPU. Some examples include:

- Glyph caching for inexpensive text rendering
- CPU Tessellation uploaded to GPU storage buffers
- Random-access vector graphics[1]

4.2.2 Parallel Models

Contrary to pre-computation models, *parallel* models are techniques that do not rely heavily on caching a GPU-friendly version upfront. Hence, these techniques are optimized better for dynamism, with shape evaluation calculated on the GPU. These techniques traditionally leverage more GPU features and pipelining such as compute kernels to circumnavigate the rigidity of the graphics pipeline. Such methods are typically the only practical filter for dynamism, interactivity, or animation solutions. Some examples include:

- Parallel winding number calculation[2]
- piet-gpu[3]
- Pathfinder 3[4]

4.3 Feature Variance

Rendering techniques are difficult to compare on the GPU because the extent of hardware leveraging and features used are often elided or not quantified. Providing an analytic framework to benchmark and measure arbitrary axes of vector graphics seems necessary to encourage proving specific models and techniques with context to others. Current research claims mainly consist of cursory comparisons or time trials. Such claims are usually anecdotal, failing to provide a complete story and significance to new techniques.

[1] see: Subsection §3.2.3
[2] see: Subsection §3.3.3
[3] see: Subsection §3.1.3
[4] see: Subsection §3.1.2

An extensible API which rapidly prototypes benchmarks with visualization support would understandably mitigate speculation. By benchmarking, performance results would provide confidence in research and survey the current 2D GPU path-rendering capability. This capability would hopefully modernize expectations for vector renderers. These optics on outlying behavior can highlight lacking performance and aid in explaining obscure phenomena.

4.4 Referential Comparison

As we previously mentioned, there are competing optimization goals and varying hardware leverage in vector rendering. Given this lack of coherence and objective performance expectations for a vector renderer, a baseline would be helpful: "*What* are *the modern expectations of a vector graphic renderer?*"

Chapter 5

Design and Methodology

Our product, *vgpu-bench*, is a benchmarking framework for measuring hardware-accelerated vector graphics, emphasizing further analysis. Below we will explain our methodology, steps we take, and justifications for our design decisions. Once we detail our goals and explain our architecture, we validate and verify our analytic framework through a test case to prove by construction.

5.1 Requirements

Our analytic framework for hardware-accelerated vector graphics is engineered to be trustworthy and resourceful, establishing results one might naturally cite as evidence. To accomplish this vision, we establish functional and non-functional requirements for *vgpu-bench*.

Citing diverse optimization goals in Section §4.1, our requirements entertain the hope that our framework should be extensible and capable of rapidly assessing generic axes of interest with concrete evidence. One may hypothesize "*Where do current vector graphic approaches maximize graphic richness without sacrificing frame rate across a range of hardware and scene complexity?*" Hence, we present the following requirements for our framework below to answer questions like these and drive a broad set of design goals.

5.1.1 Functional Requirements

- The system should be capable of collecting arbitrary measurements.
- The system should be capable of GPU metric sampling.
- The system should be capable of rapid-prototyping.
- The system should provide conveniences such as macros, common trait implementations, and conversions.
- The system should be capable of writing collected measurements.

- The system should be capable of visualizing collected measurements.

5.1.2 Non-Functional Requirements

- The system should collect measurements accurately with precise timing and synchronization.
- The system should integrate into proprietary software APIs for GPU metric sampling for GPUs within the last five years.
- The system should provide serializers and writers to write measurements.
- The system should provide plotting utilities to visualize measurements.
- The system should encourage adoption through features and pre-written examples, with foreign language interfaces.
- The system should incur no costly consequences with foreign function interfacing.

5.2 Architecture

The architecture of our benchmarking framework was designed in part to accentuate our functional requirements. We chose a design tailored to optimize extensibility and accuracy in a data flow, deriving the concept of containerization. At the highest level, our API orchestrates *drivers*, which are customized runtime executors for *benchmarks*. Benchmarks are containerized function closures that return something discretely *measurable*. Benchmarks also allow augmentation via *monitors*, which poll supplementary measurements at specified frequencies. Measurements are then output by *writers* and *plotters* conveniently for the developer. See Figure 5.1 below for a simplified organizational diagram.

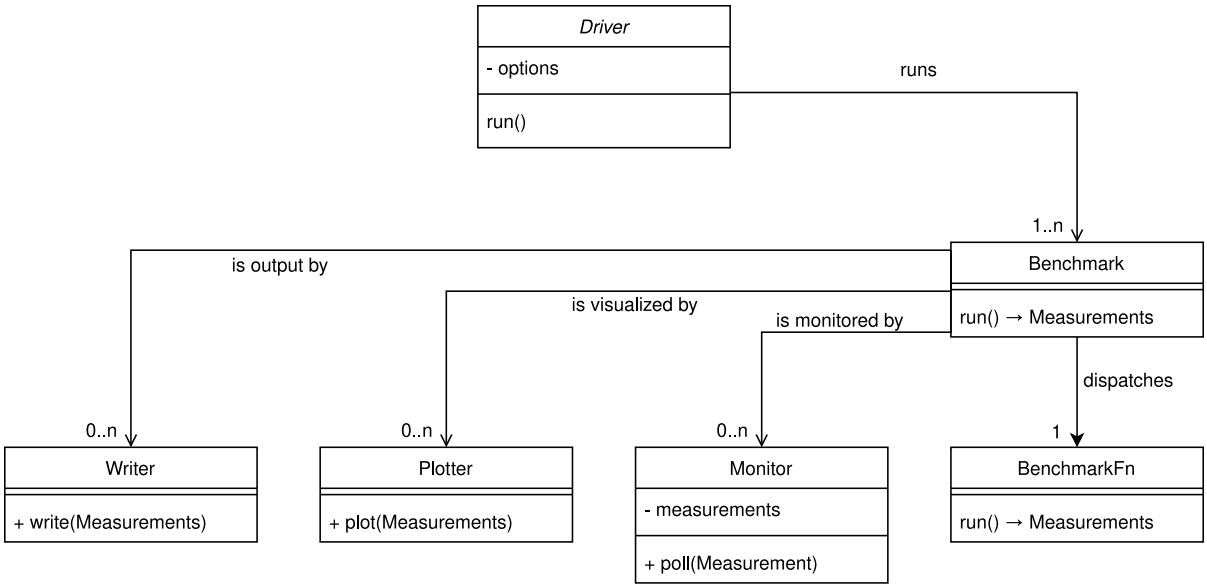


Figure 5.1: A simplified organization of *vgpu-bench*.[1]

5.2.1 Data Flow

A driver's runtime will orchestrate the execution of independent benchmark closures sequentially to prevent interference among benchmarks, with resulting measurements collected synchronously. If benchmarks are augmented with monitors, monitors will poll supplemental measurements in parallel during the runtime of a benchmark. Various atomics and barriers synchronize events because of the parallelism at runtime between monitors and benchmarks. After benchmarks are complete, measurements collected by all entities are passed as a bundle to writers and plotters for the archiving of data and visualizations, respectively.

Below in Figure 5.2, we present a simplified sequence diagram of the general data flow in *vgpu-bench*, starting with the `Driver`. Note that this diagram is purely supplemental for reference. Refined explanations are presented in the following sections.

[1] attribution: By Spencer C. Imbleau, MIT/Apache 2.0

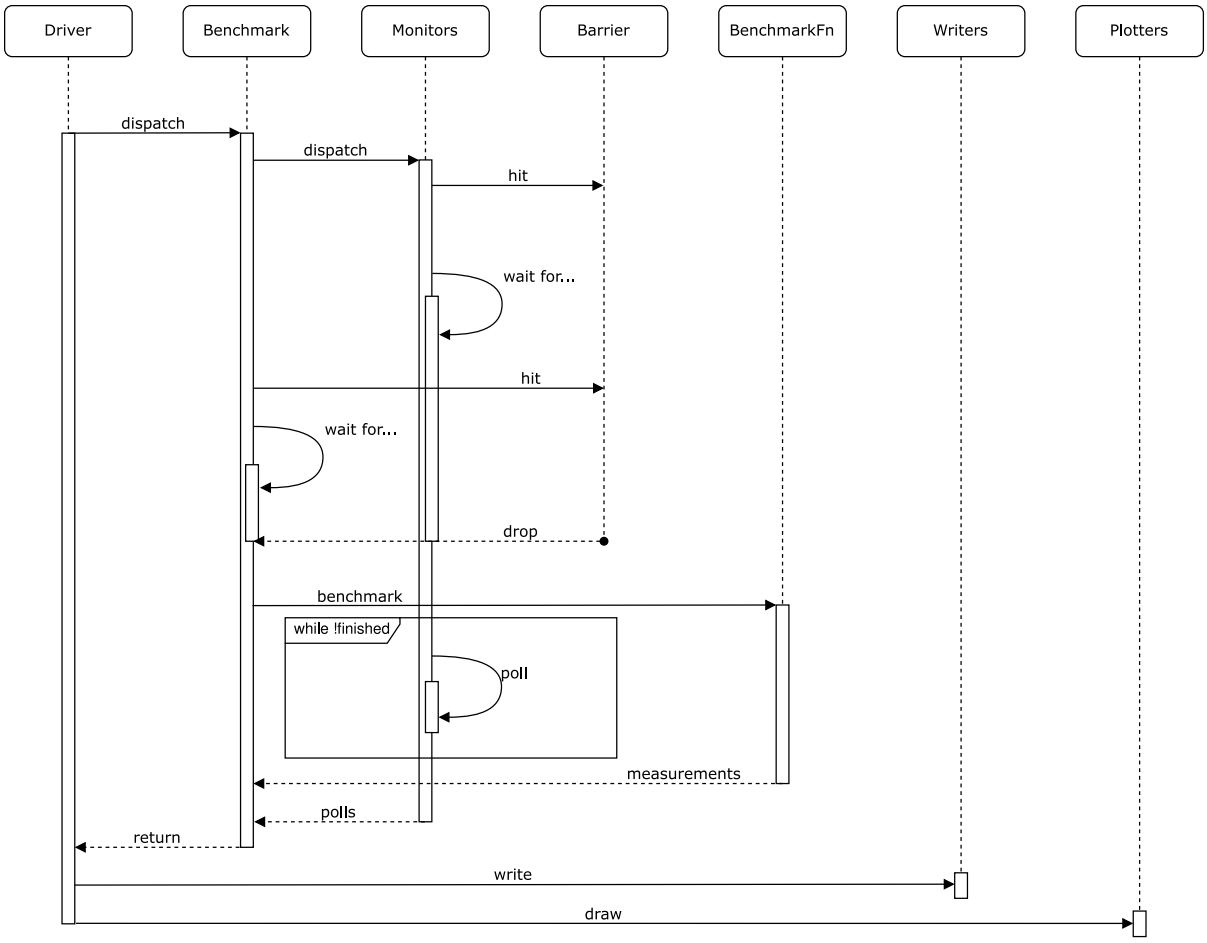


Figure 5.2: The sequencing of *vgpu-bench*.[2]

Measurable

One should primarily be acquainted with the `Measurable` trait. This trait is the only way to collect data through *vgpu-bench*. `Measurable` is, however, simply a trait alias for the constraints `Serialize`[3], `Debug`[4], `Send`[5], and `Sync`[6]. These traits are derivable for every primitive, complex structs, and enums within Rust. Most of what one considers serializable intrinsically could be automatically derived into a `Measurable` through macros. Our library provides metaprogramming macros making it trivial to derive this behavior, explained later in a Subsection §5.2.5.

[2] attribution: By Spencer C. Imbleau, MIT/Apache 2.0
[3] see: https://docs.serde.rs/serde/trait.Serialize.html
[4] see: https://doc.rust-lang.org/std/fmt/trait.Debug.html
[5] see: https://doc.rust-lang.org/std/marker/trait.Send.html
[6] see: https://doc.rust-lang.org/std/marker/trait.Sync.html

BenchmarkFn

Once a developer has something to measure, a benchmark is written in the form of a function closure that returns `Measurements`, a data structure that collects `Measurable` trait objects. This closure is encapsulated in a `BenchmarkFn`, preserving the behavior and measurements, but automatically annotating GPU tracers around the closure, compatible with NVIDIA©'s *Tools Extension SDK*, further referred to as *NVTX*. In simpler terms, `BenchmarkFn` is synonymous with a function closure with GPU *NVTX* annotations. When a binary executes a `BenchmarkFn` through NVIDIA©tools such as NVIDIA©*Nsight Systems*[7], these GPU tracers are observed, making GPU metric sampling and integration trivial for developers.

Benchmark

A `Benchmark` is the wrapping data structure which encapsulates a `BenchmarkFn`. The parent benchmark struct performs execution of the inner `BenchmarkFn` and allows parallel supplemental measurement polling through one or more `Monitor` data structures.

Monitors

The `Monitor` trait requires a frequency for polling and a function closure that returns something `Measurable`. Then, during runtime execution, time-sensitive wake-ups orchestrated by the `Benchmark` request the `Monitor` to poll and return a measurement. These polled `Measurement`s are collected automatically. A `Monitor` is a way to extend a benchmark's behavior easily by tacking-on supplemental measurements to record.

Driver

Finally, the `Driver` is a runtime executor responsible for one or more `Benchmark`s. `Driver`s are created through a `DriverBuilder` which builds the runtime execution behavior with various options. Options include writing mode, target directory, and others, such as whether to continue on errors.

Writers and Plotters

Writers and plotters are avenues of outputting data in desired formats. A `Writer` does exactly what its name implies, write `Measurements` to a file, while a `Plotter` outputs graphs through foreign function interfacing (*FFI*) to Python's graphing library `matplotlib`. Several plotters are provided for general use cases, such as numeric line graphs, which abstract the difficulty of FFI away from the developer. In general cases, few requirements are imposed on the developer, such as ceremoniously choosing configuration parameters for plotting. One may also ignore these conveniences and plot through one's favorite spreadsheet or data visualization application.

[7] see: https://developer.nvidia.com/nsight-systems

5.2.2 Data Sampling

Data collection and sampling accuracy are paramount concerns in building a benchmarking framework. The following sections will explain what instrumentation we integrate for GPU metrics and how we guarantee the accuracy for polled CPU metrics.

GPU Instrumentation

We briefly introduced `BenchmarkFn` in Subsection §5.2.1, where we exposed that each function closure in wrapped in GPU tracer annotations. These tracer annotations are invocations to the NVIDIA©*Tools Extension SDK (NVTX)*[8]. *NVTX* performs GPU and CPU profiling for NVIDIA©line hardware through a feature-rich CLI and GUI profiler, which can identify hardware starvation, insufficient parallelization, expensive algorithms, and more. Integrating our analytic framework into *NVTX* is a *necessity*, given NVIDIA©'s market share across desktop-grade GPUs.

NVTX provides a C-based API for annotating events, code ranges, and resources in applications. Although it is a C-based API, we can interface the C API in Rust with identical behavior and no overhead through foreign function interfacing (FFI) [29]. Our library will leverage tracer annotations automatically for the developer across the architecture. We also provide this FFI binding to developers with the respective implementation details elided. This binding allows developers to add additional annotations and markers using our framework without knowledge of *NVTX*. See Code Example 5.1 below for example code and Figure 5.3 for the code observed in NVIDIA©*Nsight Systems*[9].

