ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

High-performance 2D graphics rendering on the CPU using sparse strips

Master Thesis

Laurenz Stampfl

October 10, 2025

Advisors: Prof. Dr. Ralf Jung, Dr. Raph Levien
Department of Computer Science, ETH Zürich

Abstract

2D rendering is omnipresent in our digital lives: Whether it be a web browser displaying web pages or an email software showing the contents of an email, they all rely on 2D rendering to be able to present their contents on a screen. While there already exist a range of solutions for 2D rendering for both GPU and CPU, the question of which fundamental rendering technique is the "best" so far still has no clear answer. In the pursuit of contributing to this research question, this thesis presents a new rendering paradigm that is based on so-called *sparse strips*, a memory-efficient run-length encoded intermediate representation of a rendered vector path that can be used in multiple ways, for example for subsequent rasterization or caching. To demonstrate the feasibility of the approach, a CPU-based 2D renderer that fully embraces the paradigm and supports the most common operations expected of a 2D renderer, including path filling and stroking, clipping, image and gradient rendering as well as blending and compositing, is implemented as part of this work. It is also shown that the renderer is very amenable to SIMD optimizations and multi-threading, allowing it to fully utilize the capabilities of modern CPUs. The evaluation demonstrates that the renderer achieves compelling performance and outperforms many of the existing CPU-based renderers in a wide range of benchmarks, validating the thesis that the sparse strips approach is promising and could serve as the foundational architecture for the next generation of 2D renderers.

Acknowledgements

First and foremost, I would like to thank Professor Jung for giving me the opportunity to turn my project idea into a fully-fledged master's thesis and for guiding me throughout the whole process.

I also want to express my gratitude towards all members in the Linebender community that I have been able to collaborate with in these past few months and who supported me during the whole journey. In particular, I want to thank Raph Levien for his excellent research in 2D rendering that made this project possible in the first place, and for being willing to mentor and support me during this project.

Finally, I would like to acknowledge the team behind Typst for creating a really powerful typesetting system that made the creation of this write-up a breeze and a very enjoyable experience.

Contents

Contents iii

1 Introduction 1
1.1 Contribution notice ... 3

2 Background 6
2.1 2D rendering ... 6
2.2 Drawing primitives ... 8
2.2.1 Lines ... 8
2.2.2 Quadratic Bézier curves ... 9
2.2.3 Cubic Bézier curves ... 10
2.3 Fills and strokes ... 10
2.4 Fill rules ... 11
2.5 Colors ... 12
2.6 Opacity ... 13
2.6.1 Premultiplied alpha ... 14
2.7 Compositing ... 14
2.8 Blending ... 16
2.9 Anti-aliasing ... 17
2.10 Complex paints ... 18
2.10.1 Gradients ... 19
2.10.2 Images ... 20
2.11 Clip paths ... 22

3 Implementation 23
3.1 API ... 23
3.2 Architecture overview ... 25
3.3 Stroke expansion ... 28
3.4 Flattening ... 29
3.4.1 Optimizations ... 31
3.5 Tile generation ... 32
3.6 Strips generation ... 34

3.6.1 Motivation ... 34
3.6.2 Merging tiles into strips ... 35
3.6.3 Calculating strip-level winding numbers ... 35
3.6.4 Calculating pixel-level winding numbers ... 37
3.6.5 Tile size ... 40
3.6.6 Sparseness ... 40
3.7 Coarse rasterization ... 42
3.8 Fine rasterization ... 43
3.8.1 Computing pixel color ... 44
3.8.2 `Fill` vs. `AlphaFill` commands ... 47
3.8.3 Compositing and blending ... 47
3.8.4 `u8`/`u16` vs. `f32` ... 49
3.9 Packing ... 50
3.10 SIMD ... 52
3.10.1 Library ... 52
3.10.2 Implementation ... 53
3.11 Multi-threading ... 56
3.11.1 Path rendering ... 57
3.11.2 Coarse rasterization ... 59
3.11.3 Rasterization ... 59
3.11.4 Alternative approaches ... 60
3.12 Clip paths ... 61

4 Comparison **64**
4.1 Raqote ... 64
4.2 Blend2D ... 66

5 Evaluation **68**
5.1 Introduction ... 68
5.2 Setup ... 70
5.3 Single-threaded rendering ... 72
5.3.1 Filling ... 72
5.3.2 Stroking ... 76
5.3.3 Paints ... 77
5.4 Multi-threaded rendering ... 80

6 Conclusion & Future Work **84**

Bibliography **87**

Chapter 1

Introduction

2D rendering lies at the heart of virtually all our interactions with digital devices. Whether it be web browsers on our computers, news apps on our phones or ticket machines at the nearest train station: They all rely on 2D rendering to present a user interface that we can interact with.

In most cases, it is preferable to make use of GPUs during the rendering process. GPUs excel at performing hundreds of thousands of computations at the same time, which fits nicely into the 2D rendering paradigm where the colors for millions of pixels need to be computed as quickly as possible. The importance of GPUs in this area is reflected by the fact that the vast majority of research in this direction makes use of them [1] [2] [3].

With that said, utilizing the GPU does come at a cost, and there are situations where it makes more sense to accept the trade-off of (often) slower rendering times by running the rendering pipeline on the CPU instead. To name three concrete advantages of doing so:

- **Portability**: By relying on the GPU, it is implicitly assumed that the host system actually has a GPU available. While this might be the case for most modern consumer-grade devices, it might not be the case for embedded devices or server appliances. In addition to that, when utilizing the GPU, there are many different competing graphic APIs to choose from, such as Vulkan[1] or OpenGL[2], which come with their own set of trade-offs with regard to platform compatibility. In contrast, by removing any dependency on external GPUs and solely relying on the CPU, it is guaranteed that the program will run on any kind of device as long as a supported target architecture is used.
- **Complexity**: Interfacing into the GPU usually entails a whole lot of additional complexity that is necessary to properly manage the resources and pass data between the GPU and the host system. For applications with high levels of interactivity, like graphical user interfaces, where having the best performance is absolutely critical, this is often a trade-off worth taking. Still, there are many other situations where

[1] https://www.vulkan.org (accessed on 15.09.2025)
[2] https://www.opengl.org (accessed on 15.09.2025)

maintaining low code complexity and small binary sizes has higher priority, making the simplicity of CPUs more attractive.
- **Performance**: In comparison to CPUs, GPUs undoubtedly have the upper hand when it comes to processing millions of pixels at the same time. Yet, GPUs do also have performance cliffs. For example, there can be a significant latency during the first startup as the GPU context needs to be initialized and set up. This latency might not be a problem for GUI applications where a one-time startup latency is barely noticeable, but it could become cumbersome in situations where we only want to run a one-time rendering operation, for example when rendering a single image.

Doing 2D rendering on the CPU is in some sense an already solved problem, as there already exists a plethora of libraries that can do the job, including for example Skia[3] or Cairo[4], which have both been existing for more than a decade. However, the question of which *fundamental* approach to rendering is the best from the standpoint of performance and efficiency is an open research question. This is especially true as the capabilities of our CPUs have been constantly evolving, with the addition of features like SIMD and multi-threading offering new opportunities for exploring and researching different rendering paradigms. This is evidenced by the recent emergence of Blend2D[5], a CPU-renderer that for the first time offers multi-threading capabilities and beats all existing renderers by a wide margin in many benchmarks according to [4], in part thanks to a novel rendering architecture based on just-in-time compilation.

The main motivation for this thesis is two-fold. On the one hand, I wish to fill an important gap in literature, as 2D rendering is a subfield of computer graphics that has generally received little academic interest in recent years. My hope is that this work can serve as an introduction into this field by explaining relevant foundational knowledge in an approachable way based on the inner workings of a feature-complete and performant CPU-based 2D renderer in an approachable manner.

On the other hand, I want to make a contribution to the research question of finding the "best" rendering technique by exploring a novel rendering paradigm that is based on so-called *sparse strips*, a new sparsely encoded representation of rendered vector paths. This paradigm borrows some key ideas from the concept of so-called *merged boundary fragments* as they are presented by R. Li, Q. Hou, and K. Zhou [2] (although originally intended for GPU-based rendering), but provides better compatibility with CPUs by extending the idea in a number of ways to make it more amenable to SIMD optimizations

[3] https://skia.org (accessed on 15.09.2025)
[4] https://www.cairographics.org (accessed on 15.09.2025)
[5] https://blend2d.com (accessed on 15.09.2025)

and multi-threading. In order to validate the idea, I spearhead the development of a new CPU-based renderer called *Vello CPU*[6] as part of this master's thesis. It is fully based on the sparse strips idea and supports most features that are usually expected from a 2D renderer. The implementation is done using the Rust programming language.

The remainder of this thesis is structured as follows: In Section 2, some of the basic concepts of 2D rendering will be explained, as they are prerequisites for the following chapters. In Section 3, a holistic overview of the whole architecture of Vello CPU and its implementation is given by providing detailed explanations of the functionality of each part of the rendering pipeline. Two additional subsections will be dedicated to describing how the pipeline is SIMD-optimized and made compatible with multi-threading. In Section 4, we will contrast the design of Vello CPU against two other renderers to highlight similarities but also emphasize the novelties of our architecture. In Section 5, the performance of Vello CPU will be comprehensively evaluated by running it against the Blend2D benchmark suite [4] and comparing the performance against many other available CPU-based 2D renderers. Finally, in Section 6, the main findings will be summarized and suggestions for potential future work will be given.

1.1 Contribution notice

It needs to be noted that the project of building Vello CPU is part of a bigger collaboration involving external contributors, where another goal is to build another 2D renderer based on the sparse strips paradigm that also utilizes the GPU to achieve even better performance. Additionally, Vello CPU was not created completely from scratch during this thesis, but borrows some code from an earlier prototype. Therefore, the description of some parts of the pipeline in Section 3 are included as they are indispensable to gain a full understanding of how the renderer works, but were at least partly co-implemented by other parties. For full transparency, a description of the timeline leading up to the start of the thesis as well as the contributions that have been specifically made as part of the thesis is provided below.

The sparse strips approach was first sketched out by Raph Levien in January 2024 in the form of an informal document[7] that summarizes the main idea. After a few months of experimentation with a focus on integrating the idea into a GPU renderer, Raph developed a CPU-only prototype[8] as part of a week-long research retreat in late 2024,

[6] https://github.com/linebender/vello/tree/main/sparse_strips/vello_cpu (accessed on 15.09.2025)

[7] https://docs.google.com/document/d/16dlcHvvLMumRa5MAyk2Du_MsjF32w0G-n7L9NOJUbRI/edit?tab=t.0#heading=h.9lokdzrz8r5x (accessed on 05.10.2025)

[8] https://github.com/linebender/piet/pull/589 (accessed on 03.10.2025)

demonstrating that the idea can in principle be implemented on the CPU. Nevertheless, the prototype still left many pressing questions unanswered. On the one hand, it was unclear whether approach could be extended to support commonly-expected features like rendering gradients and images as well as doing blending and compositing. On the other hand, it had not been fully determined yet in which ways SIMD could be integrated in the pipeline and whether a performant multi-threaded rendering approach was possible. Resolving these open research question to arrive at a clear conclusion on whether a renderer based on sparse strips can outperform or at least compete with existing renderers formed the main motivation of this thesis.

To this purpose, I made the following contributions: First, I rewrote the existing version of the fine rasterization and packing stages (see Section 3.8 and Section 3.9) completely from scratch to add support for a `f32`-based (32-bit floating point numbers) as well as `u8`-based (8-bit unsigned integers) rendering mode. I also extended the fine rasterization stage to support rendering gradients and images and performing blending and compositing.

Next, I did an in-depth analysis of the performance profile using Apple Instruments to identify bottlenecks and discover optimization opportunities in all parts of the pipeline. As part of this, I have for example come up with two crucial optimizations for curve flattening (as explained in Section 3.4.1) that lead to drastic speedups.

In order to support SIMD, I made major contributions to the fearless_simd[9] library to support the necessary arithmetic operations for NEON and SSE4.2. Using that, I rewrote the flattening, strips generation, fine rasterization and packing stages (see Figure 18) to actually make use of SIMD.

The next major contribution implementation-wise was the multi-threaded rendering mode. This included coming up with the right architecture as well as experimenting with different approaches and synchronization primitives to support the design.

Another feature I worked on is support for clip paths. While a basic clipping algorithm was already previously implemented by another collaborator[10], I designed and implemented a completely novel algorithm (described in Section 3.12) that is more performant than the previous one. While the pull request[11] with the implementation has not been merged at the time of writing, it is expected to replace or at least complement the current implementation in the near future.

[9] https://github.com/linebender/fearless_simd (accessed on 03.10.2025)
[10] https://github.com/linebender/vello/pull/878 (accessed on 08.10.2025)
[11] https://github.com/linebender/vello/pull/1203 (accessed on 08.10.2025)

Finally, as described in Section 5, I created C bindings to integrate three different Rust-based renderers, including Vello CPU, into the Blend2D benchmark harness, ran the evaluation and did the analysis and interpretation of the results.

Chapter 2

Background

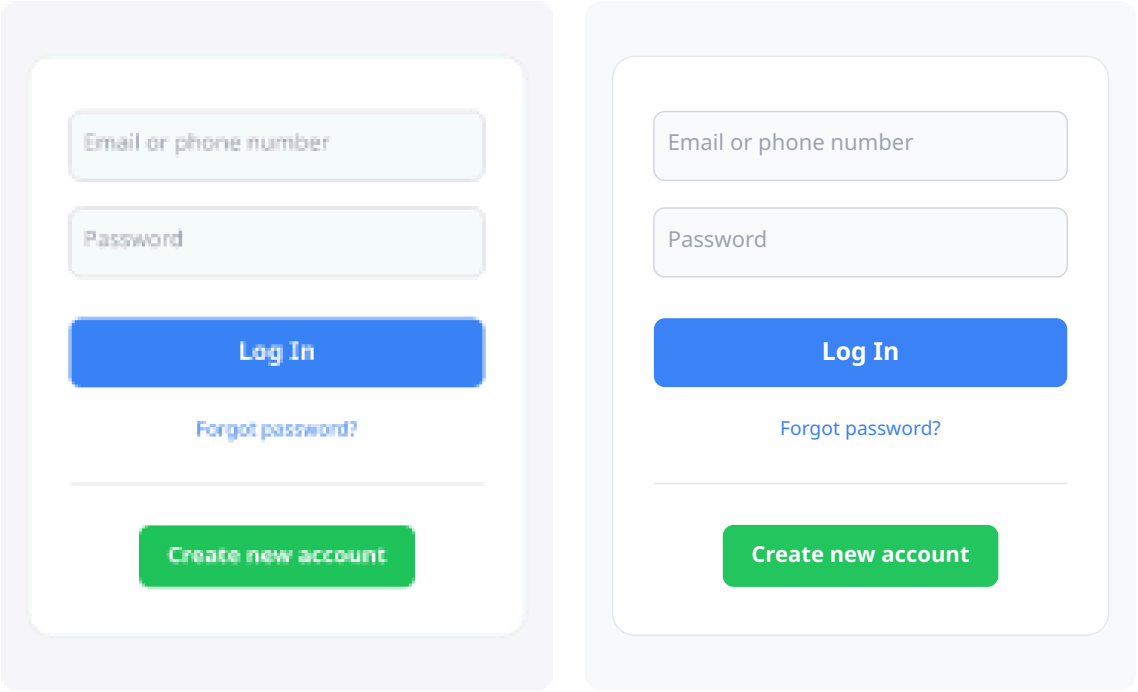
In this chapter, some of the basic notions of 2D rendering and a number of important concepts will be introduced, including for example lines and curves, the distinction between filling and stroking as well as anti-aliasing.

2.1 2D rendering

Nowadays, people mostly take it for granted that they can use their computers to engage with digital content seamlessly and without any hiccups. This is possible thanks to a tight feedback loop, where users can observe the current *state* of their system via their displays and based on this make decisions on what to do next by controlling their mouse and keyboard. For example, they expect to be able to navigate and scroll through web pages without significant delay. When writing an e-mail, the typed words should immediately show up on the display so that they can be seen and edited immediately.

However, something that is often not appreciated is that there actually is a lot of work happening in the background to ensure that the user has a seemless experience when doing the above-mentioned activities. One fundamental reason for this is that there exists a gap between the representation of graphical information in our applications and the way displays can actually show information to the user. This gap needs to be bridged in some way.

Computer displays only understand one language: the language of pixels. They are made up by a rectangular grid of small individual pixels (usually anywhere between 1000 and 4000 pixels in a single direction) that can emit varying intensities of red, green and blue at the same time. By mixing and matching those intensities in certain ways, other intermediate color such as orange, purple or white can be generated. By making each pixel emit a specific color, we can simulate different graphical effects that can then be interpreted by the user. Figure 1a shows a Facebook login modal as it is displayed on a screen with a resolution of 180x225 pixels. When looking at this picture from afar, it is easy to discern the login modal. However, a considerable disadvantage of this pixel-based graphics model is that it is inherently lossy: Once you render the modal at a specific pixel resolution and approximate its contents by pixels, there is no way to recover the original information anymore. As a result, when trying to zoom into



(a) The modal as a 180x225 image. (b) The modal as a vector graphic.

Figure 1: A comparison between a rasterized image and a vector graphic, based on a recreation of the Facebook login modal [5].

Figure 1a to scale it up, instead of becoming more readable, the result will contain very noticeable pixel artifacts and become even harder to read.

This is in stark contrast to the graphics model used by web browsers and other applications, where the contents of a graphics object are instead represented using *vector drawing instructions*. Conceptually, the viewable area is usually interpreted as a continuous coordinate system. Inside of this coordinate system drawing instructions can be emitted, such as *draw a line from point A to point B* or *draw a curve from point C to point D*. The semantics of these basic primitives will be defined more precisely in Section 2.2.

By combining these primitives in various ways, the outline of virtually any arbitrary shape can be defined in a mathematically precise way. This includes simple shapes like for example rectangles or circles, but also extends to more complex shapes such as whole letters of the alphabet. Finally, by combining multiple shapes and specifying the color those shapes should be painted with, nearly any kind of graphical object can be produced, including the modal in Figure 1b. An important consequence of this type of representation is that the representation is *resolution-independent* and thus makes the

object arbitrarily scalable at any resolution. No matter how much the user zooms into Figure 1b, the text and the shapes always remain crisp in quality.

However, this divergence between the way applications represent graphics and the way computer screens display them means that there must be some intermediate step that, given a specific pixel resolution, performs the (inherently lossy) conversion from continuous vector space to the discrete pixel space, as fast and accurately as possible. Performing this translation step is the fundamental task of a *2D graphics renderer*.

2.2 Drawing primitives

As mentioned above, a set of basic drawing primitives is required to be able to define the outlines of graphical objects. By combining dozens or even hundreds of these primitives, we can build nearly any arbitrarily complex shape. There is no unanimously recognized set of such building blocks, and different specifications have different requirements in this regard. For example, the PDF (portable document format) specification only defines lines and cubic Bézier curves as the basic path-building primitives [6, p. 132-133], while the SVG (scalable vector graphics) specification additionally also allows using quadratic Bézier curves and elliptic arc curves [7, ch. 8].

Nevertheless, in general, there are three path-building primitives that are commonly used, and any other primitives that might be defined in certain specifications can usually be approximated by them: *Lines*, *quadratic Bézier curves* and *cubic Bézier curves*.

Each type of primitive has a start point P_0 and an end point P_x (the end point is denoted as P_1 for lines, P_2 for quad curves and P_3 for cubic curves) defined in the 2D coordinate system. We can then define a parametric variable $t \in [0.0, 1.0]$ as well as a parametric function F such that $F(0) = P_0$, $F(1) = P_x$, and $F(t) = Q_t$, where Q_t simply represents the position of the interpolated point for the given drawing primitive. Conceptually, we then evaluate the function *infinitely* many times for all values in the interval $[0, 1]$ and can then plot its exact representation.

2.2.1 Lines

The definition of lines is relatively straight-forward and illustrated in Figure 2. Given our start and end points P_0 and P_1, we can use the formula $F(t) = P_0 + t \cdot (P_1 - P_0)$ to perform a simple linear interpolation and evaluate it [8, p. 218]. When doing so for all $t \in [0, 1]$, we end up with a straight line that connects the two points.

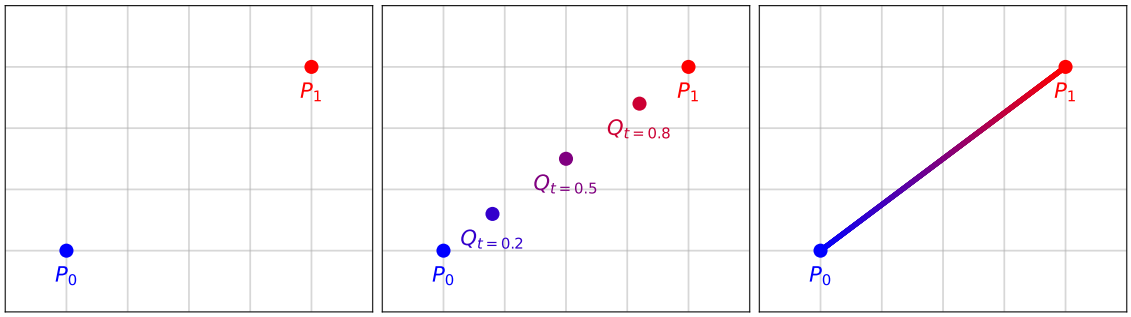


Figure 2: The course of a straight line between two points P_0 and P_1. For easier visualization, $t = 0.0$ is painted in blue, $t = 1.0$ in red and intermediate values with an in-between color.

2.2.2 Quadratic Bézier curves

For quadratics Bézier curves, things are a bit different. While we still have the start and end points P_0 and P_2, we have a third point P_1 which is called the *control point*. Given these points, the formula for evaluating the curve is given by $P_0 \cdot (1-t)^2 + 2 \cdot (1-t) \cdot t \cdot P_1 + P_2 \cdot t^2$ [8, p. 239]. The evaluation of a quadratic Bézier curve can be nicely visualized by thinking of it as a linear interpolation applied twice, as can be seen in Figure 3.

Assume we want to evaluate the curve at $t = 0.3$. We first start by finding the point P_0P_1 by linearly interpolating the points P_0 and P_1 with our given t. We do the same for the line spanning the points P_1 and P_2 to end up with the point P_1P_2. Then, we simply connect the points P_0P_1 and P_1P_2, and perform another round of linear interpolation with our value t, which will then yield the final point on the curve. Similarly to simple line segments, we perform this evaluation for all $t \in [0, 1]$ to end up with the final curve as it is visualized on the right in Figure 3.

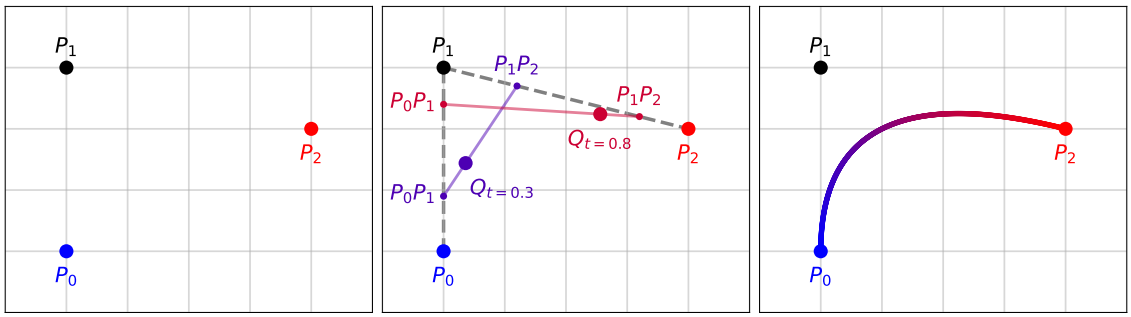


Figure 3: Visualizations of the evaluation of a quadratic curve.

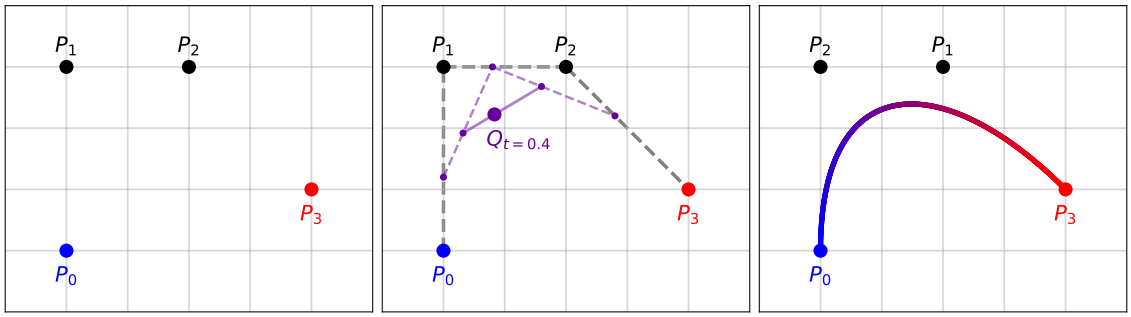


Figure 4: Visualizations of the evaluation of a cubic curve.

2.2.3 Cubic Bézier curves

Cubic Bézier curves follow the same pattern as quadratic curves, the only difference being that we have an additional control point P_2, and therefore need to run three rounds of linear interpolation to evaluate a point on the curve. The formula is given by $P_0 \cdot (1-t)^3 + P_1 \cdot 3t \cdot (1-t)^2 + P_2 \cdot 3 \cdot t^2 \cdot (1-t) + P_3 \cdot t^3$ [8, p. 240]. In Figure 4, we can once again gain a better intuition of this formula by visualizing the whole process of evaluation by repeatedly subdividing the curve using linear interpolation with our parametric value t, until we have computed the final point.

2.3 Fills and strokes

We now know how we can define the outline of a shape using lines and curves, but how can we actually *draw* it? In general, we distinguish between two different types of drawing modes: *filling* and *stroking*. Figure 5 illustrates the difference between those. In

(a) The outline of a dragon.

(b) The dragon painted using a blue fill.

(c) The dragon painted using a blue stroke of width 2 pixels.

Figure 5: Illustration of the different drawing modes.

Figure 5a, we can see the outline of a dragon, which has been specified using the basic building blocks we just defined in Section 2.2.

In the case of *filling*, we determine all of the areas on the drawing canvas that are *inside* of the outline we defined (how exactly these are determined will be elaborated in Section 2.4) and paint them using the specified color, as can be seen in Figure 5b.

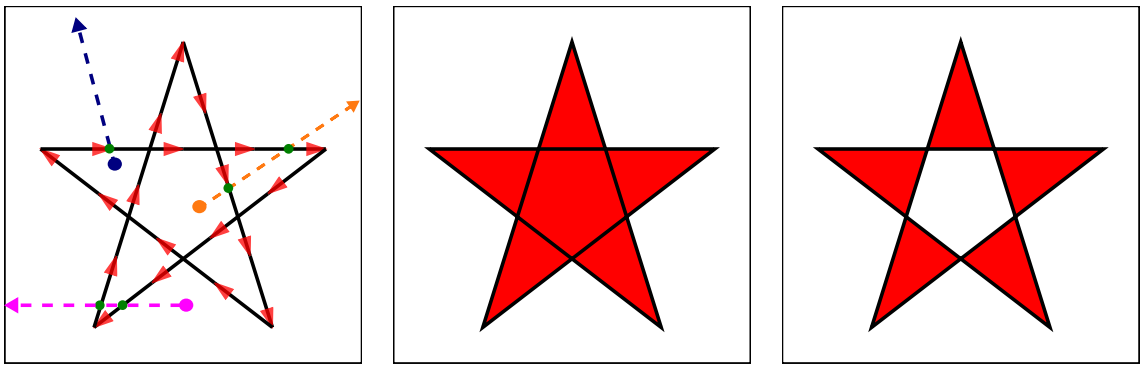
Stroking on the other hand uses a different approach. Stroking a shape can be seen as equivalent to using a marker with a specific color and width and using it to trace the outline of the shape. In doing so, the contour of the shape will be painted in that color. The visual effect of this drawing mode can be observed in Figure 5c.