[8] see: https://docs.nvidia.com/gamework.../nvtx/nvidia_tools_extension_library_nvtx.htm
[9] see: https://developer.nvidia.com/nsight-systems

Code Example 5.1: *NVTX* markers through macros provided in *vgpu-bench*.

```rust
use std::{thread, time::Duration};
use vgpu_bench::prelude::*;

#[measurement]
struct TessellationMeasurement {
    tessellation_time: f32,
}

pub fn main() -> Result<()> {
    BenchmarkFn::new(|| {
        let mut measurements = Measurements::new();
        // Annotating steps of a benchmark...
        nvtx::mark!("Step 1 - Begin");
        thread::sleep(Duration::from_secs_f32(0.5));
        measurements.push(TessellationMeasurement {
            tessellation_time: 0.5,
        });
        nvtx::mark!("Step 2 - Begin");
        thread::sleep(Duration::from_secs_f32(0.35));
        measurements.push(TessellationMeasurement {
            tessellation_time: 0.35,
        });
        // Benchmarking done!
        Ok(measurements)
    })
    .run("Benchmark Test")?;

    Ok(())
}
```

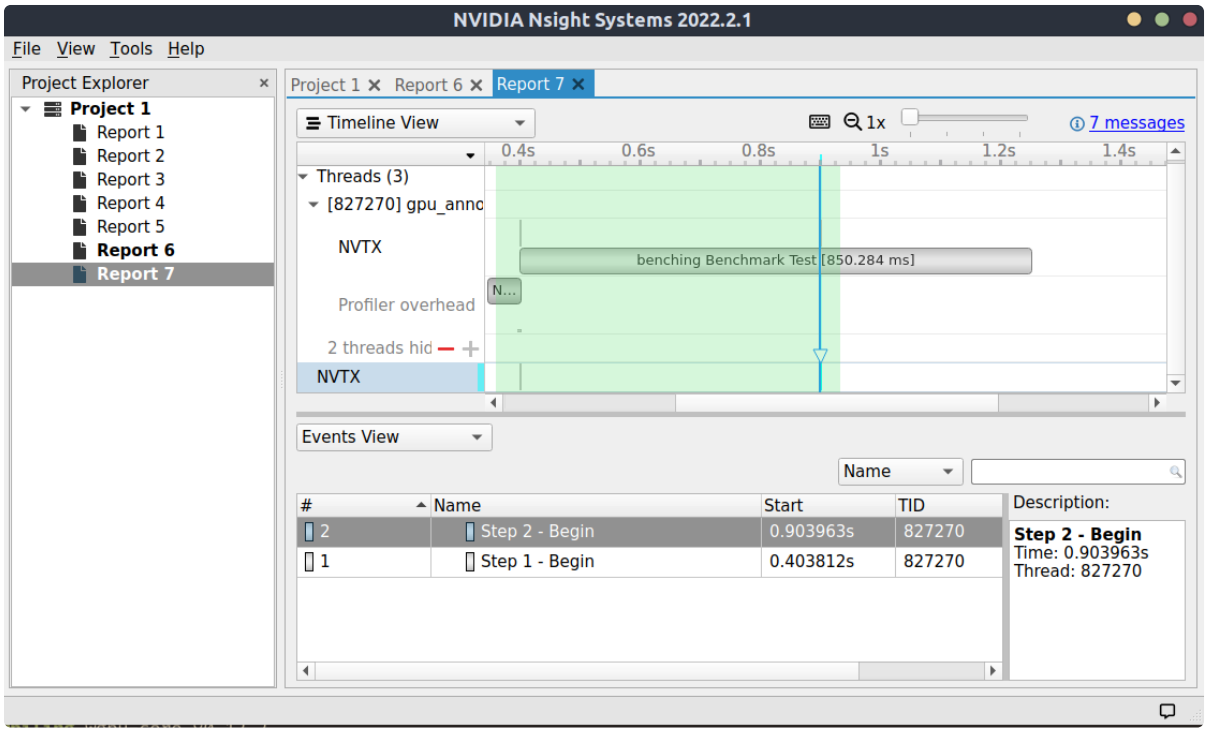


Figure 5.3: *NVTX* annotations observed in Code Example 5.1.[10]

GPU Metric Sampling

GPU metric samples may be necessary to collect to prove the efficacy of varying parallel models[11]. For example, one may need to dissect hardware starvation, count compute shaders in flight, recognize poor parallelization, or identify expensive algorithms across hardware in a benchmark. Hence, this is why we provide NVIDIA©instrumentation automatically to our library so we may annotate these anomalies with annotations. In Figure 5.4 below we show a GPU metric sample example on an NVIDIA©GeForce RTX 3060.

[10]attribution: By Spencer C. Imbleau, MIT/Apache 2.0
[11]see: Subsection §4.2.2

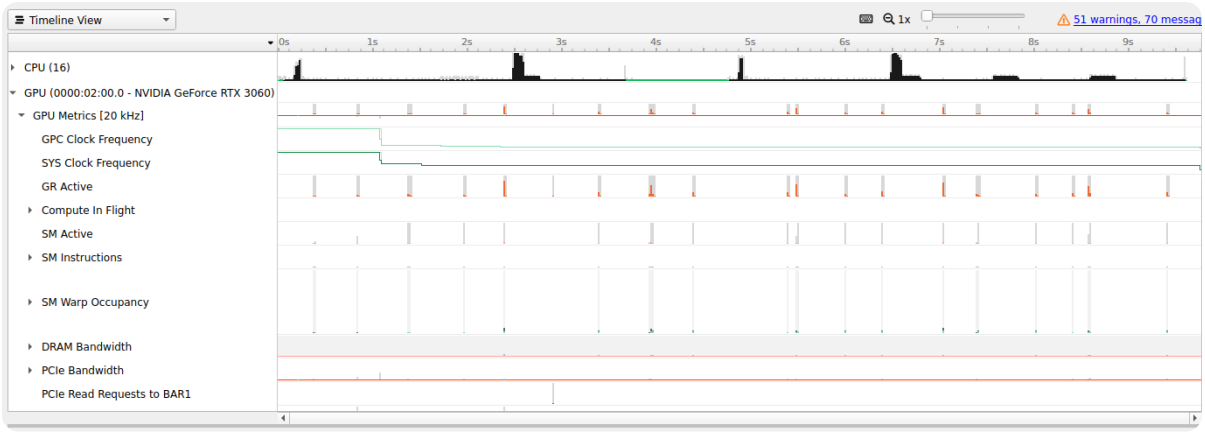


Figure 5.4: GPU metric sampling on an NVIDIA©GeForce RTX 3060.[12]

While the architecture in *vgpu-bench*is open and flexible, there are some restrictions to access GPU sampling through NVIDIA©*Nsight Systems*[13], imposed by the hardware developers, namely:

Operating system The currently supported operating systems for NSight Systems are given below.

- Ubuntu 18.04 and 20.04
- CentOS 7+
- Red Hat Enterprise Linux 7+

Hardware and drivers Graphics cards required must be at least Turing architecture or newer, with minimum driver versions provided below.

- NVIDIA Turing architecture TU10x, TU11x - r440
- NVIDIA Ampere architecture GA100 - r450
- Ampere architecture GA100 MIG - r470 TRD1
- Ampere architecture GA10x - r455

Accuracy guarantees

One philosophy exercised was to elect Rust's nightly features[14] if those features encouraged a subjectively better API. However, we restricted code impacting data handling to

[12]attribution: By Spencer C. Imbleau, MIT/Apache 2.0
[13]see: https://developer.nvidia.com/nsight-systems
[14]see: https://doc.rust-lang.org/rustdoc/unstable-features.html

stable, standard library features exclusively. This philosophy allows us to make a strong guarantee of memory integrity and safety.

While data integrity is a non-issue, parallelized data sampling across `Monitor`s was susceptible to race conditions. Therefore, we enforced atomic synchronization at execution start with the use of a `Barrier`[15]. During parallel data sampling, `Montior`s have a set frequency for polling. For every measurement collected by a `Monitor`, a *delta-time* is measured to ensure data collection was delivered in the strict frequency specified by the `Monitor`, while logs emit warnings if deadlines are not kept. Reported time error may be visualized with built-in support for standard deviation within given `Plotter`s.

5.2.3 Language Choice

This section aims to justify our decisions on language choice. We chose Rust as the programming language for a benchmarking framework for many reasons. Among those concerns are speed, safety, utility, and popularity.

Speed

The first reason we chose Rust is because of speed. Rust is built on the notion of zero-cost abstractions. Zero-cost abstractions give the ability to move certain behaviors to compile-time execution or analysis, incurring no runtime cost [30]. This guarantee provides ergonomic abstractions by providing easy to understand code without runtime overhead. Hence, runtime speed is approximately equivalent to that of C++. In addition, method calls and hooks through foreign function interfaces to another language's application binary interface with identical speed to the foreign language itself [29]. These zero-cost abstractions make benchmarking overhead agnostic to the target language.

Safety

Rust was the first language to popularize a memory-safe programming model that tries to guarantee no undefined behavior. Undefined behavior can lead to misleading measurements, unstable control flow, or *really* anything. Although unsafe code is permissible with explicit annotations, unlike C and C++, the language is built to guarantee the integrity of memory, with operations such as dereferencing raw pointers being disallowed [31].

Utility

Among the most important use-cases for vector graphics is web rendering, given its high impact among daily web browsing users. Fortunately, the companies which own the two most major web browsers currently use Rust to test their research, providing added portability. One company, Google, is developing *Spinel*[16] partly with Rust. The other

[15] see: https://doc.rust-lang.org/std/sync/struct.Barrier.html
[16] see also: Subsection §3.1.4

company, Mozilla, has written Firefox core and Servo [32] in Rust, with Mozilla engineers being the original creator of Rust [33].

Web rendering also begs the consideration of portability. Thankfully, Rust is a cross-platform systems programming language with fine control over memory where needed, and is capable of targeting all major operating systems with transpilation while featuring tiered support to several architectures.

Popularity

Rust is an elective choice for most new technologies in the experimental and academic corner involving hardware-accelerated vector graphics. In fact, many of the modern pieces we discuss in the recent years such as *Pathfinder*, *piet-gpu*, and *Lyon* are written entirely in Rust. Rust has also been voted the most loved language for over five years in the annual *Stack Overflow* developer survey [34].

5.2.4 Extensibility

Extensibility is a concern with our framework because of varying contexts and optimization goals in vector graphics, discussed in Section §4.1. We aim for a "plug-n-play" solution that fits into almost any existing solution with an effort to minimize glue required by a developer.

Generics

We have provided the `Driver`, `Benchmark`, and `BenchmarkFn` to return generics to allow the developer to specify user-defined accuracy returning user-defined measurements. Technically, these data structures are implemented as `Driver<M>`, `Benchmark<M>`, and `BenchmarkFn<M>`, such that `M` implements `Measurable`. Use of generics here allows arbitrary accuracy and data control to the developer.

Serialization

One of the only constraints of a `Measurable` data structure is implementation of `Serialize`[17]. The constraint requires the data structure to have a defined policy to convert data into an easily transmittable form, such that it may be ingested by a `Writer` or `Plotter`.

5.2.5 Software API

The ergonomics of our architecture should lead to an intuitive, decoupled API for developers which makes sense and enables rapid prototyping. We will provide code examples and show how we accomplish these goals to fit our functional requirements.

[17] see: https://docs.serde.rs/serde/trait.Serialize.html

Intuitiveness

Our software API follows all conventions and API guidelines established by the Rust-language team [35]. These guidelines include eagerly implementing common traits which play well with other libraries, providing documentation, and following best practices, such as semantic versioning[18], to ensure user-friendliness. We go beyond the checklist, dually specializing in rapid prototyping. We support rapid prototyping by reducing boilerplate code where possible. We provide a prelude, well-behaved macros, and take advantage of our architecture's indirection with support for conversions.

Prelude Providing a prelude allows easy and quick access to almost all significant types through a universal import. Although this is not practical if the binary size is a concern, it can be an excellent way to quickly import everything one may use in a benchmark.

Code Example 5.2: The prelude import statement for *vgpu-bench*.

```
use vgpu_bench::prelude::*;
```

Macros Macros provide code that writes other code, also known as metaprogramming. Rust has macro-support that enables functionality similar to functions but without runtime cost. Building upon Rust's philosophy of zero-cost abstractions and rapid prototyping, our software supports many well-behaved macros which increase developer productivity.

For example, the architecture of `Measurable` is a type alias constrained to any data structure which may be serialized, debugged, and is safe to send and synchronize across thread boundaries. These requirements alias the traits `Serialize`[19], `Debug`[20], `Send`[21], and `Sync`[22]. These are many trait constraints, and hence, it would often be burdensome and anti-thetic to the idea of rapid prototyping as a requirement to implement every trait. As a solution, we provide a procedural macro attribute, `#[measurement]`, among others, to automatically derive these traits in-line at compile time. See Code Example 5.3 below.

[18] see: https://semver.org/
[19] see: https://docs.serde.rs/serde/trait.Serialize.html
[20] see: https://doc.rust-lang.org/std/fmt/trait.Debug.html
[21] see: https://doc.rust-lang.org/std/marker/trait.Send.html
[22] see: https://doc.rust-lang.org/std/marker/trait.Sync.html

Code Example 5.3: Deriving the `Measurable` trait with a procedural macro.

```
#[measurement]
struct ToleranceMeasurement {
    tolerance: f32,
    polygons: u32,
}
```

Indirection Our API attempts to reduce boilerplate and complexity where possible by taking advantage of indirection. One may easily opt-out of extended features available in the architectural wrappers `Driver` and `Benchmark`. For example, a `BenchmarkFn` closure may be executed alone if there is no need for `Monitor` orchestration provided by the `Benchmark` wrapper. A `BenchmarkFn` will still incur the benefits of automated GPU annotations on behalf of *vgpu-bench*. See Code Example 5.4 below for an example.

Code Example 5.4: Rapid-prototyping execution using only `BenchmarkFn`.

```rust
use vgpu_bench::prelude::*;

#[measurement]
struct ToleranceMeasurement {
    tolerance: f32,
    polygons: u32,
}

pub fn main() -> Result<()> {
    BenchmarkFn::new(|| {
        let mut measurements = Measurements::new();
        // Collect real measurements here...
        for i in 0..10 {
            measurements.push(ToleranceMeasurement {
                tolerance: 1_f32 / i as f32,
                polygons: i * i,
            });
        }
        // Benchmarking done!
        Ok(measurements)
    })
    .run("Tolerance Test")?
    .write("output/tolerance.csv")?;

    Ok(())
}
```

Effortless conversion Conversions traits are eagerly implemented, allowing individuals requiring additional complexity to easily upgrade items such as a closure into `BenchmarkFn`, into a `Benchmark`, into a `Driver`. See Code Example 5.5 below for an example.

Code Example 5.5: Effortless conversions of data structures in *vgpu-bench*.

```
use std::{thread, time::Duration};
use vgpu_bench::{monitors::CpuUtilizationMonitor, prelude::*};

#[measurement]
struct RenderTime {
    render_time: u32,
}

pub fn main() -> Result<()> {
    let closure = || {
        let mut measurements = Measurements::new();
        // Collect real measurements here...
        for i in 0..5 {
            let render_time_ms = 1.0 + 0.5 * (i as f32).sin();
            measurements.push(RenderTime { render_time_ms });
            thread::sleep(Duration::from_secs_f32(render_time_ms));
        }
        // Benchmarking done!
        Ok(measurements)
    };
    // Convert closure into GPU-annotated `BenchmarkFn`
    let benchmk_fn: BenchmarkFn<RenderTime> = closure.into();
    // Create `Benchmark` from `BenchmarkFn`
    let benchmark: Benchmark<RenderTime> = Benchmark::from(benchmk_fn)
        // Attach a monitor
        .monitor(CpuUtilizationMonitor {
            name: "CPU Utilization Monitor",
            frequency: MonitorFrequency::Hertz(1),
        });
    // Convert `Benchmark` into `Driver`
    let driver: Driver<RenderTime> = benchmark.into();
    // Execute
    Ok(driver.run()?)
}
```

5.2.6 Features

Our product *vgpu-bench*offers additional features for various reasons. As of this publication, our product offers an *svg* generator, tessellation renderer, and pre-written rendering and tessellation benchmarks.

Including these features only requires the developer specify the desired features to

their project's `Cargo.toml` file. See Code Example 5.6 for an example `Cargo.toml` file.

Code Example 5.6: Importing feature dependencies from *vgpu-bench*.

```
[package]
name = "An example benchmark"
version = "0.1.0"
authors = ["Spencer C. Imbleau <spencer@imbleau.com>"]
edition = "2021"

[dependencies]
vgpu_bench = {
    version = "*",
    features = ["svg-generator",
                "render-kit",
                "tessellation-kit"]
}
```

Cargo.toml Features		
Feature	Provides	Default?
svg-generator	An *svg* file generator with options for scale, amount, and primitive used.	No
render-kit	Access to pre-written benchmarks and a baseline GPU-centric renderer for comparison.	No
tessellation-kit	Access to pre-written benchmarks and a baseline tessellator for comparison.	No

Table 5.1: Features of *vgpu-bench*.

SVG Generator

The `svg-generator` feature injects an *svg* generator crate into the root library. This crate allows the generation of *svg* files with varying primitives, amounts, and rotations. This handy crate quickly mocks *svg* data, which is the established vector standard for web rendering. These files can be manipulated or used directly in tests. It is also possible to define and generate custom primitives.

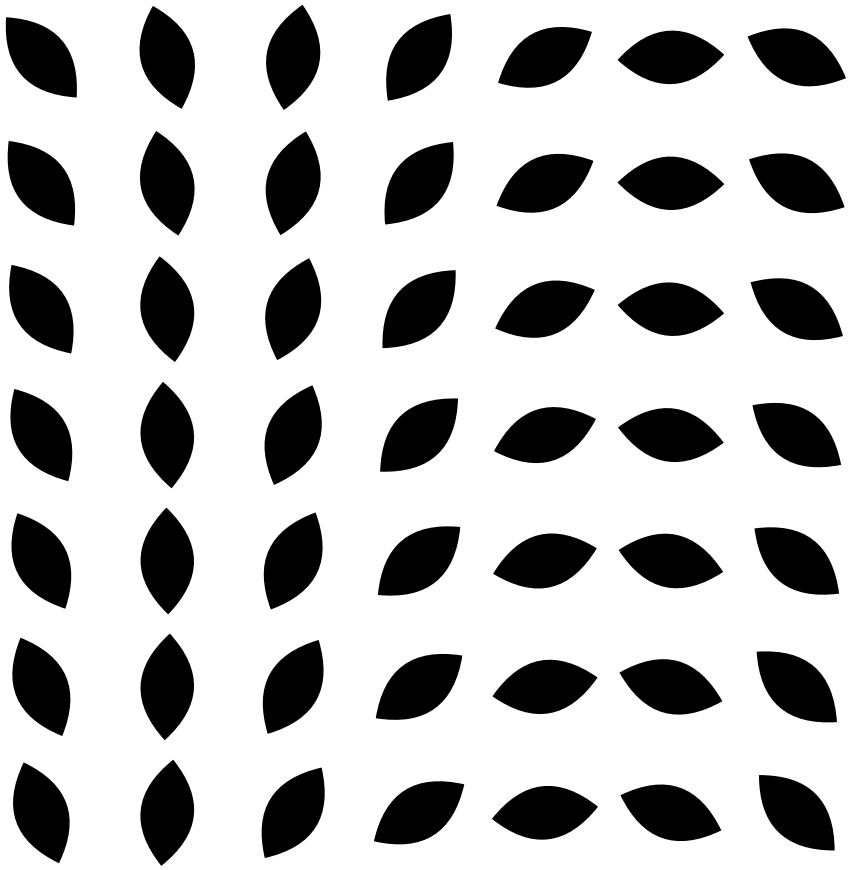


Figure 5.5: A generated *svg* file containing fifty curves.[23]

Render Kit

The `render-kit` feature injects a crate full of pre-written tests which accept a data structure implementing the `Renderer` trait, as well as a proto-type GPU-centric renderer as a baseline reference that already implements the `Renderer` trait. The `Renderer` trait intends to link an arbitrary renderer into a collection of pre-written tests with a small amount of implementation glue. Moreover, the trait and renderer facilitate easy integration for competitors wishing to test against each other quickly; adding a test that operates on a discrete `Renderer` extends all implementations.

The *render-kit* feature also provides an in-house renderer implementing the `Renderer` trait. The provided renderer transmutes vector data through tessellation and provides basic hardware acceleration. The renderer can adjust zoom, pan, wireframe view, and anti-aliasing at runtime. Otherwise, the GPU features include read-only storage buffers purposed to read tessellation data and MSAA. The implementation depends on on *wgpu-rs*[24] as a graphics abstraction, which is an implementation of the *WebGPU*specification in Rust. Moreover, wgpu-rs can be transpiled and chooses a backend such as Vulkan

[23]attribution: By Spencer C. Imbleau, MIT/Apache 2.0
[24]see: https://wgpu.rs/

or Metal deterministically according to the user's hardware. This backend provides an optimized runtime performance but may fall back to a software rendering implementation. In addition, the renderer provided in *render-kit* requests minimal GPU features as a benefit of tessellation, and MSAA. This renderer is engineered as a baseline to record the minimum time necessary for rendering a tessellated model while still being hardware-accelerated with GPU caching.

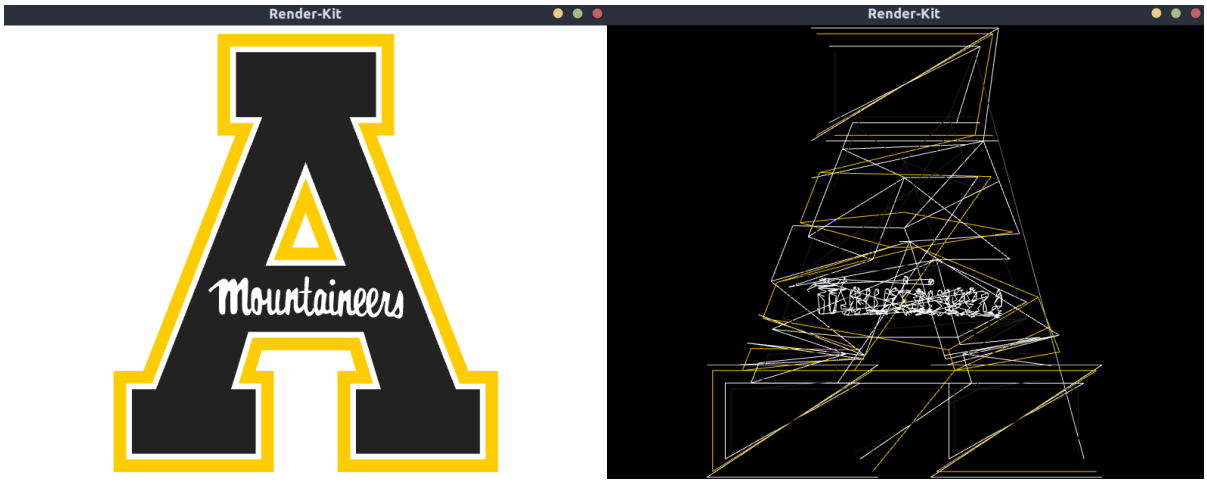


Figure 5.6: Our `render-kit` GPU-centric tessellation renderer showing *svg* rendering (left) and wireframe *svg* rendering (right).[26]

Tessellation Kit

Similar to the *render-kit*, the `tessellation-kit` feature injects a crate full of pre-written tests which accept a data structure implementing the `Tessellator` trait. However, contrary to the *render-kit*, we do not provide an in-house minimalist tessellator. Instead, we expose *Lyon* [19] with glued trait implementation, which dually provides *libtess2* through Lyon as an alternative backend. The intention for the `Tessellator` trait is to link an arbitrary tessellator into a collection of pre-written tests with a small amount of implementation glue, just as is the purpose for the *render-kit's* `Renderer` trait.

[26] attribution: By Spencer C. Imbleau, MIT/Apache 2.0

Chapter 6

Results

As a method of verification of our work, we ask "*Are we building our framework right?*." One way to verify our framework is to use it in a test trial and thereby prove through construction. Therefore, we will offer a test trial to prove our framework's concept, design, and resourcefulness. We will then discuss the results in the following discussion section.

6.1 Test Case

This section describes the test case scenario, poses analysis questions, and describes the benchmarks performed with data used.

6.1.1 The "Web Browser" Case

Web browsers are a deserving application for vector graphic analysis because of how ubiquitously used the browser technology is. Currently, Skia, discussed in Subsection §3.1.1, is the graphics library that is used in modern web browsers. There is significant ongoing development that goes into optimizing web image rendering, given the empirical consequences. While formats such as SVG are generally smaller and faster to travel over the net in web pages[1], a slow rendering speed negates these benefits.

Our test trial will analyze a classic user story of vector graphics: static *svg* content rendering. We will quantify the use of tessellation by rendering static graphics against three renderers which test if a pre-computation model such as tessellation may have usefulness in such a case.

6.1.2 Questions for Analysis

We provide several questions that *vgpu-bench*will utilize to pilot our test case.

- What are some consequences of tessellation?

[1] see: Subsection §2.3.2

- What are some consequences of a pre-computation model?
- Can hardware acceleration improve performance?

6.1.3 Benchmarks

We have coded several benchmarks using the *vgpu-bench*library to answer the above-mentioned questions. All benchmark source code in the "thesis" branches of the *vgpu-bench*repository[2], however results are taken from varying development stages of *vgpu-bench*, so our examples on the "master" branch[3] may provide the forward reader better content examples.

- Path command output for several vector examples
- Tessellation triangle output for several vector examples
- Tessellation timing for several primitives and amounts
- First frame output time of several vector examples
- Continuous frame times of several vector examples
- CPU Utilization of a complex example

6.1.4 Instrumentation

Below we provide justifications for our test data, tessellation backend, and rendering backends.

Hardware

Unless otherwise specified, all GPU benchmarks are recorded with an NVIDIA© *GeForce 1060 3GB*, a middle-grade desktop-class GPU released in 2016.

Test data

We use practical examples with varying complexity for a test set of vector graphics, supplemented with dynamically generated examples. Our dynamic examples have varying amounts of rotated primitives at a constant scale, generated with *svg-generator*[4] to more consistently assess scalability.

In Figure 6.1 we present five images that are practical in complexity and encountered naturally on the web. These images were chosen to represent assets such as logos, icons, and designs. Such images are purposed to represent the organic complexity of generalized vector graphics.

[2]see: https://github.com/simbleau/vgpu-bench
[3]see: https://github.com/simbleau/vgpu-bench/tree/main/examples
[4]see: Subsection §5.2.6

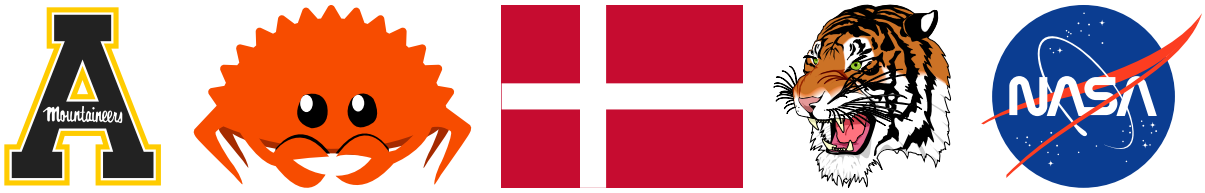


Figure 6.1: Several common vector graphics encountered on the web.[5]

We also acknowledge that the web has apps that may utilize more paths and data than the images above. Fields such as graphic design, geographic information systems, and computer-aided design may require more computer resources. As such, we have cherry-picked one such example for analysis to represent a complex, resource-greedy image, which we present in Figure 6.2.

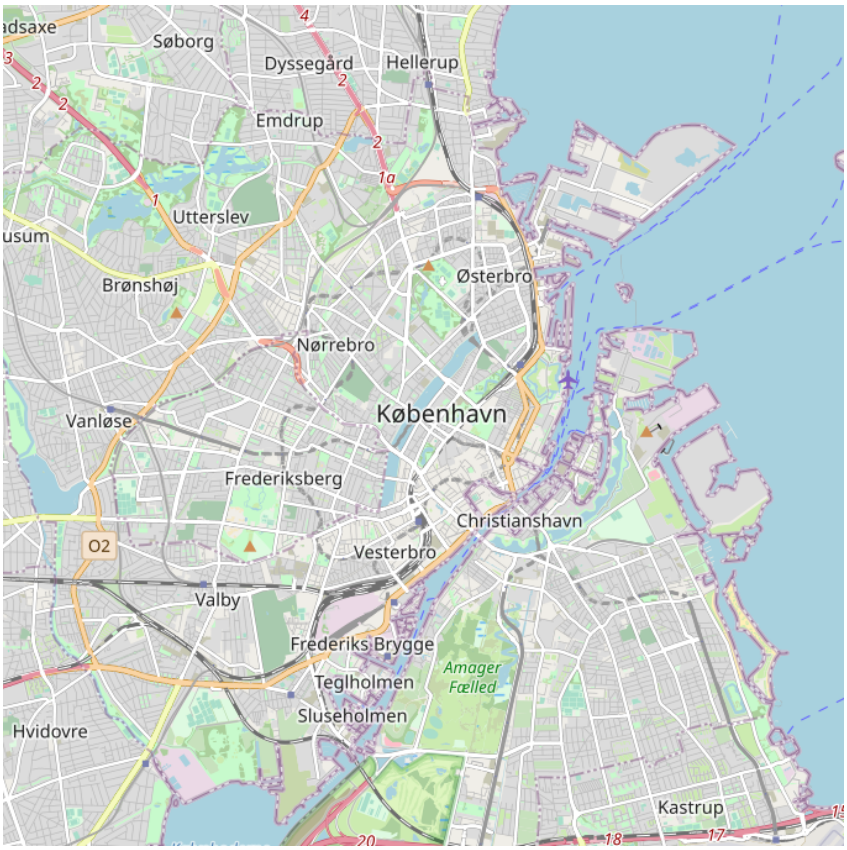


Figure 6.2: A complex vector image, "*København_512.svg*", for benchmarking.[6]

[5]attribution: By Spencer C. Imbleau, MIT/Apache 2.0
[6]attribution: By Spencer C. Imbleau, MIT/Apache 2.0

Tessellation

In all benchmarks involving tessellation, *Lyon*[7] will be used as a backend library, given its academic profile [7], performance [20], and modern application[8]. We will use a tolerance of 0.1 in all aspects where needed, which is a subjectively okay approximation.

Rendering

We will use three backend renderers to render all the provided test data above at the same scale and resolution. The first of which is *resvg*[9], an optimized CPU-based renderer paralleling Skia. Secondly, we use *Pathfinder*[10], a compute-centric sophisticated hardware-accelerated rendering library. Finally we use *render-kit*'s own renderer[11] as a tessellation-based renderer. Additional reasons and justifications may be found below, such that we may draw apt comparisons from varying rendering approaches.

resvg We have chosen *resvg* because it is an abstraction over Skia[12], closely paralleling the optimization. Using a small subset of bindings from CPU-based Skia rendering, donned tiny-skia[13], tiny-skia is about 20-100% less efficient than Skia[14]. Despite using no GPU features, *resvg* is still one of the fastest CPU-based renderers for *svg* images. We will also ignore any caching potential and produce every image as a dry run to provide optics on how beneficial caching may be.

Render-Kit As a feature of *vgpu-bench*, the "*render-kit*" feature provides a minimal tessellation-based GPU renderer described in Subsection §5.2.6. Since *Pathfinder* uses an implementation of *WebGPU*[15], it is portable and may compile to Web Assembly (WASM)[16] as a hardware-accelerated web target, making this a practical candidate runtime for web browsers. WebGPU is developed by the *W3C GPU for the Web Community Group* with engineers from Apple, Mozilla, Microsoft, Google, and others [36], such to extend hardware acceleration to the respective company's web browsers. *Pathfinder* also uses minimal GPU features, namely a storage buffer for caching and multi-sample anti-aliasing.

Pathfinder Our last renderer for instrumentation is Pathfinder, with examination in Subsection §3.1.2. *Pathfinder* was engineered for work in Servo, an embedded web engine project. *Pathfinder* offers an analytic approach to GPU-centric rendering, defended

[7] see: https://github.com/nical/lyon
[8] see: Subsection §3.1.5
[9] see: https://github.com/RazrFalcon/resvg
[10] see: https://github.com/servo/pathfinder
[11] see: Subsection §5.2.6
[12] see: https://skia.org
[13] see: https://github.com/RazrFalcon/tiny-skia
[14] see: https://razrfalcon.github.io/tiny-skia/x86_64.html
[15] see: https://www.w3.org/TR/webgpu/
[16] see: https://webassembly.org/

academically and publicly [28]. Pathfinder, as a parallel model[17], would provide a stark contrast to a tessellation-based model such as Render-Kit, or a heavily optimized CPU-based approach such as *resvg*.

6.2 Data Collection

Data collected from the benchmarks in Subsection §6.1.3 are given here. A discussion of these results will be found our discussion, Chapter 7.

6.2.1 Profiling

Results in this section are designed to profile several *svg* images to classify images into a frame of reference for complexity.

SVG complexity

In Figure 6.3 below, we plot data corresponding to the amount of path commands in each respective *svg* image. Files are located on the x-axis. The volume of path commands in the respective file is plotted on the y-axis.