2.4 Fill rules

Another important problem to be aware of is the question of which parts of a shape are actually considered to be on the "inside" and thus should be colored. For simple shapes such as rectangles or circles, it is intuitively obvious which areas are inside of the shape. But when trying to analyze more complex, self-intersecting paths, just relying on intuition is not sufficient anymore. There is a need for a clear definition of "insideness" such that it is always possible to unambiguously determine whether a point in the drawing area is inside of the shape or not.

In order to do so, we first need to introduce the concept of *winding numbers*. Remember that our shapes are built using lines and curves, which always have a start and an end point. Consequently, each path has an inherent direction. This is illustrated in Figure 6a, where we have the outline of a star as well as red arrows that indicate the direction of each line. In order to determine whether any arbitrary point is inside of the shape, we keep track of a winding number counter (which is initially 0) and shoot an imaginary ray into any arbitrary direction. Note that the exact direction of the ray does not matter as long as it is "infinitely" long, as the end result will always be the same. Every time the ray intersects a path of the shape, we check the direction in which the path intersects our ray. We increase the winding number counter if the direction is left-to-right, and decrease it if it is right-to-left [7, ch. 11].

Let us consider the orange point and its corresponding ray in Figure 6a first. With the given ray, it intersects the shape twice and in both cases the direction is left-to-right. As a consequence, the winding number of the orange point is two. The blue point only has one left-to-right intersection with a path, and therefore has a winding number of one. Finally, the fuchsia point first has a right-to-left intersection, resulting in an intermediate



(a) Analysis of winding number in two locations .

(b) The star painted using the non-zero fill rule.

(c) The star painted using the even-odd fill rule.

Figure 6: Illustration of the two most common fill rules.

winding number of −1. However, it then intersects the path a second time left-to-right, resulting in a final winding number of zero.

We conceptually repeat the above calculation for each point in the drawing area. Once we know all winding numbers, we can simply apply the fill rule to determine whether a point should be painted or not: For the *non-zero* fill rule, we paint the point if and only if the winding number is not equal to zero. For the *even-odd* fill rule, we paint the point if and only if the winding number is an odd number. The difference becomes apparent when contrasting Figure 6b and Figure 6c. In both cases, the fuchsia point remains unpainted, since the winding number is zero. The blue point *is* painted in both cases, since one is both not equal to zero and also an odd number. Things are different when looking at the orange point, though. According to the non-zero rule, the point *is* painted since two is not equal to zero. However, it is not painted according to the even-odd rule, because two is not an odd number.

2.5 Colors

In order to be able to paint shapes using certain colors, we need to be able to somehow *specify* those colors. The specification of colors is a multi-faceted and complex topic; covering all the details that are involved in the different ways colors can be defined is beyond the scope of this work. Instead, we will limit our explanations to defining RGB colors in the sRGB color space [9]. The sRGB color space is used frequently in the context of computer devices and is the default color space in many web graphics specifications such as SVG [7, ch. 12] or HTML Canvas [10, ch. 4.12].

In this color model, we define our colors using the three primaries red, green, and blue, which can be activated with varying degrees of intensity. How these intensities

Figure 7: The result of mixing the color primaries with different intensities.

are described depends on the underlying number type that we use. When using 32-bit floating point numbers (from now we will shorten this to `f32`, as is the convention in Rust), 0.0 usually stands for no activation at all, while 1.0 stands for full activation. It is also common to use 8-bit unsigned integers (often shortened to `u8`) to represent the RGB intensities, in which case 0 stands for no activation and 255 stands for full activation. Figure 7 shows some of the resulting colors that can be achieved by mixing intensities in certain ways: Enabling none of the primaries gives you a black color, while fully enabling all results in white. Fully activating one of the primaries while disabling all other ones results in the primary color itself. And finally, by using various combinations of intensities, the different intermediate colors can be created.

2.6 Opacity

So far, we have only considered the situation of drawing a *single* shape with a specific color. In doing so, we expect all areas covered by the shape to be painted using the specified color. However, what happens if we draw 2 shapes in different colors that overlap each other? How will the area that contains the overlaps be painted?

The answer depends on the *opacity* (also known as alpha value) of the color. In Section 2.5, it was mentioned that a color is specified by the three components red, green and blue. In reality, there usually is a fourth component that is called *alpha*. The alpha value specifies how transparent the color should be. If the value is 0.0 (0%) it means that the color is completely *transparent*, i.e. completely invisible. If the value is 1.0 (100%), the color is completely *opaque*, i.e. fully visible. By choosing a value between 0.0 and 1.0, we can make a color semi-transparent. Similarly to colors, we can also specify opacity using values between 0 and 255 instead. We use the term *RGBA* to denote storing RGB colors with an additional alpha channel.

The effect of varying the opacity can be observed Figure 8, where a green rectangle with varying opacities is drawn on top of a fully opaque red rectangle. In case the opacity is 0%, the green rectangle cannot be seen at all. In the case of 100%, the area of the green rectangle is painted completely in green. In all other cases, the background still shines

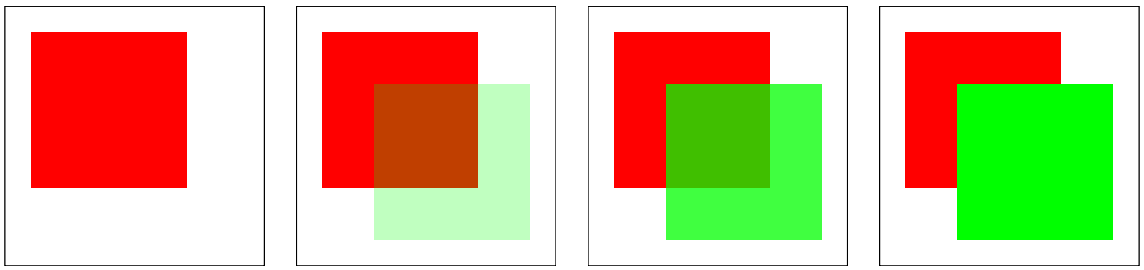


Figure 8: Painting overlapping shapes with varying opacities and a white background. From left to right: 0%, 25%, 75% and 100% opacities of the green rectangle in the foreground.

through to a certain degree, depending on how high the transparency is. As a result, the overlapping area of the two rectangles takes on a color that is somewhere "in-between" red and green.

2.6.1 Premultiplied alpha

Another important concept related to representation of color is the distinction between *premultiplied* vs. *non-premultiplied* alpha. We now know that we can store the RGBA colors using four numbers, each number representing one channel. A fully green color with 50% opacity can be compactly represented using the tuple $(0.0, 1.0, 0.0, 0.5)$. Storing the alpha explicitly as a separate channel is referred to as *non-premultiplied alpha* representation.

However, as will be described in Section 2.7, many compositing formulas require multiplying the RGB channels with the alpha value. Redoing this computation every time is expensive, giving rise to the idea of performing this multiplication *ahead of time* and storing the color implicitly with the alpha channel multiplied. This is referred to as *premultiplied alpha* representation [11].

For example, given our above example $(0.0, 1.0, 0.0, 0.5)$, in order to convert it into premultiplied representation we simply need to multiply the RGB channels with the alpha value, which results in the tuple $(0.0 \cdot 0.5, 1.0 \cdot 0.5, 0.0 \cdot 0.5, 0.5) = (0.0, 0.5, 0.0, 0.5)$. Doing calculations using premultiplied alpha whenever possible is important to ensure high performance, as it can drastically reduce the number of computations that need to be done per pixel.

2.7 Compositing

In Figure 8, we have demonstrated what happens when one shape is drawn on top of another one. In case the shape is fully opaque, it will completely replace the colors of any

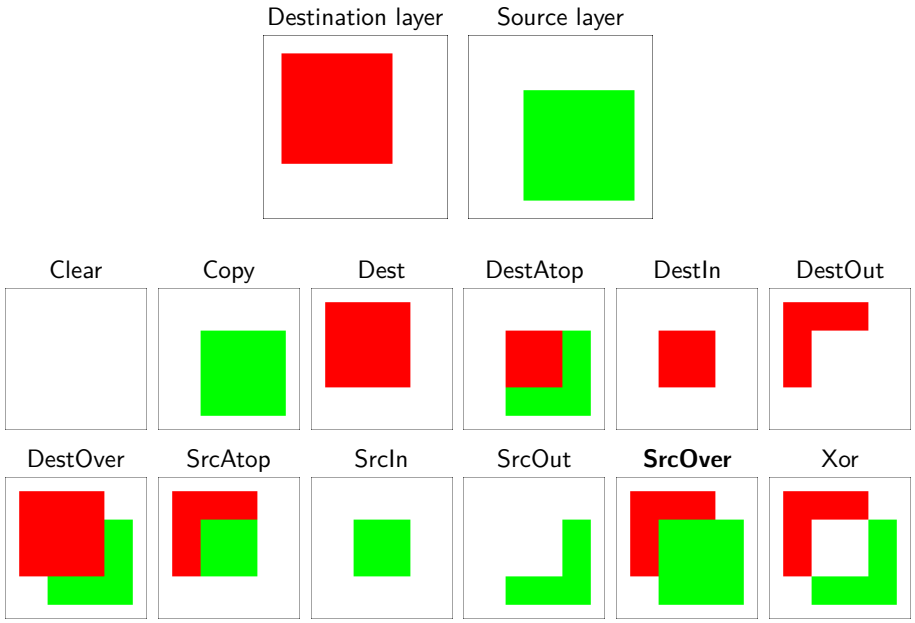


Figure 9: The effect of compositing two layers with the composition operators presented in [11]. Illustration adapted from [12].

previously drawn pixels in the overlapping areas, while if the shape has transparency, the background will shine through to some degree.

While the above-described behavior is the most commonly expected one, there actually exists a generalization of how a *source layer* (containing the shape you are about to draw) can be combined with a *destination/background layer* (containing everything that has been drawn so far). The foundational algebra for this is established in T. Porter and T. Duff [11]. To put it briefly, that seminal paper introduces an algebraic model that can be used to combine two layers in different ways. The visual effect of these different operators can be observed in Figure 9.

The source-over composition operator is by far the most commonly used one and intuitively equivalent to placing the new shape *on top* of the existing one, as was done in Figure 8. This is in contrast to the destination-over composition mode, where the new shape is instead drawn *below* the existing background. Another interesting combination is the Xor mode, where only the non-overlapping parts of the source and background are visible and all other parts are cleared. Finally, the source-in compositing operator can be used to clip a shape to another one, which is a very common operation in 2D rendering. By combining the algebraic building blocks introduced in that paper in different ways, eight other composition modes can be derived, though some arguably are less useful in practice and simply a result of exhaustively enumerating all possibilities.

Despite their age, the Porter Duff compositing operators form an important part of the CSS and HTML canvas specification [12] and are therefore highly relevant and implemented by most 2D renderers.

2.8 Blending

Blending is closely related to compositing, in the sense that it allows changing the behavior of how a source and destination layer are merged. However, while compositing is more about defining the visible areas, blending is about how the colors are mixed together. Figure 10 illustrates this behavior. By far the most common blend mode is *Normal*, where no additional mixing of colors happens and it therefore corresponds to a no-op (i.e. has no effect at all). In contrast, other blend modes such as *Color Dodge* or *Difference* result in the image taking on a noticeably different tone. The exact semantics of each blend mode are defined by mathematical formulae that describe how the colors of the source and destination should be mixed together [12].

There are two additional points worth mentioning regarding blending and compositing.

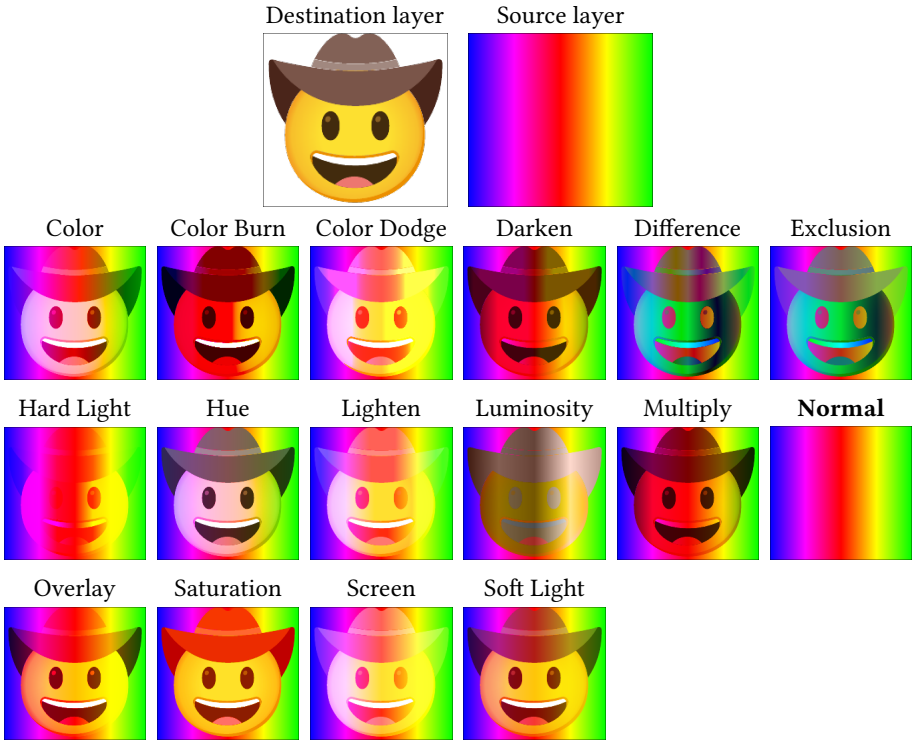


Figure 10: The effect of blending two layers with different blend modes. Illustration inspired by a similar figure in [12].

- They are not exclusive concepts but *complementary*. Whenever we want to draw a new shape and integrate it into the background, we *first* perform blending for each pixel to determine a (possibly) new source color and *then* perform compositing to determine how the result is integrated into the existing drawings.
- While certain blend modes and composition operators are occasionally used to achieve specific visual effects, especially in web graphics, by far the most common operation is to use the normal blend mode in combination with source-over compositing, which is a relatively straight-forward operation and has the visual effect of placing the new shape on top of the existing ones. The combination of these two modes is also referred to as *simple alpha compositing* [12].

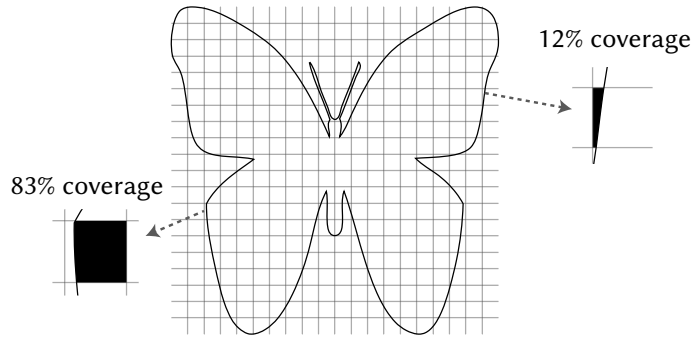
2.9 Anti-aliasing

As was elaborated in Section 2.1, the main goal of 2D rendering is to convert vector graphics into pixel representation. However, a fundamental problem is that since vector graphics are defined in a continuous space, it is possible that certain parts of the shape only *partially* cover a pixel, as can be seen for example in Figure 11a. Since a pixel can only emit one specific color, there is no direct way of retaining that information after the conversion process. There are two ways this problem can be dealt with.

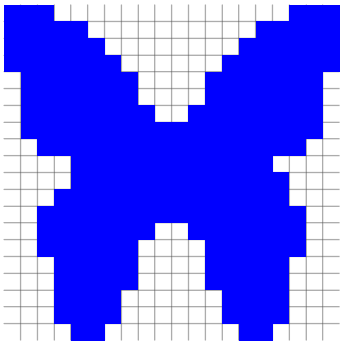
First, we can completely discard information about partial coverages and fully paint the pixel with the color if more than a specific percentage of the pixel is covered and not paint it at all in the other case. The result can be seen in Figure 11b. By using this method, the resulting shape will look noticeable blocky. These artifacts are referred to as *aliasing artifacts* [13] and are usually undesirable. For example, when rendering text at a very low resolution, the letters might become hard to read.

Because of this, it is common to render with *anti-aliasing* enabled. When looking at Figure 11c, it is apparent that the edges of the butterfly are much smoother and easier to look at. This effect is achieved by simulating the partial coverage of pixels by applying an additional opacity to the color so that parts of the background still shine through. In Figure 11c, all pixels that are entirely within the shape are painted using a fully opaque, blue color, while edge pixels appear lighter due to the additional opacity.

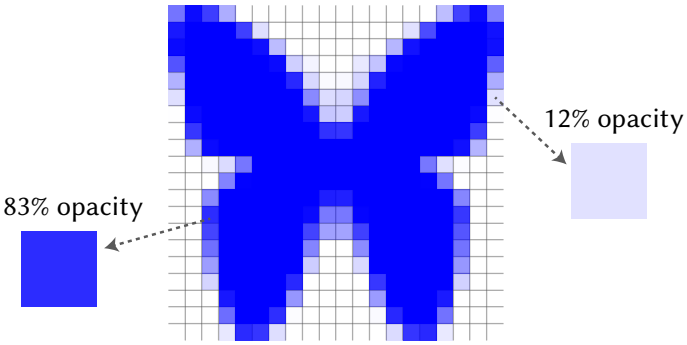
It is worth highlighting that by doing anti-aliasing this way, we are *conflating* two distinct concepts: The alpha value of a color and the coverage of a pixel are not inherently related to each other, but when rasterizing images, using color alpha to approximate pixel coverage usually works well in practice. However, this approach is not flawless and can lead to so-called *conflation artifacts* [1]. The effects of this phenomenon can be



(a) Examples of pixel coverages.



(b) Rasterizing without anti-aliasing.



(c) Rasterizing with anti-aliasing.

Figure 11: A butterfly rasterized to a 20x20 screen with and without anti-aliasing.

observed in Figure 12, where we are drawing two fully opaque triangles that overlap each other.

Since the green triangle completely overlaps the red one, there should be no visible red paint. However, upon rasterization, the following happens: When we first draw the red triangle, we use an opacity of 50% for the pixels that are only partially covered by the red shape. The same happens when drawing the second triangle in green. The crucial point here is that since we previously *converted* pixel coverage to color opacities for the edge pixels, we will compose a green pixel with 50% opacity on top of a red pixel with 50% opacity, resulting in a brownish color along the edges instead of a fully green one.

2.10 Complex paints

Up until now, we have always painted our shapes using a single color. While this is the most common operation, there are many different kinds of fills that can be used. The actual set of filling primitives that is available can vary: For example, in the case of SVG and HTML Canvas, the two main types of paint are *gradients* and *patterns* [7, ch. 13]

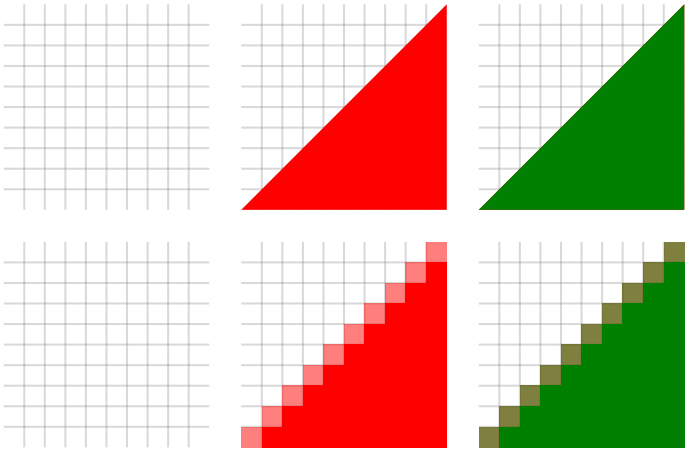


Figure 12: Drawing and rasterizing two completely overlapping triangles. The edge pixels will assume a brown color instead of a green one due to conflating pixel coverage and opacity

[10, ch. 4.12.5.1.10]. The PDF specification, however, defines some additional paints, including triangle meshes and Coons patch meshes [6, p. 192-201]. In this section, we will narrow our focus on two paints commonly used in web rendering. Gradients will be explained in Section 2.10.1 and images, which can be seen as a subset of pattern fills, will be described in Section 2.10.2.

2.10.1 Gradients

Conceptually, gradients represent transitions between two or multiple colors. We consider a parametric variable t that ranges between 0.0 and 1.0 and assign a number of colors to a specific position on that range. For example, we could assign the color blue to the position 0.0, the color red to the position 0.4, the color yellow to the position 0.7 and finally the color green to the position 1.0. All of the other positions that have not been explicitly specified are calculated by doing a linear interpolation between the given stops. The result of mapping out the whole range is visualized in Figure 13.

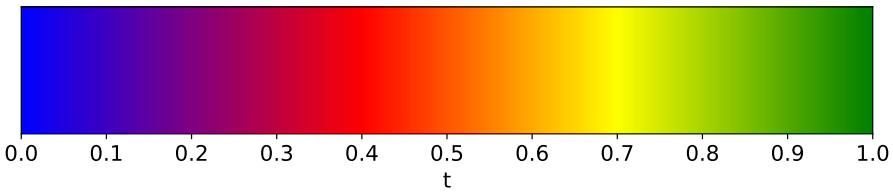
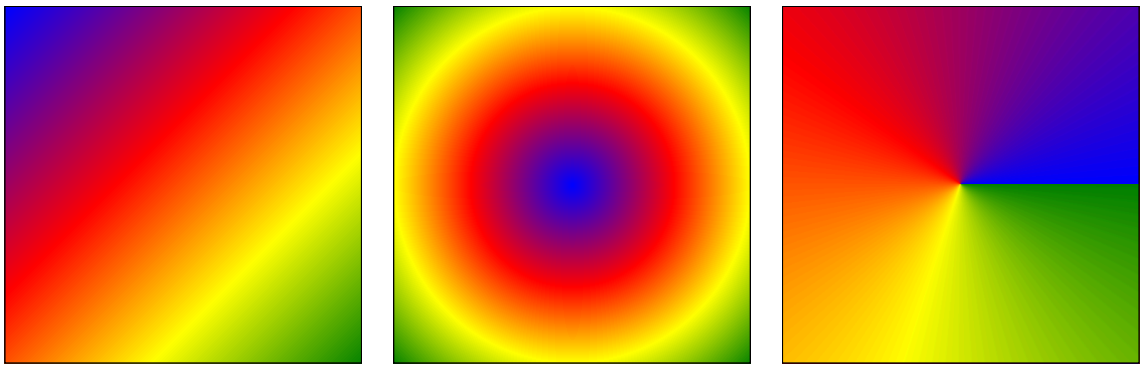


Figure 13: Visualization of a gradient line with the stops `(blue, 0.0)`, `(red, 0.4)`, `(yellow, 0.7)` and `(green, 1.0)`.



(a) Linear gradient. (b) Radial gradient. (c) Sweep gradient.

Figure 14: The shape of a rectangle filled using a linear, radial and sweep gradient.

Once we have a mapping from t values to a color, we simply need to define another mapping from the (x, y) position of a pixel to a t value, so that we know how to color that pixel. There are three commonly used types of gradients that define this mapping in different ways: *Linear gradients*, *radial gradients* and *sweep gradients*. Examples of applying those paint types to a rectangle can be seen in Figure 14.

In the case of a linear gradient, we define a start and end point along which the gradient should interpolate. In the case of Figure 14a, the start point is in the top-left corner and the end point in the bottom-right corner. The visual effect will be a linear variation of the gradient line in a diagonal direction.

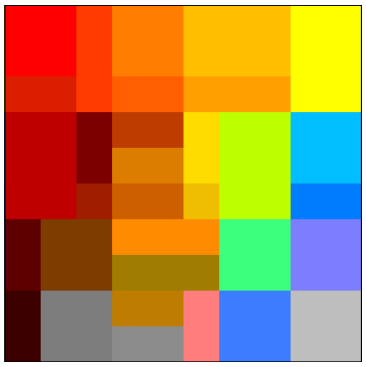
For radial gradients, we define the position and radius of a start circle as well as an end circle. In Figure 14b, the center points of both circles both coincide with the center of the rectangle, while the start radius is set to 0 and the end radius to the maximum. The visual result of this gradient is a progression of the interpolated colors in a circular fashion, as if the inner circle "expanded" to the outer circle while varying the color.

Finally, sweep gradients are colored by setting a center point as well as a start and end angle. In Figure 14c, the center point has been set in the middle, and the start and end angles are 0° and 360° respectively. In the end, the colors of the gradient line will vary as the angle of the position of the pixel from the center increases.

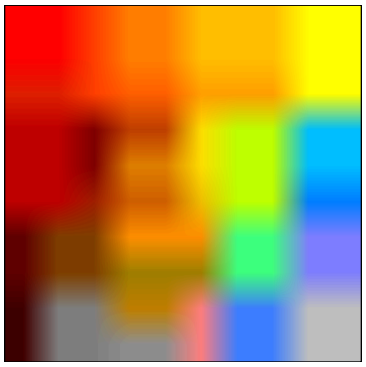
2.10.2 Images

As was mentioned in Section 2.1, it is highly desirable to represent content as vector graphics whenever possible, as it allows for arbitrary scaling without any loss of precision. However, it is clear that this is not always possible, because many objects cannot be represented as vector graphics, like for example images taken with a camera.

(a) The input image.



(b) Nearest-neighbor interpolation.



(c) Bilinear interpolation.

(d) Bicubic interpolation.

Figure 15: The shape of a rectangle filled with an image after applying a scaling factor of 50 by using nearest-neighbor, bilinear and bicubic interpolation.

A fundamental difficulty of rendering images in 2D graphics is that the input image might not have the same resolution as the rendered image. For instance, if an input image has a resolution of 1000x800 pixels but our output display has a resolution of 1350x1080, we need to apply a scaling factor of 1.35 to the image for it to render correctly. To do this, we need to *resample* the image to determine what color each pixel on the display should be to faithfully reproduce the original image at the higher resolution. In order to achieve this, three methods are commonly used: *Nearest-neighbor interpolation*, *bilinear interpolation* and *bicubic interpolation* [14, p. 87-89]. Figure 15 contrasts the different scaling methods using the 10x10 pixels input image in Figure 15a scaled by a factor of 50.

In the case of nearest-neighbor interpolation, the algorithm is straight-forward: It simply calculates the position corresponding to the new pixel in the old image by multiplying it with the inverse scale and copies the color value of the closest pixel. The result in Figure 15b shows that by using this interpolation method, the “block-like” structure of the original input image is preserved. In certain cases, this can be a desirable property, but in many cases, this interpolation method can cause artifacts and is therefore not often used [14, p. 88]. The main advantage is that it is computationally very cheap.