[17] see: Subsection §4.2.2

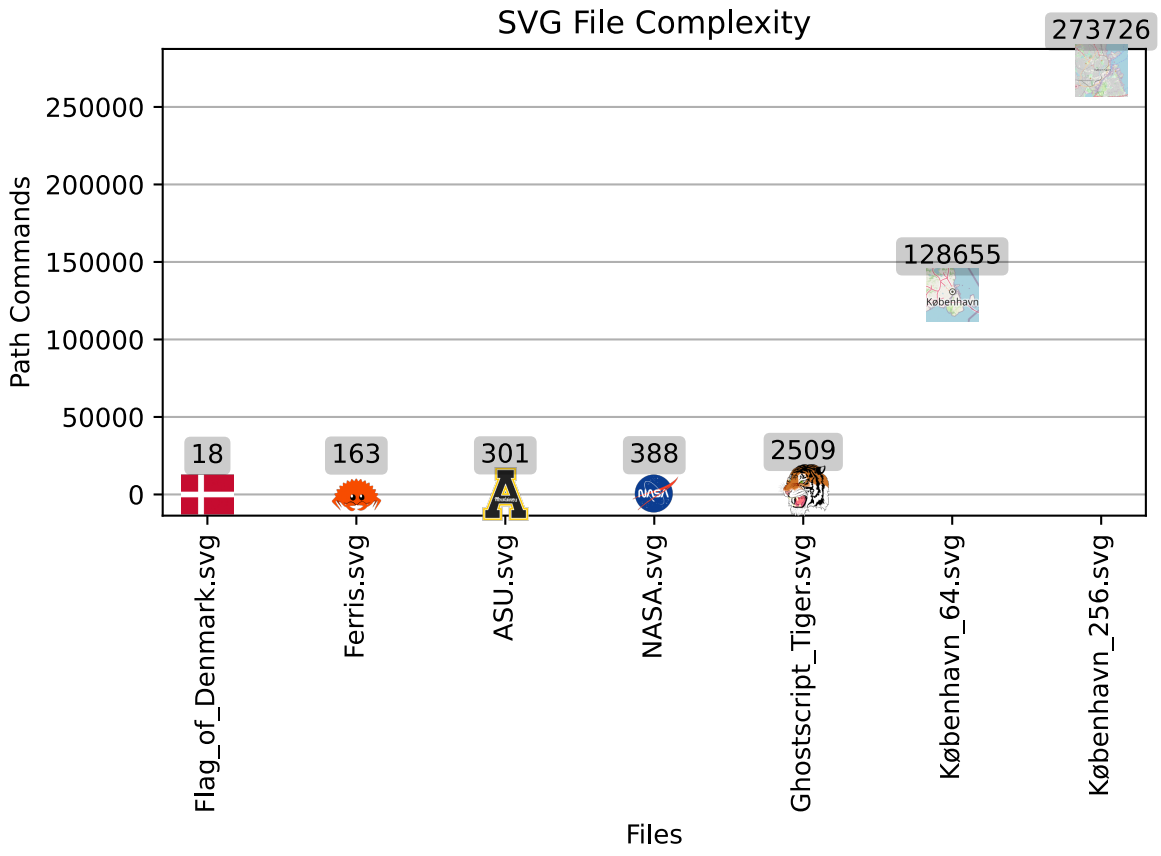


Figure 6.3: Total path commands in various *svg* examples.[18]

SVG tessellation complexity

In Figure 6.4 below, we plot data corresponding to the amount of output triangles for each respective *svg* image after tessellation. Files are located on the x-axis. The volume of triangles produced in the respective file is plotted on the y-axis.

[18]attribution: By Spencer C. Imbleau, MIT/Apache 2.0

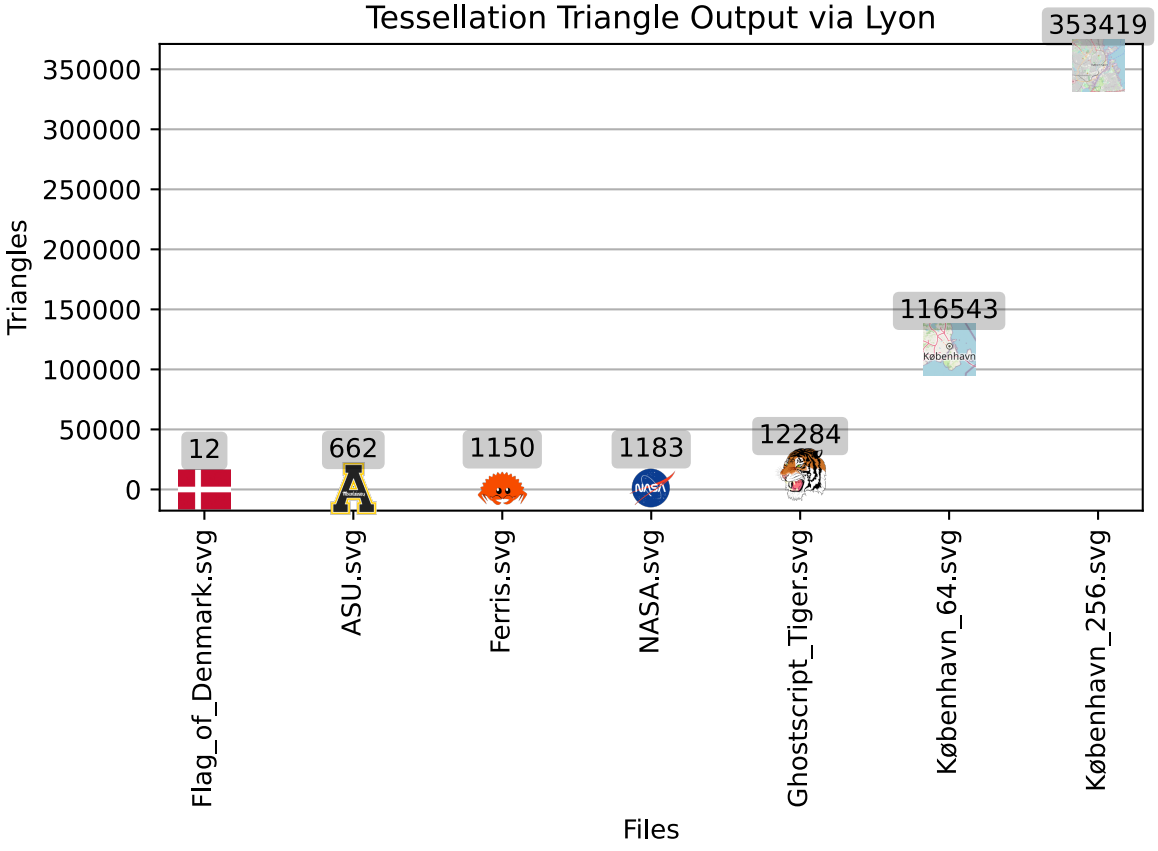


Figure 6.4: Total tessellated triangles in various *svg* examples.[19]

6.2.2 Tessellation

Results in this section are designed to collect time trials relating to tessellation to understand more about the consequences of tessellation using our instrumentation tessellator, *Lyon*.

Low primitive count

In Figure 6.5, Figure 6.6, and Figure 6.7, we plot the amount of time performed both in initialization and tessellation for low amounts of traditional vector primitives. The amount of primitives tessellated is located on the x-axis. The total time expense of both initialization and tessellation is recorded on the y-axis.

[19]attribution: By Spencer C. Imbleau, MIT/Apache 2.0

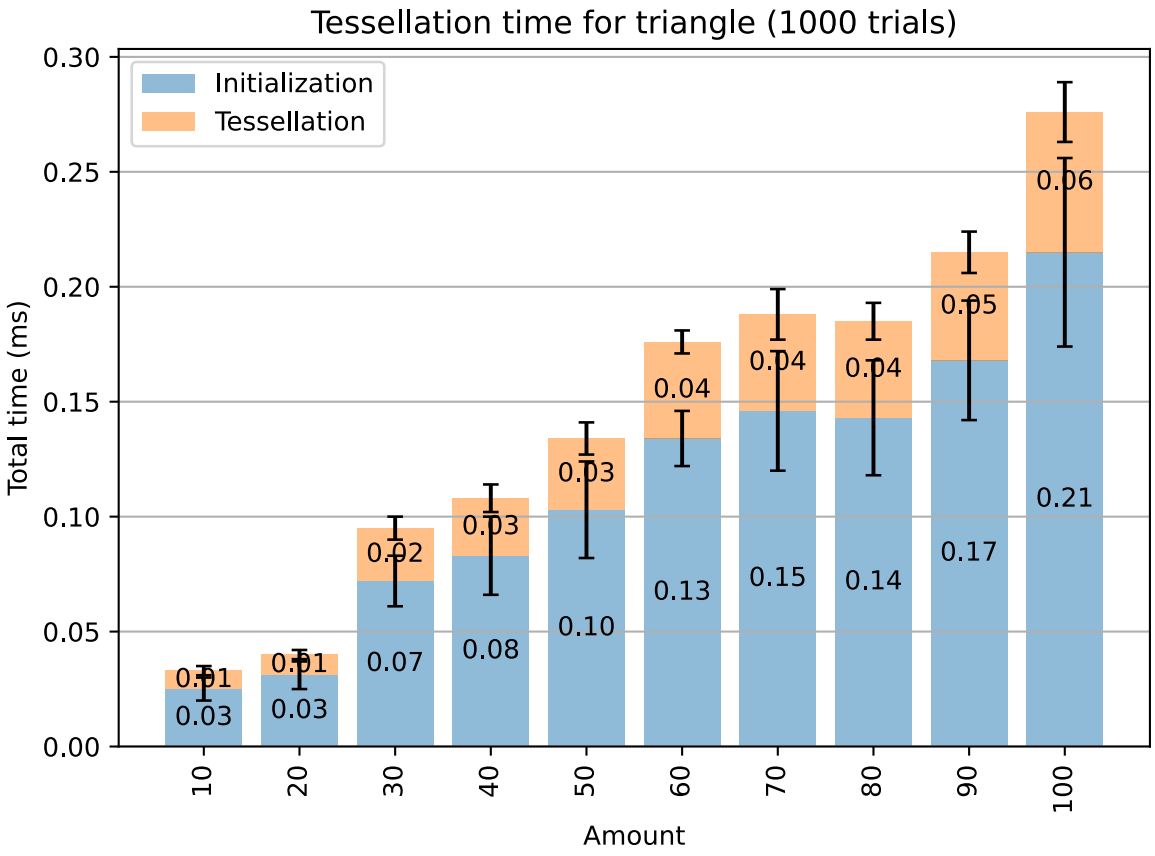


Figure 6.5: Loading and tessellation time for low amounts of *svg* triangle primitives.[20]

[20]attribution: By Spencer C. Imbleau, MIT/Apache 2.0

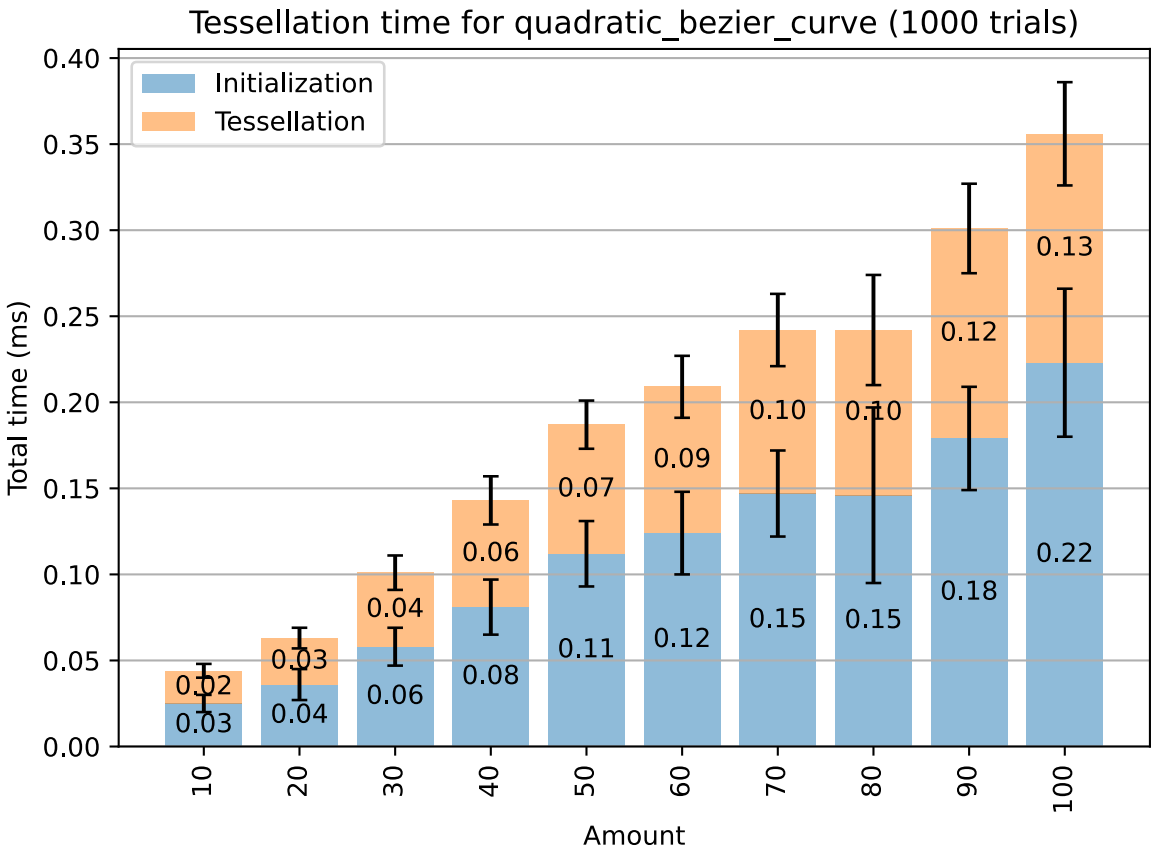


Figure 6.6: Loading and tessellation time for low amounts of *svg* quadratic Bézier curve primitives.[22]

[22]attribution: By Spencer C. Imbleau, MIT/Apache 2.0

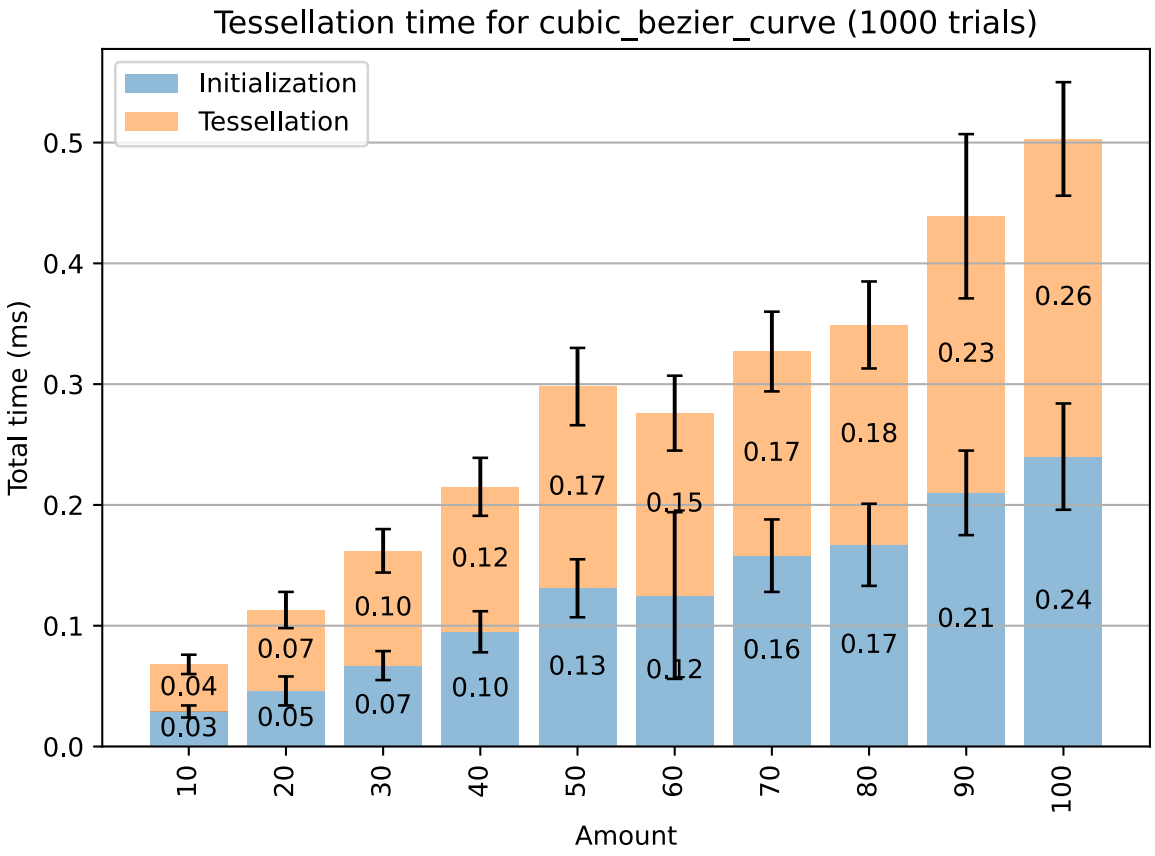


Figure 6.7: Loading and tessellation time for low amounts of *svg* cubic Bézier curve primitives.[24]

High primitive count

In Figure 6.8, Figure 6.9, and Figure 6.10, we plot the amount of time performed both in initialization and tessellation for high amounts of traditional vector primitives. The amount of primitives tessellated is located on the x-axis. The total time expense of both initialization and tessellation is recorded on the y-axis.