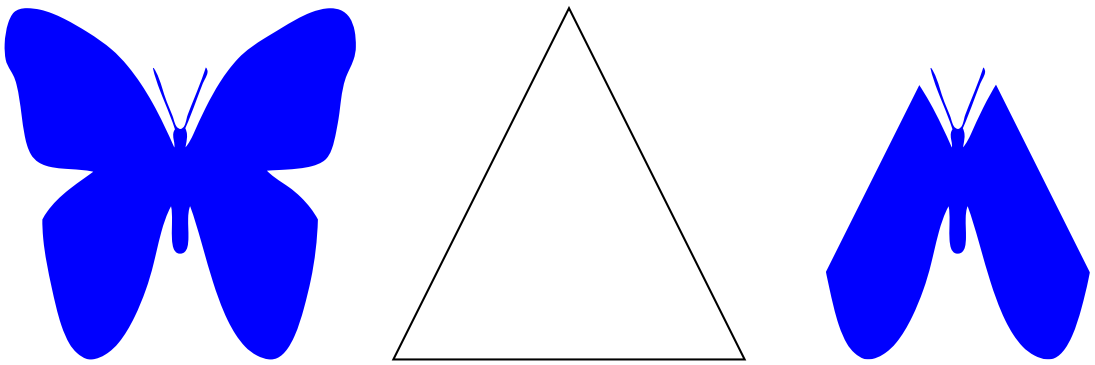
When performing bilinear interpolation, we do not only consider a single nearest neighbor, but actually the four nearest neighbors. We then assign a weight to each neighbor based on the location we are sampling and perform a weighted averaging across those

4 pixels. A similar concept applies to bicubic interpolation, with the only difference that we consider 16 neighbors instead [14, p. 88]. The results for bilinear interpolation can be observed in Figure 15c, where the boundaries appear smoother due to the averaging process. At first glance, the bicubic interpolation in Figure 15d has a very similar effect to the bilinear interpolation in Figure 15c, but when looking at it closer, it becomes apparent that certain areas in the bilinear version appear less smooth than in the bicubic version. However, the cost for the slightly better quality is a much higher computational demand per pixel.

2.11 Clip paths

Clip paths allow cropping an existing shape to another shape, ensuring that only the intersection of both is painted. While this idea is conceptually simple, actually implementing this behavior in a 2D renderer in an efficient manner is challenging but crucial as clip paths are used very extensively.

The visual effect of applying a clip path is visualized in Figure 16. First, we need an input shape which in our case is the blue butterfly. Next, we define a clip shape that determines which parts of the butterfly should actually be visible. In our case, the clip shape is a simple triangle. As can be seen in Figure 16c, after applying the clip path, only the parts of the butterfly that fall within the area of the triangle will be visible, while all other parts are clipped away and thus become invisible.



(a) The input shape, in this case a butterfly.

(b) The clip shape, in this case a triangle.

(c) The result of the clip operation.

Figure 16: The effect of a clip path illustrated. When applying a clip shape to some input shape, only the areas of the original shape that fall within the clip area will be visible.

Chapter 3

Implementation

In this section, we will first showcase an example code snippet to demonstrate how Vello CPU's API works. Afterwards, the majority of the section will be dedicated to explaining each part of the rendering pipeline. Finally, we will illustrate how the pipeline was accelerated by employing SIMD and multi-threading and explain the algorithm used for clip paths.

3.1 API

Listing 1 provides a small example that demonstrates how to use Vello CPU's Rust API for basic rendering.

In the beginning, the user can specify settings that should be applied while rendering. The `level` property indicates which SIMD level should be used during rendering. By default, Vello CPU will dynamically detect the highest available level on the host system, but the user can override this to, for example, force using SSE4.2 instructions, even though the system supports AVX2. The `num_threads` property allows to enable multi-threaded execution (see Section 3.11) or enforce single-threaded execution by setting the value to `0`. Finally, the `render_mode` property indicates whether to prioritize speed or quality during rasterization by using `u8` or `f32` numbers, respectively (see Section 3.8).

The user then creates a new `RenderContext` by specifying a width and height as well as the render settings. An important property of the `RenderContext` is that it is *reusable*. This means that if the user renders a certain scene and then wants to render a second scene, they can just *reset* the context and then reuse it instead of having to initialize a new one. This is important because during the rendering process, *Vello CPU* needs to make a number of memory allocations which can have an impact on runtime. By reusing the same `RenderContext`, Vello CPU can reuse the existing memory allocations in subsequent rendering operations, leading to better performance. This property is especially useful in the context of GUI rendering, where it is common to render the same scene dozens of times per second over an extended period of time.

Now, the user can start rendering items to the scene. In this specific example, we first render a centered rectangle of width 50 using a fully opaque blue, followed by a smaller, semi-transparent red rectangle in the bottom right. In the end, we draw a green stroked

```rust
let settings = RenderSettings {
    level: Level::new(),
    num_threads: 0,
    render_mode: RenderMode::OptimizeSpeed,
};

let mut ctx = RenderContext::new_with(100, 100, settings);

// Draw different shapes in various colors.
ctx.set_paint(BLUE);
ctx.fill_path(&Rect::new(25.0, 25.0, 75.0, 75.0).to_path(0.1));
ctx.set_paint(RED.with_alpha(0.5));
ctx.fill_path(&Rect::new(50.0, 50.0, 85.0, 85.0).to_path(0.1));
ctx.set_paint(GREEN);
ctx.stroke_path(&Circle::new((50.0, 50.0), 30.0).to_path(0.1));

// Flush all existing operations (only necessary for multi-threaded
rendering).
ctx.flush();

let mut pixmap = Pixmap::new(100, 100);
// Render the results to a pixmap.
ctx.render_to_pixmap(&mut pixmap);

// Encode the pixmap into a PNG file.
let png = pixmap.into_png().unwrap();
```

Listing 1: Example usage of Vello CPU's API.

circle. For multi-threaded rendering, we also need to make a call to the `flush` method in the end, whose purpose will be explained in Section 3.11.

By specifying drawing instructions in the render context, we are not doing the full rendering yet. In order to produce an actual picture, we first need to create a new `Pixmap`. A pixmap is basically a "fancy" wrapper around a `Vec<u8>` (a vector of bytes in Rust) that can store raw, premultiplied RGBA pixels. Remember that in Section 2.5, we explained that RGBA values can either be stored as `f32` or `u8`. In this case, we are simply storing the value of each pixel in the pixmap as integers in row-major order. For example, if we have a pixmap of size 2x2 where the top-left pixel is green, the top-right pixel red, the bottom-left pixel blue and the bottom-right pixel white, we would store this in one contiguous vector containing the bytes `[0, 255, 0, 255, 255, 0, 0, 255, 0, 0, 255, 0, 255, 255, 255, 255]`.

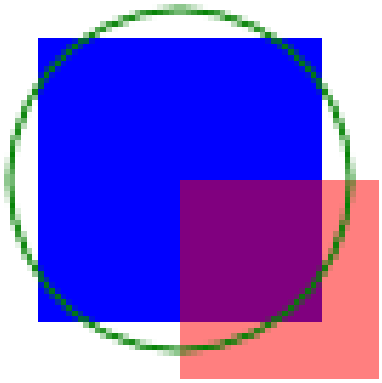


Figure 17: The rendered result of the code in Listing 1.

By calling the `render_to_pixmap` method, we instruct Vello CPU to render the previously recorded rendering instructions into the pixmap. The pixmap can then be further processed by, for example, converting it into a PNG file and saving it to disk. The rendered result of Listing 1 can be seen in Figure 17.

3.2 Architecture overview

In order to convert a vector image into a raster image, there are various intermediate steps to arrive at the final result. While a few steps are virtually universal and done by most renderers in a similar form, the exact implementation of the whole rendering pipeline varies a lot across different implementations.

An interesting aspect of the design of Vello CPU is that it has a highly modular architecture where each step takes a specific kind of data structure as input and produces some intermediate output which can then be consumed by the next stage in the pipeline. There are no cross-dependencies between non-consecutive stages. As will be seen in the following subsections, this property makes it easy to explain the functionality of each module in isolation and visualize the intermediate results that are produced after passing through each stage. The stages of the rendering pipeline and their dependencies are illustrated in Figure 18. A summary of each stage is provided below, and more details on them will be given later on.

The pipeline is illustrated in Figure 18. Overall, the steps can be grouped into three categories: *Path rendering*, *coarse rasterization* and *rasterization*.

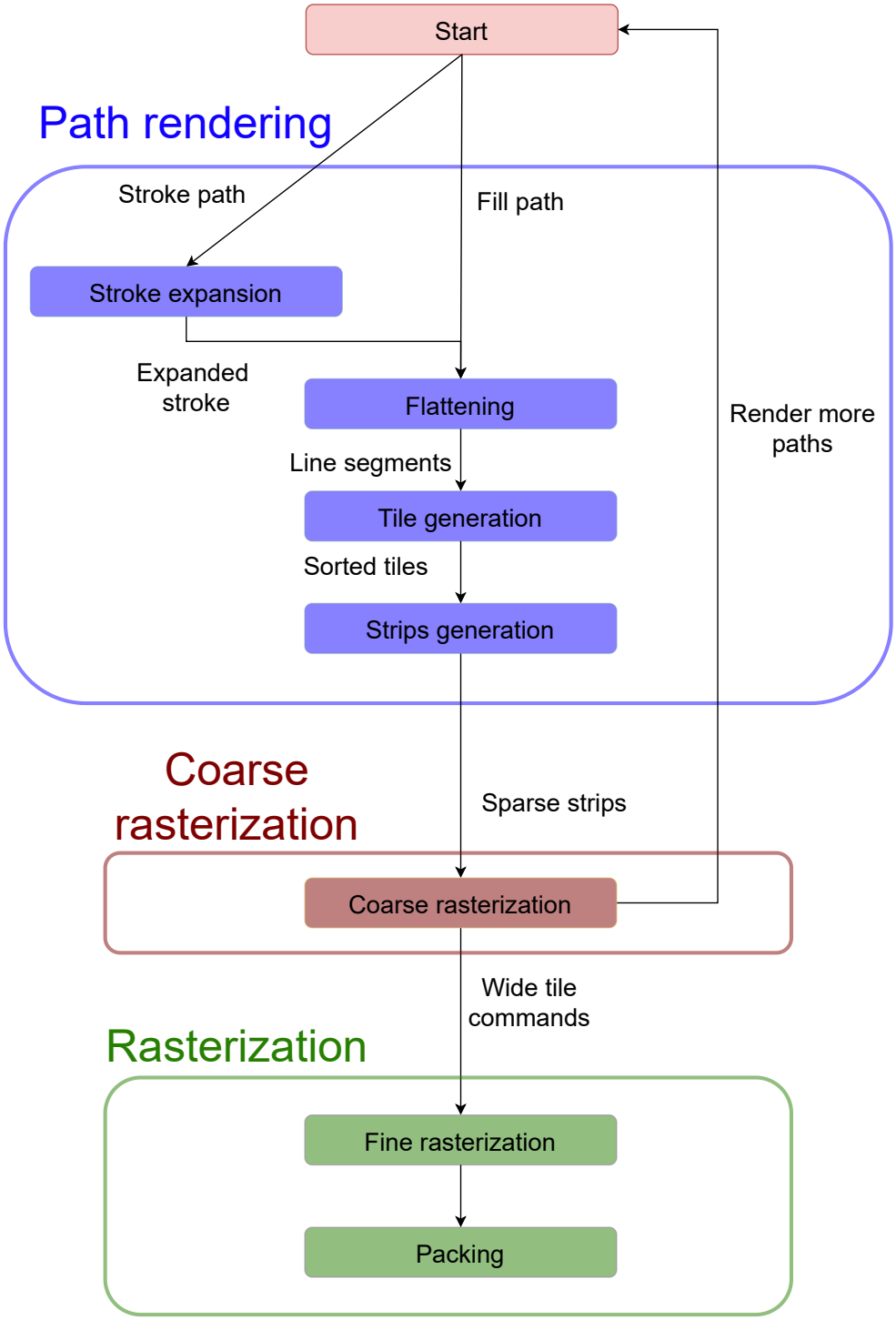


Figure 18: An overview of the rendering pipeline in Vello CPU.

The user starts by creating a render context and specifying drawing instructions, such as filling a rectangle or stroking a circle. The only difference between stroking and filling is that stroking requires an additional step called *stroke expansion* in the beginning, which manipulates the input path in such a way that filling the area of the new path has the same visual effect as stroking the old one. By doing this, we can treat filled and stroked paths the same in subsequent stages.

The next stage is called *flattening*. As was mentioned in Section 2.2, the two basic primitives used for drawing are lines and curves. As part of this stage, the path is converted into a different representation such that the whole geometry is solely described by lines. Converting curves to lines is naturally a lossy operation since curves inherently cannot be represented by just lines, but by using a sufficiently large number of lines, the error in precision will be so small that it cannot be noticed in the final result. The advantage of this conversion step is that the follow-up stage in the pipeline only needs to process line primitives instead of curves *and* lines, which greatly reduces the complexity.

After that, *tile generation* is performed. In this stage, we conceptually segment the whole canvas into areas of size 4x4. For each path, we calculate all of the areas that it crosses, generate a 4x4 tile for it and associate it with the given line. In the end, all generated tiles are sorted by their y and x coordinates.

The following *strips generation* stage is arguably the most crucial step in the pipeline: In this stage, we iterate over all tiles in row-major order (this is why tile generation has a sorting step in the end) and merge horizontally adjacent tiles into strips. For each pixel covered by a strip, we calculate the alpha value that is necessary to perform anti-aliasing. The output of this stage is a vector of strips that sparsely encodes the rendered representation of the shape by only explicitly storing anti-aliased pixels and representing filled areas implicitly.

Once the path has been converted into sparse strips representation, they are passed to the *coarse rasterizer*. The coarse rasterizer segments the canvas into so-called *wide tiles* (not to be confused with plain tiles from the tile generation stage) which all have a dimension of 256x4 pixels. We then iterate over all strips and generate the actual rendering commands that should be applied to each wide tile.

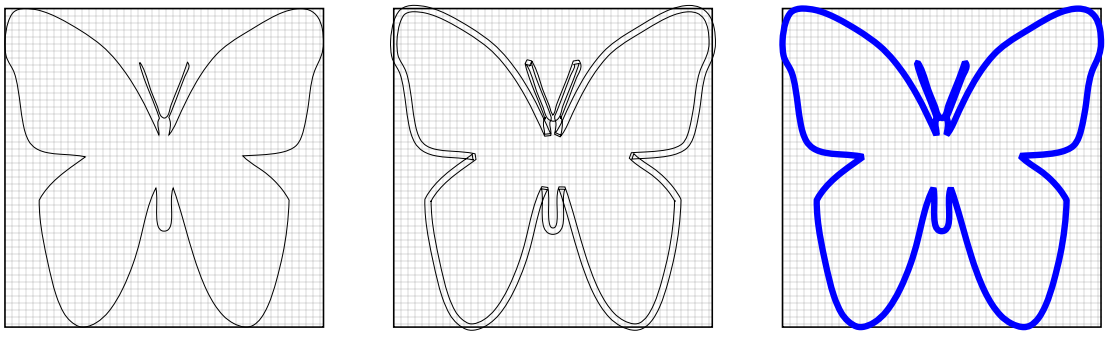
Once this is done, control is passed back to the user, in which case they have two choices: Either, they keep rendering more paths, in which case the above-mentioned stages are re-run for the new paths, or the user calls the `render_to_pixmap` method, in which case *fine rasterization* kicks off.

Fine rasterization is where the actual coloring and compositing of pixels happens. This works by iterating over all commands in each wide tile and performing operations like “fill this area using a blue color” or “fill this area using a gradient color”. For reasons that will be explained later, all of this happens in an intermediate buffer instead of the user-supplied pixmap. Once fine rasterization is complete, we start the process of *packing*, where the contents of the intermediate buffer are copied into the pixmap that the user provided. In the end, the fully rendered result is stored in the pixmap and can be processed further by the user.

3.3 Stroke expansion

In Section 2.3, it was mentioned that there are two types of drawing mode that need to be implemented: Stroking and filling. There are different ways in which stroking can be implemented. In our case, we decided to use the so-called *stroke expansion* approach. When doing stroke expansion, the original path description is transformed in such a way that filling the areas of the newly extended path has the same visual effect as stroking the outline of the original path. The idea is illustrated in figure Figure 19. If we want to draw a stroke with width 1 pixel, we compute two new *offset curves* that are each offset by half the stroke width inward and outward. Filling this area of the new path results in a painted line that is centered on the original outline description of the shape.

The main advantage of the stroke expansion approach is that it allows us to treat filled and stroked paths in the exact same way throughout the whole rendering pipeline, allowing for further simplification. The only difference is that stroked paths go through an additional step in the beginning of the pipeline.



(a) The outline of the butterfly shape.

(b) Expansion of stroke with width 1.

(c) Filling of the expanded path.

Figure 19: Reducing the problem of stroking a shape to the problem of filling its expanded version.

Given that our renderer is implemented in Rust, we decided to use the Kurbo[12] library for this purpose, as it provides an implementation for stroke expansion. It takes a path description, the settings of the stroke as well as a *tolerance* parameter as input and returns the expanded version of the stroke as the output. The *tolerance* parameter represents a trade-off that needs to be carefully balanced: Creating a mathematically completely accurate offset curve is not always possible. If we choose a lower tolerance, the expanded stroke will be more accurate at the cost of containing more path segments, which ultimately means more work for later parts of the rendering pipeline. If we choose a higher tolerance, the expanded stroke will contain fewer path segments, but the chance of visual artifacts being introduced is higher. In our case, we settled on a tolerance of 0.25, which has so far shown to be sufficient.

As outlined in D. F. Nehab [15], stroke expansion is a complex and mathematically challenging problem. One of the main difficulties is handling the various edge cases correctly, some of which even many of the mainstream renderers do not. Since implementing the stroke expansion logic was not part of this project, we will not dive into the internals of Kurbo's algorithm and leave it at this high-level description to give an intuition.

3.4 Flattening

Now that we have an expanded version of our stroke or a shape the user wants to fill, we reach the next step in the pipeline: flattening. The idea of having this step is as follows: We want to convert our input shape, which consists of lines and curves, into a new representation which consists of just lines. By doing so, later steps in the pipeline only need to consider line segments as the basic building blocks, an assumption that will largely simplify the logic.

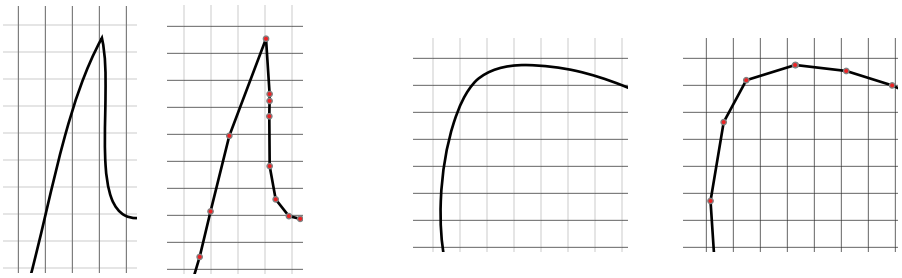


Figure 20: Parts of the butterfly shape flattened to lines. The first and third figures represent the original path segments, the second and fourth figures their respective flattened version. The red points indicate the start/end points of the line segments.

[12]https://crates.io/crates/kurbo (accessed on 15.09.2025)

However, line segments usually cannot accurately model curve segments. This can also be seen in Figure 20. While the flattened versions of the shape *overall* still look curvy, zooming in makes it apparent that it is approximated by a number of connected lines. For plain vector graphics, doing such a simplification would clearly be unacceptable. The crucial point here is that the simplification will be barely noticeable once the shape is rendered to pixels, because as part of the discretization process, the information whether a line or curve was used is lost; all that is left is an approximation of pixel coverage in the form of color opacity. And assuming that the number of used line segments is sufficiently large, the change in pixel coverage will be so small that it is unnoticeable to the naked eye.

In order to achieve flattening of curves, we once again resort to the implementation that is provided in the Kurbo library (though as mentioned in Section 3.10, the implementation was adapted to be SIMD-optimized), which is based on an algorithm described in a blog post [16]. Similarly to stroke expansion, the method for flattening takes a path with arbitrary curve and line segments as well as a *tolerance* parameter as input. The tolerance parameter indicates what the maximum allowed error distance between a curve and its line approximation is and represents another trade-off: A smaller tolerance will yield higher accuracy but result in more line segments, higher tolerance will result in less emitted lines but might cause noticeable artifacts when rendering. We once again settled on the value 0.25 for this parameter, which means that the error distance can never be larger than one fourth of a pixel.

The algorithm works roughly as follows: We iterate over each path segment in the input geometry and perform one of the following operations on it.

In the case of lines, there is no additional work to be done and they can just be re-emitted.

For quadratic curves, things start to get more interesting. One possible approach to flattening them would be doing a *recursive subdivision*, which means that we try to build a line between start and end point, and in case the maximum error is too large, we *subdivide* our current curve in the center and perform the same operation recursively on both halves, until the spanned line of all subdivisions is within the given error bound [16]. While this approach works, it has a tendency to generate more lines than necessary, a side effect that emerges from always doing the subdivision at the center of the curve.

The algorithm in the blog post instead presents a different approach that is based on mathematical analysis: We first calculate the number n of line segments that are needed for the subdivision using an integral that is derived from a closed-form analytic

expression, and only then determine the actual subdivision points by subdividing the interval into n equal-spaced points and applying the inverse integral on them [16]. The result will be an approximation that usually uses fewer line segments than the recursive subdivision approach.

For cubic curves, the Kurbo implementation simply first approximates the original cubic curves by quadratic curves and then applies the same algorithm that was outlined above.

3.4.1 Optimizations

As part of doing the profiling, it became apparent that the initial implementation of flattening was often somewhat slow, especially for small shapes with lots of curves. Two optimizations were implemented that, as will be shown in Section 5, lead to a significant decrease in runtime for curve-heavy geometries.

To understand the first optimization, it is important to realize that curve flattening is time-consuming, even if the curve we are processing is actually small, as some of the computations such as estimating the number of subdivisions and actually doing the subdivision need to be performed regardless of the size. The core insight leading to the first optimization is that in the case of small curves, there is a good chance that all of that work is unnecessary as we will simply end up approximating the whole curve by a single line segment, as the maximum distance between the straight line and the curve does not exceed our threshold. Therefore, before starting the flattening process, we first use the position of the start and end points as well as the control points to determine whether it is even possible that the curve exceeds the tolerance threshold. If this is not possible, we just emit one single line segment instead of doing curve flattening in the first place. As will be shown in Section 5, this gives us a considerable edge over the other renderers when drawing shapes with small curves.

The second optimization relates to the method used to estimate the number of quad curves needed to subdivide a cubic curve. The original implementation looked like shown in Listing 2.

Without diving into the exact semantics of this function, note that the function contains a call to `powf`, which is necessary for correctness but also showed up as a significant bottleneck in performance profiles. The core insight for the second optimization is that while the call is necessary for correctness, we do not actually care about the precise fractional result of the call, as the resulting value is always rounded up to the next integer and clamped between 1 and 16. Because of this, a much faster approach is possible where we precompute a lookup table containing the sixth power of all integer numbers between 1 and 16 and determining the right one by simply doing a linear search

```
fn estimate_num_quads(c: CubicBez, accuracy: f32) -> usize
{
    const TO_QUAD_TOL: f32 = 0.1;
    const MAX_QUADS: usize = 16;

    let q_accuracy = (accuracy * TO_QUAD_TOL) as f64;
    let max_hypot2 = 432.0 * q_accuracy * q_accuracy;
    let p1x2 = c.p1.to_vec2() * 3.0 - c.p0.to_vec2();
    let p2x2 = c.p2.to_vec2() * 3.0 - c.p3.to_vec2();
    let err = (p2x2 - p1x2).hypot2();
    let n_quads = ((err / max_hypot2).powf(1. /
6.0).ceil() as usize).max(1);

    n_quads.min(MAX_QUADS)
}
```

Listing 2: The original function for estimating the number of quad curves for subdividing a cubic curve.

to find the first value that is less than or equal to `err / max_hypot2`. Implementing this optimization completely eliminated the `powf` bottleneck and lead to a significant speedup in nearly all benchmarks that involved curves.

3.5 Tile generation

After converting the shape into flattened lines, the next step is tile generation. The main purpose of tile generation is to determine the pixels on the canvas that could *potentially* be affected by anti-aliasing. A pixel *can* have anti-aliasing if and only if a line crosses that pixel, as anti-aliasing is a correction procedure that is only triggered by the edges of shape contours.

In order to do so, we conceptually segment our drawing area into smaller sub-areas of size 4x4 pixels. Note that there is no inherent reason why we have to choose this specific size and we could also opt to choose a size like 2x2 or 8x8 pixels instead. However, there is a trade-off to balance here which will be elaborated further in Section 3.6.

Then, we iterate over all lines in our input geometry and generate *one* tile for each area (indicated by a blue square in Figure 21) that the line covers. We do this by first calculating the bounding box of the line in tile coordinates, which in this case encompasses 7x3 tiles. Then, we iterate over them in row-major order and calculate whether the line has any intersection point with the tile area. If so, we generate a new tile at that location. If not we, just ignore the location and proceed to the next location.

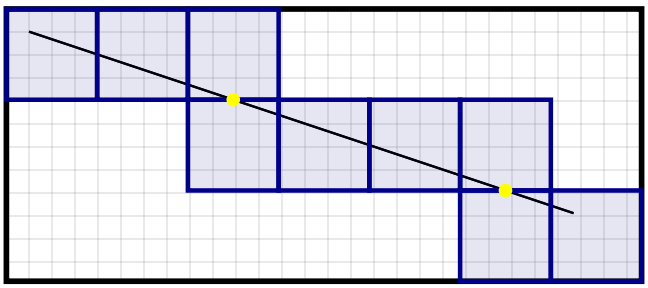


Figure 21: Generating tiles for a line. Yellow points represent the locations where the line intersects the top of a tile.

```
struct Tile {
    x: u16,
    y: u16,
    line: Line,
    has_winding: bool
}
```

Listing 3: The information stored inside of a tile.

As can be seen, tiling is very similar to the concept of conservative rasterization [17], where the goal is to conservatively determine which pixels are covered by the shape. The main difference lies in the fact that we are dealing with line primitives instead of triangles, and the calculation is based on 4x4 tile areas instead of individual pixels.

The information that is stored for each tile is shown in Listing 3. One the one hand, we store the x and y coordinates (in tile coordinates instead of pixel coordinates), but on the other hand, we also keep track of the line associated with the tile. We also store a special boolean flag `has_winding` which will be activated for any tile where the line intersects the top part, as indicated by the yellow points in Figure 21. This information will be needed later.

Applying this algorithm to our familiar butterfly shape, we end up with the representation in Figure 22. There are two aspects worth highlighting: First and foremost, it is *not* the case that one location can only have one tile. We generate one tile for *each* line at a certain location, meaning that multiple tiles can be generated if multiple lines cover the same tile square. And secondly, note how this representation really achieves our initial goal: Any pixel that could potentially have anti-aliasing is strictly contained within a tiled region. Any area that is not covered by a tile is either strictly within the shape, and thus will always be painted fully, or strictly outside of the shape and should therefore not be painted at all.

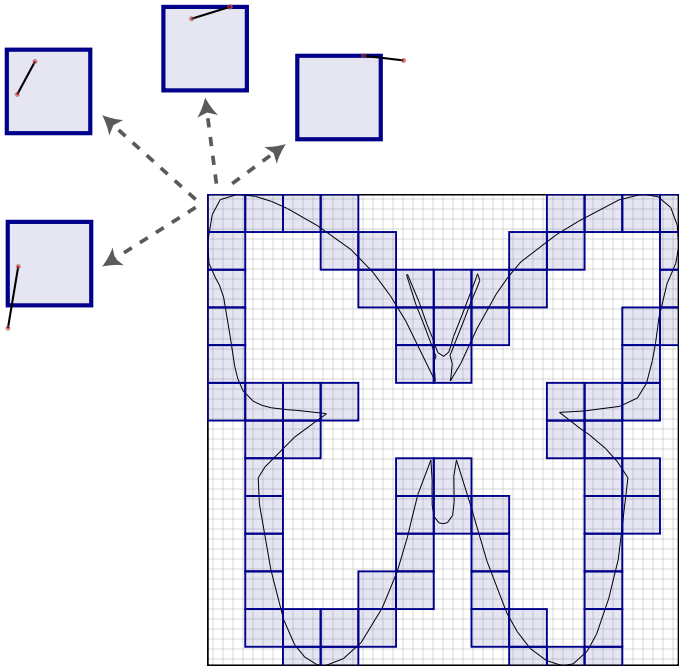


Figure 22: The generated tiles for the butterfly shape. Each line segment generates at least one tile, and there can be multiple overlapping tiles at the same location.

As a final step, we sort our buffer that stores all tiles first by ascending y-coordinate and then by ascending x-coordinate to ensure that the tiles are stored in row-major order from top to bottom. In order to do so, we use the `sort_unstable` method provided by the Rust standard library.

3.6 Strips generation

We now arrive at the most integral part of the pipeline, namely strip generation. There are a lot of details and subtleties to unpack here, and it is therefore worth re-exploring the original motivation before diving into the implementation.

3.6.1 Motivation

When rendering a shape, there are three types of computations that may be performed:

1. For pixels strictly inside of the shape, not a lot of work needs to be done. We just need to set the value of the pixel to the given color. This is computationally speaking relatively cheap.
2. For pixels strictly outside of the shape, nothing should be done at all.
3. For pixels on the edge, more work needs to be done: We need to check how the line intersects the pixel and calculate the area coverage in order to determine the correct

opacity for anti-aliasing. This is a much more expensive operation and we therefore want to perform it only for pixels where it is really necessary.

One of the core goals of strip generation is to calculate the opacity values for all pixels that could potentially be affected by anti-aliasing. However, in addition to that, we also group the pixels in a smart way and store additional "metadata" such that all the information that is needed to reproduce the original shape for rendering (which includes the opacity values for anti-aliased pixels as well as information about which areas should be completely filled and which ones should not) can be easily inferred later on. The key innovation of sparse strips is to do that in a very storage-efficient way by only storing anti-aliased pixels explicitly while representing filled and blank areas implicitly.

3.6.2 Merging tiles into strips

The first step of the strip generation algorithm is relatively straight-forward: We iterate over all tiles (remember that they are already sorted in row-major order), and merge horizontally-adjacent tiles into single *strips*. These strips have the same height as the tiles, but the width can vary depending on how many adjacent tiles there are, as is visualized in Figure 23. The blue-colored strips represent any area where we will explicitly calculate opacity values for anti-aliasing. Any pixel not falling within a strip will *not* be explicitly stored.

3.6.3 Calculating strip-level winding numbers

Generating the merged strips is only a small part of the overall process. The next step is understanding how we can encode necessary information so that in later stages of the pipeline, we can easily determine which areas between two strips on the same row

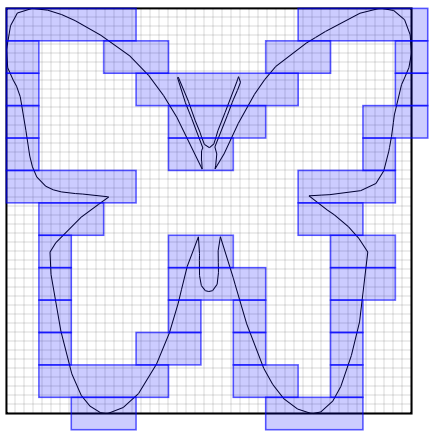


Figure 23: The areas of the generated strips, marked in blue.

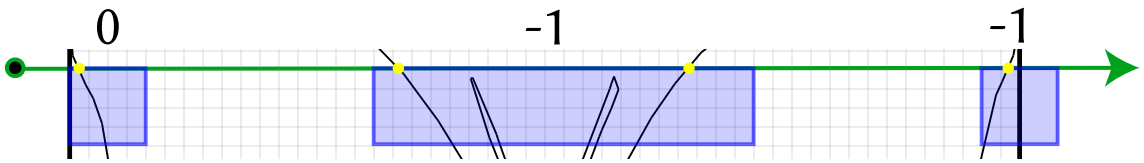


Figure 24: Calculating the strip-level winding number for each strip on the third row of the butterfly in Figure 23. The yellow points indicate the intersection points of the lines with a tile area.

should be filled and which ones should not. In order to understand how this works, we need to remind ourselves about the concept of winding numbers as they were introduced in Section 2.4: Conceptually, we shoot a ray into any direction and increase or decrease a winding number counter each time we intersect a line of the path depending on the direction of the line. In our case, there are actually two different winding numbers that we need to keep track of.

The first winding number is the *strip-level winding number*. At the top of each strip row, we conceptually "shoot a ray" from the very left to the very right. Each time we encounter a tile that intersect a line (remember that for each tile, we store a boolean flag with that information, allowing us to easily check that), we account for this while calculating the winding number of the *next* strip.

Figure 24 shows an example of doing this computation. The first strip on the very left has a start winding number of 0, as it is the leftmost part of the shape on this row. Inside of this strip, we have one tile that has a line intersection at the top. The line is defined from bottom to top, meaning that from the perspective of our imaginary ray, the intersection direction is right-to-left, and thus we *decrease* our winding counter to −1. Since there are no further intersections in this strip, the strip-level winding number of the *next* strip will be set to −1. That strip has a left-to-right intersection in its first tile area, meaning that our winding counter is temporarily reset to 0. But in the end, we have yet another intersection in the opposite direction, so we set our counter back to −1. As those are all intersections in that strip, the start winding number of the last strip in the row is going to be −1 as well.

We run this computation for all rows containing strips to assign each strip its winding number. To more easily visualize this, we can now color the strips according to the fill rule (in our case the non-zero fill rule): If the winding number of a strip is zero, we color it in green, otherwise we color it in red. The result is illustrated in Figure 25. Note in particular that just storing a simple boolean flag in each strip indicating whether the area to its left should be filled or not according to the fill rule is enough to later on reproduce

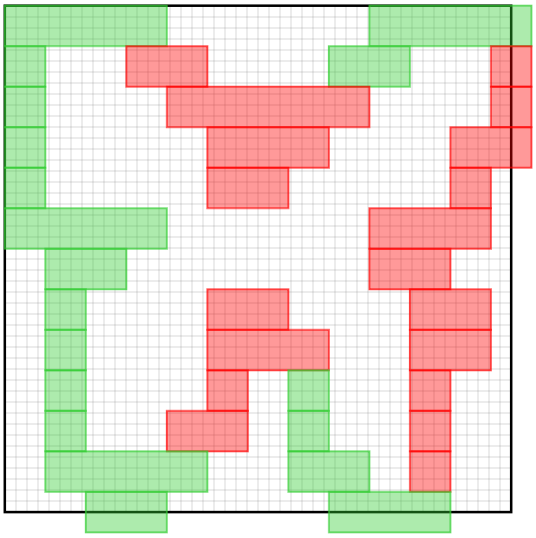


Figure 25: The areas of the generated strips, with the strips painted according to their winding number. Green areas indicate that gaps on the left should not be painted according to the fill rule, while red areas indicate that gaps on the left should be filled.

that information. For every non-covered gap, if the strip on the *right* side has a non-zero winding number (i.e. is red in the figure), the whole area is painted, otherwise it is not painted. For example, the gap in the first row in Figure 25 will not be painted since the strip on the very right has a winding number of zero. However, in the third row, both areas will be painted since the strips on the right of each gap have a non-zero winding number. Mentally applying this idea to each row, it becomes evident that this approach is sufficient to later on determine which areas need to be painted, solely based on the encoded information in the strips.

3.6.4 Calculating pixel-level winding numbers

We now have encoded the information necessary to determine fully-painted areas in later stages of the pipeline, but we have yet to determine the opacity values of the pixels *inside* of strips to apply anti-aliasing. In principle, we use a very similar "ray-shooting" approach that we used to determine the strip-level winding number, with the main difference being that we are now considering rays intersecting individual *pixel rows* (instead of strip rows) and also considering *fractional* winding numbers. The process is visualized in Figure 26 on the basis of the first strip in the first row.

For each strip, we once again look at its constituent tile areas. We initialize a temporary array of 16 floating point numbers (one for each pixel in a 4x4 tile area) to the winding

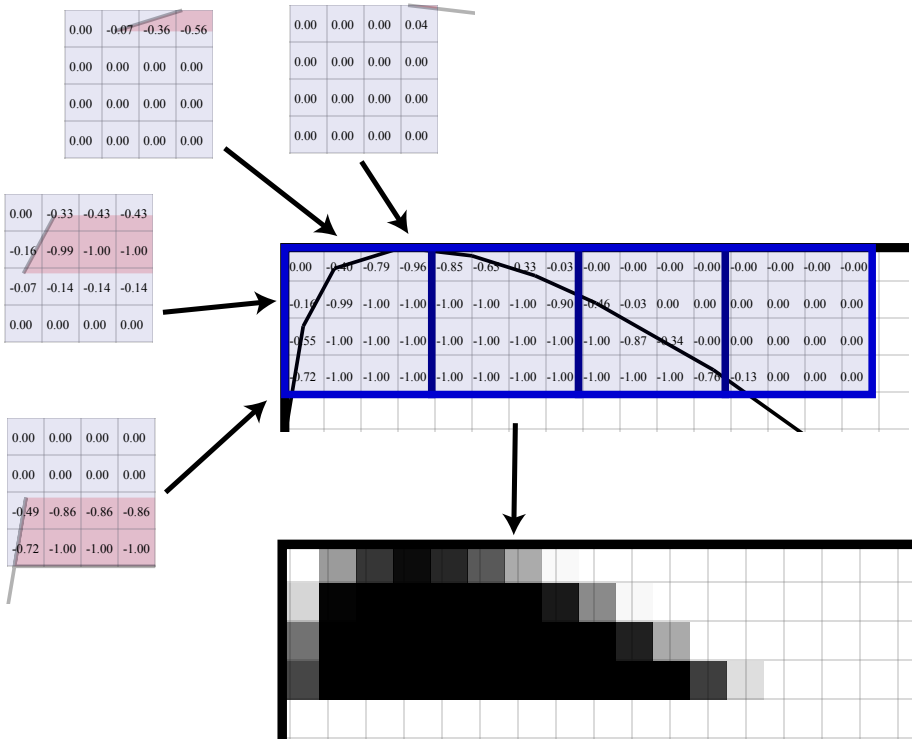


Figure 26: Calculating winding numbers of each pixel in a strip. We first calculate the winding numbers for each tile, and then sum the result to calculate the total winding number of each pixel in the whole tile area. Red areas represent the trapezoid spanned by the line.

number of the strip and store them in **column-major** order (the reasoning behind this will be elaborated in Section 3.8). We then iterate over all tiles inside the given 4x4 tile area and compute the trapezoidal area that is spanned between the line and the right edge of the tile (marked in red in the figure). For each pixel, we then calculate the fraction of its area that is covered by the trapezoid. Note that the resulting winding number can also be *negative* depending on the direction the line intersects the pixels with. We then sum the fractional windings of all tiles in the same area into our temporary array, until we end up with the final winding number for each pixel. Next, we convert the winding numbers into opacities between 0.0 and 1.0 by applying the fill rule and scale them by 255 so that we can store them as `u8`. The array of the 16 opacity values is then pushed out into a buffer and the whole process is restarted for the next tile area in the strip, using the winding numbers for the right-most column as the basis. Doing this for all strips and visualizing the calculated opacities, we end up with the representation that is shown in Figure 27.

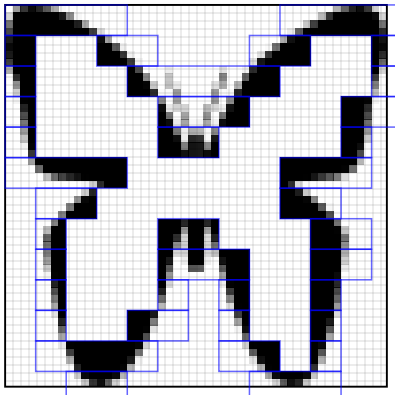


Figure 27: The opacity values for all strips. Completely black pixels represent 100% opacity, white pixels 0%, shades of grey intermediate values.

By now, we have all the information needed to fully draw the complete shape. We have calculated the opacity values of all anti-aliased pixels and represent to-be-filled areas in an implicit way, just by storing a `Vec<Strip>` that represents the whole rasterized geometry of a single shape. As can be seen in Listing 4, a strip only needs to store its start x and y positions, whether the area to its left should be filled depending on the winding number and active fill rule as well as an index into the global alpha buffer containing the opacities of each pixel in the strip. Note in particular that there is no need to explicitly store the width of the strip, as it can be inferred by looking at the `alpha_idx` of the next strip. For example, if one strip has an alpha index of 80 and the next strip an index of 160, the width of the strip is $\frac{160-80}{4} = 20$ pixels, since a strip always has a height of 4.

With the sparse strip representation of the path generated, the path rendering stage of the pipeline (as shown in Figure 18) concludes. The unique aspect of sparse strips is that the representation is very versatile and can be processed in multiple ways. On the one hand, we can pass it on to the next stage to commence the actual rasterization process and render it onto the screen.

```
struct Strip {
    x: u16,
    y: u16,
    alpha_idx_fill_gap: u32,
}
```

Listing 4: The information stored inside of a strip. The `alpha_idx_fill_gap` represents a bit-packed field where the first 31 bits are used to store the index into the alpha buffer that contains the opacity values for each pixel in the strip, and the last bit represents a boolean indicating whether the sparse area to the left of the strip should be filled.

On the other hand, we could for example use it for caching purposes: Imagine that we want to draw a GUI at 60 frames per second. Over the course of time, it is likely that the vast majority of the GUI remains static and only small parts change each frame. Because of this, we can opt to compute the sparse strips representation of elements like buttons and text fields during the first run and cache them, so that in subsequent runs we do not have to rerun the whole path rendering stage for those elements, saving a lot of time. The usual approach for caching would be to render the elements to a bitmap image with a given size, but by virtue of having this intermediate representation, we can achieve the same goal in a much more memory-efficient way.

3.6.5 Tile size

Initially, it was claimed that the point of strip rendering is to only calculate the opacities of *anti-aliased pixels*. However, looking at Figure 27, it becomes apparent that this is not entirely true: There are many pixels that either have full opacity or no opacity at all and therefore do not require any anti-aliasing at all. They are only included by virtue of being in the *vicinity* of anti-aliased pixels and being part of the same tile.

In principle, it is very much possible to use different tile sizes, as is shown in Figure 28. However, both increasing and decreasing the tile size come with their own caveats. In the case of 8x8 pixels, we have fewer tiles which means lower overhead when generating and sorting them. However, the disadvantage is that our tiles cover *many more* non-anti-aliased pixels, implying higher memory requirements and also many more pixel-level anti-aliasing computations that need to be performed, which are expensive. Using tile sizes of 1 and 2 on the other hand *reduce* the number of performed anti-aliasing computations, but the downside is that the bottleneck will instead shift toward tile generation and sorting, as we will need to generate many more tiles. A tile size of 4 is a good balance; the tiles are not too large and we therefore do not need to perform too many unnecessary anti-aliasing computations and reduce the sparseness of the representation, but also not too small, resulting in reasonable performance during tile generation and sorting. In addition to that, as will be demonstrated in Section 3.10, a tile size of 4 hits the sweet spot for efficient SIMD optimizations.

3.6.6 Sparseness

With a good understanding of how sparse strips work, it is worth analyzing the sparseness property in more detail. One area the sparse strips intermediate representation is particularly useful for is path caching. As was already briefly mentioned, in traditional renderers, the usual way to implement patch caching is to render the path into an intermediate bitmap image and then just paint the whole image when reusing it in the

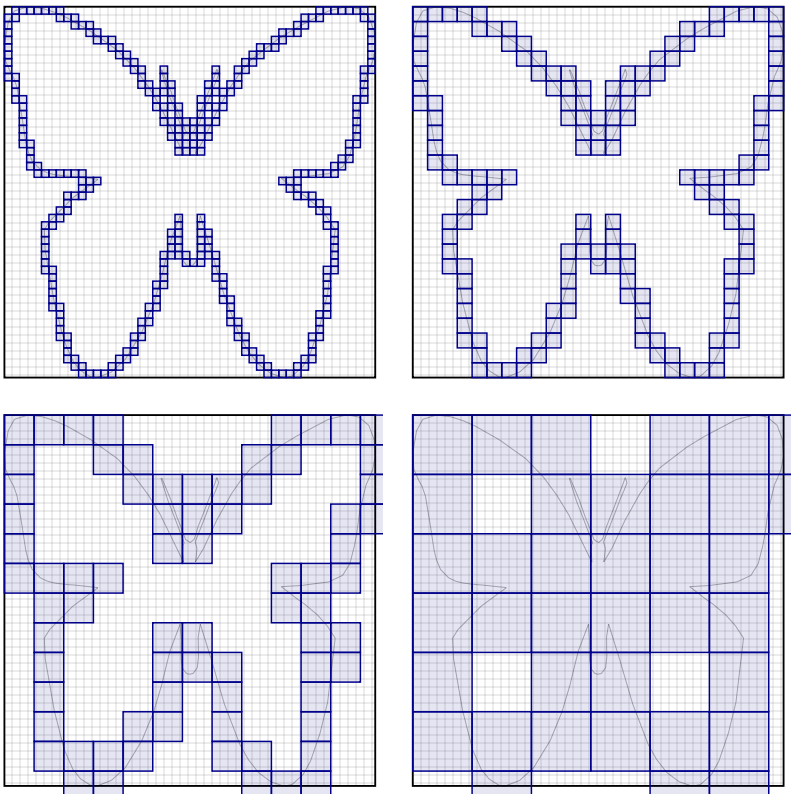


Figure 28: The butterfly processed with tile sizes 1, 2, 4 and 8 from top-left to bottom-right.

future. In particular, this requires us to explicitly store the value of each pixel covered by the shape. If we were to cache the butterfly shape rendered to a 256x256 resolution in greyscale, we would therefore need $256 \cdot 256 \approx 65$KB of storage. For 512x512, we would need around 262KB and for 1024x1024 around 1MB. As can be seen, the storage requirements increase linearly with the overall number of pixels.

This is not the case for sparse strips, because we are only explicitly storing information about pixels at the path boundary, while large filled areas are represented implicitly. According to our calculaions, rendering the butterfly to 256x256 results in the generation of 259 strips and 7296 alpha values. Since a single strip has a memory footprint of 64 bytes and a single alpha value is stored as u8, the necessary storage amounts to around $259 * 64 + 7296 \approx 24$KB. For 512x512, we get 544 strips and 15024 alpha values, resulting in $544 \cdot 64 + 15024 \approx 50$KB. For 1024x1024, we get 1159 strips and 30848 alpha values, resulting in around 105KB of storage, representing a nearly tenfold decrease compared to the 1MB of the traditional caching approach.

```
struct CmdFill {
    x: u16,
    width: u16,
    paint: Paint,
    blend_mode: BlendMode,
    compose: Compose
}
```

```
struct CmdAlphaFill {
    x: u16,
    width: u16,
    alpha_idx: usize,
    paint: Paint,
    blend_mode: BlendMode
    compose: Compose
}
```

Listing 5: The structure of of fill and alpha fill commands.

3.7 Coarse rasterization

Coarse rasterization serves as a preparatory step before actually rendering the shape into a pixmap. We start by conceptually splitting the complete drawing area into into so-called *wide tiles* (to be distinguished from the 4x4 tiles introduced earlier) which always have a dimension of 256x4 pixels. For example, in case we are rendering to a 50x50 screen, we would have $\lceil \frac{50}{256} \rceil \cdot \lceil \frac{50}{4} \rceil = 13$ wide tiles in total. If we increase the width to 312, we instead end up with $\lceil \frac{312}{256} \rceil \cdot \lceil \frac{50}{4} \rceil = 26$ wide tiles.

Each wide tile contains an (initially empty) vector of commands that represent rendering instructions. As can be seen in Listing 5, we distinguish between two main commands: *Fill* and *AlphaFill* commands.

They are very similar and mostly contain the same fields: The `x` field indicates the horizontal starting position of the command and the width how many pixels it spans horizontally. Note that there is no need to store a `y` or `height` coordinate, as a command always applies to the full height of the wide tile it is stored in. The `paint` indicates how the pixels affected by the command should be painted. It can either be a solid color, or a complex paint like a gradient or a bitmap image. Alongside, we also store which blend mode and composition operator should be used when applying the command.

The only difference between fill and alpha fill commands lies in the fact that alpha fills have an additional `alpha_idx` property. Normal fill commands are used for filling the areas *in-between* strips where there is no anti-aliasing, while alpha fill commands are used for the regions inside of strips which require an additional opacity factor to be applied to the pixels.

We now iterate over the sparse strips representation of our path to generate the appropriate commands for each wide tile. Conceptually, this is not difficult as each row of strips has a 1:1 correspondence to a row of wide tiles. Figure 29 demonstrates this for the second row of strips of our butterfly.

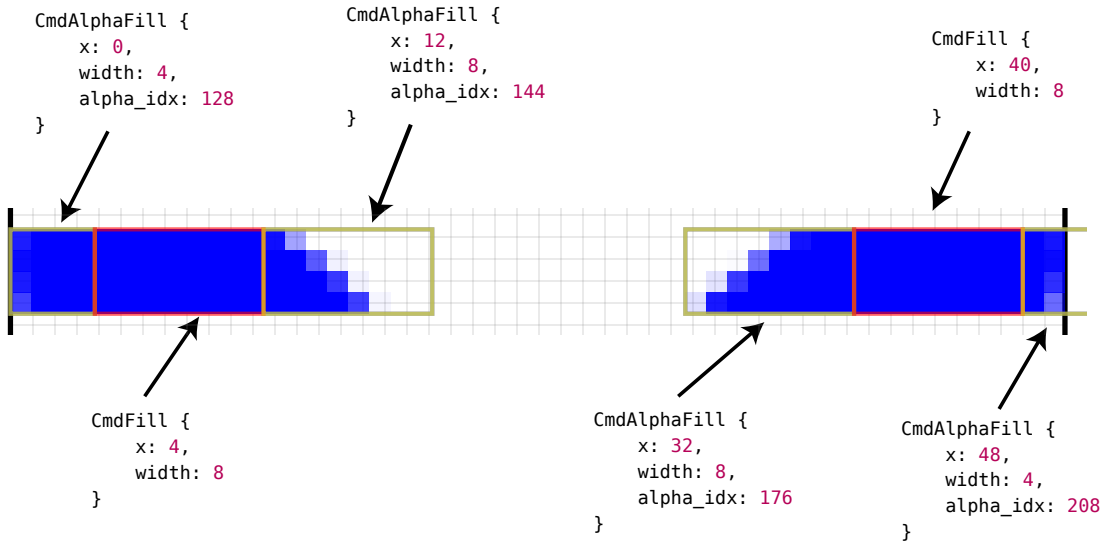


Figure 29: Generating wide tile commands for a row of strips. Green rectangles represent the strips, red ones the implicitly to-be-filled areas. The `paint`, `blend_mode` and `compose` fields have been omitted for brevity.

For strips, we more or less just need to copy the `x` and `alpha_idx` properties of the corresponding strip and calculate the implicitly represented width to generate a new alpha fill command. For the gaps between strips, we proceed as previously outlined: In case the right strip has `fill_gap` set to `true`, we generate a corresponding fill command, as is the case for the gap between strip 1 and 2 as well as strip 3 and 4 in Figure 29. Otherwise, we simply leave the area untouched, like for example the gap between strip 2 and 3.

The whole procedure is performed for all wide tile rows, until all commands have been generated. Once this is done, the first phase of rendering is completed and control is handed back to the user. Either, the user decides to render additional paths, in which case the whole cycle of path rendering is repeated and more drawing commands will be pushed to the wide tiles, or the user decides to finalize the process by kicking off the rasterization process via a call to `render_to_pixmap`.

3.8 Fine rasterization

Given a set of wide tiles that contain drawing commands, the actual rasterization process will now be started. As part of this stage, we calculate the RGBA values of each pixel by iterating over each wide tile and processing them in isolation. Every time we go to a new wide tile, an empty scratch buffer with the dimension 256x4 pixels (i.e. the same

dimension as a wide tile) is created and each pixel is set to the RGBA value (0.0, 0.0, 0.0, 0.0), so a fully transparent color. Note that pixel values are stored in *column-major* order. The rationale for doing so is that it improves cache efficiency, since the pixels that are processed during each command will be stored contiguously in memory.

When processing the commands, we first need to determine the range of pixels that we want to fill, which is straightforward because each command stores that information. For example, if we are processing the third row of wide tiles and encounter a `Fill` command with `x` set to 24 and `width` set to 24, we can infer that all pixels that fall within the rectangle spanned by the points (24, 12) to (48, 16) should be painted with the paint stored in the command. Note that since the height of a strip is always 4, the height of a command is also always 4. The steps that are outlined below then have to be done for each pixel in the area.

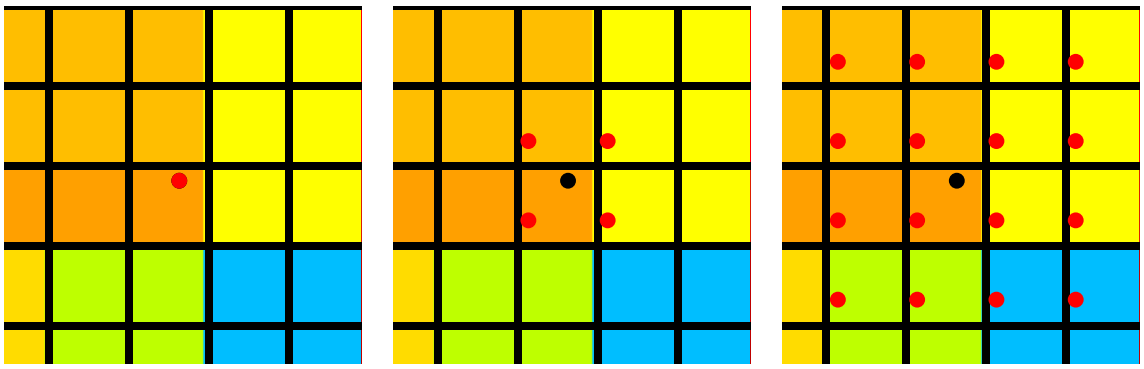
3.8.1 Computing pixel color

The first step in fine rasterization is to calculate the *raw* color value of the pixel, which is determined by the paint that we are using. In the case of a solid color, this is trivial as the pixel just assumes that same color. However, the story is different for image and gradient paints, where the color value depends on the exact location of the pixel.

Image paints

In Figure 15, we illustrated the concept of image paints based on a 10x10 input texture which is scaled up by the factor of 50 to fill a rectangle with the dimensions 500x500. Assume for example that we are currently processing the pixel at the location (385, 110). If we want to determine the color of that pixel using the nearest-neighbor method, we simply need to *reverse* the scaling process and then sample the color value of the closest pixel value in the input texture. Since our scaling factor is 50, we simply divide by that factor, which results in the position $\left(\frac{385}{50}, \frac{110}{50}\right) = (7.7, 2.2)$. As can be seen in Figure 30a, the nearest neighbor of this fractional location is the pixel at the location (7, 2) which is orange. Therefore, the pixel at the location (385, 110) in the new image will also assume the color orange.

The story is a bit different when using bilinear filtering instead. In this case, we sample the color values of the four *surrounding* locations (like in Figure 30b) and perform a linear interpolation of those samples by weighting them based on their exact fractional location within the pixel. The resulting color will then neither be a clean orange or yellow, but instead some intermediate color. Doing this for all pixels achieves the effect of smoothing the edges between pixels with varying colors, as was previously shown



(a) Nearest neighbor. (b) Bilinear filtering. (c) Bicubic filtering.