[24]attribution: By Spencer C. Imbleau, MIT/Apache 2.0

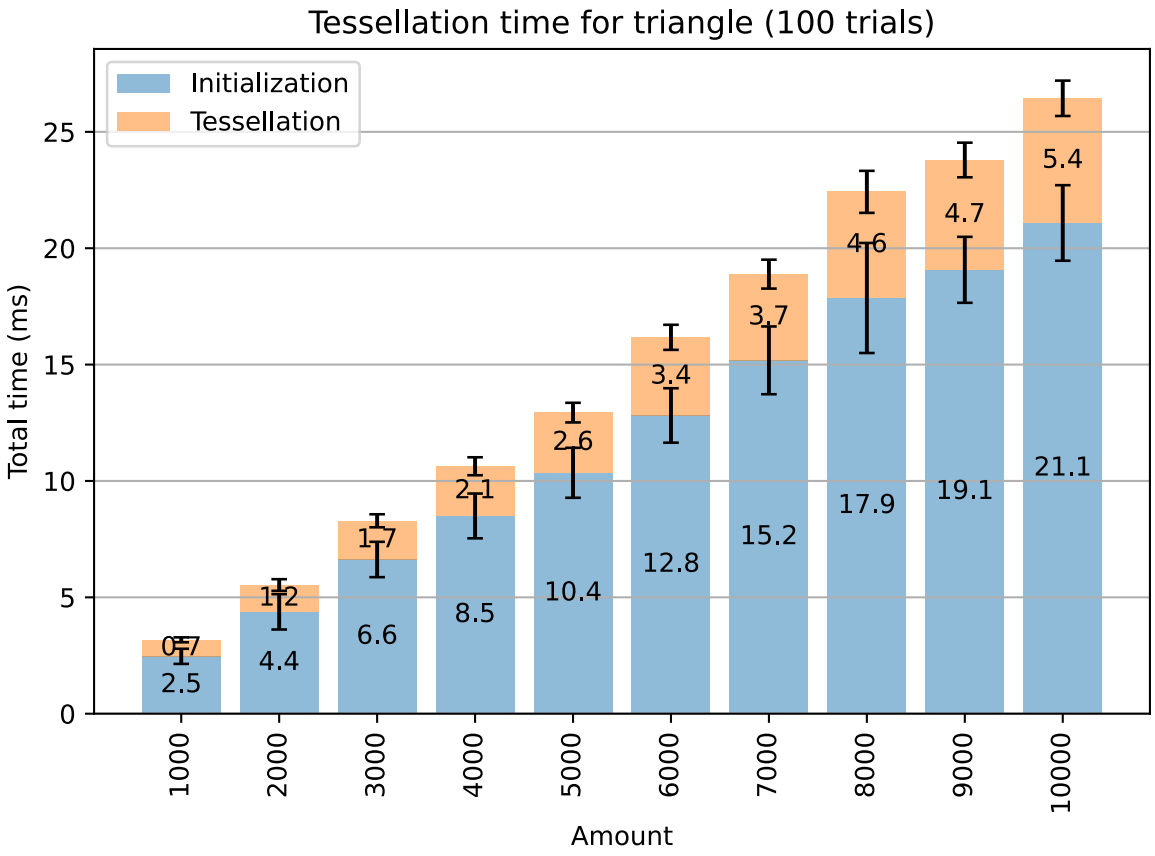


Figure 6.8: Loading and tessellation time for high amounts of *svg* triangle primitives.[25]

[25]attribution: By Spencer C. Imbleau, MIT/Apache 2.0

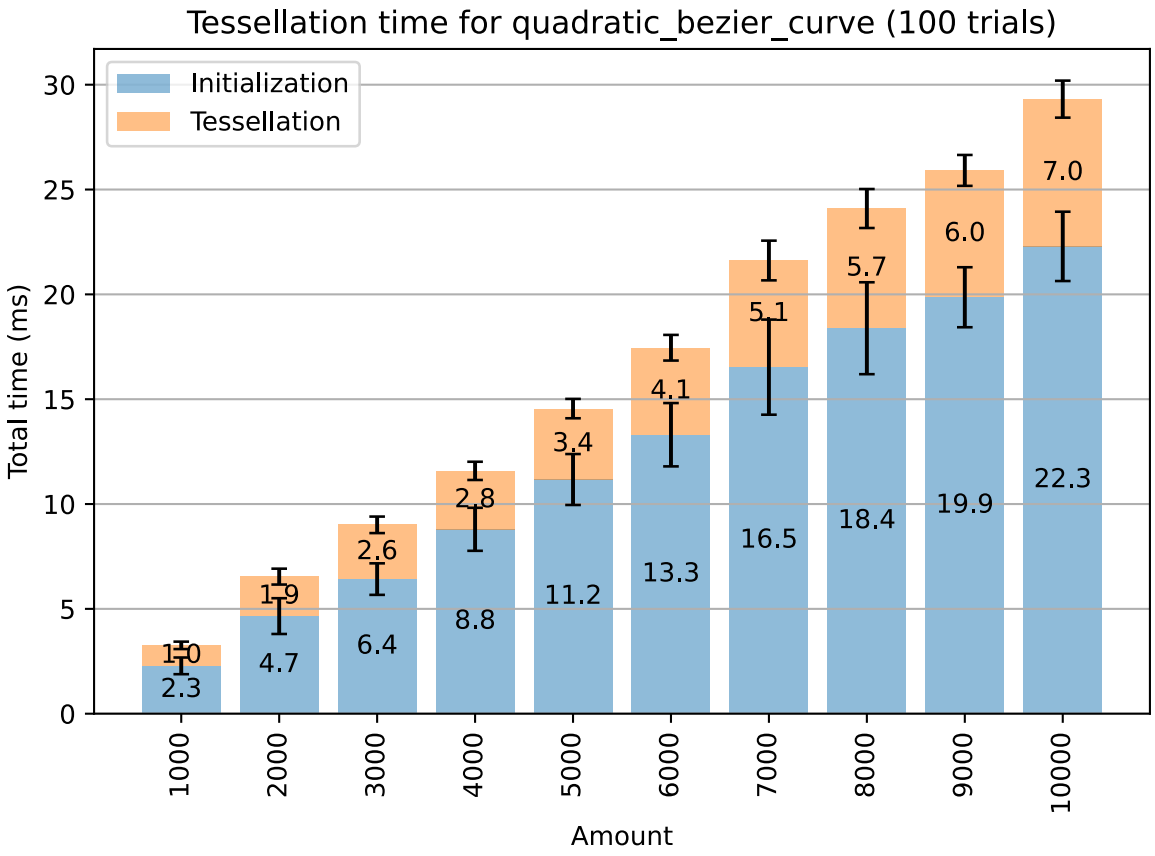


Figure 6.9: Loading and tessellation time for high amounts of *svg* quadratic Bézier curve primitives.[27]

[27] attribution: By Spencer C. Imbleau, MIT/Apache 2.0

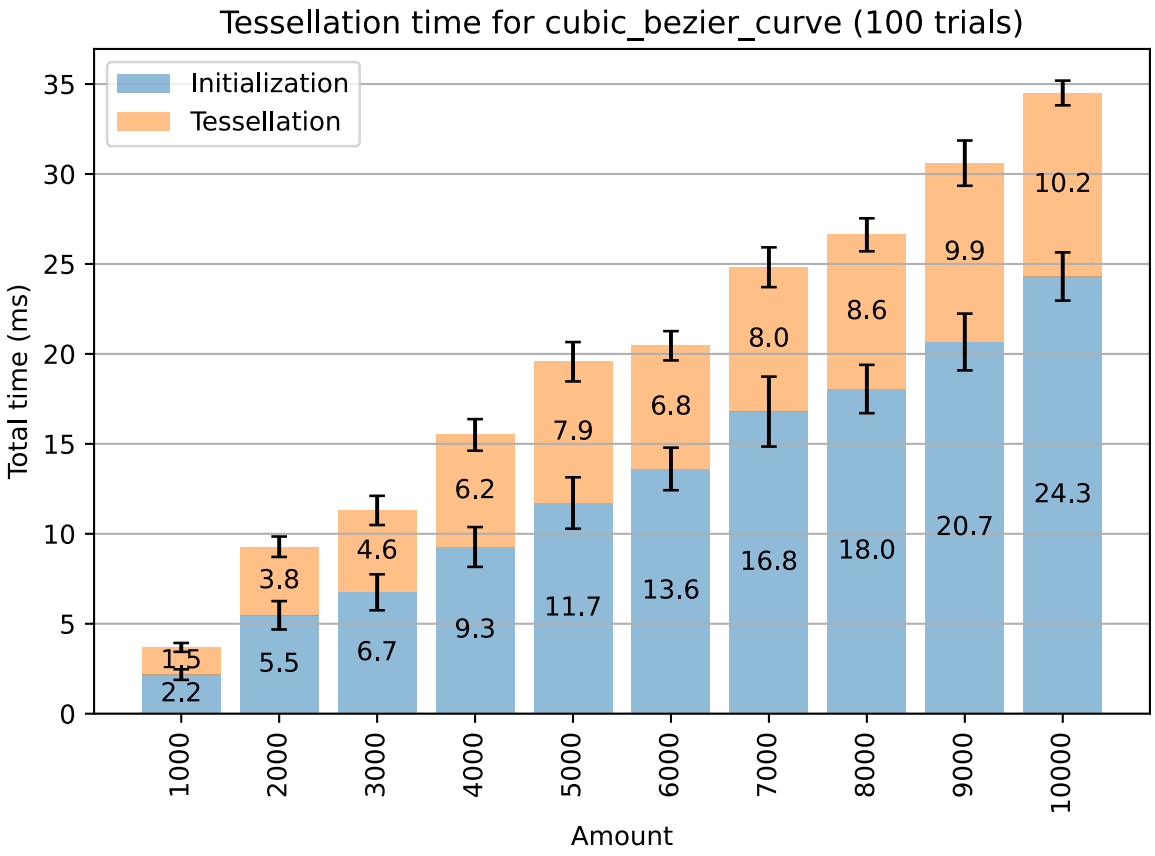


Figure 6.10: Loading and tessellation time for high amounts of *svg* cubic Bézier curve primitives.[29]

6.2.3 Rendering Trials

Results in this section are designed to collect time trials relating to rendering to understand more about the the performance of the differing renderers we use for instrumentation in Subsection §6.1.4. We do so by benchmarking dry frametimes, which are frametimes without any former computation, and wet frametimes, which may be accelerated through caching or initial processing.

Dry frametime for test data

In Table 6.1, Table 6.2, and Table 6.3, we record the amount of time required to render each *svg* image file one time as a dry run without any previous caching. These statistics include any required setup such as tessellation.

[29] attribution: By Spencer C. Imbleau, MIT/Apache 2.0

Dry Frametime, Render-Kit	
File	Frametime
ASU.svg	111.122763ms
Ferris.svg	110.129153ms
Flag_of_Denmark.svg	113.356655ms
Ghostscript_Tiger.svg	119.625961ms
København_512.svg	813.996279ms
NASA.svg	110.726647ms

Table 6.1: Dry frametime rendering for test data images with *Pathfinder*.

Dry Frametime, Resvg	
File	Frametime
ASU.svg	0.845686ms
Ferris.svg	2.819766ms
Flag_of_Denmark.svg	0.149882ms
Ghostscript_Tiger.svg	5.847163ms
København_512.svg	883.884497ms
NASA.svg	6.051327ms

Table 6.2: Dry frametime rendering for test data images with *resvg*.

Dry Frametime, Pathfinder	
File	Frametime
ASU.svg	2.328747ms
Ferris.svg	2.279044ms
Flag_of_Denmark.svg	2.116318ms
Ghostscript_Tiger.svg	3.38733ms
København_512.svg	80.38817ms
NASA.svg	5.456171ms

Table 6.3: Dry frametime rendering for test data images with *Pathfinder*.

Wet frametimes for test data

In Figure 6.11, Figure 6.12, and Figure 6.13, we plot the frametimes of our test data. Frames are measured by continuously rendering *after* setup steps such as tessellation, staging, or initialization. The frame rendered is recorded on the x-axis. The total time expense of rendering is recorded on the y-axis.

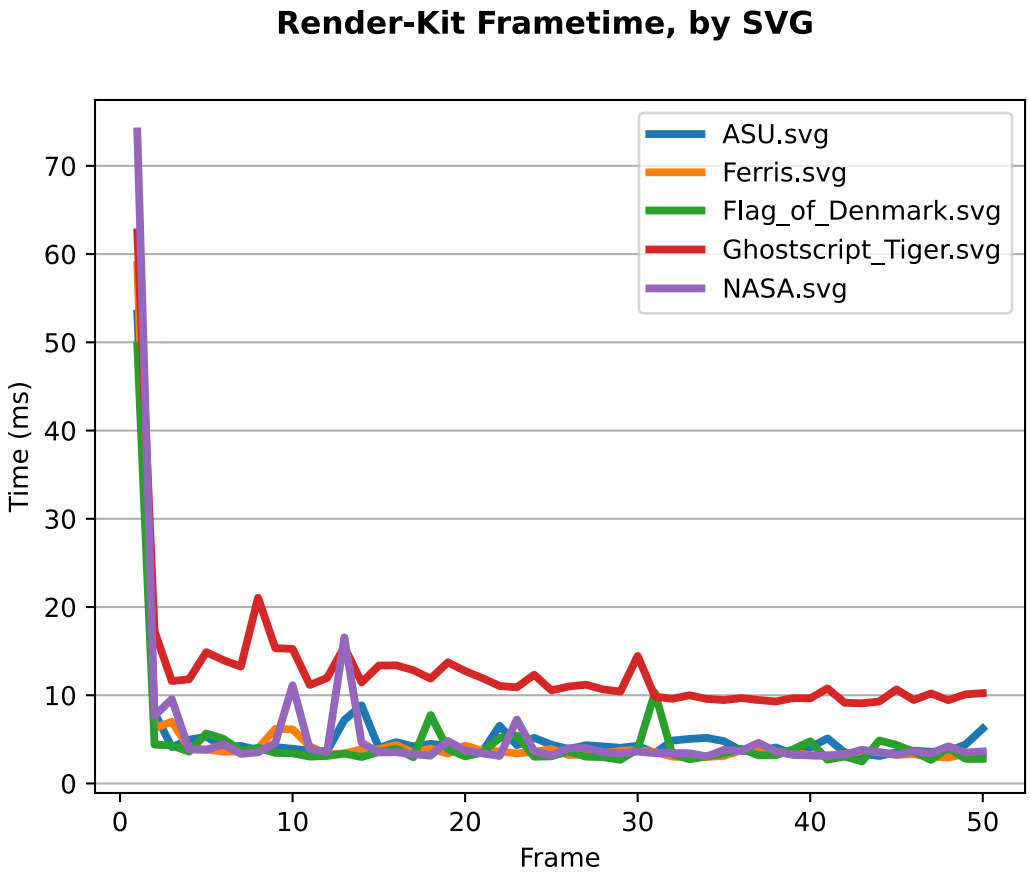


Figure 6.11: Frametime stability of all test data over 50 frames, rendered by *Pathfinder*.[31]

[31] attribution: By Spencer C. Imbleau, MIT/Apache 2.0

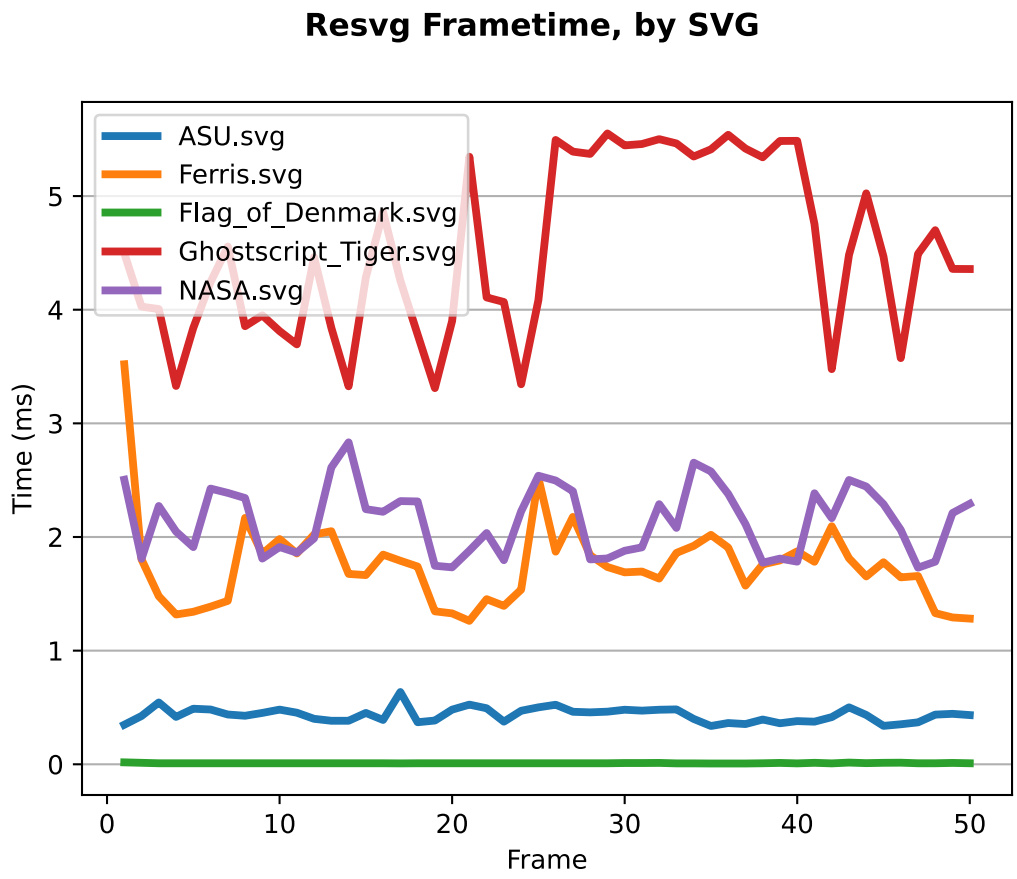


Figure 6.12: Frametime stability of all test data over 50 frames, rendered by *resvg*.[32]

[32]attribution: By Spencer C. Imbleau, MIT/Apache 2.0

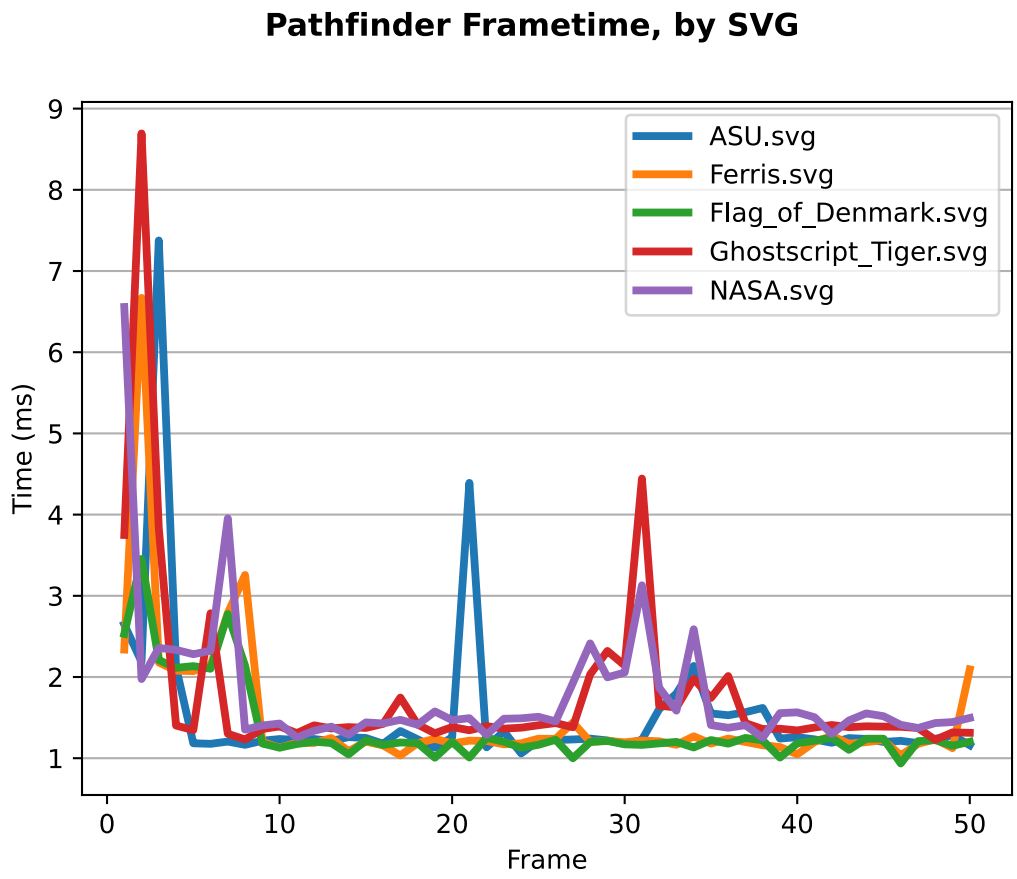
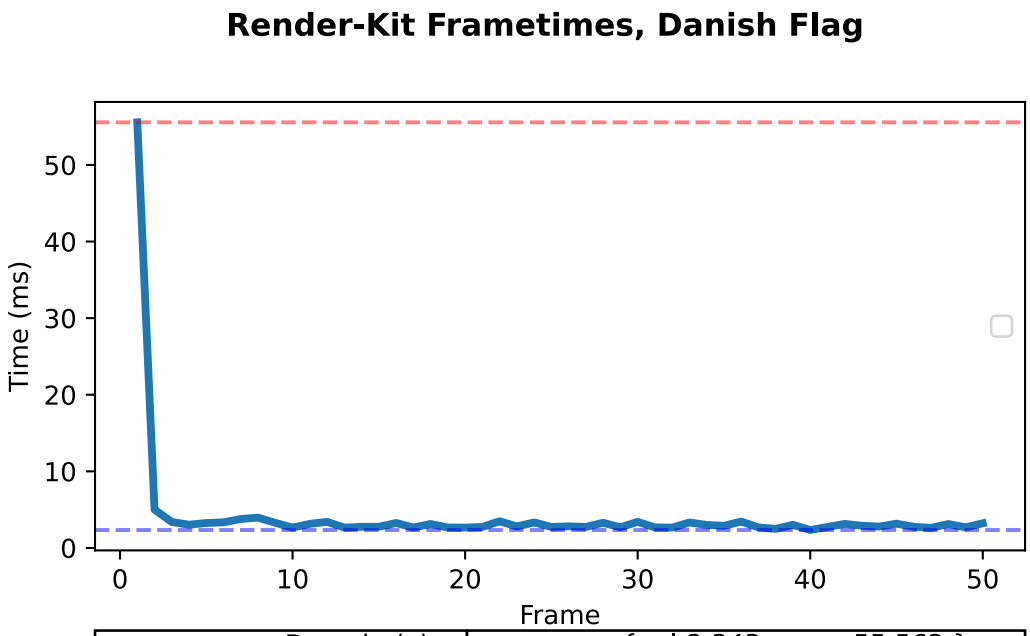


Figure 6.13: Frametime stability of all test data over 50 frames, rendered by *Pathfinder*.[34]

Wet frametimes for a simple image

In Figure 6.14, Figure 6.15, and Figure 6.16, we plot the frametimes of our most simple item of test data, "*Flag_of_Denmark.svg*." Frames are measured by continuously rendering *after* setup steps such as tessellation, staging, or initialization. The frame rendered is recorded on the x-axis. The total time expense of rendering is recorded on the y-axis.

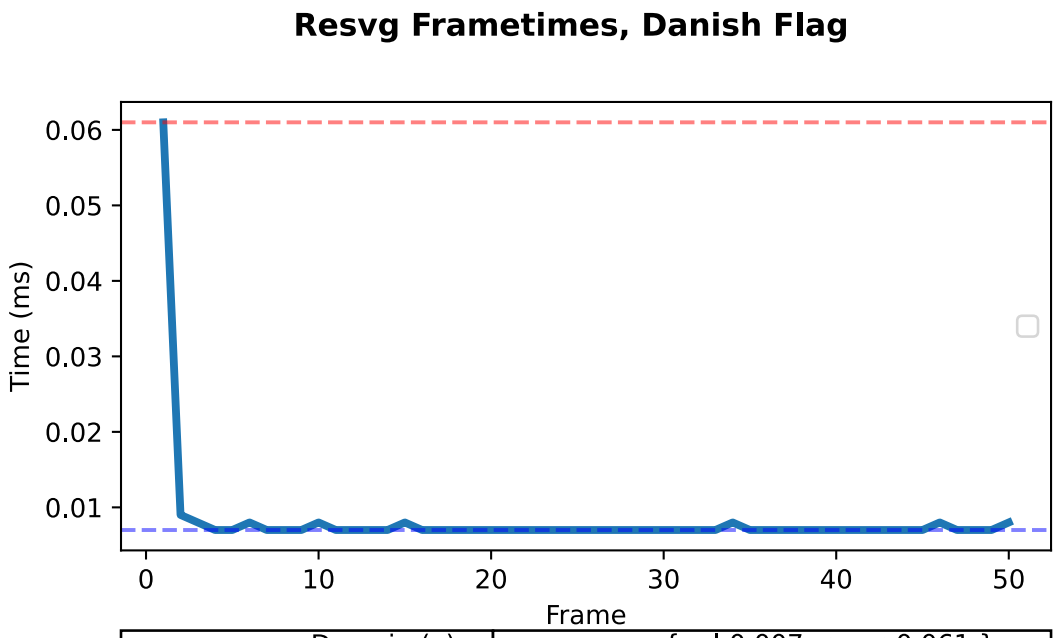
[34]attribution: By Spencer C. Imbleau, MIT/Apache 2.0



Domain (y)	{ y | 2.343 ≤ y ≤ 55.562 }
Median ($\tilde{y}$)	2.951
Mean ($\bar{y}$)	4.0838
Std. Deviation (σ)	7.3672

Figure 6.14: Frametime stability of a simple *svg* "*Flag_of_Denmark.svg*" over 50 frames, rendered by *Pathfinder*.[36]

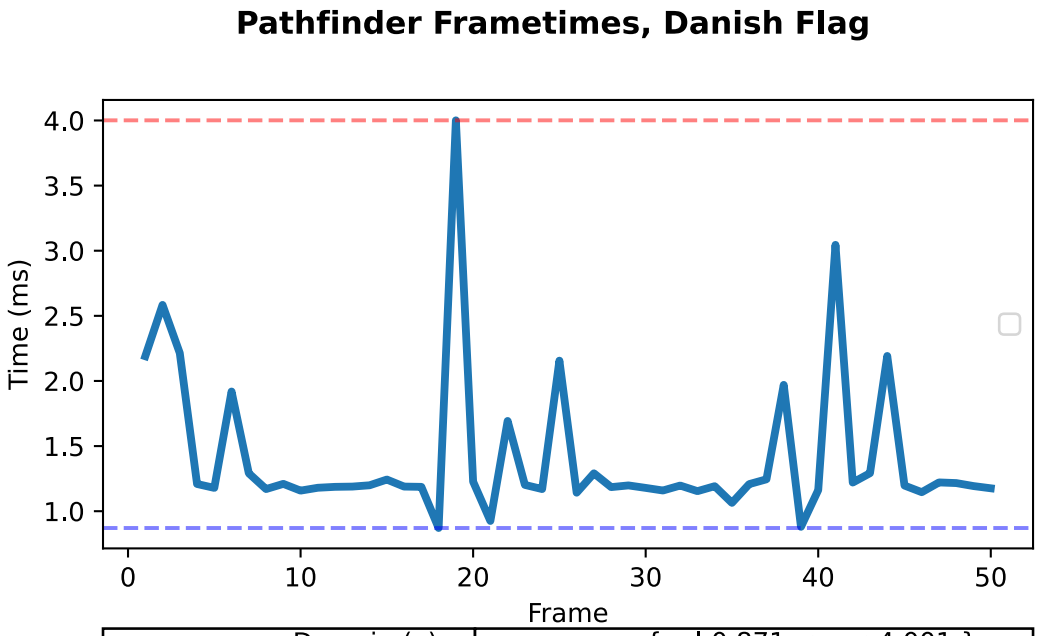
[36]attribution: By Spencer C. Imbleau, MIT/Apache 2.0



Domain (y)	{ y | 0.007 ≤ y ≤ 0.061 }
Median ($\tilde{y}$)	0.007
Mean ($\bar{y}$)	0.0083
Std. Deviation (σ)	0.0075

Figure 6.15: Frametime stability of a simple *svg* "*Flag_of_Denmark.svg*" over 50 frames, rendered by *resvg*.[38]

[38] attribution: By Spencer C. Imbleau, MIT/Apache 2.0



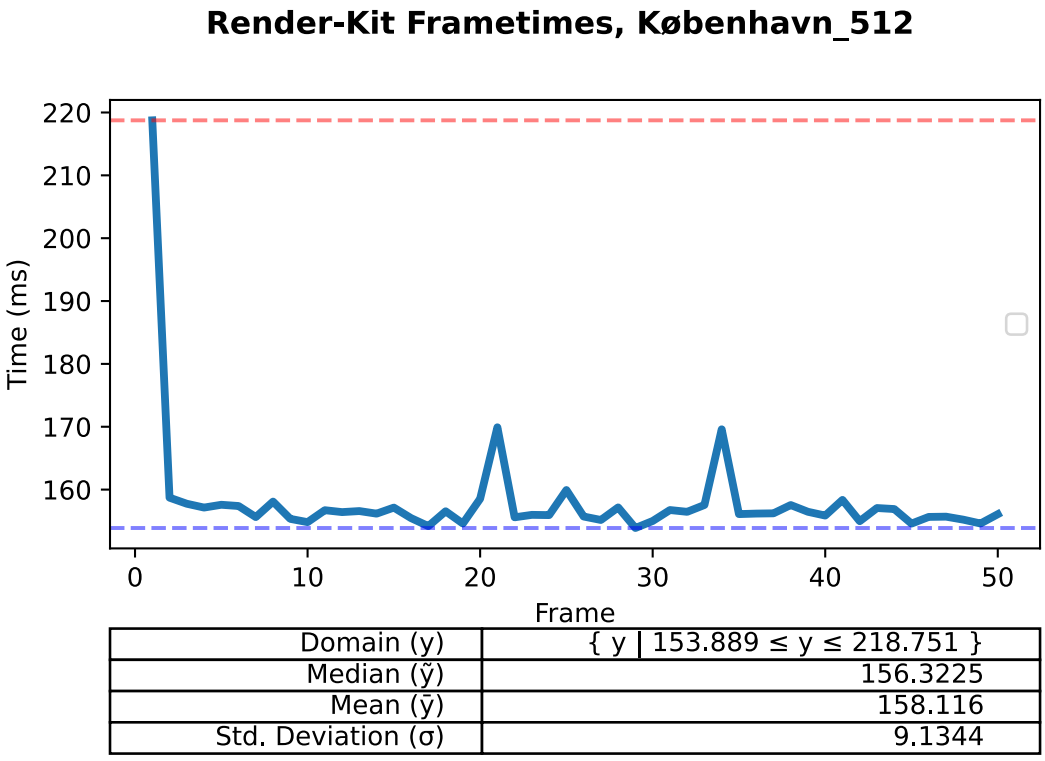
Domain (y)	{ y | 0.871 ≤ y ≤ 4.001 }
Median (ỹ)	1.1975
Mean (ȳ)	1.4173
Std. Deviation (σ)	0.573

Figure 6.16: Frametime stability of a simple *svg* "*Flag_of_Denmark.svg*" over 50 frames, rendered by *Pathfinder*.[40]

Wet frametimes for a complex image

In Figure 6.17, Figure 6.18, and Figure 6.19, we plot the frametimes of our most complex item of test data, "*København_512.svg*." Frames are measured by continuously rendering *after* setup steps such as tessellation, staging, or initialization. The frame rendered is recorded on the x-axis. The total time expense of rendering is recorded on the y-axis.

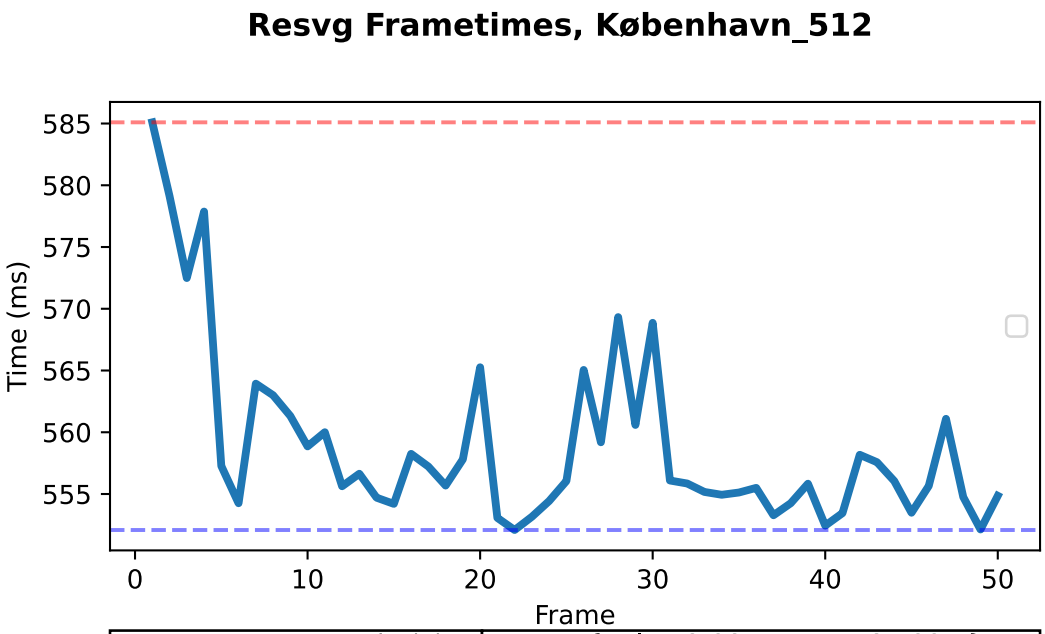
[40]attribution: By Spencer C. Imbleau, MIT/Apache 2.0



Domain (y)	{ y | 153.889 ≤ y ≤ 218.751 }
Median (ỹ)	156.3225
Mean (ȳ)	158.116
Std. Deviation (σ)	9.1344

Figure 6.17: Frametime stability of a complex *svg* "*København_512.svg*" over 50 frames, rendered by *Render-Kit*.[42]

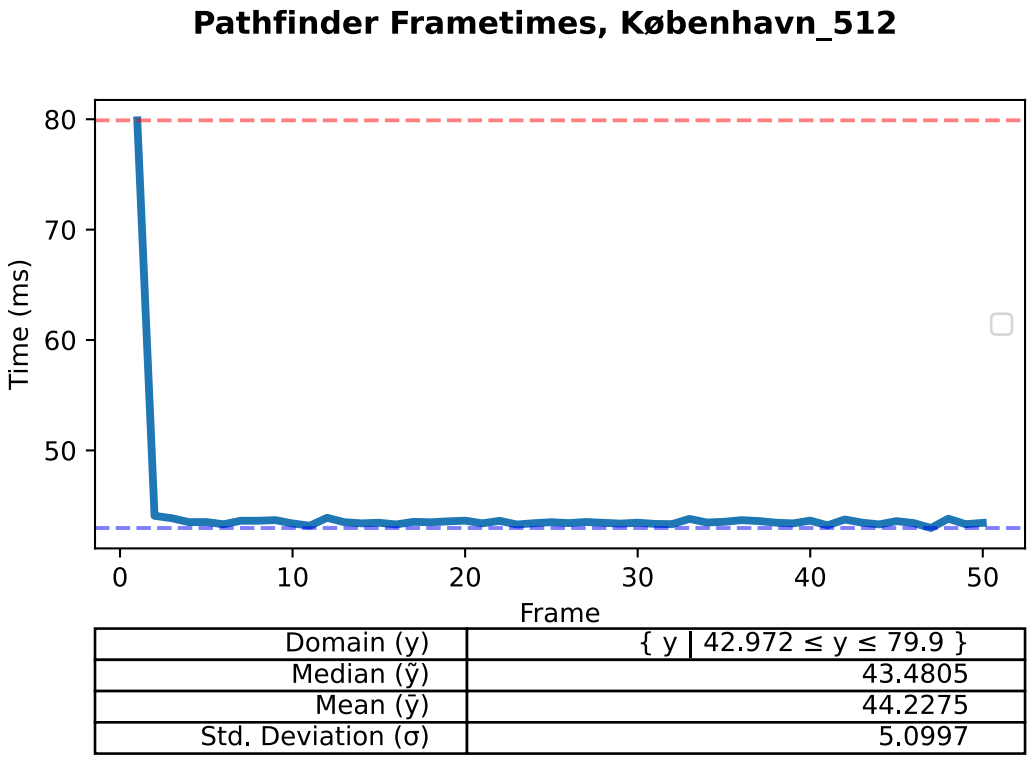
[42]attribution: By Spencer C. Imbleau, MIT/Apache 2.0



Domain (y)	{ y | 552.085 ≤ y ≤ 585.095 }
Median (ỹ)	556.0515
Mean (ȳ)	558.9224
Std. Deviation (σ)	7.1488

Figure 6.18: Frametime stability of a complex *svg* "*København_512.svg*" over 50 frames, rendered by *resvg*.[44]

[44]attribution: By Spencer C. Imbleau, MIT/Apache 2.0



Domain (y)	{ y | 42.972 ≤ y ≤ 79.9 }
Median (ỹ)	43.4805
Mean (ȳ)	44.2275
Std. Deviation (σ)	5.0997

Figure 6.19: Frametime stability of a complex *svg* "*København_512.svg*" over 50 frames, rendered by *Pathfinder*.[46]

6.2.4 Monitoring

Results in this section are designed to monitor consequences incurred by a file with a heavy resource footprint.

In Table 6.4, Table 6.5, and Table 6.6, we record the cpu utilization for ten seconds while rendering our most complex item of test data, "*København_512.svg*." The process responsible for rendering is initiated by a user.

[46] attribution: By Spencer C. Imbleau, MIT/Apache 2.0

CPU Utilization Rendering København_512.svg, Render-Kit					
Second	Idle	Interrupt	Nice	System	User
1	0.467693	0.0	0.0	0.037544124	0.49476284
2	0.39563155	0.0	0.0	0.067370936	0.5369975
3	0.495704	0.0	0.0	0.021988489	0.48169076
4	0.53159183	0.0	0.0	0.02529972	0.44310844
5	0.5278983	0.0	0.0	0.02840928	0.44369245
6	0.46954525	0.0	0.0	0.045535572	0.4849192
7	0.45059177	0.0	0.0	0.0710343	0.47837391
8	0.50600004	0.0	0.0	0.09704739	0.39521644
9	0.5059884	0.0	0.0	0.051762626	0.41293645
10	0.46378028	0.0	0.0	0.034738675	0.50148106

Table 6.4: CPU Utilization over ten seconds of rendering a complex *svg* "*København_512.svg*" with *Render-Kit*.

CPU Utilization Rendering København_512.svg, Resvg					
Second	Idle	Interrupt	Nice	System	User
1	0.84086835	0.0	0.0	0.023640312	0.1219778
2	0.85577947	0.0	0.0	0.0039034719	0.13270414
3	0.8671819	0.0	0.0	0.0009431307	0.12520833
4	0.85061646	0.0	0.0	0.020844596	0.12853892
5	0.78110397	0.0	0.0	0.054991208	0.16390486
6	0.85544133	0.0	0.0	0.014675165	0.1298835
7	0.84812915	0.0	0.0	0.014469341	0.13740154
8	0.80173135	0.0	0.0	0.013854485	0.13774753
9	0.8667649	0.0	0.0	0.04184088	0.091394216
10	0.85680187	0.0	0.0	0.0139503265	0.12924781

Table 6.5: CPU Utilization over ten seconds of rendering a complex *svg* "*København_512.svg*" with *resvg*.

CPU Utilization Rendering København_512.svg, Pathfinder					
Second	Idle	Interrupt	Nice	System	User
1	0.92592514	0.0	0.0	0.013025147	0.06104972
2	0.7673017	0.0	0.0	0.16913769	0.06356061
3	0.97890556	0.0	0.0	0.00616342	0.014931006
4	0.97474915	0.0	0.0	0.007291886	0.017959006
5	0.8870437	0.0	0.0	0.0076014614	0.052989975
6	0.85821474	0.0	0.0	0.011255654	0.063322365
7	0.91479445	0.0	0.0	0.005	0.08020559
8	0.9161637	0.0	0.0	0.017694628	0.06614172
9	0.9035666	0.0	0.0	0.025771506	0.070661925
10	0.93193793	0.0	0.0	0.00093627756	0.06712583

Table 6.6: CPU Utilization over ten seconds of rendering a complex *svg* "*København_512.svg*" with *Pathfinder*.

6.3 Test Case Analysis

This section interprets the several benchmarks and data collected in the results above. Precisely, we will frame findings in the context of our analysis questions in the test case.

6.3.1 Consequences of Tessellation

In our questions for analysis, we asked "What are some consequences of tessellation?." Below we will explain our findings for this query.

Primitive count

Tessellation does not always output more complexity than the original vector image. In the example of "*Flag_of_Denmark.svg*" in our profiling results[47] we notice the original file contains 18 path commands, and the tessellation outputs 12 triangles. This intuitively makes sense, as the flag may be represented with two triangles for each rectangle, with the flag being able to be described as six rectangles.

Tolerance

One may point out that "*Ferris.svg*" has far less path commands than "*ASU.svg*" in their original *svg* files, but produces far more triangles during tessellation. Upon further investigation, this is because of curve flattening and a tolerance, described more in Section §2.5. "*ASU.svg*" has many more paths due to the text "*Mountaineers*" over the logo, which are subtracted away during curve flattening simplification, a function of tolerance.

[47] see: Subsection §6.2.1

Tessellation costs

Given an svg with varying amounts of primitives, tessellation costs a lot. If we ignore all initialization cost required to de-serialize an *svg*, which is usually higher than tessellation cost itself according to our results, tessellation is still expensive. Performing a simple linear regression on time residuals gives fairly precise predictions of tessellation time cost as volume of primitives increases. These results suggest that a few thousand primitives will start to incur several milliseconds of cost regardless of type.

Triangle tessellation cost

$$f(x) = 0.00044ms * x - 0.1022ms \tag{6.1}$$

where x is the amount of triangle primitives to tessellate.
Correlation: $r = 0.996$
R-squared: $r^2 = 0.993$

Quadratic Bézier curve tessellation cost

$$f(x) = 0.00035ms * x + 1.0694ms \tag{6.2}$$

where x is the amount of quadratic Bézier curve primitives to tessellate.
Correlation: $r = 0.998$
R-squared: $r^2 = 0.997$

Cubic Bézier curve tessellation cost

$$f(x) = 0.00039ms * x + 3.9174ms \tag{6.3}$$

where x is the amount of cubic Bézier curve primitives to tessellate.
Correlation: $r = 0.968$
R-squared: $r^2 = 0.937$

6.3.2 Consequences of Pre-Computation

In our questions for analysis, we asked "What are the consequences of a pre-computation model?." Below we will explain our findings for this query.

Cache-friendliness

Pre-computation is proven useful in situations where vectors do not have to be deformed or rescaled, such as in the web browser case. Furthermore, Pre-computation may use caching to reduce computation in future rendering iterations. For example, in our benchmarks recording continuous "dry" and "wet" frametimes, we recorded a single frame turnaround and continuous frametimes for three renderers. Since we did not use any caching features with *resvg*, *resvg*'s dry frametimes are approximately equal to its wet frametimes. On the contrary, render-kit's only GPU feature is a storage buffer binding to tessellation data for computation re-use. This removed the need to re-tessellate per frame, improving the frametime of subsequent frames by a magnitude of 10.

Interactivity

While pre-computation may help to reduce recalculation and improve performance through recycling, the model is anti-thetic to interactivity such as animation or live deformation such as scaling. A reasonable goal is to render an image within $16ms$, the reciprocal of $60fps$ (frames per second), a standard convention for interactivity. In our case we only consider static content, so this is not such an issue, but it should be noted that our test case is both narrow and naive for brevity.

6.3.3 Hardware-Acceleration

In our questions for analysis, we asked "How can hardware-acceleration improve performance?." Below we will explain our findings for this query.

GPU latency

Hardware acceleration always brings latency when interacting with the GPU, so in some cases, hardware acceleration is not the magic solution some believe. In elementary vector images with low complexity, *resvg* beat both *Pathfinder* and *Render-Kit* which both leverage the GPU. In the case of *Render-Kit*, there is an nonnegotiable $110ms$ of submit latency in buffer allocation and transfer required for an initial frame. Viewing the NVTX annotations while running *Pathfinder* provides us with the details to prove this. We have annotated the first frame as "Strange Behavior" in NVIDIA©*Nsight Systems*[48] to show this behavior in Figure 6.20. The metric samples show a build-up to a GPU queue submit, thereby triggering a compute dispatch to commit DRAM for reading subsequent frames.

[48] see: https://developer.nvidia.com/nsight-systems

Figure 6.20: Initial GPU latency of *Render-Kit*, annotated by *vgpu-bench*.[49]

Since *Pathfinder* interprets vector graphics mainly through shaders, there is minor caching or pre-computation, and less ceremony is required for an initial frame. However, even with no caching, there still exists an unnegotiable $2ms$ of GPU latency on our test hardware.

Compute-centricity

Pathfinder mollifies historical pipeline rigidity by utilizing compute shaders for parallel winding number computation. This pipeline results in efficient rendering on the GPU in almost all cases, except for elementary ones. Against a traditional raster pipeline, *Render-Kit* provided no competition, with *Pathfinder* being exceptionally better in all cases.

[49] attribution: By Spencer C. Imbleau, MIT/Apache 2.0

CPU Utilization

There is also significantly lower CPU utilization monitored with a compute-centric approach, suggesting an intention of increased GPU leveraging and parallelization.

GPU-caching

Although *resvg* offers one of the most optimized backends for rasterizing vector graphics, the renderer failed to outperform the minimal tessellation-based renderer in *render-kit* by a lack of caching ability.

Chapter 7

Discussion

Our discussion connects interesting findings and discourse on our test case results and interprets our analytic framework's performance in a test trial.

7.1 Test Case Discussion

In the context of our test case, we analyzed several axes of measurement for static *svg* content. We extrapolated many patterns and consequences for our analysis questions through many data artifacts and plots provided by *vgpu-bench*. These artifacts proved how dated tessellation is in a modern context for static content. Additionally, results support compute-centric approaches may provide better results.

When tessellation input was a simple *svg* file, obstacles such as tessellation costs, initialization costs, and GPU latency crushed any potential of a fast initial frame. As a pre-computation model, tessellation also suffers from obstruction by other means, such as hostility towards deformations and rescaling. Benefits of hardware acceleration benefitting tessellation were only noticed with *Render-Kit*'s GPU cache on subsequent frames, even outperforming an extremely optimized renderer like *resvg*. However, these benefits came at the cost of higher computer resources. Moreover, the results of GPU leveraging in *Render-Kit* paled comparatively to *Pathfinder*'s sophisticated compute-centric rendering in every benchmark.

The test case implies that a compute-centric approach provides faster initial frame-time and subsequent frame times with evidence. Compute-centricity in *Pathfinder* was capable of higher parallelization and utilizing fewer CPU resources, mitigating the impact on business logic and system performance. While faster initial frame times were observed with CPU rendering by *resvg* in the most simple examples, this observed benefit only exists until render time exceeds GPU latency.

Specifically, hardware acceleration shows incredible benefits for rendering vector graphics for our test case, especially with compute-centric approaches. Tessellation stood dominated in our test case results by compute-centric pipeline, and *feels* dated as a symptom.

7.2 Product Retrospective

Our research *is* our product and methodology. We prove our framework's ingenuity through use; the benchmarks deliberated to support our synthesized theories and test cases prove that. An extended test trial rewarded itself through valid results and feedback, and the features and API provided are *useful*.

We feel successful in engineering a product to analyze vector graphics with finer granularity. Our framework made capturing benchmarks on image complexity, tessellation costs, and rendering easy. Moreover, all aspects of our framework's methodology were utilized in our test case, including integration into NVIDIA©*Nsight Systems*[1] for further analysis in the discussion, proving value to each design choice.

[1] see: https://developer.nvidia.com/nsight-systems

Chapter 8

Conclusion

8.1 Review

Vector graphics pose unique properties which make the imaging model ideal for users who value resolution independence, storage footprint, or seek to benefit from implicit modeling. While the field is optimistic with experimentation and research, new and old technologies lack comprehensive performance comparison. Users seeking to integrate a rendering backend have little more than cursory time trials or *Big-O* to encourage adoption, which is often insufficient.

This entanglement of information among technology is an opportunity for further understanding. Analyzing performance on the GPU is *hard*. Our research sets a precedent to deobfuscate the field of hardware-accelerated vector graphics with a novel benchmarking framework. Our tool's extensible design and integration into GPU analysis tools will begin to rectify the inadequate comparative research. We justified our framework's design decisions through methodology and a pilot test trial, which collected results defending our synthesized theories.

While *vgpu-bench*is the first step in bringing enhanced optics and context to eclectic options, there is still available work.

8.2 Future Work

In this section, we provide opinions on how to improve both the imaging model for vector graphics and our framework's usefulness.

8.2.1 Research Focus

Results presented in our test case support a theory that compute-centric approaches which extend the flexibility of compute-shaders to leverage more parallelism in the vector imaging model are promising. On the contrary, tessellation and pre-computation-based approaches may be convenient for static vector rendering but do not encourage further research, given their anti-thetic consequences to the imaging model. New research is

needed to extend the flexibility of low-level GPU features and maximize parallelism in a way that does not inhibit any benefits discussed in Chapter 2.

8.2.2 Tooling

Currently, tooling for vector graphics is poor. Most people may be familiar with excellent software such as Adobe Illustrator[1] or Inkscape[2] for composing vector graphics, but there is almost no free or open-source tooling for animation. This lack of tooling has likely discouraged adoption for artists and developers alike. Failed standards on how to encode animation such as the "*SMIL*" format have also come and gone, failing to reach adoption with eventual deprecation[3].

8.2.3 Encoding

The *svg* specification is built on *xml*, an extremely verbose format with repeating tags and redundant information. While this format is still generally more lightweight than raster graphics, further elaborated in Subsection §2.3.2, compression can improve file storage and empirical benefits such as network throughput.

Another issue is standardization. The bloated *svg* specification is an inhibitor of vector graphic rendering implementations, with full implementations being relatively rare, even in web browsers with commercial support. Future specifications should abbreviate current features, such as subdividing higher-dimension Bézier curves into piece-wise quadratic Bézier curves or flattening text into paths. A simpler specification would facilitate faster standardization but require tooling to adopt such output formats, which is a hard sell.

8.2.4 API Enhancements

Since Rust is still in its infancy as a language, it is missing some key language features which would empower a more intuitive API.

Integration

As the framework's ecosystem receives adoption, people will want to test against certain renderers or tessellators. The traits provided in the "*render-kit*" and "*tessellation-kit*" features provide a convenient interface and pre-written tests, although it would be beneficial if users could add modular dependencies which provide certain renderers, such as *Pathfinder*[4], or certain tessellators, such as *Lyon*[5], to test against.

[1] see: https://www.adobe.com/products/illustrator.html
[2] see: https://inkscape.org/
[3] see: https://developer.mozilla.org/en-US/docs/Web/SVG/SVG_animation_with_SMIL
[4] see: https://github.com/servo/pathfinder
[5] see: https://github.com/nical/lyon

Variadic generics

Variadic generics are the ability to enable traits, functions, and data structures to be generic over a variable number of types. Currently, a monitor delegated to a `Benchmark` is passed as a `Box<dyn Monitor>`, where `Monitor` is a trait. Thus, trait objects are handled by a collection (`Vec`) for dispatch when polling the collection of monitors.

This relatively minor inconvenience incurs some runtime overhead due to dynamic dispatch. On the other hand, variadic generics would make polling invocations and memory access slightly faster with static dispatch and stack-allocated monitors. An example of what variadic generics could be semantically is in Code Example 8.1 below.

Code Example 8.1: Theoretic variadic generic usage in *vgpu-bench*.

```
fn poll_monitors<...M: Monitor>(monitors: (...M)) {
    for monitor in ...monitors {
        monitor.poll();
    }
}

let cpu_mon = (CpuUtilizationMonitor::new());
let hb_mon = (HeartbeatMonitor::new());
let mixed_mon = (CpuUtilizationMonitor::new(),
                 HeartbeatMonitor::new());

poll_monitors(cpu_mon);
poll_monitors(hb_mon);
poll_monitors(mixed_mon);
```

Parallel runtime execution in `Driver`

The `Driver` is designed in such a way to execute benchmarks sequentially, as to eliminate interference. However, one may be a concerned with "how x performs while y." In such a case, this can currently be performed by launching two threads with two drivers, or two threads within a closure, but this is tedious ceremony that we would like to provide an API for.

Asynchronous API

Currently, our `Driver` data structure is a synchronous runtime executor for benchmarks. While this works, extending the runtime further with parallel computing and asynchronous programming should be possible. Independent tasks, such as polling with a `Monitor`, could be faster and less resource-hungry asynchronously than with multi-threading.

Currently, async closures are unstable as of Rust 1.59. Our methodology prohibited using unstable features in data collection code as a design philosophy. Therefore, *vgpu-bench*must wait for feature stabilization to declare asynchronous benchmark declarations. Async closures would also provide the ability for users to run async benchmarks in differing runtime executors built for futures, rather than relying on `Driver` as the only option. See Code Example 8.2 for a theoretical example.

Code Example 8.2: Async flow in *vgpu-bench*.

```
use futures::executor::block_on;
use vgpu_bench::prelude::*;

pub async fn benchmark() -> AsyncBenchmark {
    AsyncBenchmarkFn::new(async || {
        let mut measurements = Measurements::new();
        measurements.push(something_to.await);
        Ok(measurements)
    })
}

fn main() -> Result<()> {
    block_on(benchmark())?.write("results.csv")?;
    Ok(())
}
```

Live Monitoring

Sometimes visualization is more important than accuracy, and in such cases, we want to provide the ability to visualize a live, updating plot. This has the benefit of seeing live impact in an interactive demo, as opposed to annotating the behavior. Such a plot would update when a `Monitor` returns a polled value.

Bibliography

[1] A. M. Noll, "Scanned-display computer graphics," *Commun. ACM*, vol. 14, p. 143–150, mar 1971.

[2] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, pp. 720–738, June 1989.

[3] I. E. Sutherland, "Sketch pad a man-machine graphical communication system," in *Proceedings of the SHARE Design Automation Workshop*, DAC '64, (New York, NY, USA), p. 6.329–6.346, Association for Computing Machinery, 1964.

[4] Y. Reizner, "Resvg." https://github.com/RazrFalcon/resvg/tree/5e8c634457a70f9ac2656dc59e40da841a8fbe9b#svg-support, 2022.

[5] Pomax, "A primer on bézier curves," Jan 2022.

[6] A. H. Barr, "Global and local deformations of solid primitives," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, (New York, NY, USA), p. 21–30, Association for Computing Machinery, 1984.

[7] RustFest 2018, *Vector graphics rendering on the GPU in Rust with Lyon*, May 2018.

[8] Microsoft, "Geometry realizations overview," May 2018.

[9] C. Loop and J. Blinn, "Resolution independent curve rendering using programmable graphics hardware," *ACM Trans. Graph.*, vol. 24, p. 1000–1009, July 2005.

[10] D. Nehab and H. Hoppe, "Random-access rendering of general vector graphics," *ACM Trans. Graph.*, vol. 27, Dec. 2008.

[11] F. Ganacim, R. S. Lima, L. H. de Figueiredo, and D. Nehab, "Massively-parallel vector graphics," *ACM Transactions on Graphics (Proceedings of the ACM SIGGRAPH Asia 2014)*, vol. 33, no. 6, p. 229, 2014.

[12] R. Li, Q. Hou, and K. Zhou, "Efficient gpu path rendering using scanline rasterization," *ACM Transactions on Graphics*, vol. 35, no. 6, 2016.

[13] R. L. Levien, "A sort-middle architecture for 2d graphics," *Raph Levien's blog*, Jun 2020.

[14] G. LLC, "Skia." https://skia.googlesource.com/skia, 2022.

[15] P. Walton, "Pathfinder 3." https://github.com/servo/pathfinder/tree/581eadfbeb61a973f73691f4672ad40d6e70e7b5#features, 2022.

[16] N. Silva, "A look at pathfinder," May 2019.

[17] G. LLC, "Spinel." https://fuchsia.googlesource.com/fuchsia/+/refs/heads/main/src/graphics/lib/compute/spinel, 2022.

[18] G. LLC, "Spinel." https://fuchsia.googlesource.com/fuchsia/+/refs/heads/main/src/graphics/lib/compute/spinel/README.md, 2022.

[19] N. Silva, "Lyon." https://github.com/nical/lyon, 2018.

[20] N. Silva, "Tessellator." https://github.com/nical/lyon/wiki/Tessellator#sweep-line, 2022.

[21] L. P. Chew, "Constrained delaunay triangulations," in *Proceedings of the Third Annual Symposium on Computational Geometry*, SCG '87, (New York, NY, USA), p. 215–222, Association for Computing Machinery, 1987.

[22] M. Qi, T.-T. Cao, and T.-S. Tan, "Computing 2d constrained delaunay triangulation using the gpu," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, (New York, NY, USA), p. 39–46, Association for Computing Machinery, 2012.

[23] D. Eberly, "Triangulation by ear clipping," Nov 2002.

[24] C. Green, "Improved alpha-tested magnification for vector textures and special effects," in *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, (New York, NY, USA), p. 9–18, Association for Computing Machinery, 2007.

[25] V. Chlumský, J. Sloup, and I. Šimeček, Sep 2017.

[26] S. Laine and T. Karras, "High-performance software rasterization on gpus," in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, (New York, NY, USA), p. 79–88, Association for Computing Machinery, 2011.

[27] M. J. Kilgard and J. Bolz, "Gpu-accelerated path rendering," *ACM Trans. Graph.*, vol. 31, Nov. 2012.

[28] P. Walton, "Gpu rasterization, the orphan rules, and rocket," Dec 2018.

[29] A. Crichton, "Rust once, run everywhere: Rust blog," Apr 2015.

[30] I. Dursun, "Rust zero cost abstractions in action," Feb 2020.

[31] P. Domain, "What unsafe can do." https://doc.rust-lang.org/nomicon/what-unsafe-does.html.

[32] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin, "Engineering the servo web browser engine using rust," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 81–89, 2016.

[33] P. Domain, "Hello world!." https://foundation.rust-lang.org/news/2021-02-08-hello-world/, Feb 2021.

[34] S. Overflow, "Stack overflow developer survey 2021." https://insights.stackoverflow.com/survey/2021, 2021.

[35] P. Domain, "Rust api guidelines." https://rust-lang.github.io/api-guidelines/checklist.html, Mar 2022.

[36] W3C®, "Webgpu," Jun 2021.

A Methodology for Table 2.1

This explains the methodology for Table 2.1. All files used to replicate results can be found at https://github.com/simbleau/simbleau/tree/research/thesis-master. We used linux system binaries and *inkscape* for SVG → PNG file exporting.

First we parsed the file "*assets/Impossible_Cubes.svg*" for viewport information to obtain the canonical size the *svg* was saved in. The metadata in the image indicates the dimensions are roughly 375x429.

```
viewBox="0 0 374.95 429.34"
```

Thus, to export at $1x$ scale, we used the following *inkscape* command:

```
inkscape -w 375 -h 429 Impossible_Cubes.svg -e Impossible_Cubes.png
```

Upscaled dimensions are modified through the `-w` and `-h` options. File savings were measured in bytes with the formula $f(x, y) = 100(1 - \frac{x}{y})$, where $f(x, y)$ is the percentage of storage savings, x is the amount of original file bytes, and y is the new amount of file bytes.

B Methodology for Figure 3.2

There are many ways to simulate an image without occlusion culling. The first option is to use the blending hardware; when rendering geometry with any GPU API, specify the "add" blending operator and render "1" into the target. The target will result in a map containing the number of writes per pixel. Afterward, one can then take that as input of another shader that translates that number into a color that is easy to see.

That being said, we took a rudimentary approach, as detail was not imperative. We took an *svg*, "GhostScript_Tiger.svg", and ungrouped all paths in *Inkscape*. We then selected all paths and modified the opacity to 0.2 and the fill color to white. This process is shown in Figure 1.

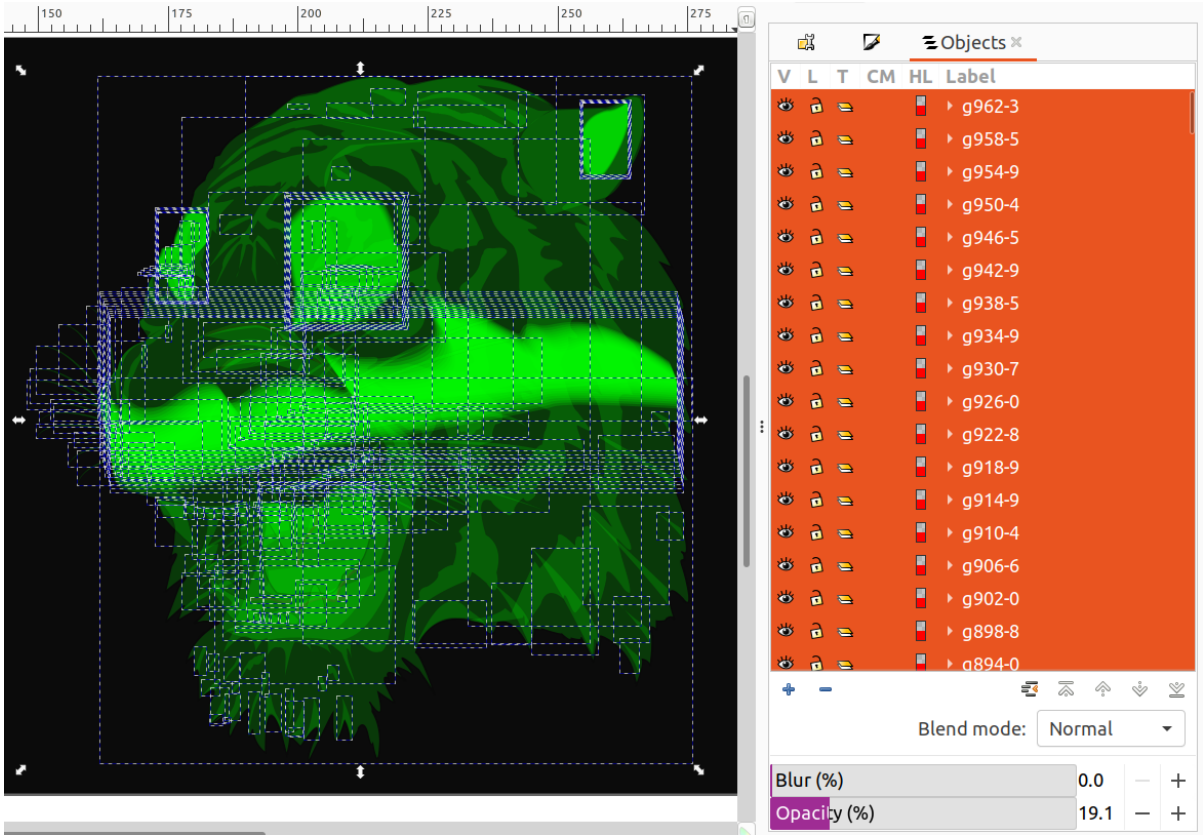


Figure 1: Changing fill and opacity for paths in *Inkscape*.[6]

[6]attribution: By Spencer C. Imbleau, MIT/Apache 2.0