Figure 30: The different image sample strategies. The black dot (in Figure 30a at the same location as the red dot) represents the original location, the red dots the sampled locations.

in Figure 15c. Bicubic filtering operates on a similar basis, but instead samples the surrounding 16 pixels and then uses a cubic filter for weighting the contributions of each sample based on proximity. The result will be an even smoother blending between pixel edges, as was visible in Figure 15d, at the cost of a higher computational cost.

Gradient paints

As was mentioned in Section 2.10.1, gradients allow us to represent smooth transitions between different colors along a certain trajectory. When doing fine rasterization, we need to calculate the exact position of the pixel along the gradient line so that the color that the pixel should be painted with can be determined. How exactly this is achieved depends on the gradient type.

Linear gradients are defined by a start and end point that define the line used as the basis for the color gradient, as was previously shown in Figure 14a. In order to calculate the parametric `t` value for any arbitrary pixel position, we just need to calculate the intersection of the *perpendicular* line that passes through the pixel position. If the intersection point lies exactly at the start point, the value of t will be zero. If it instead lies at the end point, the value will be one. Otherwise, for any position in-between, the value will be some fractional value between zero and one, as seen in Figure 31.

Radial gradients instead are defined by a start and end circle which each have their respective center points and an associated radius. In Figure 31, both circles are centered in the middle and the radius of the start circle is zero. The visual result is a circle that keeps expanding its radius while constantly changing the color depending on the t value. In order to calculate the t value at any arbitrary pixel position, we simply need

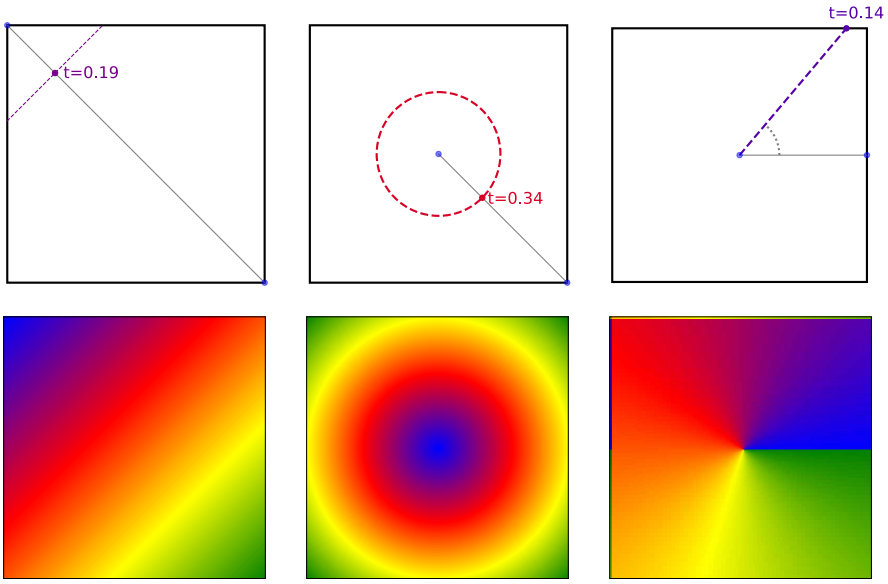


Figure 31: Determining the t-value of a pixel for each gradient type.

to calculate the distance of the pixel to the gradient center and then divide it by the radius of the outer circle, giving us the t value in the range 0.0 to 1.0. Note that the above calculation assumes that the start and end circles are concentric and the start radius is 0, which represents the simplest case. However, there are many other possible variations, in which case the above method is not sufficient anymore. For example, the start radius could be larger than 0 or the center points of the circles could be at different locations. To account for these, we decided to adopt the same approach as Skia[13], where the radial gradient is assigned to a specific category based on its properties and an appropriate formula is then used to compute the value of t for a specific location. However, since the details of this approach are relatively intricate, we will not elaborate them here and leave it at this high-level description.

For sweep gradients, we only need to determine the angle of the pixel in relation to the center point, which can be easily done using the `atan2` function. In our case, similarly to Skia we do not actually use the `atan2` function but instead compute the angle with the help of a polynomial approximation, as this makes it easier to compute the angle for multiple pixels at a time using SIMD (see Section 3.10).

After having determined the t value, we need to calculate the corresponding color value. This is achieved by looking at the two surrounding color stops and doing a linear inter-

[13]https://github.com/google/skia/blob/a38a531dec1de335b5ffdf174e2a97b2d450c8d6/src/shaders/gradients/SkConicalGradient.cpp (accessed on 21.09.2025)

polation based on the position between the two stops. For example, let us assume that we have a blue color stop at the position $t = 0.15$ and a red color stop at the position $t = 0.40$. We now want to determine the color that should be used for the value of $t = 0.33$. We first scale the t value to the range [0.0, 1.0] so that 0.0 stands stands for fully blue and 1.0 for fully red: $\frac{0.33-0.15}{0.40-0.15} = 0.72$. Doing a simple linear interpolation will yield the final color: $0.28 \cdot (0.0, 1.0, 0.0, 1.0) + 0.72 \cdot (1.0, 0.0, 0.0, 1.0) = (0.72, 0.28, 0.0, 1.0)$. A feasible approach would be to run this calculation for every pixel after we have determined its t value, but doing so would be very costly.

Instead, what we do is that we precompute a LUT (look-up table) that contains pre-calculated color values for specific t values between 0.0 and 1.0. Depending of the number of color stops, the number of entries is either 256, 512 or 1024, inspired by the approach used in Blend2D. After calculating the t value of a pixel, we then look up the color value of the entry that is the closest to our t value and use that. By doing so, the necessary work for each pixel is reduced from computing the linear interpolation to a simple memory access, which is significantly faster. The downside is that the color values will not always be a 100% accurate, but thanks to the high resolution of the LUT, the differences will be so small that they are not noticeable in the usual case. Another downside is that we need to do a lot of computations ahead of time, even though not all of them might be needed in the end, but especially for larger-sized geometries this initial overhead is fully eclipsed by the performance improvement that arises from reducing the per-pixel workload.

3.8.2 `Fill` vs. `AlphaFill` commands

Previously, we introduced the distinction between the `Fill` and `AlphaFill` commands, the only difference being that alpha fill commands apply an additional opacity to account for anti-aliasing. To account for that, in case we are processing an alpha fill command we multiply the calculated raw pixel value by the opacity for anti-aliasing before processing it further. Apart from this small additional step, the two commands can be treated the same.

3.8.3 Compositing and blending

After having determined the raw pixel color, we arrive at the core part of fine rasterization that does blending and compositing. If we are only drawing a single shape, we could simply copy the raw pixel values into the scratch buffer and we would be done. However, the key is that a rendering engine needs to account for the fact that multiple shapes with different paints can be overlapping each other. In particular, the

scratch buffer might contain colored pixels from previous drawing instructions which we now need to combine with the generated pixels from our current drawing command according to the rules that were outlined in Section 2.7 and Section 2.8.

Alpha-compositing

The simplest and by far the most common operation is simple alpha compositing, which involves no blending and uses the source-over compositing operator. Due to its prevalence, Vello CPU contains a specialized and highly-optimized code path that handles this specific case. The following formula is used to perform alpha-compositing [12]:

$$\alpha_s \cdot C_s + \alpha_b \cdot C_b \cdot (1 - \alpha_s) \tag{1}$$

α_s stands for the opacity of the source color, C_s for the source color, C_b for the background color and α_b for the opacity of the background color. Note in particular that there are two expressions that require us to multiply the source/background color with their alpha value. This ties to the previous discussion in Section 2.6.1, where it was mentioned that storing color values using premultiplied alpha can save lots of computations, and this formula should make this point more clear. If both the source color C_s (which we just computed) and background color C_b (which is stored in the scratch buffer) are *already* stored using premultiplied alpha, the formula reduces itself to $C_s + C_b \cdot (1 - \alpha_s)$. The original five arithmetic operators were reduced to just three, which leads to a substantial performance improvement given that this calculation is performed for every pixel. Applying this formula results in a new color value which will override the existing value in the scratch buffer to eventually be reused as the new background color in future computations.

It is worth analyzing Equation 1 more closely, as it makes it clear that it is essentially yet another linear interpolation based on the opacity of the source pixel. Assume for example that the background pixel is a fully opaque red, denoted by the tuple (1.0, 0.0, 0.0, 1.0).

First, let us try to compose the background with a fully opaque blue pixel, given by (0.0, 0.0, 1.0, 1.0). Since the α_s is 1.0, the compositing formula reduces itself to just C_s, which matches our intuition that a fully opaque blue should completely override the previously existing red. On the other hand, if we consider a fully transparent blue with (0.0, 0.0, 1.0, 0.0), pre-multiplying this yields just (0.0, 0.0, 0.0, 0.0). Since α_s is 0, the formula changes to just C_b, which also matches the intuition that if we compose a fully transparent pixel on top of another one, it should have no effect at all and the background should still be visible like before. Finally, assuming an alpha value of 0.15, we end up with the

calculation $(0.0, 0.0, 0.15, 0.15) + (1.0, 0.0, 0.0, 1.0) \cdot 0.85 = (0.85, 0.0, 0.15, 1.0)$, also confirming the intuition that the result should have a touch of blue, but overall still be mostly red.

Normal blending and compositing

In case the user specified a different composition operator or blend mode, we cannot use the above shortcut path and instead use a less-optimized code path which can handle all the different modes. The same basic principles that apply to alpha-compositing also apply here, the only difference being that the pixel-level calculations become much more extensive and complex, as many of the assumptions that make simple alpha compositing so efficient to implement do not apply anymore. Since this code path mostly consists of applying the well-documented formulas for the given blend mode and compositing operator [12], the details will not be elaborated here. The final result is the same: A new color value for the given pixel that will be stored in the scratch buffer so that it can be used as the new background color in future compositing operations.

3.8.4 `u8`/`u16` vs. `f32`

As was just shown, the vast majority of the fine rasterization pipeline consists of additions and multiplications of color values. We can choose to run the calculations using either 32-bit floating point numbers that are normalized between 0.0 and 1.0, or 8-bit unsigned integers ranging from 0 to 255.

For example, assume that we have two colors given by their RGBA values:

$$c_1 = (255, 0, 0, 255)$$
$$c_2 = (0, 128, 0, 255)$$

We now want to determine the color that results from linearly interpolating between the two with a t value of 0.3 using the formula $t \cdot c_1 + (1.0 - t) \cdot c_2$.

For `f32`, we first normalize the numbers by dividing by 255, resulting in $c_1 = (1.0, 0.0, 0.0, 1.0)$ and $c_2 = (0.0, 0.5, 0.0, 1.0)$. Next, we do the interpolation: $0.3 \cdot (1.0, 0.0, 0.0, 1.0) + 0.7 \cdot (0.0, 0.5, 0.0, 1.0) = (0.3, 0.35, 0.0, 1.0)$. If desired, we can then scale and round the result back to `u8`, resulting in the value (77, 89, 0, 255).

For `u8`, we instead scale up the fraction and change the formula to $t \cdot c_1 + (255 - t) \cdot c_2$ to perform the calculations using only integers, in the end dividing the result by 255 to normalize the result back to the range of a `u8` number: $\frac{77*(255,0,0,255)+178*(0,128,0,255)}{255} =$

$\frac{(19635,0,0,19635)+(0,22784,0,45390)}{255} = (77, 89, 0, 255)$. It should be noted that we need to use `u16` numbers for intermediate results to prevent overflows.

As can be seen, at least in this particular case both calculations lead to the same result. However, there is a delicate trade-off between the two methods that becomes especially relevant as the number of calculations for the same pixel increases (which will happen if multiple shapes overlap the same pixel). The integer-based pipeline is usually faster as it allows processing more pixels at the same time due to only requiring 8/16 bits instead of 32 bits per channel. We also use an efficient approximation of a division by 255 using a bitshift and addition instead of actually doing the division, as it would otherwise be significantly slower. However, the clear disadvantage is that in contrast to `f32`, we lose more precision when using integers due to quantization. These rounding errors can become bigger as the number of pixel-level calculations increases. As these rounding errors are usually not noticeable to the naked eye, using the `u8`-pipeline is often preferable, but there are use cases where the higher precision can play an important role. Therefore, Vello CPU gives the user the freedom to decide themselves which pipeline to use.

3.9 Packing

After completing fine rasterization, the scratch buffer of each wide tile stores the final RGBA values for each pixel that is covered by the wide tile. However, there are four problems that need to be addressed:

- The wide tiles (and in particular their buffers) are not stored contiguously in memory. However, users usually expect one continuous buffer containing pixel data for the *whole* drawing area.
- Wide tiles have a fixed size of 256x4 pixels, but a pixmap could have a size that is not a multiple of that, for example 53x71 pixels.
- The wide tiles themselves store pixel data in column-major order, while the user expects pixels in row-major order (see Figure 32).
- In case we are running the high-precision rasterization pipeline, the RGBA values will be stored as normalized `f32` values, but the user usually expects `u8` values for the final pixel values, meaning that we must convert them first.

To address these points, there is a final stage called *packing*. As part of this, we iterate over all wide tiles and copy each pixel in the wide tile buffer to the appropriate location in the user-supplied `Pixmap`. Pixels whose position lie outside of the pixmap are simply ignored and not copied. In case the values are stored as f32, we multiply the values by 255 and then round them to the appropriate `u8` values. Once this done, all pixels are stored

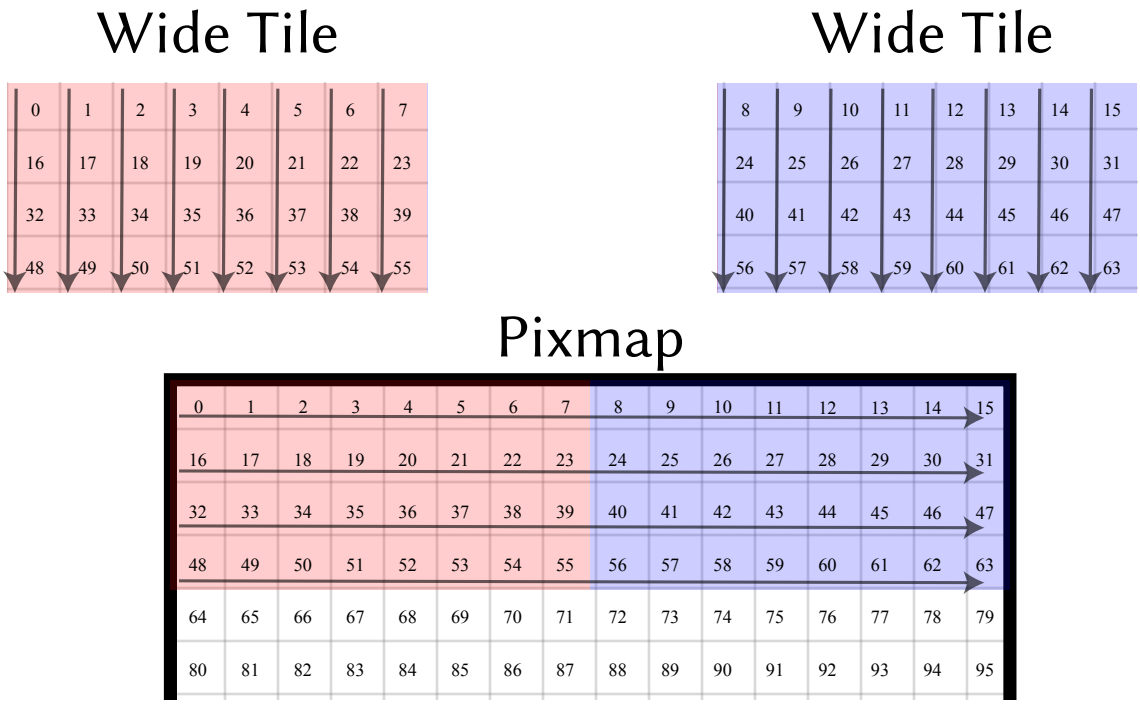


Figure 32: The packing process visualized based on a 16x16 pixmap and wide tiles of size 8x4 (for easier illustration). As indicated by the arrows, when copying the pixels into the pixmap, we need to transpose from column-major to row-major order.

in the pixmap as premultiplied RGBA values and the user can process them further, for example by encoding the image into a PNG file and storing it on disk. The final rasterized version of our butterfly can be seen in Figure 33.

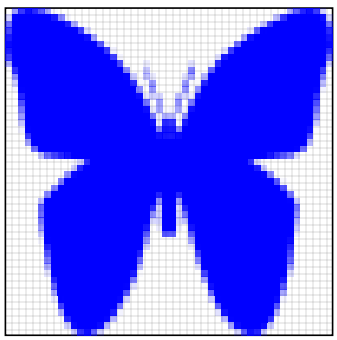


Figure 33: The final rasterized version of the butterfly.

3.10 SIMD

In order to achieve the best performance, implementing SIMD optimizations is indispensable. In many cases, it can be sufficient to rely on auto-vectorization for that purpose, but in our case, this is unsatisfactory for two reasons: Firstly, while the compiler often does a good job at detecting auto-vectorization opportunities, it does not do so 100% reliably and making the intended vectorizations explicit by using SIMD intrinsics is therefore more desirable. Secondly, since the Rust compiler by default needs to produce portable code, it can often only rely on a very reduced set of intrinsics. For example, in order to instruct the compiler to make use of AVX2 features, it is necessary to explicitly enable the `avx2` target feature, as a result of which the compiled code cannot be run on x86 devices that do not support these instructions.

3.10.1 Library

For Vello CPU, our two main goals for the SIMD implementation were:

1. The SIMD code should be written in a target-agnostic way. We wanted to write our code **once** using an abstraction over SIMD vector types and then have the ability to generate target-specific SIMD code on-demand.
2. The implementation should support runtime dispatching, so that we can for example leverage AVX2 instructions on targets that support it, while falling back to SSE4 or even scalar instructions on targets that do not.

There already exist Rust libraries that attempt implementing such an abstraction, like for example pulp[14]. However, after evaluating available options, we came to the conclusion that it would be most advantageous to build our own abstraction, such that we have full control over the architecture and the set of implemented instructions. To this purpose, we[15] built fearless_simd[16], a SIMD abstraction library that exposes abstract vector types such as `f32x4` or `u8x32` as well as functions for arithmetic operations and calls the appropriate SIMD intrinsics behind the scenes. At the time of writing, support is limited to NEON, WASM SIMD and SSE4.2, all of which operate on 128-bit vector types. A fallback level also exists for platforms without SIMD support. Larger vector types (like `u8x32` or `u8x64`) are also supported but will simply be poly-filled multiple 128-bit vectors.

[14]https://github.com/sarah-quinones/pulp (accessed on 11.09.2025)

[15]Other collaborators have contributed to the creation of this library and it was not exclusively built as part of this thesis. As was mentioned in Section 1.1, the main contribution of this thesis was adding the necessary operators to support all usages in Vello CPU.

[16]https://github.com/linebender/fearless_simd (accessed on 11.090.2025)

```rust
trait Simd {
  fn add_f32x4(self, a: f32x4<Self>, b: f32x4<Self>) -> f32x4<Self>;
  fn add_f32x8(self, a: f32x8<Self>, b: f32x8<Self>) -> f32x8<Self>;
  fn add_u8x16(self, a: u8x16<Self>, b: u8x16<Self>) -> u8x16<Self>;
  fn sqrt_f32x4(self, a: f32x4<Self>) -> f32x4<Self>;
  fn sub_u16x8(self, a: u16x8<Self>, b: u16x8<Self>) -> u16x8<Self>;
}
```

Listing 6: A selection of functions defined by the `Simd` trait.

One of the main features of fearless_simd is that it is based on code generation. Writing manual code for all different combinations of arithmetic operators (which often are only distinguished by the different name of the called intrinsic) and vector sizes would be hard to maintain. Because of this, the core logic for generating the intrinsic calls happens in a separate Rust crate using the proc-macro2[17] and quote[18] crates, and the fearless_simd crate then simply contains the output of the auto-generated code.

At the core of fearless_simd is the `Simd` trait which defines the methods for all possible combinations of vector types and arithmetic (or boolean) operators. A very small selection of these is displayed in Listing 6. Note in particular how the actual vector types such as `f32x4` are also generic over `Simd`.

The different available SIMD levels are then represented as zero-sized types that implement the functions for the given architecture. In order to prevent the user from arbitrarily creating levels on platforms that do not support it, the types contain an empty private field so that they can only be constructed from within the crate, where an instance of the struct will only be returned at runtime if the current system supports the given capabilities.

An example of the implementation of the `Simd` trait can be seen in Listing 7. For `Fallback`, we use normal scalar arithmetic to implement the addition of two floating point numbers, while for `Neon`, we make a call to the `vaddq_f32` intrinsic. Using these capabilities, we can define the main functions in Vello CPU to be generic over the `Simd` trait, which allows us to implement them in a platform-agnostic way while still leveraging SIMD capabilities.

3.10.2 Implementation

Certain parts of the pipeline (such as coarse rasterization) are not obviously SIMD-optimizable and would only benefit very little from it, if at all. However, the main stages

[17] https://github.com/dtolnay/proc-macro2 (accessed on 11.09.2025)
[18] https://github.com/dtolnay/quote (accessed on 11.09.2025)

```rust
pub struct Fallback {
    _private: (),
}

impl Simd for Fallback {
    #[inline(always)]
    fn add_f32x4(self, a: f32x4<Self>, b: f32x4<Self>) -> f32x4<Self> {
        [
            f32::add(a[0usize], &b[0usize]),
            f32::add(a[1usize], &b[1usize]),
            f32::add(a[2usize], &b[2usize]),
            f32::add(a[3usize], &b[3usize]),
        ]
            .simd_into(self)
    }
}

pub struct Neon {
    _private: (),
}

impl Simd for Neon {
    #[inline(always)]
    fn add_f32x4(self, a: f32x4<Self>, b: f32x4<Self>) -> f32x4<Self> {
        unsafe { vaddq_f32(a.into(), b.into()).simd_into(self) }
    }
}
```

Listing 7: Example implementations of the SIMD trait for `Fallback` and `Neon`.

are fortunately very much amenable to such optimizations and profit vastly from it. A major focus of this work was therefore rewriting the existing parts of the pipeline to make use of SIMD capabilities. At the time of writing, the flattening, strips generation, fine rasterization as well as packing stages are SIMD-optimized. The last remaining candidate that could *potentially* profit from such optimizations is stroke expansion, but this has not been implemented yet.

For the flattening stage, the SIMD optimization only applies to cubic curves. As was explained in Section 3.4, cubic curves are flattened by first approximating them by multiple quadratic curves and then flattening each quadratic curve. What we therefore do is to use SIMD compute the number of necessary subdivisions for multiple quadratic curves in parallel, and finally emit the line segments for each curve. Our benchmarks confirmed that this has a noticeably positive impact on performance, as the original

implementation of flattening on Kurbo was written in a way that makes it hard to auto-vectorize for the compiler and also operated on `f64` instead of `f32` values.

For strips generation, the vectorization is conceptually also relatively simple. Remember that the bulk of the work in this stage comes from iterating over all 4x4 tiles in a strip, calculating the winding number of each pixel, adding them and then converting them into `u8` opacities. In principle, it would be possible to do this calculation independently for each pixel, but the current implementation uses the winding number of the *left* pixel as the basis for the area calculation, meaning that horizontally consecutive pixels cannot be processed in parallel. However, there are no dependencies in the vertical direction. Since strips always have a fixed height of 4, we can therefore always process a whole column of pixels in parallel. As we are using `f32` to compute winding numbers, we can conveniently use 128-bit vector types for this. This does however mean that the current approach cannot currently fully utilize 256-bit or 512-bit vectors. Adding support for bigger vector types is possible in principle, but would require changes to make the calculation of the winding number of a pixel completely independent of its neighbors.

The fine rasterization stage makes it even easier to add such optimizations, as the calculations for compositing and blending pixels are completely independent and have no dependencies to neighboring pixels. Because of this, it would theoretically be possible to use 512-bit vectors for most calculations. For the f32-pipeline, a whole column of pixels could be processed at the same time, since a column consists of 4 pixels and each pixel stores four `f32` values for the red, green, blue and alpha channels. Since storing `u8`s only needs one fourth of the space, the u8-pipeline could therefore even process chunks of 4x4 pixels at the same time. However, performing computations with such large vectors has the risk of causing high register pressure on targets that do not support such a large width natively. Therefore, in practice, the current implementation uses 256-bit vector types by default, and switches to 128-bit or 512-bit in certain places if appropriate.

Finally, the last part that makes use of SIMD optimizations is the packing stage of the u8 pipeline. In Section 3.9, it was explained that one of the main challenges of packing is that we need to "transpose" the pixels from column-major order in the wide tiles to row-major order in the pixmap. The main bottleneck here is first loading and then storing each pixel one at a time. Fortunately, at least some instruction sets have dedicated intrinsics that makes this a lot easier. For example, NEON has the `vld4q_u32` instruction which allows loading 16 `u32` values (in this case, the RGBA values of a single pixel are interpreted as a single `u32` value instead of 4 `u8` values) and interleaving them all in one step. By doing so, we can directly store each vector in the corresponding row in the

pixmap. Our benchmarks showed that handling these pixels in bulk using SIMD leads to a speedup higher than 3x, so this is a very important optimization.

3.11 Multi-threading

One core motivation for exploring the sparse strips approach was that we believed it to be compatible with multi-threading, something that is currently only supported by one other mainstream renderer, namely Blend2D. Figure 34 provides a rough overview of the architecture that enables this rendering mode. In general, path rendering and rasterization run completely in parallel on different threads, while coarse rasterization is still exclusively performed on the main thread, therefore forming the only serial bottleneck in the pipeline.

Everything starts with the creation of the `RenderContext` (see Section 3.1), where the user can set the number of threads that should be used for rendering. In case it is set to 0, no multi-threading will be activated and the code path for single-threaded rendering will be used. Otherwise, the `RenderContext` will spawn a thread pool

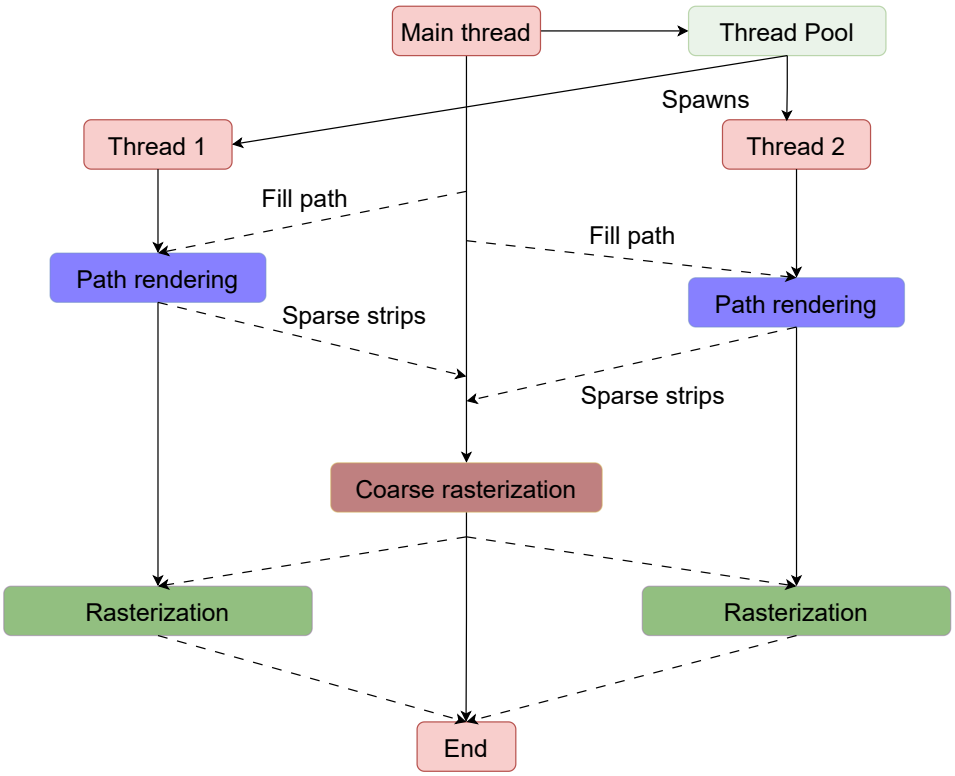


Figure 34: An overview of the architecture for multi-threaded rendering.

containing `num_threads` threads, meaning that including the main thread, there will be `num_threads + 1` active threads. In order to manage the thread pool, we use rayon[19], a Rust library for data parallelism.

3.11.1 Path rendering

The core insight necessary to understand the first part of multi-threading is that path rendering (ranging from stroke expansion or flattening to strips generation) is essentially a pure function that takes a single path as input and returns a vector of strips as the output. Since there are no other dependencies during this stage, the rendering of a single path can be completely outsourced without requiring any additional communication between the main thread and the child thread until the strips have been generated. This stage of the pipeline often takes up the largest chunk of time. Being able to run these stages in parallel can therefore lead to considerable speedups.

In practice, the process starts by the user emitting a command like `fill_path` or `stroke_path`, upon which the main thread first stores the command inside of a local queue. The reason for doing this instead of directly sending the command to a thread is that we make use of a batching mechanism to increase the efficiency of the whole process. While multi-threading itself is powerful, a considerable problem is that if the amount of work farmed out to the threads is too small, the overhead that arises from context switches and the bookkeeping done by rayon will be so large that any benefit of parallelism is essentially rendered useless.

Therefore, instead of dispatching fill and stroke commands individually, the main thread collects multiple drawing commands and keeps track of a counter that tries to estimate the cost of rendering all currently queued paths. Only once a certain threshold is reached does the main thread dispatch the whole batch of render tasks to a thread in the thread pool, which will then generate the sparse strips for the whole batch.

The cost estimation function works in a very simplistic way by considering relevant information about the path, including the number of line and curve segments, the overall length of the path and whether we are performing filling or stroking. While naive, this approach has been empirically validated to work relatively well. With this batching mechanism, if the user for example decides to draw many small rectangles, the main thread will collect dozens of these commands before dispatching them. In contrast, if larger geometries with many curves are drawn, only few of them will be batched before being processed. To enable the communication between the main thread and the worker

[19] https://github.com/rayon-rs/rayon (accessed on 11.09.2025)

threads, we use the crossbeam-channel crate[20] that provides primitives for implementing the single-producer multi-consumer paradigm easily.

While path rendering itself can be parallelized in a very straight-forward way, there is a very central issue that needs to be addressed: How do we ensure that the generated strips arrive in the correct order back at the main thread? After all, if the user first draws a red rectangle and then a green one, we need to ensure that they are processed in that very same order during coarse rasterization. Due to the inherent indeterminism of multi-threading, it could happen that the second thread ends up being faster than the first one, resulting in the two rectangles being processed in the wrong order.

Two key implementation details make sure that this property is always upheld. First, before dispatching a path rendering task, the main thread assigns a unique ID to the task that is based on an incrementing counter. By doing so, just checking the ID of a render task is enough to determine if it should be processed next during coarse rasterization or whether there are in-between tasks still missing. Secondly, in order to send the rendered strips back to the main thread, we use the ordered-channel[21] crate, which provides a MPSC (multi-producer single-consumer) channel primitive with the useful property that the user will always receive messages with incrementally ascending IDs. Internally, this is achieved with the help of a binary heap and a simple counter keeping track of the next expected ID in the `Receiver`. In case the receiver gets a message with the expected ID, it simply forwards it to the user. If the ID of the received message is too high, it is instead pushed to the binary heap and stored there until it is the right time to forward it. This process is repeated until the message with the right ID is received, which will then again be forwarded to the user. Using this principle, the user will always receive the messages in the ascending order, regardless of the order in which the messages where sent from each thread.

Assume for example that thread 1 sends the generated strips of path 1 via the channel, and subsequently thread 2 sends the strips for path 2. In this case, the main thread can just receive the results normally in that order and use them during coarse rasterization. However, if subsequently thread 3 sends the data for path 5 and thread 2 sends the data for path 4, while path 3 is still being processed in thread 1, the main thread will not receive any messages until thread 1 is done, after which the main thread will receive the messages with ID 3, 4 and 5, in that order. This ensures determinism in the order in which the sparse strips are received to then be further processed during coarse rasterization.

[20] https://github.com/crossbeam-rs/crossbeam (accessed on 11.09.2025)
[21] https://gitlab.com/kornelski/ordered-channel (accessed on 11.09.2025)

3.11.2 Coarse rasterization

As was mentioned previously, coarse rasterization is currently the only part of the pipeline that runs strictly sequentially and always on the main thread, the reason being that it is not trivially parallelizable. In the current setup, coarse rasterization happens in two phases.

In the first phase, as the user submits commands for filling and stroking paths, each time the main thread sends a new batch of render commands to a child thread, it subsequently checks whether new sparse strips from previous rendering commands are already available. If not, instead of blocking the thread, control is simply passed back to the user so that new render commands can be submitted. If there are strips available for processing, the main thread does coarse rasterization for them and only passes back control to the user once all currently available strips have been handled. As can be seen, the phase is basically an interleaved process where on the one hand fill and stroke commands are dispatched to other threads, but on the other hand coarse rasterization is done eagerly for strips that have already been generated.

As was previously shown in Listing 1, once the user is done submitting commands, they need to call the `flush` method, which starts the second coarse rasterization phase. Similarly to the first phase, the main thread will try to receive new strips from the queue and do coarse rasterization for them. However, unlike previously, the main thread will not pass control back to the user in case no strips are currently available. Instead, it will *block* by continiously polling for new sparse strips in a loop, until the very last batch of strips has been received and processed in coarse rasterization. It is important that we do this, because as soon as the rasterization stage is started, it is assumed that all render commands are already stored in their wide tiles.

3.11.3 Rasterization

The final part of the pipeline consists of fine rasterization and packing, which is trivial to parallelize. Remember that after coarse rasterization, our drawing commands are distributed over many wide tiles with the dimension 256x4 pixels which can all be processed independently from each other. Therefore, different wide tiles can easily be processed at the same time by different threads without worrying about any data dependencies. In our implementation, this is achieved with a call to the rayon-provided method `par_iter_mut` while iterating over the wide tiles.

The main difficulty arises once we reach the packing stage, as every thread will need mutable access to the user-provided `Pixmap`, which cannot easily be shared across the thread boundary due to the constraints imposed by the Rust compiler, even though

doing so would be safe because wide tiles are guaranteed to not overlap each other, ensuring that no memory location is updated simultaneously from two threads. One way of solving this would be to simply use `unsafe` code to work around that restriction. However, we decided to circumvent this issue by instead building a custom *Regions* abstraction that chops up the whole pixmap into many small mutable slices using the `split_at_mut` method, making it possible to write to distinct parts of the pixmap at the same time.

3.11.4 Alternative approaches

The description above outlines the current state of multi-threading. As will be seen in Section 5, while the current approach does lead to good speedups in many cases, it does exhibit suboptimal scaling behavior for very large thread counts, largely due to the serial bottleneck that is formed by coarse rasterization. In an attempt to alleviate this, two other ideas have been explored, and while they did not lead to any significant improvements and were discarded in the end, we still want to outline them here.

Initially, we believed that a significant bottleneck was formed by the first phase of coarse rasterization. In general, to achieve the best performance, it is important that path rendering commands are distributed fast enough such that the child threads are kept busy at all times. The theory was that if the main thread spends too much time doing coarse rasterization in the first phase, it might lead to a slowdown since there is a larger gap between dispatching render commands, and some threads might therefore stay idle due to not being assigned enough work. This gave rise to the idea of shifting the burden of coarse rasterization to the child threads instead. This was achieved by putting the wide tiles behind a mutex and letting the child threads take turns in doing coarse rasterization. In this design, coarse rasterization is still a fundamentally serial process, but the advantage is that the main thread will be freed up and can therefore "focus" on distributing the work to child threads as fast as possible. Unfortunately, benchmarking showed that this did not lead to any improvement. Upon closer inspection with a profiler, it became clear that only a small fraction of the time is spent during the first phase of coarse rasterization and that the real bottleneck was actually the second phase. Therefore, the root cause of the suboptimal scaling behavior for large thread counts was that the serial time needed to perform all coarse rasterization operations exceeded the time each child thread spends during path rendering, which cannot be addressed by this method.

This motivates a different approach to coarse rasterization: While there is a serial dependency between the commands *within* a single wide tile, different wide tiles are

completely independent from each other. This makes it possible to instead group multiple rows of wide tiles into larger bands (somewhat similar to what Blend2D does) and assign each band to a specific thread, which is then responsible for doing the whole coarse rasterization only for that band. By doing this, the burden of coarse rasterization is once again shifted away from the main thread. But in addition to that, coarse rasterization happens in parallel since different wide tile bands can be handled at the same time. Unfortunately, as part of our experiments, we determined that this approach only seems to lead to a slowdown. While we were not able to determine the exact reason, profiling revealed a much more significant communication overhead, which can be explained by the fact that the sparse strips of each path were broadcasted to all other threads instead of just being sent back to the main thread. Additionally, another problem is that, given the sparse strips representation of a path, there is no easy way for a thread to determine which strips are relevant for its band of wide tiles, making it necessary to iterate over the strips until the right start point is found, which could potentially also be time-consuming. While our initial experiments did not work out, we do believe that this approach could potentially be tuned to eliminate those downsides. For example, instead of broadcasting the strips to all other threads, they could just be collected on the main thread and then shared all at once with each thread a single time. To tackle the problem of determining the right range of strips for a specific band of wide tiles, it might be possible to generate an "index" for each set of sparse strips that allows for determining the right range via a simple lookup. With that said, we leave further explorations in this area as future work.

3.12 Clip paths

As was mentioned in Section 2.11, having an efficient implementation of clip paths is crucial, as the operation is very commonly used in 2D rendering. In the following, we present an algorithm that implements clip paths using sparse strips. While the actual implementation is somewhat involved, the algorithm itself is relatively high-level and intuitive, which yet again demonstrates the convenience of having sparse strips as an intermediate representation. To explain the algorithm, we will demonstrate it based on the example in Figure 16.

On a high level, the clipping algorithm works by first generating the strips for the input shape and our clip shape, and subsequently generating a new set of strips that represent the clipped output shape. To achieve this, we consider each row of strips in an isolated fashion. Figure 35 demonstrates the process for the fourth row of strips in the butterfly and triangle. First, we subdivide the row of both shapes into segments, such that each segment covers one type of region of the input shape and clip shape. In Figure 35, we

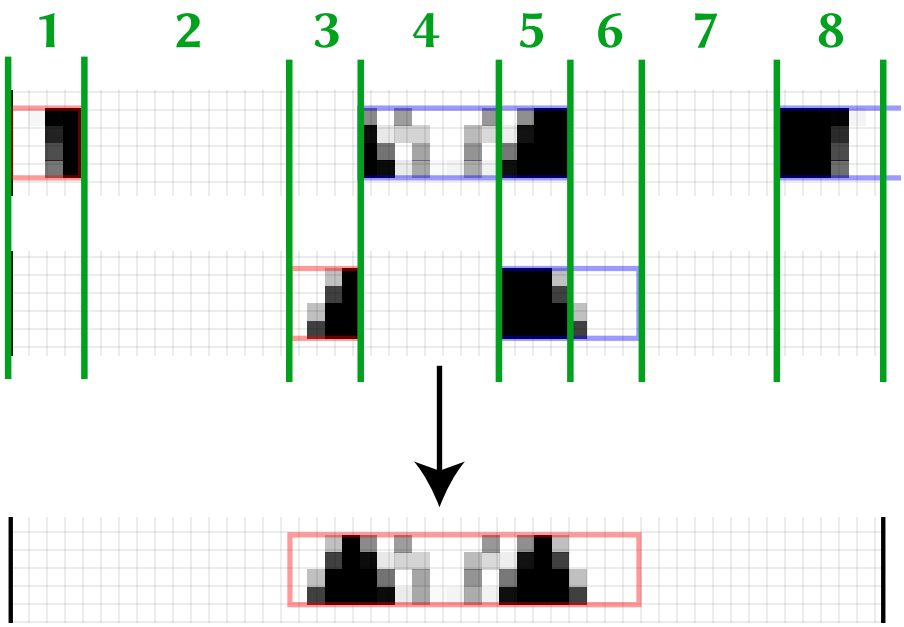


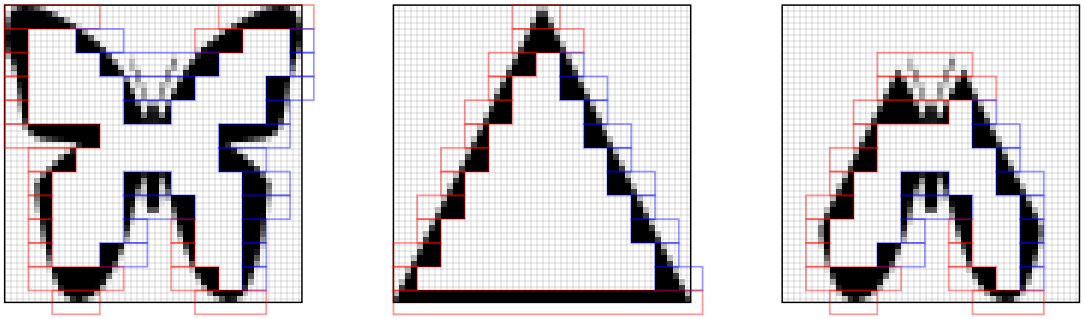
Figure 35: The clipping algorithm visualized based on the 4th row of strips.

have eight segments in total. The first segment covers a strip region of the butterfly and an unfilled region of the triangle. The second segment covers a filled region of the butterfly and another unfilled region of the triangle. The third segment covers a filled region of the butterfly and a strip region in the triangle. The same principle applies to all other segments, where either a filled, unfilled or strip region is covered on each side.

Then, we simply iterate over all the segments and produce new strips for that area depending on what types of region it covers:

- In case any of the two regions is an unfilled area, no strips are generated at all. This is the case for the regions one, two, seven and eight in Figure 35.
- In case we have one filled region and one strip region, we generate a new strip with the same pixel opacities as in the strip region. This is the case for regions three, four and six in Figure 35.
- In case we have two strip regions, we produce a new strip where each pixel opacity is the product of the pixel opacities in the two input strips. This is the case for region five in Figure 35.
- Finally, in case we have two fill regions, we simply generate another sparse fill region. This case does not appear in Figure 35.

At the bottom of Figure 35, we can see the generated row of strips for the output shape. Note also that strips produced by different segments are merged into single strips. The result of applying this algorithm to all rows of strips is visible in Figure 36. The output



(a) The strips of the input shape.

(b) The strips of the clip shape.

(c) The strips of the final clipped result.

Figure 36: The complete result of applying the clipping algorithm.

of the algorithm is simply a new set of sparse strips (like in Figure 36c) where as many regions as possible are still represented as sparse fill regions, while anti-aliased parts are covered by a strip. The beauty of this algorithm is that it is incredibly work-efficient as the only computationally expensive part in the whole process is the multiplication of opacities in case a segment happens to cover two strip regions. All other cases simply involve doing a memory copy operation or doing nothing at all.

Chapter 4

Comparison

In this section, we will briefly contrast the architecture of two different CPU-based 2D renderers against Vello CPU to highlight similarities but also differences. Since there is little literature describing the inner workings of those, we will base most of our descriptions on a best-effort analysis of their source code.

The comparison will demonstrate that certain parts of the 2D rendering pipeline are, with the exception of a few implementation details, virtually universal across different implementations. Yet, the core path-rendering algorithm of Vello CPU, underpinned by the sparse strips intermediate representation, is a distinguishing feature that separates it from other renderers.

4.1 Raqote

The first renderer is Raqote, a relatively simple yet feature-rich CPU-only renderer written in Rust.

Similarly to Vello CPU, stroking is implemented by reducing the problem to the filling of an equivalent path, but with a small difference: Raqote first flattens the stroked path to lines and then generates the stroked path based on the lines. In Vello CPU, it is the other way around, as stroke expansion happens in cubic space, and only then the resulting curve is converted into line segments.

An important difference lies in the way normal fills are handled. Unlike Vello CPU, Raqote does not flatten curves into line segments but instead uses a rasterizer that can support both line and curve segments. The disadvantage of this approach is that it complicates the rasterization logic as there is a less clear-cut separation of concerns between pipeline stages as in Vello CPU. However, the advantage is that it does not need an explicit flattening stage.

The main point of distinction between the two renderers is the actual rasterization process. As previously elaborated, Vello CPU generates so-called sparse strips that have a fixed height of 4 and cover possibly anti-aliased parts of the pixmap, while filled areas are only represented implicitly. Vello CPU does *analytical anti-aliasing*, which means that the pixel coverage is determined by analytically calculating the area of the pixel

that is enclosed by the line. The advantage of this technique is that it is relatively fast, but the downside is that it can more easily lead to conflation artifacts (as explained in Section 2.9), since the method only calculates how much of the area is covered but not actually *which parts* of the area.

Raqote on the other hand uses the classical scan-line rasterization approach as it is adopted by many other renderers. Conceptually, we iterate from top-to-bottom and consider one row of pixels at a time. At the same time, we store all the edges that are active in that particular pixel row by keeping them in a sorted list that is continuously updated as we step through the pixel rows. For each row, we iterate through the edges in windows of 2 and emit a "span" that ranges from the first edge to the second edge in case the area should be filled according to the winding rule (this is conceptually very similar to how we determine filled areas between two strips in Vello CPU). While doing so, Raqote uses *multi-sampled anti-aliasing*: Each pixel is viewed as a 4x4 grid, as can be seen in Figure 37. Above, we mentioned that the scan-line approach works on a per-row basis, but in reality it actually operates on a *sub-pixel row* basis, and when processing the edges from left to right we determine the sub-pixel columns of that row that should be filled. For a single pixel, we just need to determine how many of the sub-pixel cells are filled to determine the overall opacity of the pixel. The downside of multi-sampled anti-aliasing is that it is usually more expensive, but it can help with guarding against conflation artifacts, since the approach does not discard information on which *parts* of the pixel are covered, unlike analytic anti-aliasing.

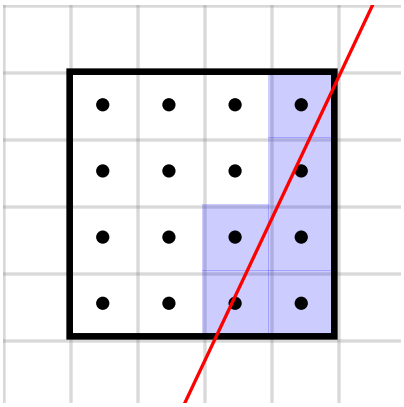


Figure 37: Calculating pixel coverage using 4x4 multi-sampled anti-aliasing. The area spanned by the line touches 6 out of the 16 cells in the sub-pixel grid, so the coverage is $\frac{6}{16} = 37.5\%$. It is possible to increase the resolution for example to a 16x16 grid for a higher resolution, but that is performance-wise usually only feasible when using the GPU.

These single spans are then processed in the usual way by computing the color with the paint and then compositing it. Taking this overall design into account, it should become apparent that there is a significant weakness in this design: It computes alpha masks for anti-aliasing for all pixels, even the ones that are strictly on the inside of the shape, while Vello CPU limits this rather expensive computation to the pixels that are within a strip, leaving the sparsely-encoded filled regions "untouched". Section 5 will show that this results in worse scaling behavior for larger geometries compared to Vello CPU.

Raqote does not make direct use of SIMD instructions and is therefore reliant on the Rust compiler to perform auto-vectorization. It also has no support for multi-threading.

4.2 Blend2D

Blend2D is a C++-based renderer that, as will become evident in Section 5, has outstanding performance. On a surface level, Blend2D does many things similarly to other renderers: Before rasterizing, strokes are converted into filled primitives and filled paths are simplified to flattened lines. Blend2D fundamentally uses a scan-line rasterizer that processes the pixmap on a per-row level but utilizes a number of clever techniques to ensure faster processing. Similarly to Vello CPU, Blend2D uses an analytical method for fast anti-aliasing calculations.

What makes Blend2D stand out compared to other renderers is not necessarily the underlying rendering technique, but instead the *way* those techniques are implemented. A look at the source code reveals that one reason for the exceptional performance of Blend2D is the sheer amount of effort that has been put into optimizations on both the algorithmic level as well as the actual implementation. These range from smaller optimizations like for example using fixed-point math instead of floating-point math when possible, to more sophisticated innovations like a customized bit vector storage that allows the compositor to quickly detect and skip empty pixels. Blend2D also makes use of an innovative JIT-compiler to generate highly-efficient SIMD-optimized machine code for parts of the pipeline. In contrast, Vello CPU consists of hand-written kernel functions written in normal Rust. While we tried to ensure that the functions compile to fast code, in the end we are still at the mercy of the compiler to actually generate efficient machine code, while being able to use custom-generated machine code gives you full control over what instructions will actually be generated.

Blend2D is the only other 2D renderer that also supports multi-threading and therefore makes for an interesting comparison in this regard. Apart from the fact that the concrete implementation is very different as Blend2D implements custom logic for managing

thread pools and scheduling jobs, while Vello CPU mostly outsources this to other Rust libraries, there also are differences on a higher level.

Blend2D divides the drawing area into so-called “bands” which can be thought of being similar to wide tiles, but they always span across the whole width instead of having a fixed width of 256 and can also have varying heights. In the first level of parallelism, Blend2D schedules commands for filling and stroking paths to worker threads which will then do stroke expansion and curve flattening in parallel. Then, each worker thread determines which line segments are part of which band and stores that information in a store. Note that this is different to Vello CPU, as in our case, in addition to expanding strokes and flattening paths, the worker threads will also perform anti-aliasing computations by converting the lines into the sparse strips representation and then letting the main thread taking care of assigning the commands to wide tiles.

The second level of parallelism then includes letting each worker thread process a single band and performing rasterization for all lines that were assigned to that band. This is conceptually similar to the second phase of parallelism in Vello CPU, with the main difference being that Blend2D also needs to compute pixel coverages in that phase.

In summary, the overall approach to multi-threading shares similarities, but many details like in which way work is distributed per thread are solved in different ways.

Chapter 5

Evaluation

Even when doing 2D rendering on the CPU instead of the GPU, having good performance is still crucial. In this section, we will analyze how Vello CPU fares in comparison to other prominent 2D renderers by evaluating its performance using the Blend2D benchmark suite [4].

5.1 Introduction

The main idea behind the benchmark harness is to run various rendering operations that exercise different parts of the rendering pipeline to varying degrees, measure the time and compare the results. Doing so allows us to easily determine both the strengths and weaknesses of each renderer. To achieve this, the harness defines a number of different parameters that can be tweaked to specify what exactly should be rendered. The configuration possibilities are visualized in Figure 38.

The first configurable knob is the *draw mode*, which should be relatively self-explanatory. Choosing *fill* will help us measure the performance of filling shapes, while choosing *stroke* instead will tell us how performant stroking (in particular stroke expansion) is.

Next, we can configure the actual *shape* that is drawn, which yet again impacts which part of the pipeline will be mainly exercised:

- **Rect A.**: Rectangles aligned to the pixel boundary. This is the most basic kind of shape you can draw and most renderers have a fast path for them.
- **Rect U.**: Unaligned rectangles. The same as Rect A., but the rectangles are not aligned to the pixel boundary and thus require anti-aliasing.
- **Rect Rot.**: Rotated rectangles. They are harder to special-case and the test case therefore demonstrates how well the renderer can rasterize simple line-based shapes.
- **Rect Round**: Rectangles with rounded corners. The reason these are worth benchmarking is that rounded rectangles are very common in graphical user interfaces and should therefore be fast to render.
- **Triangle**: Triangles. They represent the most basic form of a polygon.
- **Poly N**: N-vertex polygons. These shapes consist of many self-intersecting lines and are useful to determine the performance of anti-aliasing calculations.

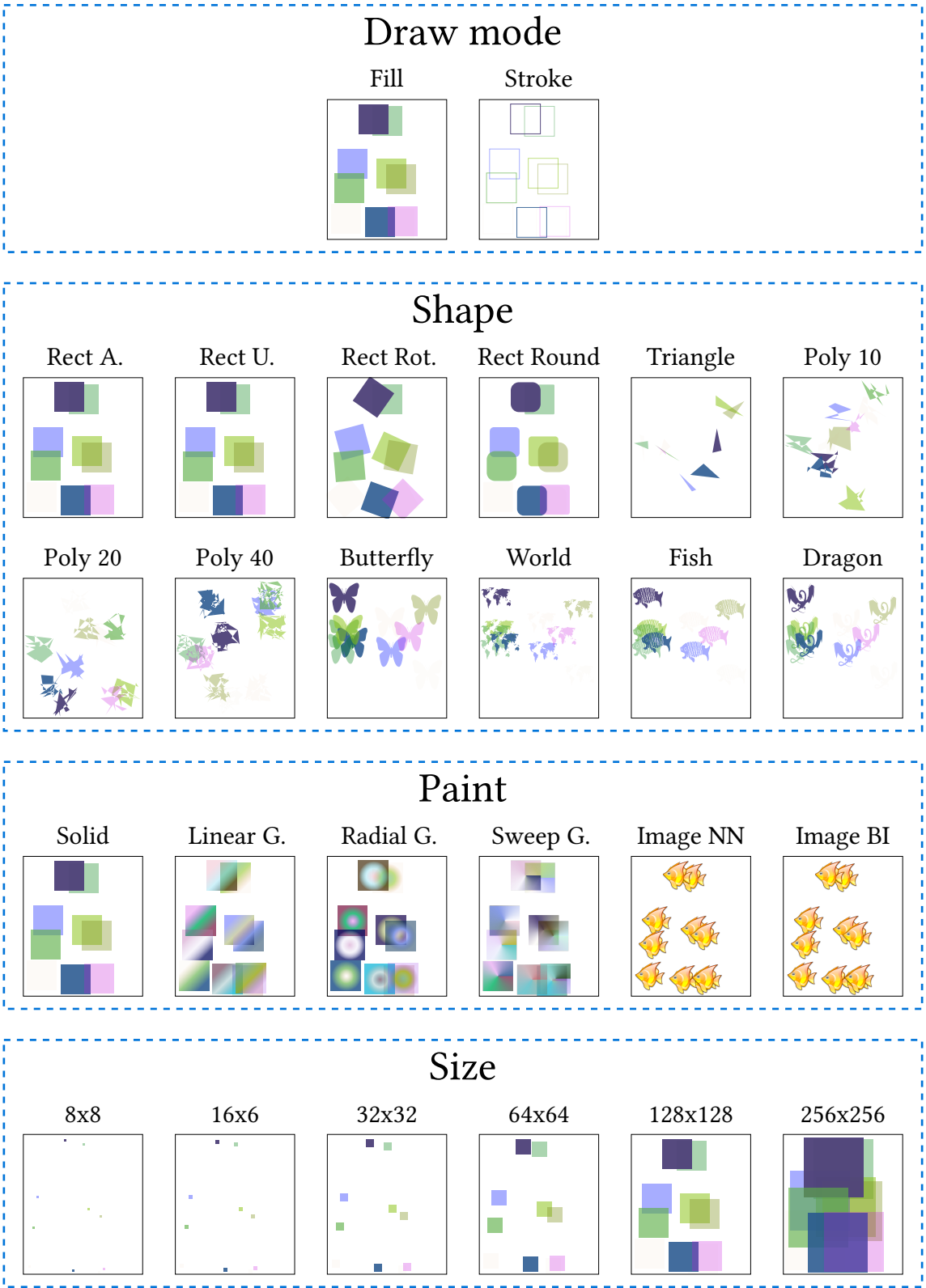


Figure 38: The four different configurable parameters of the benchmarking harness.

- **Butterfly**, **Fish**, **Dragon**: These shapes contain many curve segments, making them good candidates for evaluating curve flattening performance.
- **World**: This shape consists of just small line segments and aims to measure how effectively the renderer can rasterize a complex shape.

Specifying the paint allows us to see how fast the renderer can process gradients and image fills. And finally, tweaking the size of the shape shows how runtime scales as the dimension of the shape increases. As will be shown below, having all these different configurations is really useful as most renderers do have strengths and weaknesses in different areas.

5.2 Setup

In the interest of covering both the x86 and AArch64 architectures, we ran the benchmarks on two different machines. The first machine is a MacBook using the M1 Pro chip with 8 cores (6 performance cores and 2 efficiency cores) and 16GB of RAM. The second machine uses an Intel Core i9-14900K (8 performance cores and 16 efficiency cores) and runs on 64GB of RAM. Since the main focus of this work was optimizing the renderer using NEON SIMD intrinsics, we will only present the results of the M1 machine here. In addition, due to space reasons, we will not exhaustively list the measurements for all possible configurations but pick a small representative selection of tests that highlight the strengths but also weaknesses of Vello CPU. The full results for both architectures and all configurations are made available online[22].

The original benchmark harness is written in C++ and covers the following selection of renderers:

- Blend2D[23], a highly performant 2D renderer based on just-time-compilation that also supports multi-threading.
- Skia[24], the 2D renderer maintained by Google. Only the CPU-based pipeline is tested here.
- Cairo[25], a relatively old but still widely used 2D renderer.
- AGG (Anti-Grain Geometry)[26], yet another prominent 2D renderer written in C++.
- JUCE[27], the renderer that is shipped as part of the C++-based JUCE application framework.

[22] https://laurenzv.github.io/vello_chart/ (accessed on 11.09.2025)
[23] https://blend2d.com (accessed on 22.09.2025)
[24] https://skia.org (accessed on 22.09.2025)
[25] https://www.cairographics.org (accessed on 22.09.2025)
[26] https://agg.sourceforge.net/antigrain.com/index.html (accessed on 22.09.2025)
[27] https://juce.com (accessed on 22.09.2025)

- Qt6[28], the renderer that is shipped as part of the Qt application framework.
- CoreGraphics[29], the graphics framework that is shipped as part of Mac OS.

We decided to exclude Qt6 and CoreGraphics from our benchmarks since they turned out to be very slow in many cases and made visualizations harder.

Since Vello CPU is a Rust-based renderer, we had to create C bindings[30] to properly integrate it into the harness. We also decided that another intriguing research question was how Vello CPU compares against other Rust-based CPU renderers in particular, as none were included in the original benchmarking harness. To this end, we also created C bindings[31][32] for tiny-skia[33] and raqote[34], the two currently most commonly used CPU renderers in the Rust ecosystem. tiny-skia is more or less a direct port of a small subset of Skia to Rust, while raqote is a from-scratch implementation that still borrows many ideas from Skia. To make the comparison fair, all crates have been compiled with the flag `target-feature=+avx2` on x86 so that the compiler can make use of AVX2 instructions. On ARM, no additional flags were necessary since the compiler can assume that NEON intrinsics are available by default.

The operational semantics of the benchmark suite are very simple. Each test will be rendered to a 512x600 pixmap. When starting a test, the harness will enumerate all possible configurations and make repeated calls to render the shape with the given settings. In order to prevent caching, each render call introduces some randomness by for example varying the used colors and opacities or by rendering the shape in different locations on the canvas. The harness will perform multiple such render calls and use the measured time to extrapolate how many render calls can be made in 1 millisecond using that specific configuration. To reduce the impact of outliers, we repeat this process ten times for each test and always choose the best result.

In the following, we will show plots that display the benchmark results for a select number of configurations. For each configuration, we show the results of the given test across all shape sizes to make it easy to see the scaling behavior. It is important to note that for easier visualization, the **time axis is always log-scaled**, which has the side effect that large differences in rendering times are visually not as pronounced and can

[28] https://www.qt.io/product/qt6 (accessed on 22.09.2025)
[29] https://developer.apple.com/documentation/coregraphics (accessed on 22.09.2025)
[30] https://github.com/LaurenzV/vello_cpu_c (accessed on 22.09.2025)
[31] https://github.com/LaurenzV/tiny_skia_c (accessed on 22.09.2025)
[32] https://github.com/LaurenzV/raqote_c (accessed on 22.09.2025)
[33] https://github.com/linebender/tiny-skia (accessed on 22.09.2025)
[34] https://github.com/jrmuizel/raqote (accessed on 22.09.2025)

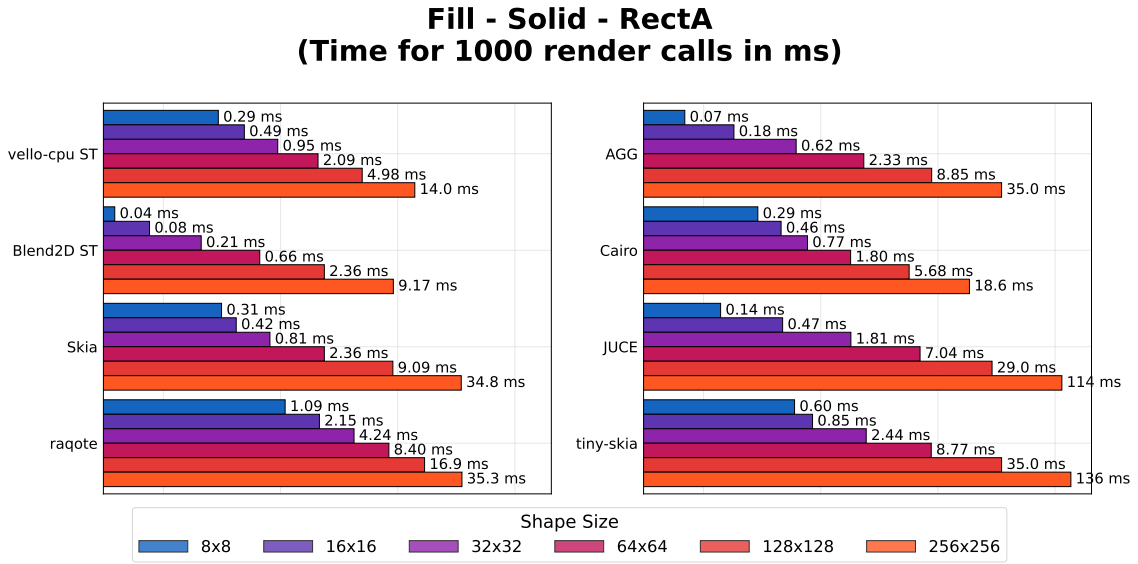


Figure 39: The running times for the test “Fill - Solid - RectA”.

only be noticed by looking at the individual time measurements. For multi-threaded rendering, we also show the speedup factor in comparison to single-threaded rendering.

5.3 Single-threaded rendering

5.3.1 Filling

We begin the analysis by looking at single-threaded rendering and considering the simplest test case, filling a pixel-aligned rectangle with a solid color. The results are visualized in Figure 39. There are two points worth highlighting in this figure as they represent a trend that, as will be shown soon, applies to most test cases.

First, it is apparent that Blend2D is the clear winner in this specific benchmark. Regardless of whether we are considering small or larger shape sizes, Blend2D consistently needs the shortest time, both compared to the C++-based as well as the Rust-based renderers and Vello CPU. As will be visible in other plots, this does not just apply to that specific test configuration but to nearly all other configurations as well, confirming the fact that the JIT-based architecture and painstaking optimizations that have been applied to Blend2D over the course of the years have their merit. Given this, for the remainder of this subsection we will mostly focus on comparing Vello CPU to the other renderers and shift our focus back to Blend2D when analyzing multi-threaded rendering.

Secondly, another trend that will become more apparent soon is that Vello CPU seems to have a general weakness for small shape sizes, but apart from Blend2D it is the only

other renderer that shows excellent scaling behavior as the shape size increases. For 8x8 and 16x16, AGG and JUCE are faster in this specific benchmark, but that difference is reversed when considering 128x128 and 256x256.

We believe this behavior can easily be explained by considering how sparse strips work. Since the minimum tile size is 4x4, in case we are drawing small geometries there is a high chance that the whole geometry will be represented exclusively by strips instead of sparse fill regions, which are more expensive to process. In addition to that, the fixed tile size has the consequence that many pixels that are actually not covered by the shape are still included and incur costs during anti-aliasing calculations and rasterization, which can represent a significant overhead given the small size of the shape. On the other hand, thanks to sparse strips, the larger the geometry, the more likely it is that it can be represented by large sparse fill regions, which are comparatively cheap to process. Other renderers do not necessarily make use of such an implicit representation and therefore exhibit worse scaling behavior.

Another reason that Vello CPU performs slightly worse in this particular case is that we decided to not include a special-case path for pixel-aligned rectangles, which becomes apparent when contrasting the results to Figure 40, where unaligned rectangles are drawn. In that figure, the results for Vello CPU more or less stay the same, while for example JUCE and AGG show worse performance. Overall, it still seems safe to draw the conclusion that Vello CPU lands a second place for those two configurations.

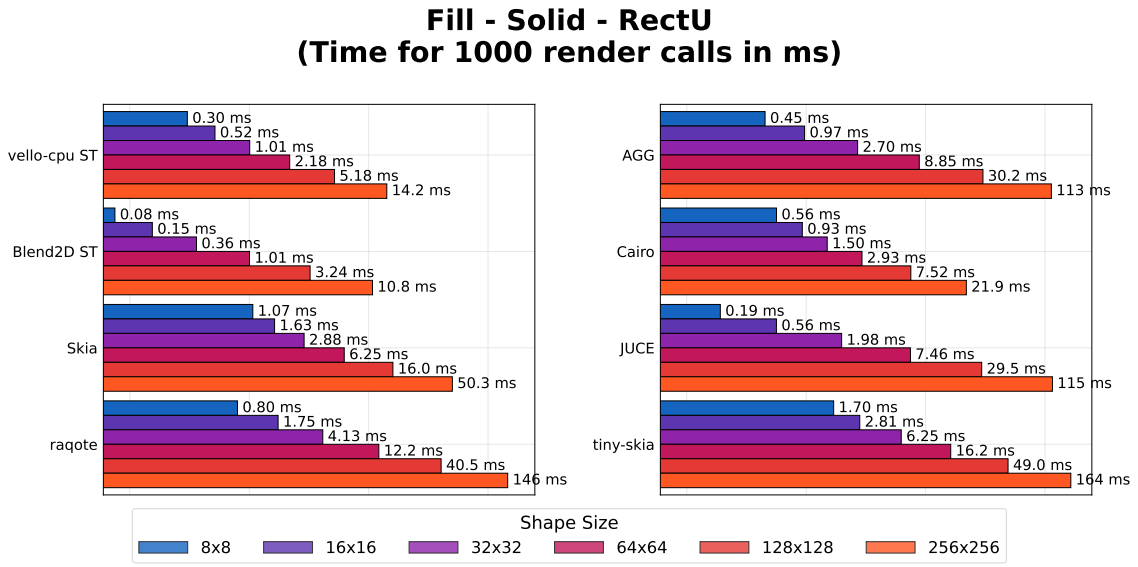


Figure 40: The running times for the test “Fill - Solid - RectU”.

Next, we want to analyze the performance of general edge rasterization by considering the "PolyNZi40" test case in Figure 41. As can be seen, this is another area where Vello CPU shines: While the gap to Blend2D is larger for small sizes, it becomes much closer for larger sizes. In particular, note how Vello CPU is significantly faster than any of the other renderers, both for smaller sizes and especially for larger sizes. It is difficult to grasp why exactly the other renderers are performing worse here, but it is possible to hypothesize. For example, remember that Raqote uses an "active" edge list to keep track of the active edges per scan-line. This means that each time the renderer moves to a new scan-line, it needs to do much more work to discard, add and sort the edges, resulting in additional overhead for every pixel row that is analyzed. On the other hand, in Vello CPU, adding more lines simply means adding another set of tiles that we need to perform anti-aliasing for, but it has no impact on the processing times of the other tiles.

Let us shift our focus to Figure 42 next, which forms the main exception to our previous statement that Vello CPU has a weakness for small shapes and actually ends up beating Blend2D for certain sizes. The reason that Vello CPU performs so well here is to a large degree due to the optimizations that were introduced in Section 3.4.1. As can be seen in the figure, having this shortcut path leads to impressive gains if the shape consists of many small curve segments. Blend2D does regain the lead as the shape sizes increases, but Vello CPU follows closely and is still significantly faster than all other implementations.

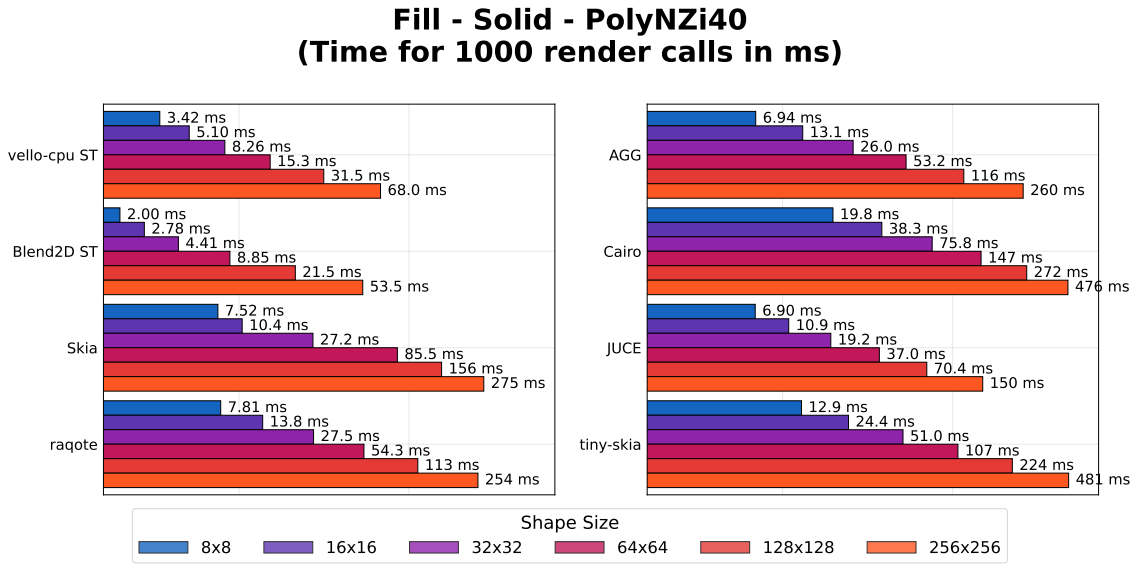


Figure 41: The running times for the test "Fill - Solid - PolyNZi40".

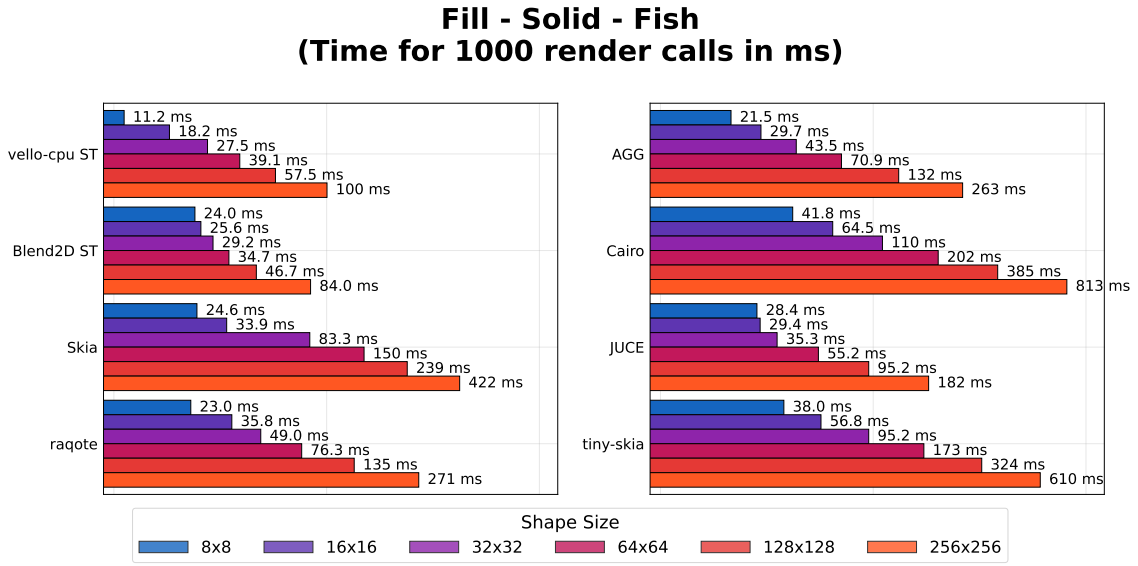


Figure 42: The running times for the test “Fill - Solid - Fish”.

Let us consider Figure 43, where the weakness for smaller geometry sizes becomes more apparent: At the smallest size, Vello CPU is more than twice as slow as Skia and raqote, with the inflection point only arriving at the size 64x64, at which point Vello CPU once again exhibits much better scaling behavior. When looking at the performance profile for the 8x8 case in Figure 44, over 85% of the time is spent in the path rasterization stage during tile generation, sorting as well as strip generation. Thinking about this in more detail, the possible problem becomes apparent. The “World” test case consists of a shape with a very large number of lines. Even when the shape is scaled down to 8x8, we still end up generating a 4x4 tile for each single line and computing the winding numbers for all 16 pixels, even though only a small part is really covered. The problem could be slightly ameliorated by relaxing the restriction on the width of strips (see Section 6), but this will still not reduce the number of generated tiles that need to be sorted, which takes up a significant chunk of the time.

There does not seem to be a straightforward solution to the problem. One approach could be introducing a kind of “line-merging” stage where multiple small lines are merged into larger ones at the cost of some precision, but this seems to be a rather complicated optimization. Otherwise, it might also be worth investigating faster sorting algorithms specialized for a large number of tiles, but the potential for gains here does not seem that significant either.

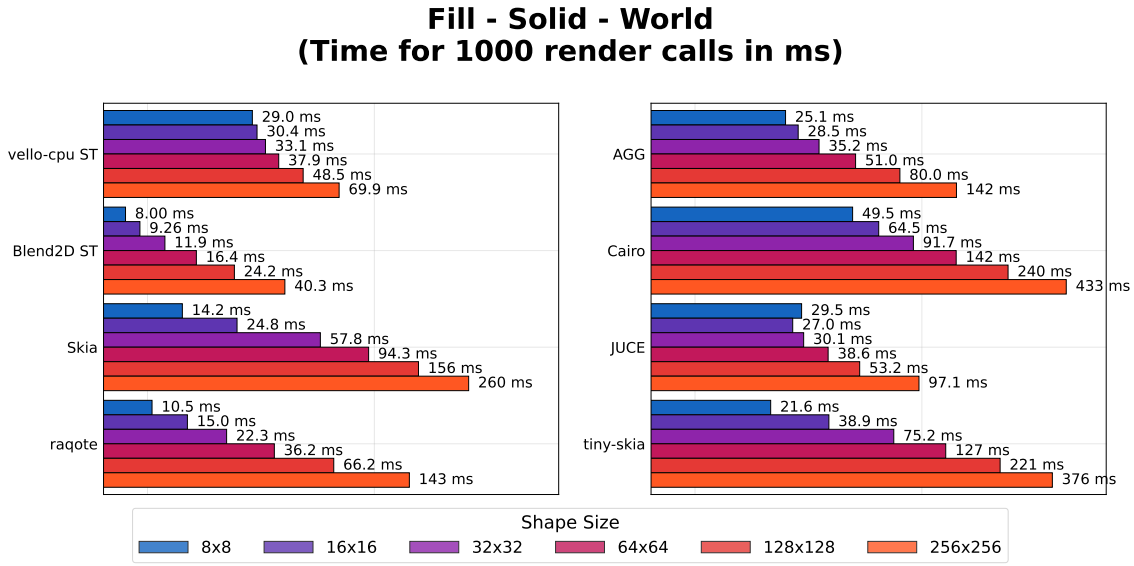


Figure 43: The running times for the test “Fill - Solid - World”.

5.3.2 Stroking

Let us devote our attention to stroking next. There are two cases that are worth exploring: Stroking of straight lines and stroking of curves. For lines, we can consider the “World” shape again as it is depicted in Figure 45. As can be seen, Blend2D once again by far leads the score, but is followed second by Vello CPU, which (together with JUCE) leads with another significant gap compared to the remaining renderers. There

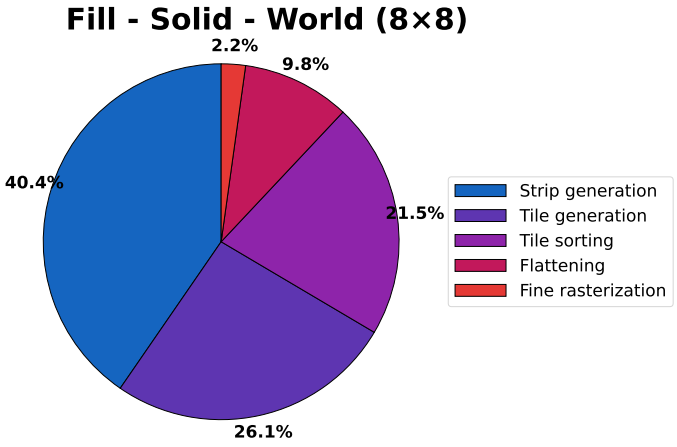


Figure 44: The percentage of the total runtime each step in the pipeline takes for the test case “Fill - Solid - World” with shape size 8x8. Note that the shape does not contain any curve segments. Therefore the “flattening” part mostly represents the overhead from iterating over all lines and re-emitting them.

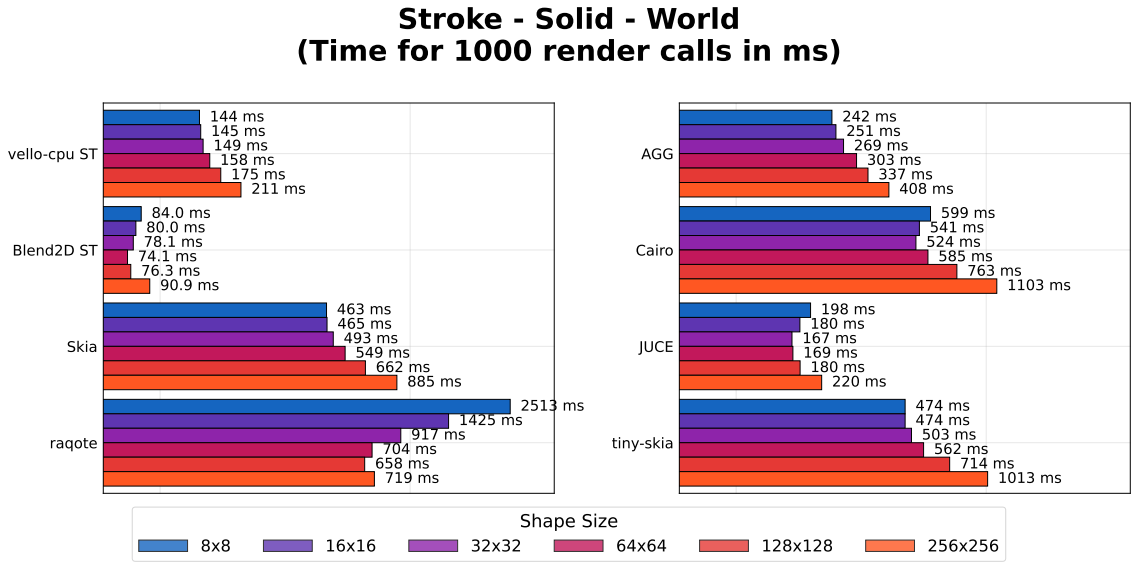


Figure 45: The running times for the test “Stroke - Solid - World”.

is also an interesting behavior where different renderers react differently to different sizes: Raqote seems to deal worse with small shapes but better with larger ones, while for other renderers the sweet spot is somewhere in the middle. Determining the exact reason for these discrepancies appears difficult, but the observation highlights the fact that different renderers use different algorithms for stroking.

In Figure 46, we can see the running times when stroking a curved shape instead. We once again observe that in this particular case, Vello CPU does not perform as well as other renderers for small shape sizes, but makes up for the differences as the dimensions of the shape increases. In this case, we do not attribute the sluggish performance for small shape sizes to the fixed size of 4x4 for a tile. A look at the profile in Figure 47 reveals that more than 70% of the time is spent in stroke expansion, leading us to the conclusion that there seems to be some inefficiency in the current algorithm. The exact reason for this issue remains to be determined, however.

Overall, we draw the conclusion that while there is more work left to be done on the stroking side of things, but performance is at least comparable to other renderers and stands out when considering larger shape sizes.

5.3.3 Paints

Finally, let us analyze the performance of Vello CPU when complex paints such as linear gradients (in Figure 48) and images (in Figure 49) are used.

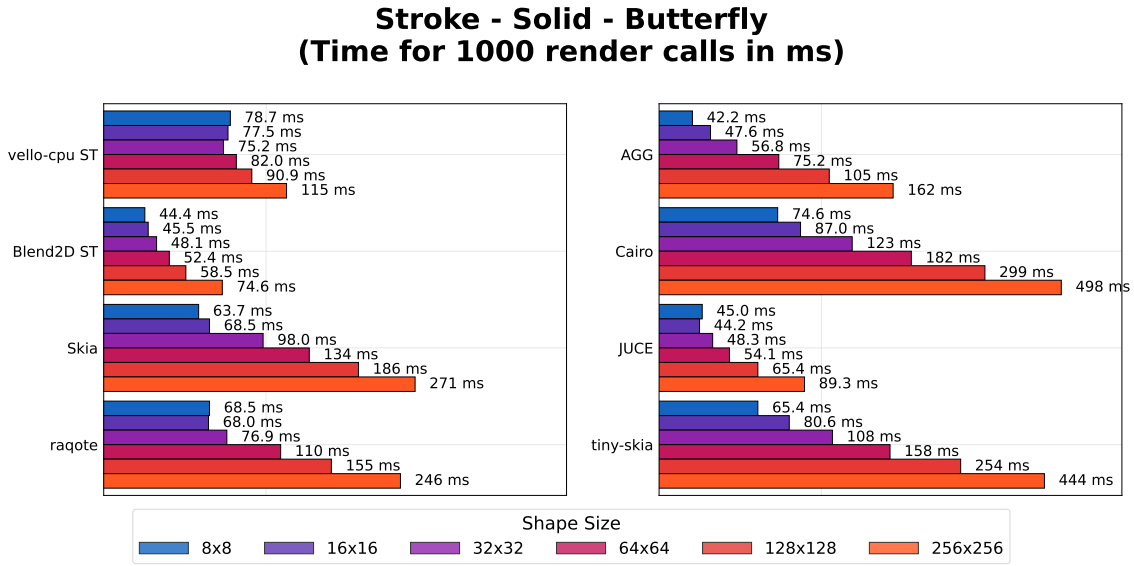


Figure 46: The running times for the test “Stroke - Solid - Butterfly”.

As is visible in Figure 48 figure, Vello CPU has a significant overhead when rendering a gradient with a small shape but has outstanding performance for larger ones. The reason for this can be explained as follows. There are two general approaches for how to deal with gradients: The first method is to precompute a lookup table with a fixed resolution and then sample from that table, as is done in Vello CPU. The second method (for example used by tiny-skia and therefore most likely also Skia) is to do no pre-computation at all but instead do the color interpolation on the pixel-level. The

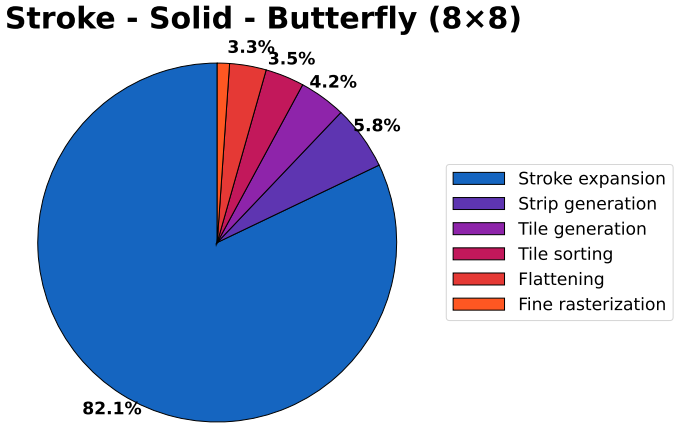


Figure 47: The percentage of the total runtime each step in the pipeline takes for the test case “Stroke - Solid - Butterfly” with shape size 8x8. Stroke expansion represents the main bottleneck.

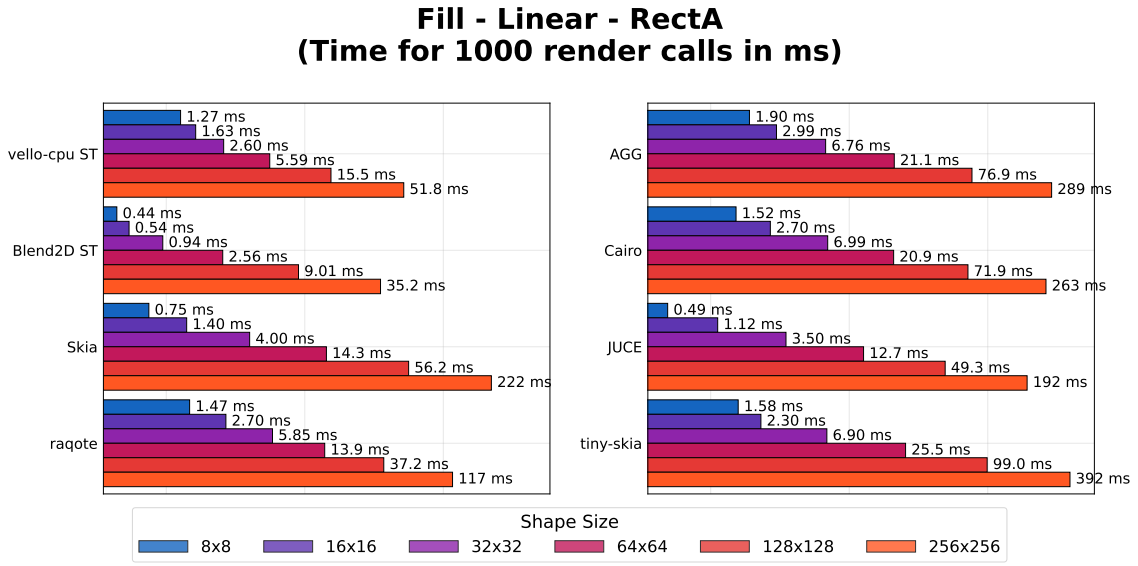


Figure 48: The running times for the test “Fill - Linear - RectA”.

disadvantage of the former method is that computing a LUT of this size is more time-consuming if the shape we will draw only covers very few pixels, but the advantage is that it exhibits much better scaling behavior as the size of the shape grows, as can be seen in the benchmark. It should be noted though that Blend2D also uses the LUT approach and still has the best runtime for 8x8, suggesting that there are probably ways of improving the computation in Vello CPU as well.

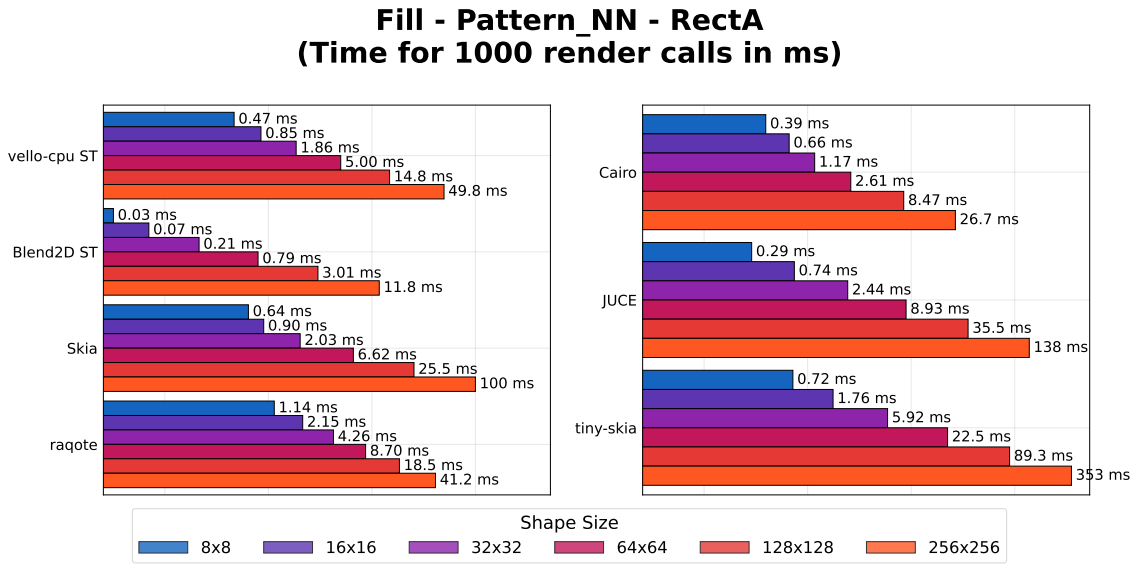


Figure 49: The running times for the test “Fill - Pattern_NN - RectA”.

For image fills, we see that Vello CPU is not doing particularly well, but performance is not bad either and matches most of the other renderers. A particular problem that we are facing is that for nearest-neighbor rendering of images, the performance is largely determined by how fast a pixel can be sampled from the original image. Since memory safety is at the core of Rust, each access to a memory location is preceded by a bounds-check, which has been shown to lead to a significant slowdown. We could have resorted to using unsafe code to circumvent this, but in the end decided that it was not worth it to potentially introduce memory-safety issues by doing so.

5.4 Multi-threaded rendering

Next, we want to more closely analyze Vello CPU's multi-threaded rendering performance. Since Blend2D is the only other renderer that supports such a mode, we will use this as the main reference point for the comparison, putting a particular emphasis on analyzing how the speedup scales as the number of threads increases. Before diving into the analysis, it is important to make a remark about the difference between how threads are counted in both renderers that makes it hard to do a fully comparable analysis. In Blend2D, the main thread will also be used as a worker thread. This means that if a test case is indicated to run with eight threads, Blend2D will use the main thread as well as seven additional worker threads for rendering. In Vello CPU, it is not possible to use the same approach as the main thread is always only responsible for distributing work and doing coarse rasterization, but unlike the worker threads cannot be used for path rendering and fine rasterization. Therefore, if a test case is indicated to run with two threads, there will actually be three threads in total: One main thread and two worker threads. Because of this, a direct comparison of the results of the two renderers with a low thread count should be done with care, as the results might be slightly biased towards Vello CPU. With that said, the main goal of this section is to analyze the scaling behavior as the thread count increases, where this discrepancy becomes increasingly irrelevant as the thread count increases, especially for the 8-thread case since our machine only possesses 8 cores in total.

As will be seen shortly, both renderers can give remarkable performance boosts depending on the exact configuration but for larger thread counts are far away from achieving completely linear speedup relative to the thread count. At least a partial explanation for this behavior is that the machine has two efficiency cores, but another factor is likely the overhead that arises from intermediate communication between the threads. As will be shown shortly, Blend2D seems to overall have the more "well-rounded" multi-threading mode, achieving decent speedup even for simple paths where Vello CPU has

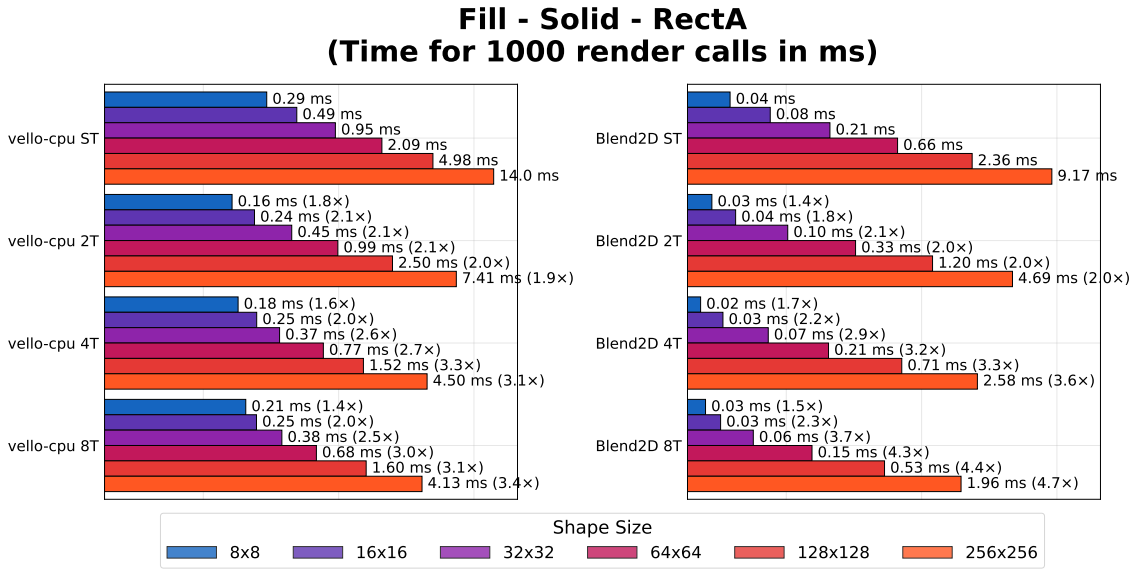


Figure 50: The running times for the test “Fill - Solid - RectA”.

some shortcomings. On the other hand, Vello CPU’s approach to multi-threading seems to lead to higher speedups as the processing time per path increases.

We begin our analysis by once again considering the simplest case of drawing simple rectangles in Figure 50. A pattern that will also become evident in other benchmarks is that the speedup for smaller shapes is usually smaller than for larger ones. The reason for this is that the needed processing time for handling small paths is so small that any speedup is eclipsed by the overhead that comes from using multi-threading. As the size of the rectangle increases, more rasterization work needs to be performed, which implies a longer processing time and therefore better utilization of different threads. A performance analysis of Vello CPU using a profiler confirms the fact that for 8x8, the majority of the time in each thread is spent on yielding and overhead that arises from the communication. For 256x256, the speedup reaches 3.4 but is still below Blend2D’s speedup of 4.7x. Profiling revealed that the main bottleneck seems to be coarse rasterization. In general, coarse rasterization consumes very little time, but since this is the only part of the pipeline that is currently not parallelized, it does become more significant as the number of threads increases. We therefore consider researching ways to lift this limitation important future work (see Section 6).

As can be seen in Figure 51, things improve as the complexity of the path increases. In the given figure, we only consider line-based shapes, meaning that the most time-consuming step is strip generation, but as can be seen, we already achieve much higher speedups in this test case. For small shapes, we are still restricted to a speedup of two to

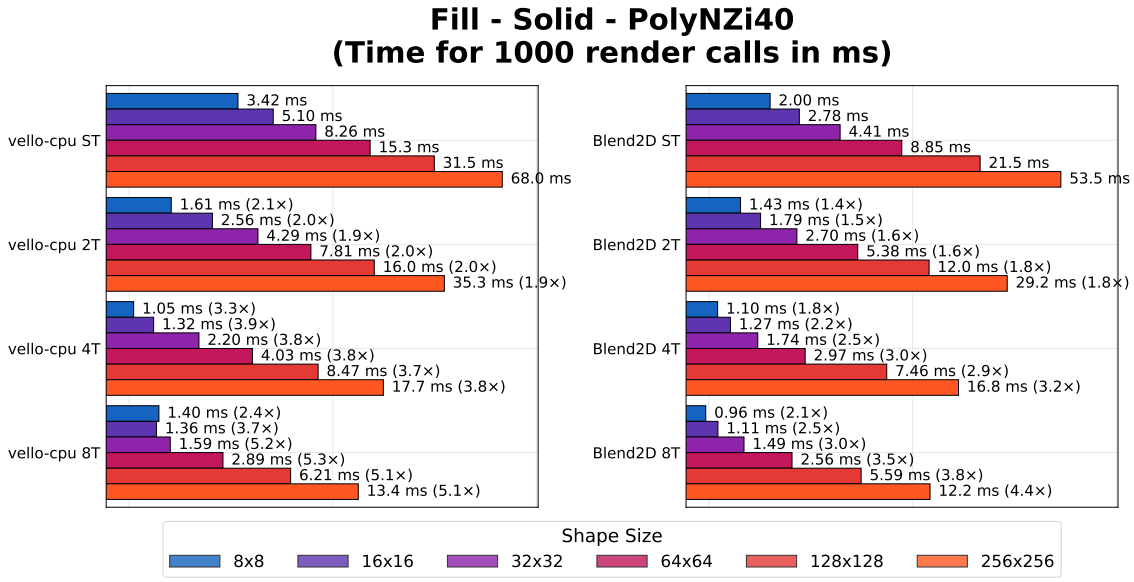


Figure 51: The running times for the test “Fill - Solid - PolyNZi40”.

three (note in particular how the speedup actually gets worse when using eight threads), but for larger versions of the shape, the speedup gets increasingly better.

Similarly high speedups are achieved once curve segments or strokes are thrown into the mix, as can be observed in Figure 52. Blend2D and Vello CPU achieve similar speedups when only using two threads, but when increasing that number two four or eight threads, Vello CPU seems to be at an advantage.

Finally, it is important to remember that the second component that can be parallelized in the pipeline concerns fine rasterization, which can become a bottleneck when drawing shapes with complex paints. As can be seen in Figure 53, there does not seem to be a clear winner when comparing Blend2D and Vello CPU. Blend2D seems to achieve higher speedups for the 256x256 case, while Vello CPU scales better for smaller sizes (for unknown reasons, Blend2D actually experiences a slowdown for very small sizes). It is surprising to us that the speedup for Vello CPU is only limited to 4.1, as the bottleneck for this test case is fine rasterization, which conceptually is much easier to parallelize since each wide tile can be processed completely independently.

We conclude that both renderers have powerful multi-threading capabilities but with different strengths and weaknesses. As was shown, there is no clear “winner” in most benchmarks. Sometimes, Blend2D achieves better scaling with smaller shapes but not with larger ones, but sometimes, it is the other way around. We believe that Vello CPU could achieve even higher speedups if a way is found to eliminate the coarse

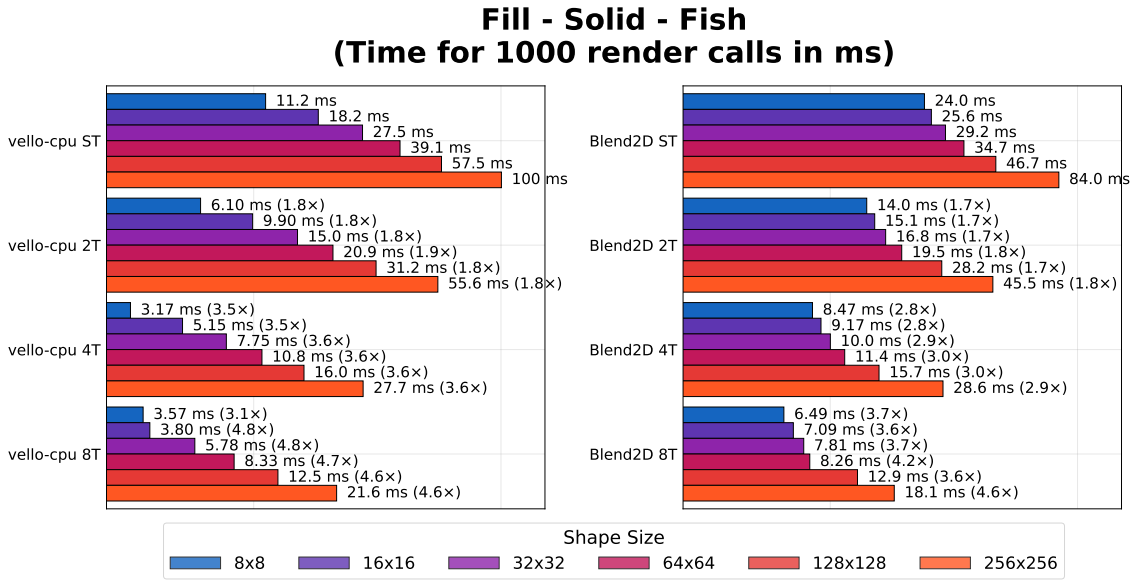


Figure 52: The running times for the test “Fill - Solid - Fish”.

rasterization bottleneck, although that remains to be empirically validated as part of the future work.

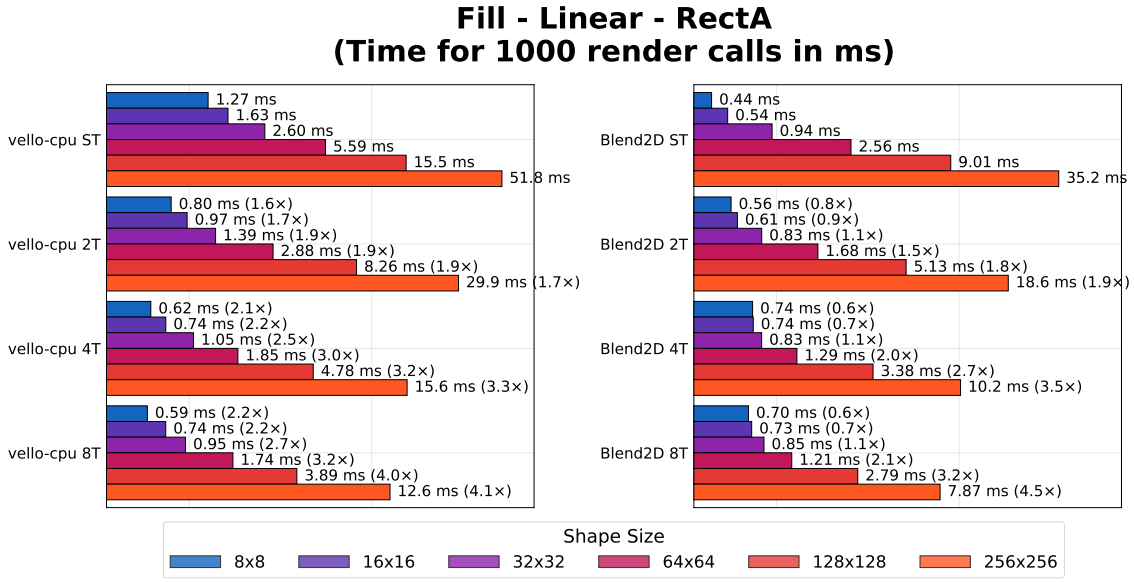


Figure 53: The running times for the test “Fill - Linear - RectA”.

Chapter 6

Conclusion & Future Work

The main goal of this thesis was to determine whether sparse strips are a suitable approach for implementing a state-of-the-art 2D renderer. To this end, we built a CPU-based 2D renderer with the given paradigm at its core to validate that all of the features expected of a 2D renderer are compatible with the approach[35]. Our experiments demonstrate that Vello CPU exhibits competitive performance in most cases for small-sized shapes and outperforms nearly all other renderers, including Skia or Cairo, as the shape size increases. An exception is Blend2D, which our renderer cannot beat in the vast majority of cases in terms of raw performance output, which we largely attribute to the efforts that have been put into optimizing the library. We also show that Vello CPU can achieve even higher speedups by making use of multi-threading, something that many other mainstream renderer cannot make use of due to architectural reasons. It is therefore valid to draw the conclusion that the sparse strips paradigm is very effective and forms a suitable and promising basis for a performant renderer.

Four avenues for future work that could be explored are now presented.

First, it would be interesting to look into how this approach could be adapted for GPU-based rendering. While CPU-based rendering has its use cases, for most interactive rendering workloads it is usually preferable to make use of hardware-acceleration for better throughput. In fact, at the time of writing, there is already an ongoing parallel attempt to create a CPU/GPU-hybrid renderer[36] where the CPU is responsible for all stages up to coarse rasterization and the final fine rasterization happens on the GPU. This current approach already works and highlights the usefulness of having a pipeline consisting of separated stages with well-defined inputs and outputs, but there is a lot more exploration work to be done. A further interesting question would be to determine whether the strip generation stage (in particular, the part of strip generation that is responsible for calculating anti-aliasing) can be moved to the GPU as well, so that all the heavy pixel-level calculations are completely performed on the GPU, while only the more sequential parts of the pipeline (flattening, tile generation and coarse rasterization)

[35] Other commonly-expected features of a 2D renderer include masks and layer isolation, which *are* supported by Vello CPU, but have not been treated in this work due to space reasons.

[36] https://github.com/linebender/vello/tree/main/sparse_strips/vello_hybrid (accessed on 11.09.2025)

are done by the CPU. As part of this, it might also be worthwhile to explore whether a different tile size (for example 8x8) is more suitable for GPU-based rendering compared to the currently-used 4x4 tile size on CPU. While doing so would mean that we need to perform more superfluous anti-aliasing calculations, this is not as much of a problem since the GPU can do those much faster than the CPU. The advantage is that it will reduce the time needed for tile generation and sorting on the CPU, which is likely to form a bottleneck in this GPU-first approach.

Next, the evaluation has shown that Vello CPU has a certain weakness when handling small geometries. It was also asserted that the most-likely culprit for this behavior is, apart from some algorithmic inefficiencies that should also be addressed, the fixed 4x4 size of tiles, resulting in unnecessary computational work during strip generation. While it is not possible to introduce variable-height strips, it *is* possible to introduce variable-width strips. The way this could be achieved is by letting each tile store the horizontal extents of the containing line and then only computing winding numbers for pixel-columns that are actually covered by a line. A possible downside would be that future steps in the pipeline can no longer assume that the width of a strip is a multiple of four, an assumption that is at the moment baked into most of the SIMD code.

Third, as was briefly touched upon in Section 3.6.4, a considerable advantage of adopting the sparse strips approach is that it allows for efficient path caching, which is especially relevant when rendering scenes with lots of text. Unlike traditional approaches where caching usually requires rendering the path to an intermediate bitmap image, in our case, it is possible to only cache the sparse strips representation of the path, leading to lighter storage requirements, especially as the size of the path increases. With that said, there are is a challenge that needs to be addressed for the technique to be fully viable. When caching a glyph[37], the usual expectation is that it can be reused multiple times in different locations, so that we can for example first render the letter “t” at the location (120, 135) and then a second time at the location (133, 160). Applying an x-shift to the sparse strips representation of a path is not too challenging as it simply involves updating the x coordinate of each strip. However, a considerable difficulty are shifts in the y direction. The problem is that strips have a fixed height of four, and it is therefore not easily possible to apply arbitrary y shifts to it, unless the shift happens to be a multiple of four. An interesting research direction would therefore be figuring out

[37] In very simplifying terms, a glyph *roughly* corresponds to a letter in the alphabet. Therefore, when talking about glyph caching, it refers to the ability to cache the rendered outline of a letter so that it can be reused in different locations on the screen.

whether it is possible to efficiently update the sparse strips representation of a path to make translations in the vertical direction possible.

Finally, the fourth major point that could benefit from more dedicated attention is multi-threading. On the one hand, this includes investigating more closely why the speedup for parallel fine rasterization is not as high as expected. On the other hand, this requires more fundamentally exploring how coarse rasterization, which currently forms the only serial bottleneck in the pipeline, can be made more parallel. As was previously mentioned, there are different approaches that could be explored, but it remains to be determined whether it can be fully parallelized in a performant way.

Bibliography

[1] M. J. Kilgard and J. Bolz, "GPU-accelerated path rendering," *ACM Trans. Graph.*, vol. 31, no. 6, Nov. 2012, doi: 10.1145/2366145.2366191.

[2] R. Li, Q. Hou, and K. Zhou, "Efficient GPU path rendering using scanline rasterization," *ACM Trans. Graph.*, vol. 35, no. 6, Dec. 2016, doi: 10.1145/2980179.2982434.

[3] F. Ganacim, R. S. Lima, L. H. de Figueiredo, and D. Nehab, "Massively-parallel vector graphics," *ACM Trans. Graph.*, vol. 33, no. 6, Nov. 2014, doi: 10.1145/2661229.2661274.

[4] "Blend2D Performance." Accessed: Aug. 21, 2025. [Online]. Available: https://blend2d.com/performance.html

[5] "Facebook." Accessed: Aug. 17, 2025. [Online]. Available: https://www.facebook.com/login

[6] Adobe Systems Incorporated, *Portable document format — Part 1: PDF 1.7*. 2008. Accessed: Aug. 19, 2025. [Online]. Available: https://opensource.adobe.com/dc-acrobat-sdk-docs/pdfstandards/PDF32000_2008.pdf

[7] "Scalable Vector Graphics (SVG) 1.1." 2011. Accessed: Aug. 19, 2025. [Online]. Available: https://www.w3.org/TR/SVG11/

[8] J. Vince, *Mathematics for Computer Graphics*, 5th ed. Springer Publishing Company, Incorporated, 2017.

[9] M. Anderson, R. J. Motta, S. Chandrasekar, and M. Stokes, "Proposal for a Standard Default Color Space for the Internet - sRGB," in *International Conference on Communications in Computing*, 1996. [Online]. Available: https://api.semanticscholar.org/CorpusID:6586058

[10] "HTML." 2025. Accessed: Aug. 20, 2025. [Online]. Available: https://html.spec.whatwg.org/

[11] T. Porter and T. Duff, "Compositing digital images," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, in SIGGRAPH '84. New York, NY, USA: Association for Computing Machinery, 1984, pp. 253–259. doi: 10.1145/800031.808606.

[12] W3C, “Compositing and Blending Level 1,” W3C Candidate Recommendation, Mar. 2024. Accessed: Sept. 01, 2025. [Online]. Available: https://www.w3.org/TR/compositing-1/

[13] M. Maule, J. L. D. Comba, R. Torchelsen, and R. Bastos, “Transparency and Anti-Aliasing Techniques for Real-Time Rendering,” in *2012 25th SIBGRAPI Conference on Graphics, Patterns and Images Tutorials*, 2012, pp. 50–59. doi: 10.1109/SIBGRAPI-T.2012.9.

[14] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. USA: Prentice-Hall, Inc., 2006.

[15] D. F. Nehab, “Converting stroked primitives to filled primitives,” *ACM Transactions on Graphics (TOG)*, vol. 39, pp. 137:1–137:17, 2020, [Online]. Available: https://api.semanticscholar.org/CorpusID:221105893

[16] R. Levien, “Flattening quadratic Bézier curves.” Accessed: Aug. 25, 2025. [Online]. Available: https://raphlinus.github.io/graphics/curves/2019/12/23/flatten-quadbez.html

[17] J. Hasselgren, T. Akenine-Möller, and L. Ohlsson, “Conservative Rasterization,” in *GPU Gems 2*, Addison-Wesley, 2005, pp. 677–690.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

- (●) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].
- ( ) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].
- ( ) I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

High-performance 2D graphics rendering on the CPU using sparse strips

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Stampfl	Laurenz

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
10.10.2025	L. Stampfl

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard