Instituto Nacional de Matemática Pura e Aplicada

A Fully-Parallel Pipeline for High-Quality Rendering of Vector Graphics Illustrations

Francisco Ganacim

Doctoral Thesis

Advisor: Diego Nehab

Co-Advisor: Luiz Henrique de Figueiredo

Rio de Janeiro, March 2015

Agradecimentos

Agradeço aos meus pais pelo apoio e dedicação ao longo de todos os anos.

Agradeço a Nara, sem seu apoio, amor e companheirismo, esta conquista não seria possível. Agradeço a Cecília, cuja presença, além de dar norte e propósito, é motivo de felicidade constante.

Agradeço aos meus orientadores Diego Nehab e Luiz Henrique de Figueiredo por compartilharem seu conhecimento e experiência. Além disso, agradeço por todas as oportunidades de crescimento profissional que me foram dadas ao longo destes anos.

Agradeço a todos os colegas de VISGRAF. Em especial aos amigos André Máximo, Djalma Lúcio, Leandro Cruz, Leonardo Koller e Rodolfo Schulz, cuja ajuda foi essencial em vários momentos. Agradeço ao professor Luiz Velho pelo ambiente único que é o VISGRAF.

Abstract

Vector graphics are a versatile form of representing visual information. They are resolution independent and can represent a broad range of content: text fonts, maps, user interfaces, and games.

Rendering vector graphics is not an easy task. High quality rendering requires huge amount of processing power. To achieve real-time performance, many applications sacrifice the quality of their resulting images. The solution is to parallelize rendering using GPUs.

In this thesis we describe a rendering pipeline that is fully parallel, and implemented on the GPU. We show how to create a novel acceleration data structure, called shortcut tree, that allow us to render high quality images using antialiasing filters of large support. Our results surpass the state-of-the-art in quality and performance.

Contents

Introduction

Vector graphics are a versatile form of representing visual information. They are resolution independent and can represent a wide variety of content; from text fonts to maps, from user interfaces to games.

In general, vector graphics follow the model created in the seminal work of Warnock and Wyatt [1982]. In their model, an illustration is composed by many layers, each layer containing a shape or other geometric primitives colored by a solid color, a gradient or a texture. The final figure is the composition of the layers in order.

Alternative models exist, such as Diffusion Curves [Orzan et al., 2008], where instead of specifying a shape's color or texture, we specify the figure's gradient at the boundary of the many shapes composing the scene. The final result is found by solving a Poisson problem. Nevertheless, the majority of the vector graphics in use today, such as SVG and OpenVG, use the classic specification. Therefore, rasterizing these figures is a problem of major importance.

Many CPU implementations of raterizers exist today, such as Cairo [Packard and Worth, 2003], Skia [2015] and others, with varying degrees of quality and performance. We believe that one way of meeting today's users' expectations of performance and quality is to heavily parallelize the rasterization. For that, the platform of choice are the GPUs.

Bringing vector graphics rasterization to the GPU is not a trivial task. The non-locality of the point-in-shape problem poses a challenge that requires a reformulation of traditional CPU methods. GPU methods can be classified into two categories: immediate mode and vector textures.

In immediate mode, shapes are sequentially processed and the results are blended into the output image. Parallelism is achieved when processing the fragments created by each shape. Many such systems approximate antialiasing by transforming per-pixel coverage into transparency prior to blending. This conflation leads to

incorrect rendering of correlated layers. This policy of immediate-mode rendering is analogous to the z-buffering algorithm for 3D rendering.

Traditionally, vector textures methods sequentially preprocess the scene in order to build an acceleration data structure that is later used for rendering. These data structures are built by exploiting spatial coherence to reduce the amount of computation needed for rendering. When rendering, samples can be collected in parallel to create the output image. Antialiasing may be computed by an analytic approximation or by supersampling. Vector textures are analogous to rendering by ray-casting in 3D. They allow samples to be freely evaluated using an acceleration data structure that is adapted to the geometry of the scene.

We built our pipeline [Ganacim et al., 2014] to be massive parallel in each stage from preprocessing to rendering. At preprocessing we create an acceleration data structure called the *shortcut tree*. When rendering, we query the shortcut tree to evaluate sample colors. Our renderer component is able to share sample colors among many pixels, allowing the use of antialiasing filters with large support. Since both preprocessing and rendering are efficient and fully parallel, we can render complex illustrations surpassing state-of-the-art quality and performance.

Our contributions include:

- The *shortcut tree*, a novel hierarchical acceleration data structure that enables efficient random access to the color of each point in the illustration;
- *Fully parallel* construction of the shortcut tree, including novel subdivision and pruning algorithms;
- New *segment abstraction* that eliminates the need for *all* intersection computations throughout the pipeline. Conversion from input segments during preprocessing requires only monotonization, splitting at double and inflection points, and implicitization;
- The *flat clipping* algorithm that supports arbitrary nesting of clip-paths without resorting to recursion or even a stack.
- A *front-to-back* rendering pipeline that aborts computation when full opacity is reached. This is done independently per sample;
- Support for user-defined effects in image space, sharing samples under wide antialiasing filters, using large sampling rates, minimizing control-flow divergence as well as memory and bandwidth usage.

We begin chapter 1 by defining a model for vector graphics. We introduce the concept of filled paths and how they can be colored to create complex illustrations.

This model is expressive enough so we can encompass most of the functionality described in the SVG (Dahlström et al. [2011]) and OpenVG (Rice and Simpson [2008]) standards. Next, we show how images can be sampled and discretized with high quality, avoiding common problems such as aliasing and noise. At the end of the chapter, we provide a complete but unoptimized algorithm for vector graphics rasterization.

In chapter 2, we review the state of the art in area and discuss the most relevant work on GPU rasterization of vector graphics.

In chapter 3, we show how the algorithm presented in chapter 1 can be accelerated. By rearranging the rendering loop, we can compute high quality images evaluating each sample only once. We also build an acceleration data structure (quadtree) to speed up the evaluation of sample colors. This chapter lays the foundation in which we build our GPU rasterization pipeline (chapter 4).

Results of our work are presented in chapter 5. The last chapter discusses future directions.

Chapter 1

Preliminaries

1.1 Vector Graphics

1.1.1 Structure of a Scene

Our model for vector graphics follows the model proposed in the seminal work of Warnock and Wyatt [1982]. We begin with an empty *canvas* over which, we paint *layers* of *colored shapes* to compose the final *image*.

The operation of painting a colored shape onto the canvas corresponds to the metaphor of a silk-screen printer pushing colored ink through a stencil onto paper. Figure 1.1 exemplifies of this operation. The ink (or paint) above defines the colors to be used at all points in the plane. This color can be constant throughout the plane or vary spatially. In the middle, geometric shapes delineate the stencil. Points *inside* the shape should be thought of as *open* and let the ink pass through the stencil to be deposited onto the canvas. Points *outside* the shape are closed, and block the ink. At the bottom, we see the result of the painting operation.

To create complex scenes we can apply the painting operation repeatedly, depositing layer upon layer of ink onto the canvas. Some examples of complex vector graphics scenes are shown in figure 1.3.

Many of the terms used above remain loosely defined. For instance, we have not defined precisely what a color is, or how we specify colors that vary spatially. How does the color of one painting operation interact with the colors already on the canvas? How do we specify shapes and what is inside in this context? We answer those questions on the next few sections.

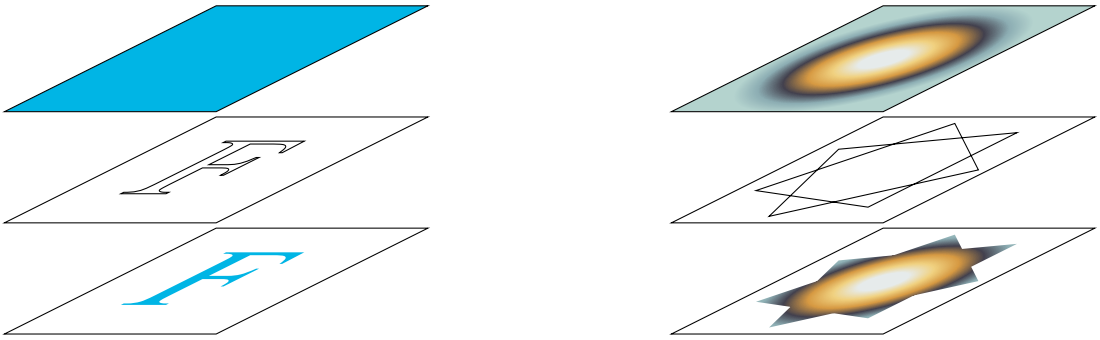


Figure 1.1: Examples of the silk-screen metaphor.

1.1.2 Spaces and Transformations

The space in which the scene (or illustration) is defined is called the *scene space* (or *world space*). The scene space is a copy of $\mathbb{R}^2$ with an affine *frame of reference*: an origin point o and two orthogonal directions $\vec{x}$ and $\vec{y}$. Coordinates is this space are called scene (or world) coordinates. Shapes are defined in their own coordinate system, called the *object space* (with object coordinates). An affine transformation on the plane maps shapes from object space into scene space. These transformations are called *modeling transformations*.

It is our ultimate goal to produce an image of a given scene. For that we select a rectangular area in scene space to be rasterized. We call this area *window*. We also choose the width w and height h of the image, in pixels. The window is mapped into a third space called *image space*. The resulting set of points is called *viewport*. In image space the viewport has its bottom left corner at $(0, 0)$ and its size is $w \times h$. In image space two consecutive samples have distance 1.

Figure 1.2 shows one shape defined in object space being mapped by three different transformations into scene space.

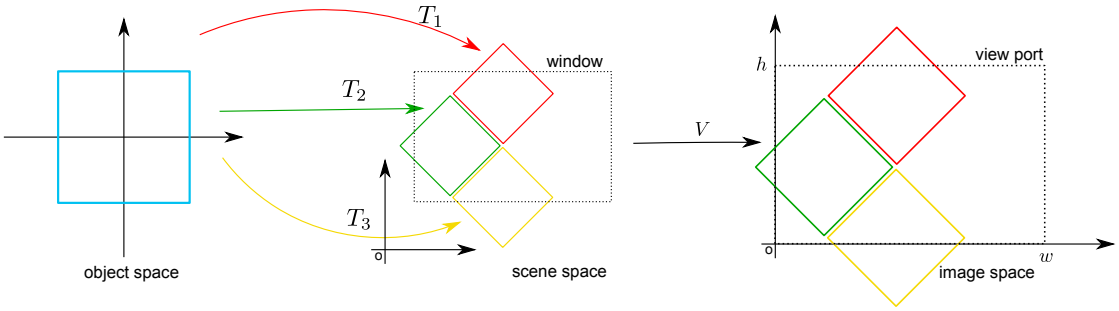
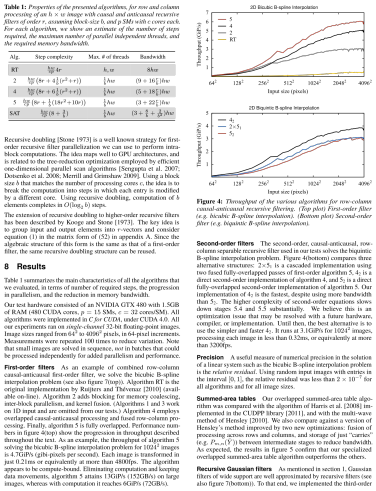
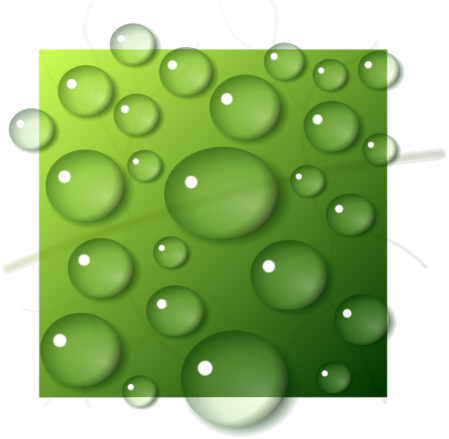


Figure 1.2: Examples of modeling transformation.

(a) Lion.

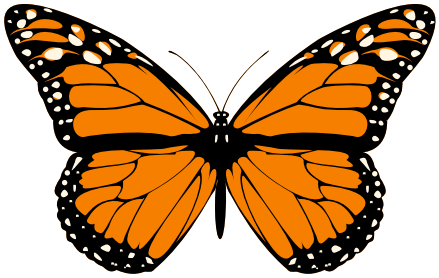


(b) Paper.

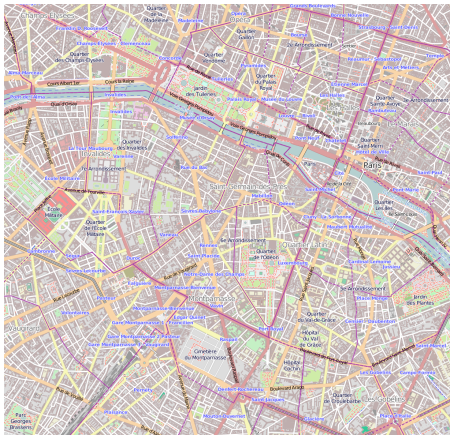


(c) Drops.

(d) Tiger.



(e) Butterfly.



(f) Map of Paris (Paris30k).

Figure 1.3: Complex vector graphics scenes.

1.1.3 Shapes and Paths

To define the geometry of shapes we use *paths*. A path is a collection of contours on the plane. A contour α is a function

$$\begin{aligned} \alpha : [a, b] \subset \mathbb{R} &\to \mathbb{R}^2 \\ t &\mapsto (x(t), y(t)) \end{aligned}$$

that is continuous and piecewise smooth. Figure 1.4 shows examples of paths: on the left a path with many contours, on the right a self-intersecting contour.

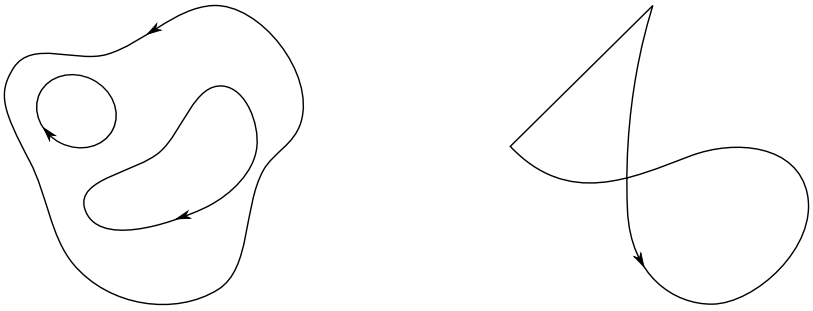


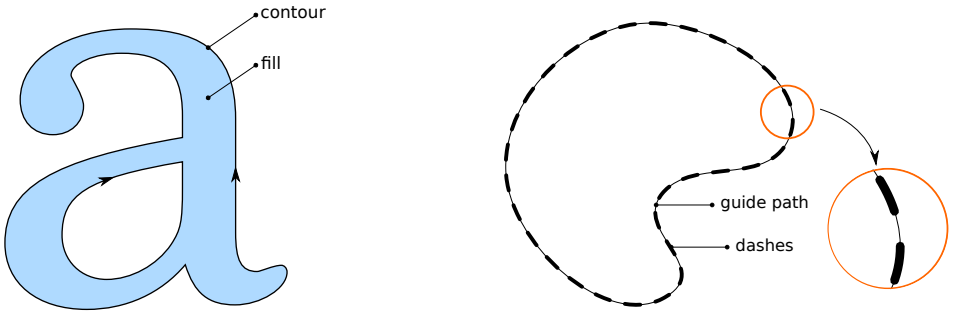
Figure 1.4: Examples of paths.

Shapes are defined by paths in two different ways: *filled paths* and *stroked paths*. Filled paths use contours to enclose a limited area of the plane, whose points are filled with some color. Figure 1.5a shows a typical use of filled paths. Stroked paths are constructed using contours as guides. These guides—along with others parameters such as *width* and *dash pattern*—define lines in a drawing. Figure 1.5b shows a stroked path.

Filled Paths

Given a *closed* contour, the following theorem gives us a rigorous notion of *inside*.

Theorem 1. *(Jordan Curve) Let α be a simple (non-self-intersecting) closed curve in the plane $\mathbb{R}^2$ and let C be its trace. Then, the open set of points $\mathbb{R}^2 \setminus C$ consists of exactly two connected components. One of these components is bounded (and is called the* **interior***) and the other is unbounded (and is called the* **exterior***), and C is the boundary of each component.*

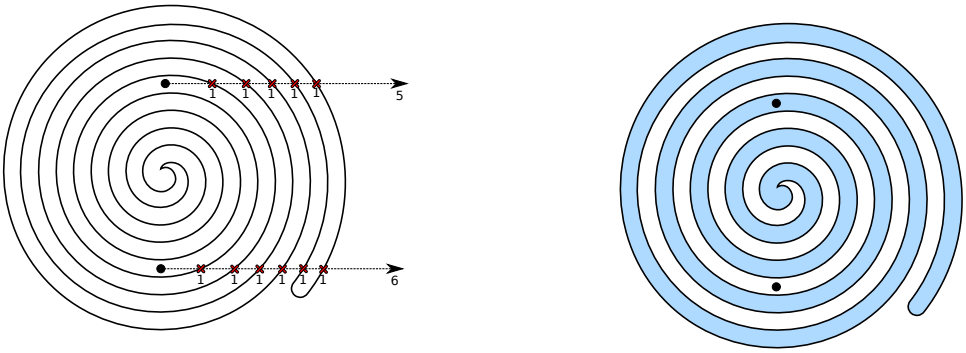


(a) A filled path. *(b) A stroked path.*

Figure 1.5: Example of filled and stroked of paths.

Therefore, a point p in the plane is inside the shape delimited by the closed contour α if p belongs to the interior component of $\mathbb{R}^2 \setminus C$, given by the Jordan Curve theorem. The theorem also gives the tool we need to define a procedure that determines whether a given point is inside any given contour.

Given a point $p \in \mathbb{R}^2$ and a simple closed contour α, we choose a direction $d \in \mathbb{R}^2$, $d \neq 0$, and define a *ray* $r(u) = p + ud$. We count the number of intersections between the ray r and the curve α. If the number is odd, then p is inside. Otherwise, p is outside. This is called the *even-odd* rule. Figure 1.6 shows an example of this method.



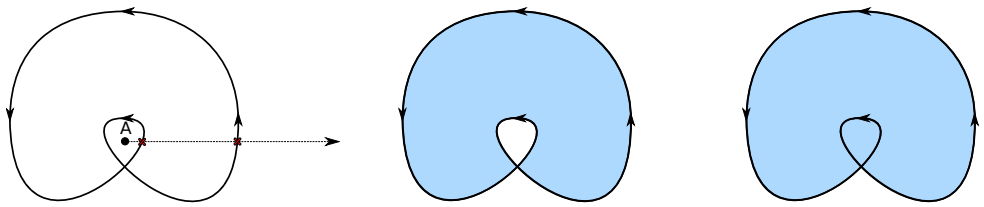
(a) Shooting rays to determine the interior of the shape. *(b) The interior region painted.*

Figure 1.6: Even-odd test.

We can justify the method as follows: Jordan's Curve theorem states that C is the boundary of each component (inside and outside). Each intersection indicates a transition from outside to inside or the opposite. Hence the number of intersection

indicates the number of transitions between inside and outside. We know that the point $r(u)$ is outside for a large value of $u > 0$, since inside is bounded. Following the parameter u as it decreases to zero, we have that the first intersection (the intersection with the largest u) is a transition from outside to inside. The next intersection will necessarily be a transition from inside to outside. This is valid for all consecutive pairs of intersections. Therefore an even number of transitions indicates that $r(0)$ is also outside. It follows that an odd number of transitions indicates that $r(0)$ is inside. We can ignore points where the curve only tangencies the ray, since there is no transition between inside and outside.

We would like to generalize our method for a larger class of contours. Consider what happens when we apply our method to contours with self intersections. Figure 1.7a shows a self-intersecting contour and the result of the method is shown in figure 1.7b. This result seems somewhat arbitrary. Region A could be considered inside since it is surrounded by the curve. Another common situation appears when we want to draw regions with holes (figure 1.5a).



(a) *(b)* *(c)*

Figure 1.7: In (a) we show that the inside/outside criteria is not well defined for region A. In (b) we show the result of applying the even-odd rule. In (c) we show the result of applying the non-zero rule.

To deal with this situation in a more consistent fashion, we need to generalize the even/odd rule. We begin by assigning, to each intersection between a ray $r(u) = p + ud$ and a contour α, a *sign*. Intersections are characterized by a pair of parameters $(\bar{u}, \bar{t})$ satisfying the following relations:

$$\begin{aligned} \bar{u} &> 0, \\ \bar{t} &\in [a, b], \\ r(\bar{u}) &= \alpha(\bar{t}). \end{aligned} \tag{1.1}$$

We define the sign of the intersection at $(\bar{u}, \bar{t})$ as $+1$ if the contour crosses $r(u)$ going counterclockwise, -1 if it crosses going clockwise, and 0 if the r is only

tangent to the curve. This is shown in figure 1.8. This definition can be made mathematically precise: we find $\varepsilon > 0$ such that no other intersection occurs in the interval $(\bar{t} - \varepsilon, \bar{t} + \varepsilon) \subset [a, b]$. Let s be the segment going from $r(\bar{t} - \varepsilon)$ to $r(\bar{t} + \varepsilon)$. If s does not cross r then the intersection is a tangent. If a crossing does occur, then the sign is given by comparing the direction d (from r definition) with the direction of the segment s using a vector product. We will give an easier way to computed this sign after defining monotonic segments.

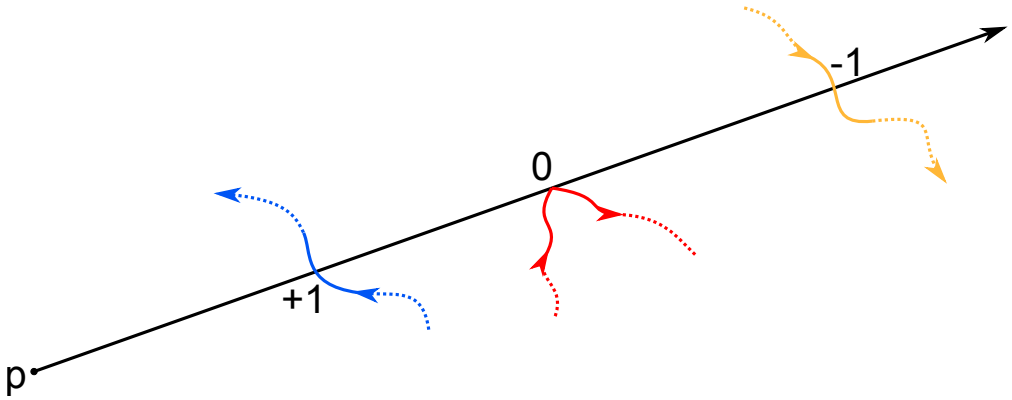


Figure 1.8: Intersection crossing.

Now we can count the intersections using the sign as defined above; this number is called the *winding number* of the point about the contour.

With the winding number, two different rules can be applied to determine if points are inside or outside the contour. The even-odd rule, which is the same as before, but now applied to signed numbers. And another rule called *non-zero* rule, which defines as inside those points that have the winding number different from zero, and define as outside those points with winding number zero. Figure 1.7b and Figure 1.7c shows examples of both rules applied to a self intersecting contour. It is worth noticing that for simple contours both rules give the same result.

The extension of the method to general paths is straightforward. We define the winding number of a point relative to a path as the sum of the winding numbers of the point relative to each contour in the path. Now we can form complex regions, including regions with holes. Figure 1.9 shows this procedure written as an algorithm.

```
function is_inside( p, S )
    -- Receives: the point 'p' in ℝ², and the path 'S' defining a shape.
    -- Returns: true if p is inside the shape, false otherwise.
    local r = ray( p, {1, 0} ) -- creates a ray in the direction (1,0).
    local wn = 0 -- sets winding number to zero.
    for c in contours( S ) do -- iterates over all contours in S.
        local I = intersections( r, c ) -- finds the intersections between r and c
        for {u, t} in I do -- iterates over the intersections
            if u > 0 and a <= t and t <= b then
                -- sum the sign to the intersection
                local wn = wn + sign( r, u, c, t )
            end
        end
    end
    -- apply the path's rule: even/odd or non-zero
    return apply_rule( C, wn )
end
```

Figure 1.9: in/out procedure.

Stroked Paths

Conceptually, stoked paths are drawn by dragging a straight-line pen held perpendicular to the path's contours, to create a "wide" line. The width is chosen by the user creating the path. Figure 1.10 illustrates this operation.

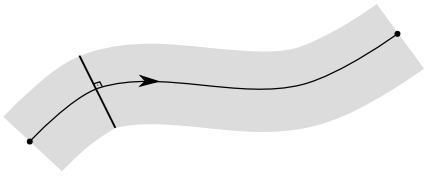
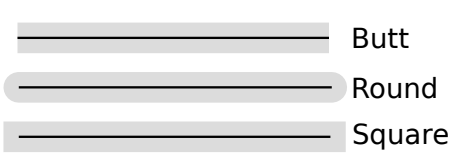


Figure 1.10: stroke definition.

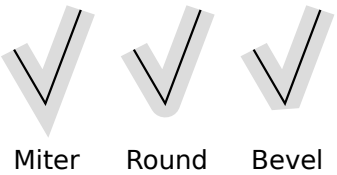
It is also possible to define dashed patterns. Dashed patterns are a periodic sequence of lengths defining segments where the pen is "up" or "down". Along with the pattern, the user may specify a dashing *phase*. The dash phase defines the starting point at the pattern that is associated with the start of the path.

If an open contour is stroked, the points at the beginning and end of each dash in the contour may be decorated with a *cap*. Many types of caps are possible, figure 1.11a shows some of them. When the pen reaches a non-differentiable point

in the path, a gap may occur. To keep the drawing consistent, we add, at theses points, a *line join*. Figure 1.11b shows examples of common joins.



(a) Stroke caps.



(b) Stroke joins.

Figure 1.11: Stroked paths decoration.

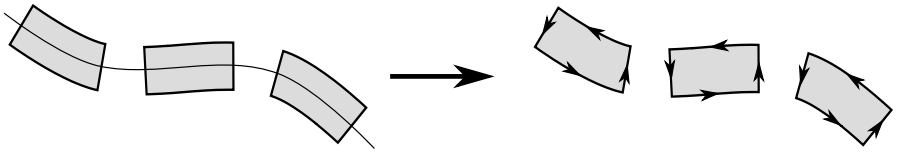


Figure 1.12: Conversion from stroked path to filled path.

To decide whether a given point p in the plane is inside the painted region of a stroked path, we could use a procedure based on the distance from p to the contours, followed by a test if p is inside a cap or line joint. But this procedure is computationally expensive and somewhat intricate. Instead, we chose to convert strokes to paths. To convert a stroked path to a filled path, we create a new path that encompass the region painted by the stroked path. This conversion is not without its difficulties, yet it allows us to deal with stroked and filled paths in a uniform fashion. Figure 1.12 exemplifies the conversion.

Path Representation

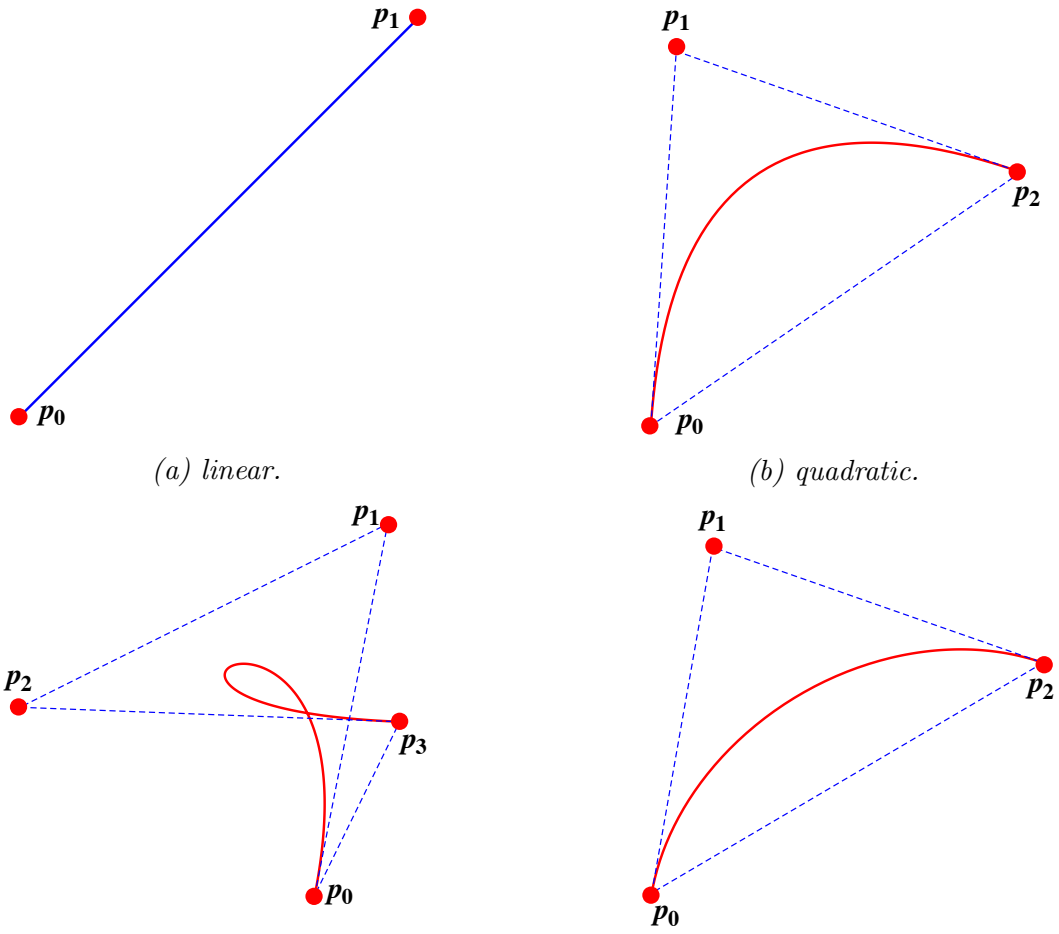
We have described, so far, an abstract model for paths. Our ultimate goal is to implement this model in a computer and to use it for practical purposes. That will limit our model to the class of paths that can be represented in a computer. In fact, we want paths that have a finite representation. We choose to use curves that are composed by sequences of integral and rational *Bézier segments* [Farin et al., 2002]. Bézier segments are defined by polynomials, and polynomials can be represented by a finite number of coefficients. We limit our model to integral Bézier segments of degree 1, 2, and 3 and rational Bézier segments of degree 2.

Each Bézier segment is specified by a sequence of control points. Two of the control points are the start and end points of the segment. For linear segments, those

points are enough, but for segments of higher degree, additional control points are necessary: one point for quadratic segments and two points for cubic segments. Rational quadratics also require an additional real number, which we call *projective weight*. The structure of Bézier segments is shown on Figure 1.13.

Integral Bézier segments is defined by:

$$b_1(t) = p_0(1-t) + p_1 t, \tag{1.2}$$
$$b_2(t) = p_0(1-t)^2 + 2p_1 t(1-t) + p_2 t^2, \tag{1.3}$$
$$b_3(t) = p_0(1-t)^3 + 3p_1 t(1-t)^2 + 3p_2 t^2(1-t) + p_3 t^3. \tag{1.4}$$



(a) linear. *(b) quadratic.*

(c) cubic. *(d) rational quadratic.*

Figure 1.13: Examples of Bézier segments.

In general,

$$b_n(t) = \sum_{i=0}^{n} B_{i,n}(t) p_i, \tag{1.5}$$

where

$$B_{i,n}(t) = \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i}. \tag{1.6}$$

are the nth-degree Bernstein polynomials. The points p_i are the *control points* of the segments. In general a n-degree Bézier segment has $n+1$ control points.

Arcs of ellipses can be represented by *rational* quadratic Bézier segments.

$$\begin{aligned} a_2(t) &= \frac{b_2(t)}{\sum_{i=0}^{2} B_{i,n}(t) w_i} \\ &= \frac{p_0(1-t)^2 + 2p_1 t(1-t) + p_2 t^2}{w_0(1-t)^2 + w_1 2t(1-t) + w_2 t^2}. \end{aligned} \tag{1.7}$$

We would like to be able to transform paths using transformations on the plane. Common transformations include scales, rotation, translation and their compositions. All these transformations can be modeled as *affine transformations* in the plane. The defining property of an affine transformation $A\colon \mathbb{R}^2 \to \mathbb{R}^2$ is: given a set of points $\{p_i \in \mathbb{R}^2, i = 1, \ldots, n\}$ and a set of real numbers $\{\lambda_i \in \mathbb{R}; i = 1, \ldots, n\}$, such that $\sum_{i=1,\ldots,n} \lambda_i = 1$, A satisfies

$$A\left(\sum_{i=1}^{n} \lambda_i p_i\right) = \sum_{i=1}^{n} \lambda_i A p_i. \tag{1.8}$$

On the other hand, the Bernstein basis satisfies

$$\sum_{i=0}^{n} B_{i,n}(t) \equiv 1. \tag{1.9}$$

This allow us to transform Bézier segments by affine transformations easily

$$A\left(b_n(t)\right) = \sum_{i=0}^{n} B_{i,n}(t) A p_i. \tag{1.10}$$

That is, paths defined by Bézier segments can be transformed by an affine transformation by simply transforming their control points with the sample transformation.

Monotonization of Bézier Segments

Each input segment is specified as a parametric curve $\gamma : [0, 1] \to \mathbb{R}^2$,

$$\gamma(t) = \big(x(t), y(t)\big).$$

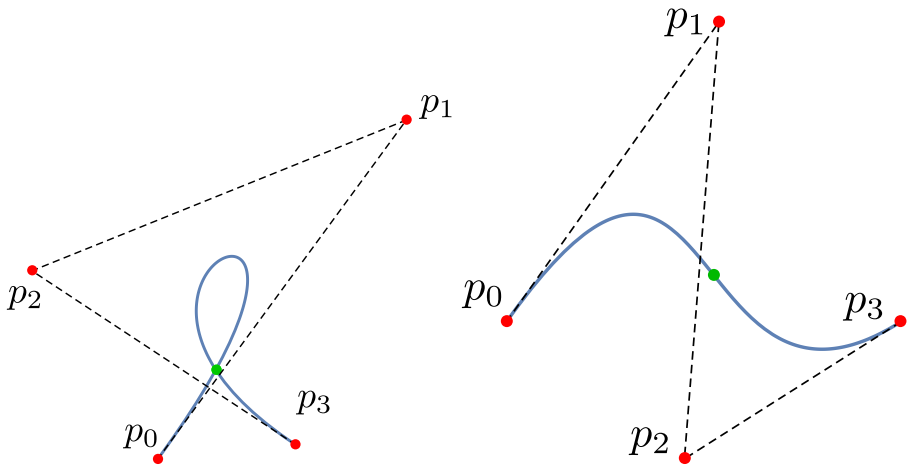
A segment γ is *monotonic* if the derivatives $x'(t)$ and $y'(t)$ never change sign in the interval $[0, 1]$. Given any segment, we can break it into monotonic pieces by finding the parameter values t_j

$$0 = t_0 < t_1 < \cdots < t_{k-1} < t_k = 1$$

that satisfy either of the equations

$$x'(t_j) = 0 \quad \text{or} \quad y'(t_j) = 0. \tag{1.11}$$

These t_j values break γ into k monotonic segments corresponding to the intervals $[t_{j-1}, t_j]$, for $j \in \{1, \ldots, k\}$. Each monotonic segments (i.e., γ restricted to $[t_{j-1}, t_j]$) can be recast as a Bézier segment in the interval $[0, 1]$. For that we use the multiaffine representation of Ramshaw [1988] to generate the control points corresponding to each parameter interval. Linear segments are monotonic on their own. Otherwise, finding the t_j leads to linear equations for integral quadratic segments, and to quadratic equations for rational quadratic and integral cubic segments. We use an algorithm by Blinn [2005] to solve the quadratics in a numerically robust way.



(a) (Loop) double point. *(b) (Serpentine) inflexion point.*

Figure 1.14: Cubic Bézier segments may have a double point or inflexion points.

We also break cubic Bézier at *double points* and *inflexion points* (see figure 1.14). Breaking at double point is useful for *implicitization*. Along with monotonization,

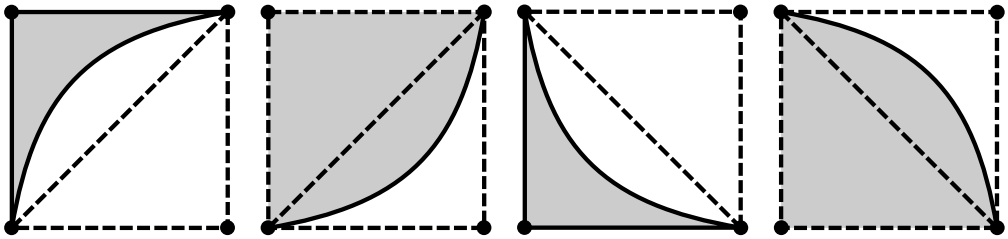


Figure 1.15: Monotonic segments never crosses the diagonal connecting its endpoints.

breaking at inflexion points guarantees that the resulting segments follow the pattern shown in figure 1.15. That is, the bounding box of the segment is the bounding box of the endpoints and the segment is either to the left or the right of the diagonal defined by the endpoints.

Monotonic segments simplify the task of computing intersection with *horizontal* rays easier. Since either $y'(t) \leq 0$ or $y'(t) \geq 0$, the y component of the segment is either a nondecreasing or nonincreasing function respectively. The y component of a horizontal ray is constant. Hence, the ray and the segment can intersect at most once. Figure 1.16 shows all possible cases of intersection. Intersections can be ruled out if the sample is above, below, or to the right of the bounding box. If it is to the left then there is an intersection. Otherwise, if the sample and segment are on opposite sides of the diagonal defined by the segment's endpoints, there is an intersection if, and only if, the sample is to the left of the diagonal. When the sample and the segment are on the same size of the diagonal, we must solve the system of equations 1.1.

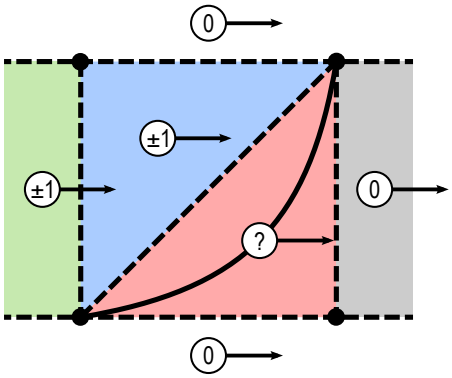


Figure 1.16: Possible cases of intersection between a ray and the segment.

1.1.4 Colors, Gradients and Textures

We have described how paths are represented. To continue with the description of our model we need to detail how to define the color layers used to fill paths. We start by defining what colors are and how we use them to fill paths. Next, we define gradients, which are a procedural way of defining color variations on the plane. After that, we describe textures, which use images as the color filling. We finish the section describing how colors in different layers interact to create the final image.

Colors

Physically, colors are a distribution of energy in the electromagnetic spectrum, more specifically in the *visible light* range of wave lengths. The set of all visible colors forms an infinite dimensional vector space. Different colors—i.e., different energy distributions—may be perceived as the same by a human observer, due to the human eyes and brain physiology. Using data from experiments with human subjects, it has been determined that the space of perceived color is a tree-dimensional space, more precisely a three-dimensional cone [Wyszecki and Stiles, 1982, Young, 1802]. Hence, we can represent visible colors by the linear combination of three basic colors, conventionally called red, green, and blue. Although these colors are not the pure spectral red, green and blue colors, they have a peak of energy at the red, green and blue regions of the spectrum. Along with the chromatic information, we carry with each color an opacity coefficient: a real number between 0 and 1. The opacity of a color determines how overlapped colored layers interact. A plausible, but not accurate, physical interpretation for the opacity is that it represents the percentage of the media covered by opaque tiny particles. Figure 1.17 illustrates this idea. A color is then represented by a tuple of tree number (r, g, b) along with the opacity coefficient α. The set of all colors is denoted by $\mathscr{C}$.

Filling Paths

A simple way of painting a filled path is by specifying one color for all points inside the path. This is called *solid color* filling and is the most common way of filling a path.

Solid colors alone are not expressive enough. For some illustrations, we need to specify colors in a richer way. For that, there are two useful methods. First, there are *gradients*, in which colors vary following a defined procedure. Second, there are *textures*, which are used to map an image onto the plane.

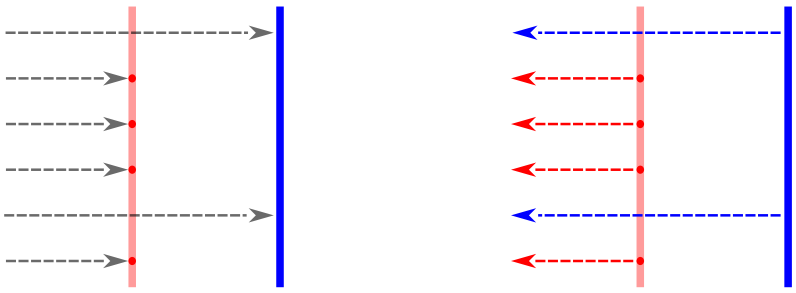


Figure 1.17: Opacity model showing incident and reflected light rays.

Gradients are composed of three different maps. The first maps the interval $[0, 1]$ into a linear transition of colors. It is called ramp. The second, extends ramp to the whole real line. The third, maps the entire plane into the real line and is defined by the type of gradient (linear or radial) and its many parameters.

To define a color ramp, we choose a set of colors with at least two elements, and to each element we associate a value in the interval $[0, 1]$. The first color associated with 0 and the last with 1. More precisely we have the set

$$S = \{(c_0, 0), (c_1, s_1), \ldots, (c_{n-1}, 1)\},$$

where c_i are colors and $n \geq 2$. We call each element of S a *stop*. Then, we define the function

$$\text{ramp}\colon [0, 1] \to \mathscr{C}.$$

Function ramp is a linear interpolation of each stop in S. The procedure to compute the color associated with a $t \in [0, 1]$ is shown in Figure 1.18.

```
function ramp( t, S )
    -- find the previous and next values of t,
    -- t_prev <= t <= t_next.
    t_prev, t_next = find_prev_and_next_t( t, S )
    d = (t - t_prev)/(t_next - t_prev)
    -- interpolates the colors in t_prev and t_next.
    return (1-d)*color(S, t_prev) + d*color(S, t_next)
end
```

Figure 1.18: Ramp.

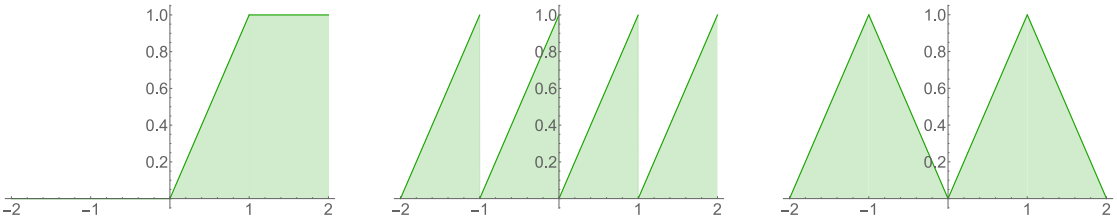
To extend ramp to all real numbers we may use one of the following three *extension functions*:

$$\text{clamp}(d) = \max(0, \min(1, d)), \tag{1.12}$$
$$\text{repeat}(d) = d - \lfloor d \rfloor, \tag{1.13}$$
$$\text{mirror}(d) = \text{hat}(2\,\text{repeat}(d/2)), \tag{1.14}$$

where

$$\text{hat}(d) = \begin{cases} d & \text{if } t \leq 1 \\ 2 - d & \text{if } t > 1 \end{cases}. \tag{1.15}$$

Their graphs are plotted in Figure 1.19.



(a) clamp. *(b) repeat.* *(c) mirror.*

Figure 1.19: Mappings between 0,1 and $\mathbb{R}$

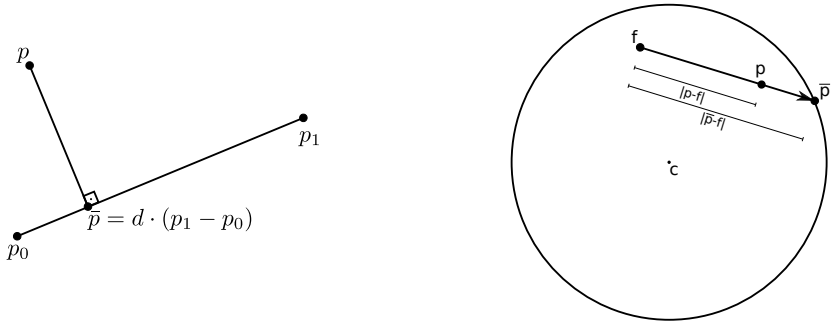
In Figure 1.20 we show how *linear* and *radial* gradients map points from plane into the real line. Linear gradients use two points to define a base segments, p_0 and p_1. Points in the plane are orthogonally projected into this segment, and the signed distance (d) between the projection and p_0 is computed:

$$d = \frac{\langle p - p_0,\, p_1 - p_0 \rangle}{\langle p_1 - p_0,\, p_1 - p_0 \rangle}, \tag{1.16}$$

where $\langle u,\, v \rangle$ is the inner-product of the vectors u and v.

Radial gradients operate in a similar way. Their definition is comprised by a circle (center and radius) and a focus (a point f inside the circle). To find the value of d, we shoot a ray from the focus in the direction of p, the intersection between this ray and the circle is our projection:

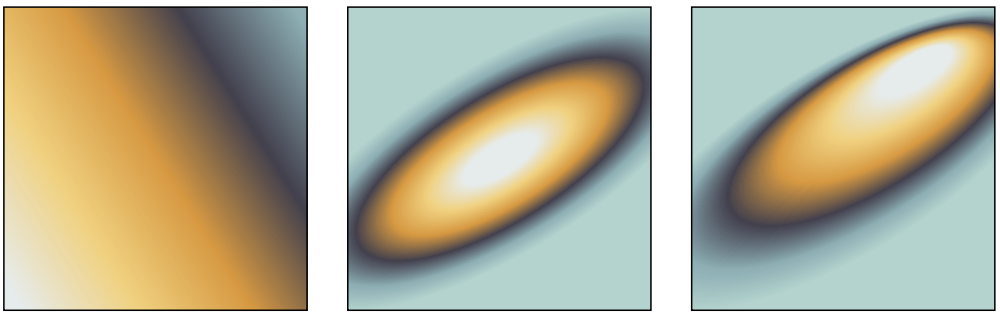
$$\bar{p} = \text{proj}_{f \to p}(p), \tag{1.17}$$
$$d = \frac{|p - f|}{|\bar{p} - f|}. \tag{1.18}$$



(a) Linear gradient. *(b) Radial gradient.*

Figure 1.20: Computing the gradient projection.

Both types of gradient can be accompanied by an affine transformation. These transformations allow us to create, for instance, ellipsoidal gradients and other deformations. We allow for f to be different from c to increase the generality of the radial gradients. The deformations created by making f different from c cannot be reproduced by changing the affine transformation of the radial gradient. We will discuss more of gradients transformations on section 1.4.1. Examples are shown in Figure 1.21.



(a) Linear gradient *(b) Radial gradient* *(c) Displaced focus*

Figure 1.21: Gradients transformed by an affine transformation.

The final form of coloring is using textures. Textures are specified by an image:

$$\begin{aligned} tex\colon [0,1] \times [0,1] &\to \mathscr{C} \\ (u,v) &\mapsto tex(u,v). \end{aligned}$$

along with an affine transformation T. To evaluate a texture we first apply the transformation T

$$(\bar{u}, \bar{v}) = T(u,v).$$

Then, we map $(\bar{u}, \bar{v})$ onto the set $[0,1] \times [0,1]$ using either one of the functions *clamp*, *repeat*, or *mirror* in each coordinate. After that, we evaluate tex to find the color.

1.1.5 Color Compositing

Now that we know how to find the color of a point inside a filled shape, we need to understand the effect of having multiples shapes in a scene. When shapes overlap, the color of the covered points are a *composition* of the color in the canvas with the color of the shape being painted. When an *opaque* (opacity 1) shape is painted on the canvas, the color of its internal points will be the color assigned to the shape, obscuring the layer below. Suppose we have a semi-transparent layer of color $C_f = (r_f, g_f, b_f)$ and opacity α_f over a layer of opaque color $C_b = (r_b, g_b, b_b)$. The final color is given by:

$$C_f \rhd C_b = \alpha_f C_f + (1 - \alpha_f) C_b \tag{1.19}$$

And its opacity is 1 since all light is reflected. The symbol $A \rhd B$ is short for "A **over** B". This definition only works for a color A **over** an opaque color B.

We would like to determine how two non-opaque colors interact. Following the computations made by Wallace [1981], we define the compositing operator in a way that the following equality holds for any pair of colors C_1 and C_2 and an opaque color C_b:

$$(C_1 \rhd C_2) \rhd C_b = C_1 \rhd (C_2 \rhd C_b).$$

Lets call the color $C_1 \rhd C_2$ by C, with opacity α, then

$$C \rhd C_b = C_1 \rhd (C_2 \rhd C_b).$$

Developing both sides we have,

$$\begin{aligned} \alpha C + (1 - \alpha) C_b &= \alpha_1 C_1 + (1 - \alpha_1) \left(\alpha_2 C_2 + (1 - \alpha_2) C_b\right) && (1.20) \\ &= \alpha_1 C_1 + (1 - \alpha_1)\alpha_2 C_2 + (1 - \alpha_1)(1 - \alpha_2) C_b, && (1.21) \end{aligned}$$

which gives

$$\begin{aligned} C &= \frac{\alpha_1 C_1 + (1 - \alpha_1)\alpha_2 C_2}{\alpha}, && (1.22) \\ \alpha &= 1 - (1 - \alpha_1)(1 - \alpha_2). && (1.23) \end{aligned}$$

One way of simplifying these computations is to keep colors *premultiplied* by their opacity [Porter and Duff, 1984]. Rewriting the example above we have

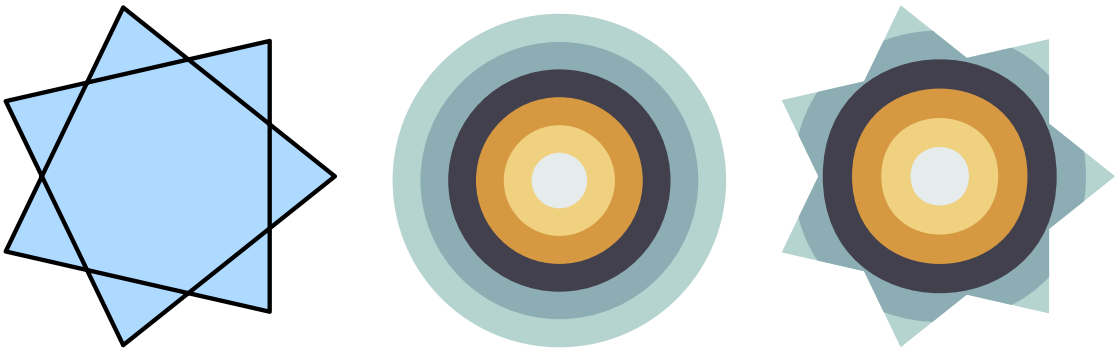
$$\begin{aligned}\bar{C}_1 &= (\alpha_1 r_1, \alpha_1 g_1, \alpha_1 b_1, \alpha_1),\\ \bar{C}_2 &= (\alpha_2 r_2, \alpha_2 g_2, \alpha_2 b_2, \alpha_2),\\ \bar{C}_b &= (\alpha_b r_b, \alpha_b g_b, \alpha_b b_b, \alpha_b).\end{aligned}$$

Then, the premultiplied color C and its opacity are given by

$$\bar{C} = \bar{C}_1 + (1-\alpha_1)\bar{C}_2, \tag{1.24}$$

$$\begin{aligned}\alpha &= 1-(1-\alpha_1)(1-\alpha_2)\\ &= \alpha_1 + (1-\alpha_1)\alpha_2.\end{aligned} \tag{1.25}$$

1.1.6 Clip Paths



(a) Clip-path. *(b) Shape to be clipped.* *(c) The result of the clipping.*

Figure 1.22: The action of the clip-path on a shape.

A *clip-path* is a closed path used as mask to other paths. The effect of a clip-path on a shape is to conceal all points of the shape that are outside the clip-path. Figure 1.22 shows an example of a clip-path applied to a shaped. Clip-paths may be grouped (union of paths) or nested (intersection of paths) to create complex regions. Clip-paths may affect many shapes at once.

1.1.7 Evaluation

We have thus far defined the structure of vector graphics and how its many components interact. This structure allows us to associate to each vector graphics

scene a function f on the plane. This function maps each point in the plane to its color. For scenes without clip-paths the algorithm to evaluate f is straightforward and is shown by figure 1.23. When clip-paths are present, the decision of whether to blend a sample's color is based on a Boolean expression involving the results of the inside-outside tests for the path and all active clip-paths. Since this expression can be arbitrarily nested, its evaluation seems to require a stack or some kind of recursion. A simplified form of evaluation, without any stack or recursion, is presented in section 4.3.

To display and manipulate these images in a computer, we need a discrete version of them. A method to discretize images in a reasonable way is the subject of the next section.

```
-- receives a scene and a point
-- returns the point's color
function evaluate( scene, p )
    color = background_color -- set the color as the background color
    -- iterate over all shapes, from the back of the scene
    -- to the front.
    for shape in iterate_back_to_front( scene )
        -- test if the point is inside the shape using the
        -- 'is_inside' function defined before.
        if is_inside( p, shape ) then
            -- compute the premultiplied color of the shape.
            shape_color = pre_multiply_by_alpha( color(shape) )
            -- use the 'over' operator to find the new color.
            color = over_compositing( shape_color, color )
        end
    do
    -- this color has a premultiplied alpha,
    -- we need to divide its components to find the
    -- final color.
    color = divide_by_alpha( color )
    return color
end
```

Figure 1.23: Basic evaluation of a sample's color.

1.2 Image Sampling

We define a continuous image as a function $f\colon \Omega \subset \mathbb{R}^2 \to \mathscr{C}$ that maps each point in the subset $\Omega \subset \mathbb{R}^2$ into a *color* in $\mathscr{C}$. The set $f(\mathscr{C})$ is the *gamut* of the image.

A *digital image* or *discrete image* is a discrete set of colors indexed by two indices corresponding to the pixel location. For instance, a discrete image of size w by h is the set

$$\left\{c_{ij} = (r_{ij}, g_{ij}, b_{ij}, \alpha_{ij}) \in [0,1]^4; i \in \{0, \ldots, h-1\}, j \in \{0, \ldots, w-1\}\right\}$$

and is usually represented as vector of tuples with 4 components and length $h \cdot w$. The process of creating a discrete image from an image is called *sampling*. On the other hand, to produce an image from a discrete image is called *reconstruction*.

1.2.1 Sampling and Reconstruction

To sample an image $f\colon \Omega \subset \mathbb{R}^2 \to \mathscr{C}$, we start by choosing a rectangular domain $V \subset \Omega$ that is aligned with the coordinate axis. V is called the *viewport* and can be fully characterized by two points: $p_{min} = (x_{min}, y_{min})$ and $p_{max} = (x_{max}, y_{max})$. To produce a discrete image of size $w \times h$, we sample f regularly at the points defined by:

$$\begin{aligned}
\Delta_x &= \frac{x_{max} - x_{min}}{w}, \\
\Delta_y &= \frac{y_{max} - y_{min}}{h}, \\
x_j &= x_{min} + \Delta_x(0.5 + j), \\
y_i &= y_{min} + \Delta_y(0.5 + i), \\
c_{ij} &= f(x_j, y_i),\ i = 0, \ldots, h-1 \text{ and } j = 0, \ldots, w-1.
\end{aligned} \tag{1.26}$$

This is shown in figure 1.24.

The reconstruction of an image from a set of discrete samples starts by the choice of a *reconstruction kernel*. A reconstruction kernel is a function $\varphi\colon \mathbb{R}^2 \to \mathbb{R}$, such that the family of functions $\{\varphi_{ij}(x,y) = \varphi(x-j, y-i); i, j \in \mathbb{Z}\}$ is linearly independent. A few examples of kernels are shown in Figure 1.25.

Now, given a chosen kernel $\varphi(x,y)$, we define

$$\varphi_{\Delta_x,\Delta_y}(x,y) = \varphi\left(\frac{x}{\Delta_x}, \frac{y}{\Delta_y}\right). \tag{1.27}$$

The reconstructed image is given by

$$\tilde{f}(x,y) = \sum_{i=0}^{h-1}\sum_{j=0}^{w-1} c_{ij}\varphi_{\Delta_x,\Delta_y}(x - x_j, y - y_i). \tag{1.28}$$

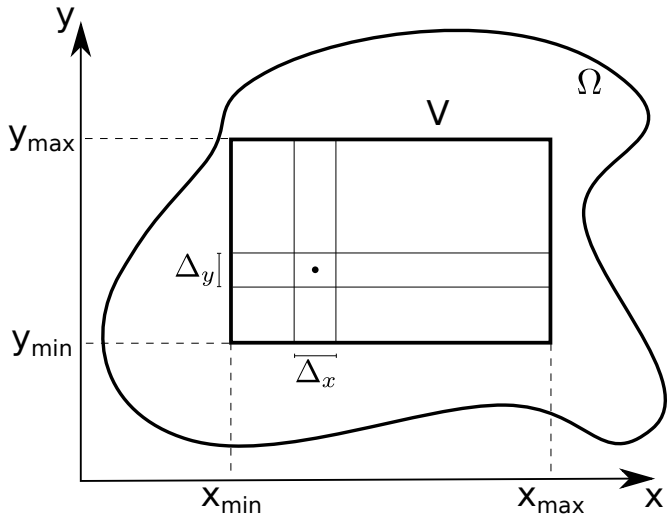
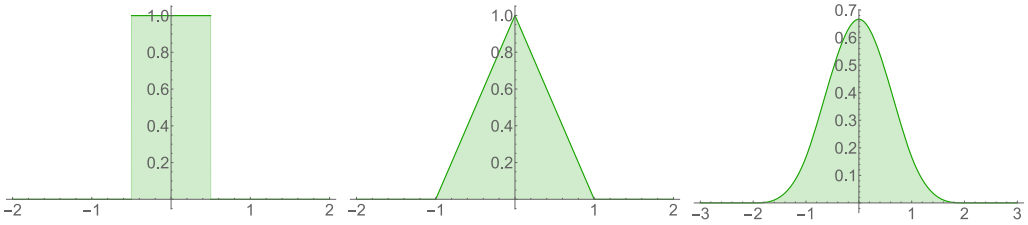


Figure 1.24: Sampling domain.



(a) Box. *(b) Hat.* *(c) Cubic B-Spline.*

Figure 1.25: Examples of reconstruction kernels.

The first question to arise is: how close are f and $\tilde{f}$? For a certain class of functions and for a specially chosen kernel, the answer is given by the Shannon-Whittaker theorem.

Theorem 2. *(Shannon–Whittaker) Let $f(t)$ be a signal (image) and $F(w)$ its Fourier transform. If F has compact support, that is, $supp(F)$ is bounded, then, there is a real number w_0 such that $w_0 = \inf\{w \in \mathbb{R}; supp(F) \subset [-w, w]\}$ and the following relation holds:*

$$f(t) = \sum_{k \in \mathbb{Z}} f\left(\frac{k}{2w_0}\right) \operatorname{sinc}(2w_0 t - k),$$

where

$$\operatorname{sinc}(t) = \frac{\sin(\pi t)}{\pi t}.$$

In other words, *band-limited* signals can be reconstructed exactly from a suitable discrete set of uniformly spaced samples, using sinc as the reconstruction kernel.

We cannot use this theorem directly when dealing with vector graphics for several reasons. First, most images do not have limited band. Second, we want to create discrete images of arbitrary sample rates, not only the ones allowed by the theorem. Finally, when displaying an image on a computer display, we do not control the reconstruction process, which is performed by the display and constrained by its pixel geometry.

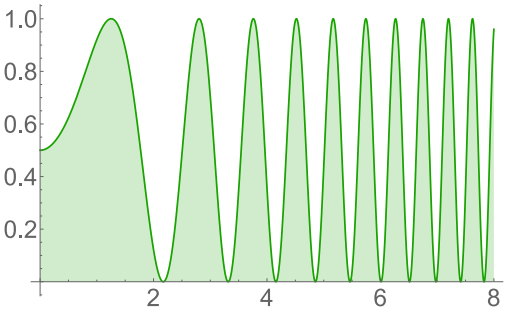
Figure 1.26 shows the result of sampling images with high frequency content at low sampling rates. The image chosen is given by the function

$$f(x, y) = (c(x, y), c(x, y), c(x, y), 1), \tag{1.29}$$

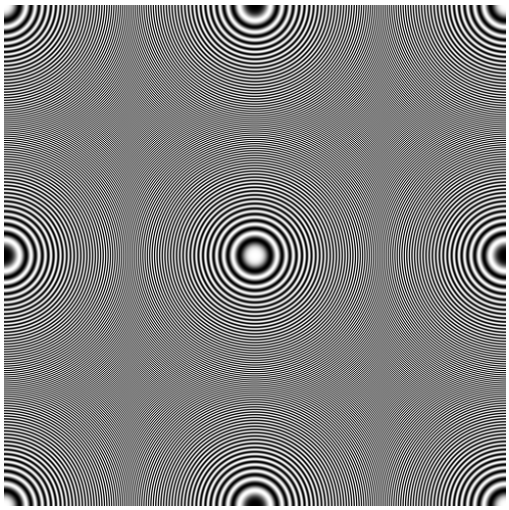
where c is given by

$$c(x, y) = \frac{sin(x^2 + y^2) + 1}{2}. \tag{1.30}$$

Figure 1.26a shows the restriction of f to the x axis in the interval $[0, 2\pi]$. We would expect a discrete image of f to show the same behavior on the plane: a sinusoidal wave of increasing frequency irradiating from the lower-left corner of the image. Figure 1.26b shows the result of sampling the set $[0, 2\pi] \times [0, 2\pi]$ with 512×512 samples.



(a) The restriction of f to the x axis.



(b) 512×512 samples.

Figure 1.26: Aliasing.

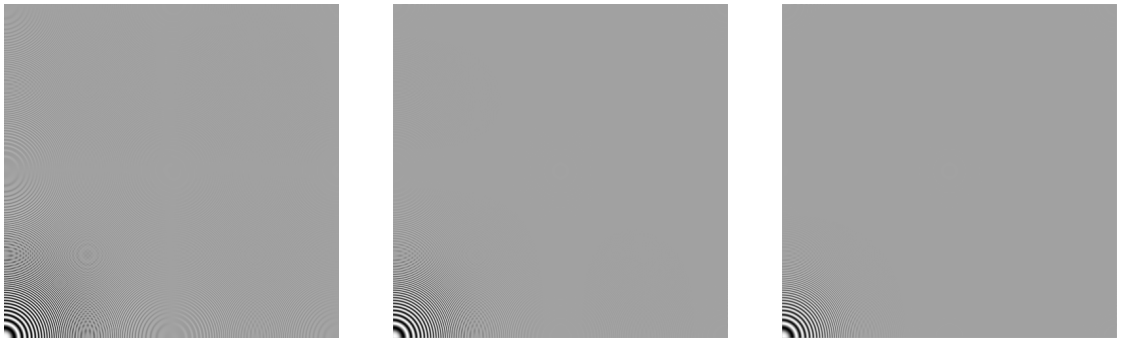
The visual artifacts created by the sampling and reconstruction processes are called *aliasing*. Aliasing occurs when high-frequency content in the signal is mixed with the low-frequency content, resulting in structured interference in the signal. A compressive analysis of aliasing is found in Nehab and Hoppe [2014].

To attenuate aliasing, we need to suppress high-frequency content *before* sampling. For that, we choose an *antialiasing* filter and, instead of sampling f, we sample the convolution of f with the filter. That is, if we choose a function ψ as the antialiasing filter, we sample $f * \psi$.

The rationale behind this method is simple: the frequency content of $f * \psi$ is given by its Fourier transform

$$\mathcal{F}(f * \psi) = \mathcal{F}(f) \cdot \mathcal{F}(\psi).$$

Therefore, choosing a band-limited φ will make $f * \varphi$ band-limited. It is difficult, if not impossible, to use a band-limited filter. Functions with compact support in the frequency domain (Fourier domain) have non-compact support in the spatial domain, making the computation of the sample values impractical. Therefore, we use antialiasing kernels that attenuate high frequencies. Searching for good antialiasing filter is an active research area. In figure 1.27, we show the result of using the Box, Hat and cubic B-spline kernels as anti-aliasing filters applied to the function f above at the same sampling rate. Figure 1.28 shows the magnitude of



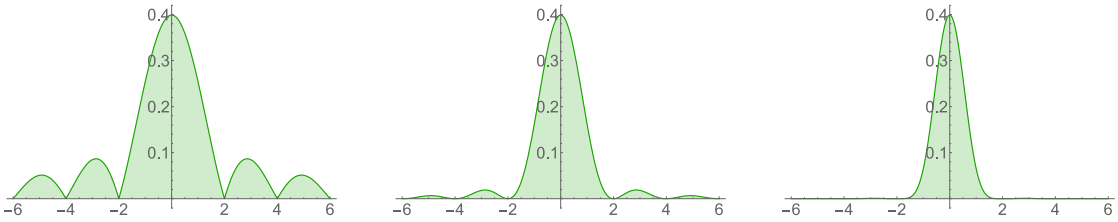
(a) Box. *(b) Hat.* *(c) BSpline3.*

Figure 1.27: Anti-aliasing filters applied to f.

the Fourier transform of those same kernels.

Going back to our first example (equations 1.26) of image sampling, we can redefine our method in the following way. Given an image f and a anti-aliasing filter ψ, we define

$$\psi_{\Delta_x,\Delta_y}(x, y) = \psi\left(\frac{x}{\Delta_x}, \frac{y}{\Delta_y}\right), \tag{1.31}$$



(a) $|\mathcal{F}(Box)|$. (b) $|\mathcal{F}(Hat)|$. (c) $|\mathcal{F}(Cubic\ B\text{-}Spline)|$.

Figure 1.28: Anti-aliasing kernels frequency response.

where

$$\begin{aligned} \Delta_x &= \frac{x_{max} - x_{min}}{w}, \\ \Delta_y &= \frac{y_{max} - y_{min}}{h}, \\ x_j &= x_{min} + \Delta_x(0.5 + j), \\ y_i &= y_{min} + \Delta_y(0.5 + i). \end{aligned} \tag{1.32}$$

And the samples values are computed by

$$\begin{aligned} c_{ij} &= (f * \psi_{\Delta_x,\Delta_y})(x_j, y_i), \\ i &= 0, \ldots, h-1, \\ j &= 0, \ldots, w-1, \end{aligned} \tag{1.33}$$

The reason why we choose to define the reconstruction and anti-aliasing kernel using

$$\varphi_{\Delta_x,\Delta_y}(x, y) = \varphi\left(\frac{x}{\Delta_x}, \frac{y}{\Delta_y}\right), \tag{1.34}$$

$$\psi_{\Delta_x,\Delta_y}(x, y) = \psi\left(\frac{x}{\Delta_x}, \frac{y}{\Delta_y}\right), \tag{1.35}$$

is that we want to draw these kernels out of a uniform (or canonical) space of functions, where kernels are independent of scale and image size (sampling rate).

1.2.2 Computing Samples

The question that remains for computing the samples c_{ij} is how to evaluate the expression

$$\begin{aligned} c_{ij} &= (f * \psi_{\Delta_x,\Delta_y})(x_j, y_i) \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y) \cdot \psi_{\Delta_x,\Delta_y}(x_j - x, y_i - x)\, dx\, dy \end{aligned} \tag{1.36}$$

which is a double integral.

We can not solve it analytically, since f can be arbitrarily complex and most probably lacks a closed form primitive (anti derivative). Since the only alternative is to use numerical methods, we chose to use Monte-Carlo estimators.

A complete treatment of Monte-Carlo integration applied to image synthesis can be found in Glassner [2014]. We can summarize the method in the following way: given an integrable function $g\colon [a,b] \times [c,d] \to \mathbb{R}$, we define the sequence

$$g_n = \frac{(b-a)(d-c)}{n} \sum_{i=0}^{n-1} g(Z_i), \tag{1.37}$$

where Z_i is a uniform random variable in the domain $[a,b] \times [c,d]$. As n increases, the expected value of the estimator converges to the value of the integral

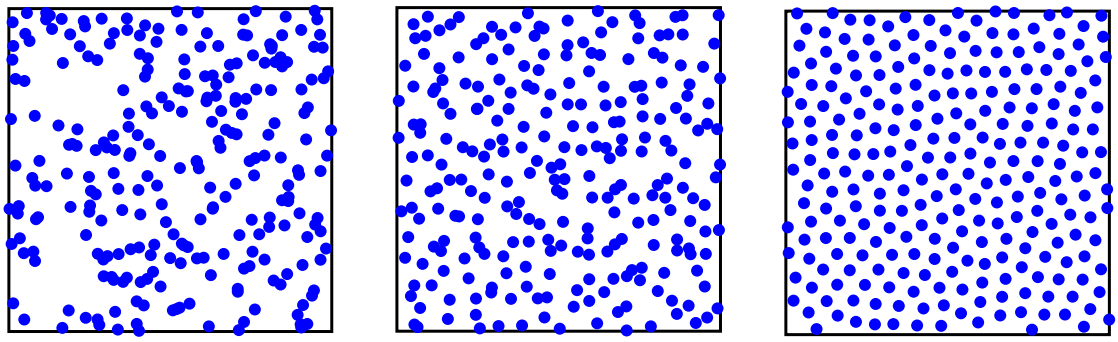
$$\lim_{n\to\infty} g_n = \iint_{[a,b]\times[c,d]} g(x,y)\, dx\, dy. \tag{1.38}$$

Since in practice we can not draw an infinite number of samples, we need to be concerned with the variance of the estimator g_n. The variance of g_n is given by:

$$\begin{aligned} \mathrm{Var}[g_n] &= \mathrm{Var}\left[\frac{(b-a)(d-c)}{n} \sum_{i=0}^{n-1} g(Z_i)\right] \\ &= \frac{(b-a)(d-c)}{n^2} \sum_{i=0}^{n-1} \mathrm{Var}[g(Z_i)] \\ &= \frac{(b-a)(d-c)}{n} \mathrm{Var}[g(Z)]. \end{aligned} \tag{1.39}$$

Since the variance of $g(Z)$ is constant, the variance of the estimator decreases with the number of samples.

The method calls for uniformly distributed samples in the plane. There are many ways of generating these samples [Dippé and Wold, 1985]. The first and most obvious way is to simply generate random numbers for the samples coordinates (figure 1.29a). This process tends to generates clusters of points, and for low number of samples this may create unwanted visual artifacts. We say that this point distribution has high *discrepancy*. A way of reducing discrepancy is to start the process with samples placed in a regular grid on the area, and then applying a small random displacement to each sample. This method is called *stratified jittering*, and it is shown in Figure 1.29b. A third class of patterns is called *low-discrepancy*. Low-discrepancy patterns distribute the samples in the area and apply a global optimization that displaces the samples in order to eliminate clusters, but keeping the randomness of the samples. One particular instance of this class is called *blue-noise*, shown in Figure 1.29c.



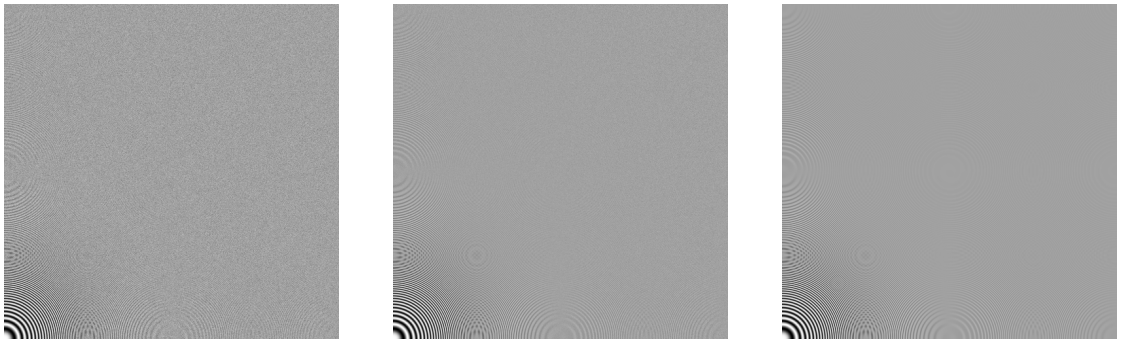
(a) Uniform. *(b) Stratified.* *(c) Blue noise.*

Figure 1.29: Sample pattern in the plane.

In figure 1.30, we show the results of using different sample patterns when evaluating the estimator 1.37 on an image f as defined in equation 1.29. We use 64 samples per pixel using Box as the antialiasing filter. While the structured artifacts (aliasing) in the image are almost the same, the amount of noise decreases as we go from uniform to stratified and then to blue noise.

The expression for the estimator g_n for equation 1.36 is somewhat convoluted. We can simplify it by rewriting our problem using a different frame of reference. We have the viewport in which we would like to discretize our image. And we have the size of the final discrete image $w \times h$. We define the following affine transformation

$$T = \begin{bmatrix} \frac{1}{\Delta_x} & 0 & 0 \\ 0 & \frac{1}{\Delta_y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & -x_{min} \\ 0 & 0 & -y_{min} \\ 0 & 0 & 1 \end{bmatrix}. \tag{1.40}$$



(a) Uniform. *(b) Stratified.* *(c) Blue noise.*

Figure 1.30: Result of applying different sample pattern.

The effect of T is to translate the viewport by $-(x_{min}, y_{min})$ and then scale the x and y axes so that the spaces between two consecutive samples (vertically or horizontally) is 1. This is shown if figure 1.31. This is, not surprisingly, analogous to the viewport transformation of section 1.1.2.

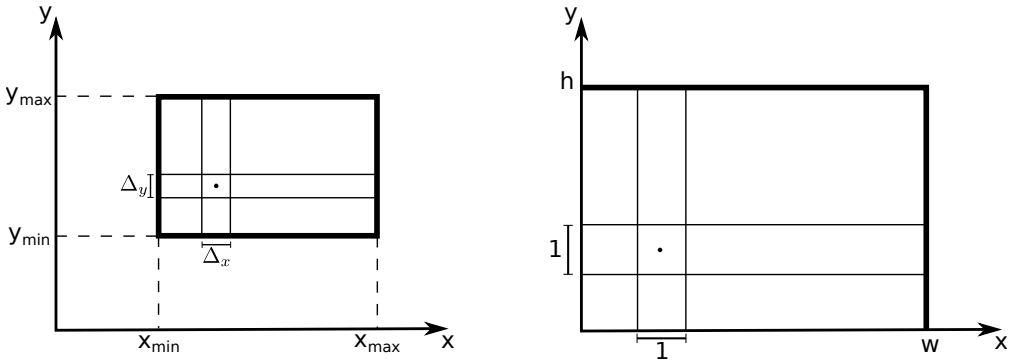


Figure 1.31: Action of T

Now, we define the image $\tilde{f}(x,y) = f \circ T^{-1}(x,y)$ and recast our problem as

$$\begin{aligned} c_{ij} &= (\tilde{f} * \varphi)(x_j, y_i), \\ x_j &= 0.5 + j, \\ y_i &= 0.5 + i, \\ i &= 0, \ldots, h-1, \\ j &= 0, \ldots, w-1. \end{aligned} \tag{1.41}$$

This simplifies the estimator g_n to

$$g_n = \frac{4\delta^2}{n} \sum_{k=0}^{n-1} \tilde{f}(x_j + X_k, y_i + Y_k)\varphi(-X_k, -Y_k), \tag{1.42}$$

with $(\bar{X}_k, \bar{Y}_k)$ uniformly distributed over $[-\delta, \delta] \times [-\delta, \delta]$. Where $[-\delta, \delta]$ is the support of φ. The simplified version of the algorithm is shown in figure 1.32.

```
function evaluate_simplified_monte_carlo( scene, p, kernel )
    -- create a sampling pattern on the support of
    -- the kernel
    local pattern = load_pattern( support(kernel) )
    -- new "empty" color
    local color = new_color(0,0,0,0)
    for s in iterate_over_samples( pattern ) do
        -- point in the scene for this sample
        local q = point(p.x + s.x, p.y + s.y)
        -- use our previous define evaluate function
        local c = evaluate( scene, q )
        -- find the kernel value for this sample
        local w = kernel( -s )
        color = color + w*c
    end
    -- number of samples in the pattern
    local n = size( pattern )
    -- return the Monte-Carlo estimator
    -- with area(kernel) = δ_x δ_y
    return (area(kernel)/n)*color
end
```

Figure 1.32: Simplified Monte-Carlo evaluation.

1.3 Rendering Vector Graphics

In the previous sections we have detailed the two main steps for creating an image for a vector graphics scene. First, we have defined the structure of scenes and created a method to evaluate our scene—viewed as a function on the plane—and called it 'evaluate' (Figure 1.23). Then, we investigated how to rasterize images and create discrete representations of the original input. Those are the tools we need to create a complete algorithm to rasterize vector graphics scenes. This is shown in Figure 1.33.

```
-- receives a scene
-- returns an discrete image
function rasterize( scene, kernel, viewport, width, height )
    -- preprocess the scene: compute f̃ = f ∘ T⁻¹
    scene = prepare_scene( scene, viewport, width, height )
    -- create an empty image
    image = create_image(width, height)
    -- iterate over all pixels
    for i=0, height-1 do
        y_i = 0.5 + i
        for j=0, width-1 do
            x_j = 0.5 + j
            p = point(x_j, y_i)
            -- evaluate color using simplified version of
            -- Monte-Carlo sampling
            color = evaluate_simplified_monte_carlo( scene, p, kernel )
            -- store the color in the image
            image.set( i, j, color )
        end
    end
    return image
end
```

Figure 1.33: Complete algorithm.

The function 'prepare_scene' receives the scene to be rasterized and the data necessary to compute the transformation T^{-1}(equation 1.40). It iterates over all shapes in the scene transforming the Bézier segments of all paths along with all gradients and textures. The resulting scene is the equivalent of the image $\tilde{f}$ defined before. The scene is sampled at the points $(j + 0.5, i + 0.5)$ using the simplified version of the Monte-Carlo method, and the colors of the output image are stored on the 'image' object, before being returned by the function.

1.3.1 Algorithm Analysis

In figure 1.34 we show the rendering time for algorithm shown in figure 1.33 when running on benchmark scenes. Each scene contains from 1 to 256 polygons. Each polygon has 4 sides and is created randomly. The graphs confirms the expected behavior: the algorithm is linear in the number of segments in the scene, since it tests all segments in the scene. Each intersection test is done in constant time. We can change this behavior using acceleration data structures or other strategies. We discuss these options in chapter 2.

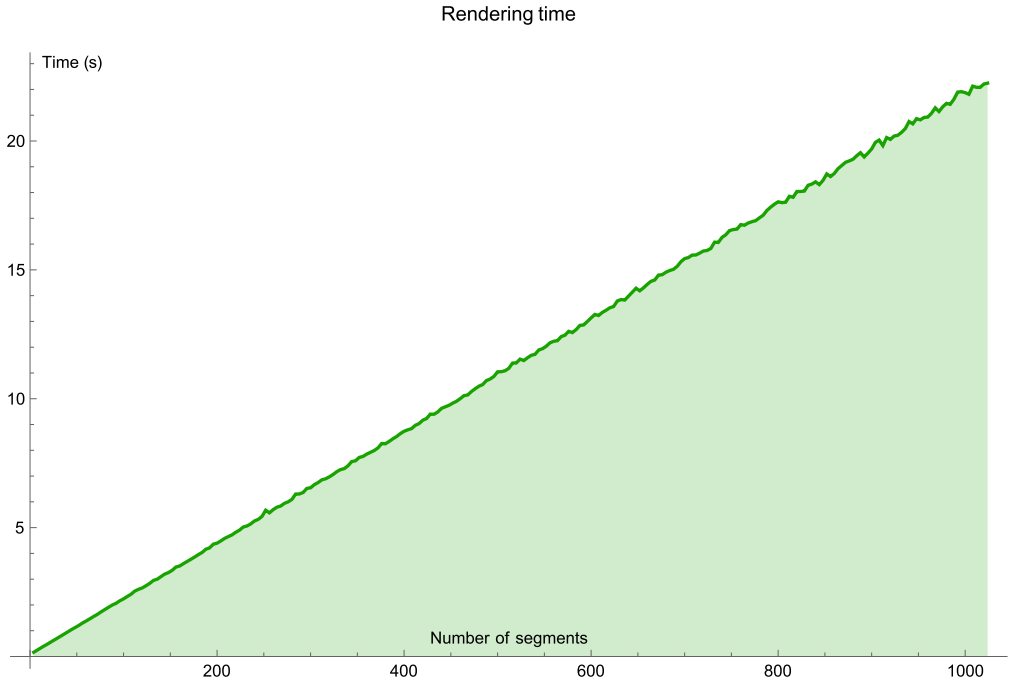


Figure 1.34: Rendering

1.4 Basic Optimization

There are some optimizations we can implement in a basic renderer. These optimizations do not change the complexity of the algorithm, but are important in common situations: illustrations with large areas covered by gradient paints or illustrations with many overlapping polygons.

1.4.1 Gradients

Gradients carry along with their defining parameters an affine transformation T (Section 1.1.4). This transformation maps points from the gradient's reference frame $\mathcal{G}$ into points in the scene's reference frame $\mathcal{S}$:

$$\mathcal{G} \xrightarrow{T} \mathcal{S}.$$

The gradient's reference frame defines the space where the gradient's parameters are defined, such as the initial and ending point for linear gradients and the center, focus and radius for radial gradients. Therefore, before evaluating the gradient on a sample p of the scene, we should apply the inverse of the transformation: $\bar{p} = T^{-1}(p)$.

This seems the opposite to what we would expect: why not specify T^{-1} instead of T? This is done so we can transform gradients (and textures) along with paths or even the whole scene. Suppose we have an affine transformation A that modifies the scene, then the new gradient transformation will be given by $A \circ T$.

In order to simplify the gradient evaluation, we define a new frame of reference and call it the *normalized gradient space* $\mathcal{N}$. We also define a transformation O that maps the gradient space into the new space $\mathcal{N}$. Once we have O computed, the new gradient transformation will follow the diagram

$$\mathcal{N} \xleftarrow{O} \mathcal{G} \xrightarrow{T} \mathcal{S},$$

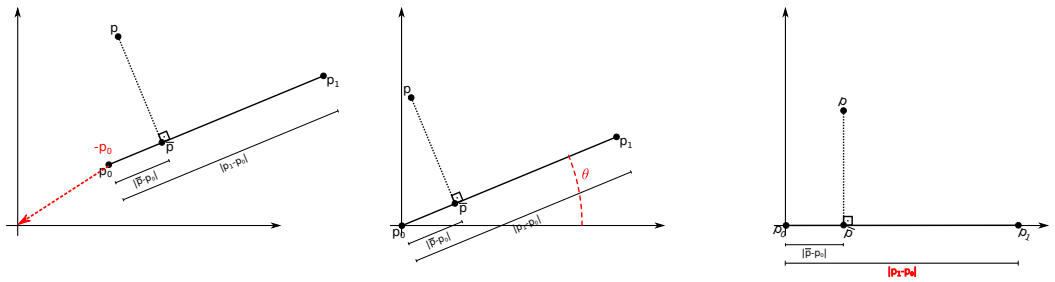
and will be given by $O^{-1} \circ T \colon \mathcal{N} \to \mathcal{S}$.

To compute O for linear gradients we follow the sequence of transformations shown in Figure 1.35. First we apply a translation t_{-p_0} to position p_0 on the origin. Then we rotate, using r_θ, the segment p_0p_1 so it lays on the positive side of the x-axis. And finally we scale by $\lambda = \frac{1}{|p_1 - p_0|}$, using s_λ, to make the point p_1 sit on the coordinates $(0, 1)$. This sequence of operations is summarized by

$$O = s_\lambda \circ r_\theta \circ t_{-p0}.$$

Now, to find the value of d (as defined for linear gradients in section 1.1.4), we simply take the x coordinate of $O \circ T^{-1}(p)$. All of these operations can be computed once for each linear gradient and $O \circ T^{-1}$ can be stored to be used for sampling. Since $O \circ T^{-1}$ is an affine transformation, it is represented as a 3×3 matrix, and since we are only interested in the x coordinate of transformed points, we can store only its first row, i.e., only 3 floating point numbers.

A similar process is employed to simplify radial gradients. First we apply a scale, both in x and y, to make circle unitary. Then, a translation is used to make f the



(a) Translation. *(b) Rotation.* *(c) Scale.*

Figure 1.35: Simplification of linear gradients.

origin. Followed by a rotation to place the center c on the x-axis. If the focus f and the center c coincide, then the value of d is the norm of p. If not, the computations are made as before, but the equations are greatly simplified.

Color ramps can be precomputed to generate textures that approximate the visual appearance of the ramp. Given a color ramp, we choose a number of samples n and sample the ramp at the points

$$x_i = \frac{0.5 + i}{n}, i = 0, \ldots, n - 1. \tag{1.43}$$

After computing d and applying the extension functions—clamp, repeat, or mirror—we find the color by sampling the linearly interpolated texture (i.e. reconstructed with the Hat kernel). This method is especially useful when dealing with GPUs that have dedicated hardware to sample textures.

1.4.2 Front-to-Back Sampling

One minor optimization in sampling comes from changing the order in which the scene is iterated. By definition (see section 1.1.5) the color of a sample is computed by

$$(C_1 \rhd (C_2 \rhd (\ldots (C_n \rhd C_b)))).$$

where $C_1, \ldots, C_n$ are the colors of the shapes containing the point being sampled, and C_b is the background color.

Using equation 1.22 and premultiplied alpha values, we are able to switch this computation order to

$$((((C_1 \rhd C_2) \rhd \ldots) C_n \rhd C_b)).$$

This means that we can traverse the scene, from front to back, computing the color of the sample. Once the composed color is opaque enough, the whole computation can be stopped. This is justified by the fact that

$$C_a \triangleright C_b = C_a,$$

for any opaque color C_a and any color C_b. The algorithm for color sampling shown in Figure 1.23 can be rewritten as shown in Figure 1.36.

```
-- receives a scene and a point
-- returns the point's color
function front_to_back_evaluate( scene, p )
    color = blank_color() -- set the color as blank
    -- iterate over all shapes, from the front of the scene
    -- to the back.
    for shape in iterate_front_to_back( scene )
        -- test if the point is inside the shape using the
        -- 'is_inside' function defined before.
        if is_inside( p, shape ) then
            -- compute the premultiplied color of the shape.
            shape_color = pre_multiply_by_alpha( color(shape) )
            -- use the 'over' operator to find the new color.
            color = over_compositing( shape_color, color )
            if color.a >= 1 -0.5/255 then
                -- alpha is 1, so no division is needed
                return color
            end
        end
    do
    -- at this point we've reached the background
    color = over_compositing( color, background_color() )
    -- we need to divide its components to find the
    -- final color (background may be transparent!).
    color = divide_by_alpha( color )
    return color
end
```

Figure 1.36: Front to back sampling.

Chapter 2

Previous Work

In chapter 1 we have shown the structure of vector graphics and a basic rendering method. We have also shown that rendering vector graphics with high quality is a computationally expensive task. While the method we presented before is sequential, much effort has been put into the parallelization of rendering methods. CPU implementations use vector instructions (SIMD) to efficiently collect samples, but are limited by the small number of cores available. On the other hand, GPUs have a massive number of processing units available, but the architecture of GPUs demands an adaptation of the rendering methods.

Real time implementations (both in CPU and GPU) of vector graphics renderers fall into one of two categories: immediate mode and retained mode. In immediate mode the scene is rendered one shape at a time. Each shape is broken into fragments and the fragment's colors are blended into a frame buffer. In retained mode, the scene is preprocessed to create an acceleration data structure, which is later used to render the output image. Methods in this category are also called vector textures.

2.1 Immediate Mode

Immediate-mode methods generally exploit spatial coherence in the scene by subdividing complex geometry into simple primitives that are easier to fill. Scanline-based algorithm follow the work of Wylie et al. [1967] and break shapes into horizontal spans that cover interior pixels [Foley, 1996]. The Direct2D [Kerr, 2009], Cairo [Packard and Worth, 2003], and Skia renderers decompose shapes into trapezoids. Loop and Blinn [2005] use a constrained triangulation, while Kilgard and Bolz [2012] use a triangle fan and the stencil buffer—we will discuss these

two works in more detail later. Each primitive is broken into fragments which are painted and blended over a frame buffer. The cost of the rendering is amortized within paths, since fragments of the primitives can be computed in parallel. We illustrate these methods in figure 2.1.

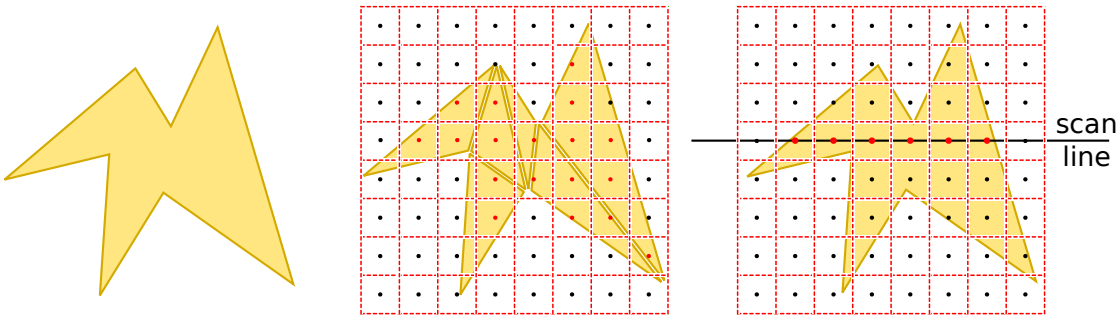
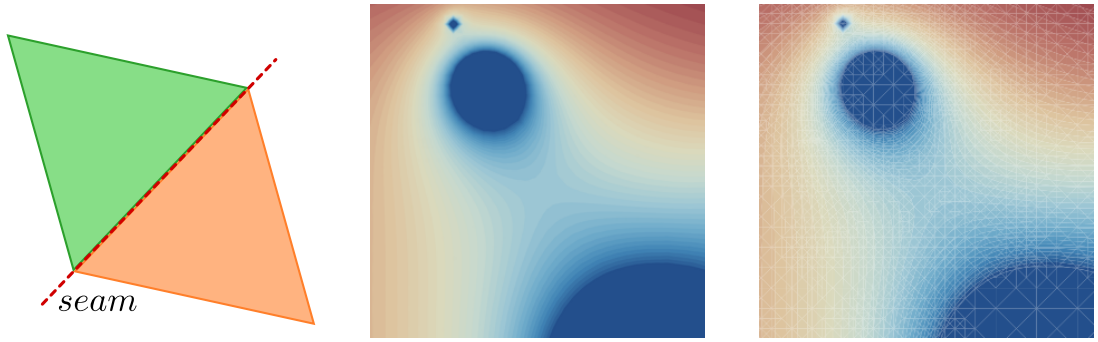


Figure 2.1: To rasterize a path (left), it can be triangulated (middle) or broken into line spans (right).

Most of these renderers use box filtering for antialiasing. Therefore, the color of a pixel is a numerical approximation of the average color of the primitive over the pixel area. As we have seen, the correct way of computing the pixel's color is to collect many samples taking in account all the shapes that overlap with the pixel's area. Instead, these methods implement an optimization where the antialiasing is computed for each shape independently. Pixel coverage is converted into semi-transparent colors prior to blending into the frame buffer. When more than one shape partially covers a pixel, the resulting blended color is not opaque, even when the union of all shapes entirely cover the pixel area [Porter and Duff, 1984]. This *conflation* of color is shown in figure 2.2. To eliminate this problem, it



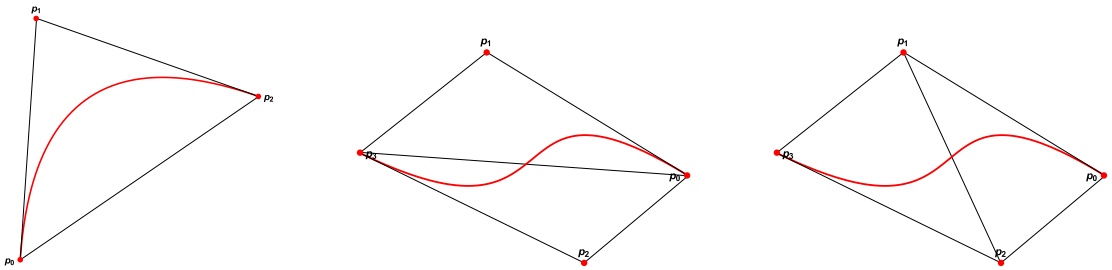
(a) Shared edge. *(b) Rendered correctly.* *(c) Color conflation.*

Figure 2.2: Color conflation.

is necessary to allocate memory for many samples per pixel. Each sample will have its color blended with the color in the frame buffer independently. The resulting pixels are computed by averaging the samples final colors (box filtering). This is the approach implemented by Kilgard and Bolz [2012].

2.1.1 Loop and Blinn [2005]

Loop and Blinn [2005] describe a way of displaying filled paths using the GPU hardware. The control polygon of each path are triangulated. Completely filled triangles are simply painted by a trivial shader. Triangles whose interior are only partially filled are treated by a special shader. This special shader uses an implicit version of the input Bézier segments to distinguish fragments that are inside from fragments that are outside the shape. Anti-aliasing is performed on boundary pixels using a distance function derived from the implicit representation's derivative.

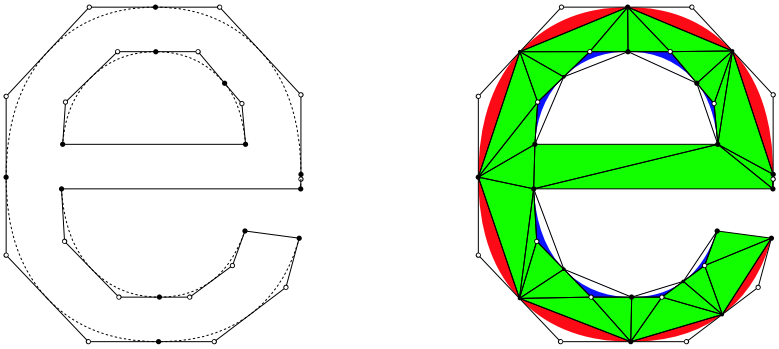


(a) Quadratic. *(b) Cubic.* *(c) Cubic.*

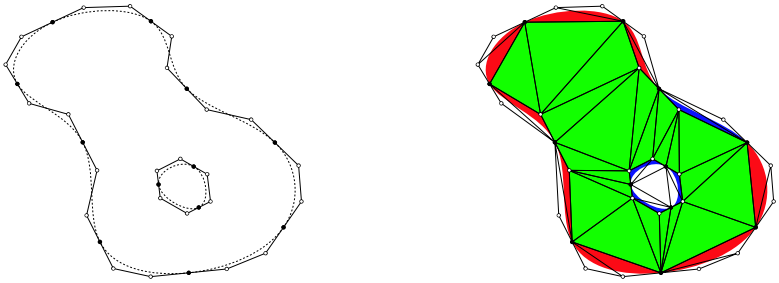
Figure 2.3: Quadratic segments have a natural triangulation given by the control points. Cubic segments may be triangulated in more the one way.

Rendering filled shapes using their work requires a preprocessing phase where paths are prepared for rendering. The preprocess stage triangulates the control points of the path to create a mesh that contains the shape to be filled. Control points of a quadratic segment naturally give a triangle containing the segment, while for cubic segments it is necessary to compute a local triangulation of the four control points, see figure 2.3. After triangulating all Bézier segments these triangles are constrained and a global Delaunay triangulation is computed. Figure 2.4 shows two input paths, followed by their triangulations.

The triangulation yields two types of triangles: completely covered triangles and partially covered triangles. The completely covered triangles are inside the path and are ready to be processed by the rendering stage. Partially covered triangles contain quadratic or cubic Bézier segments and require the definition of an *implicit*



(a) A shape with quadratic segments.



(b) A shape with cubic segments.

Figure 2.4: Constrained triangulation.
Image from Loop and Blinn [2005].

function inside the triangle: $f\colon \mathbb{R}^2 \to \mathbb{R}$. Function f is created such that $f(p) < 0$ if p is inside the path, $f(p) > 0$ if p is outside, and $f(p) = 0$ if p lies on the Bézier segment.

The triangles are broken into fragments by the GPU hardware and processed by the fragment shader. Fragments from covered triangles are painted with the path's color. Fragments from partially covered triangles are tested by the implicit function. If the fragment is inside the path, then it is painted. Otherwise it is discarded.

Finding an implicit function for both quadratics and cubics is an extensive part of their work. They use a result by Salmon [1852] that ensures it is always possible to find affine functionals $\mathbf{k}$, $\mathbf{l}$, and $\mathbf{m}$ such that the implicit function f applied to a point (in homogeneous coordinate) $\mathbf{s} = \begin{bmatrix} s_x & s_y & 1 \end{bmatrix}$ is given by

$$\text{integral quadratic:} \quad f(\mathbf{s}) = (\mathbf{k}\,\mathbf{s})^2 - \mathbf{l}\,\mathbf{s}, \tag{2.1}$$

$$\text{rational quadratic:} \quad f(\mathbf{s}) = (\mathbf{k}\,\mathbf{s})^2 - (\mathbf{l}\,\mathbf{s})(\mathbf{m}\,\mathbf{s}), \tag{2.2}$$

$$\text{integral cubic:} \quad f(\mathbf{s}) = (\mathbf{k}\,\mathbf{s})^3 - (\mathbf{l}\,\mathbf{s})(\mathbf{m}\,\mathbf{s}). \tag{2.3}$$

At the preprocessing stage, the functionals are evaluated on the vertices of the triangles and the results are stored. To evaluate the functionals at any point inside the triangle it is sufficient to compute the barycentric interpolation of the functional values stored at the triangle's vertices. This is exactly what is done by the hardware when generating fragments at the rasterization phase of the 3D pipeline.

2.1.2 Kilgard and Bolz 2012

Kilgard and Bolz [2012] build on the ideas of Loop and Blinn [2005] to create a complete rendering pipeline integrated with OpenGL. They introduce a programming interface called "Stencil then Cover" (StC). Stencil then Cover explicitly decouples the two-stage of rendering a path: deciding which point to paint (Stencil) and then paint it (Cover).

Before StC can take place all paths have to be preprocessed (*baked*). The baking process converts each path into a resolution-independent representation from which they can be stenciled. Baking a path yields four types of geometric primitives:

1. Polygonal anchor geometry, structured as a triangle fan and rendered with no shader.
2. Quadratic and cubic discard triangles, rendered with Loop–Blinn implicit shader.
3. Arc of ellipses discard triangles.
4. Conservative covering geometry, typically a triangle fan or quadrilateral.

Primitives from 1 to 3 are rendered at the stencil fill step. Fragments from front face triangles increment the stencil buffer while fragments from back facing triangles decrement it. This process is equivalent to computing the winding number of each sample and storing it in the stencil buffer. Primitives from 4 are rendered in the cover step and the stencil buffer is used to exclude outside fragments. The fragments inside the path are then covered using the path's paint computed by a fragment shader. Figure 2.5 shows the geometric primitives above.

The quality of the rendered scene depends directly on the number of samples supported by the hardware (and selected by the user) when rendering the scene. More samples per pixel implies better images at the cost of more memory to store the stencil and color buffers. The antialiasing used is also dependent of the hardware capabilities: at this time only box filtering is natively supported. It is possible to use better antialiasing filter by rendering in multiple passes. Better quality comes at the expense of a performance penalty.

Figure 2.5: The geometry yield by the baking process. From left to right: the path to be filled, its control points, the triangle fan, the discard geometry for curved segments, the conservative covering geometry.
Image from Kilgard and Bolz [2012].

2.2 Vector Textures

In retained mode, all layers in the illustration are sampled in a single pass without conflation. A particular class of retained-mode methods are vector textures [Kilgard, 1997, Frisken et al., 2000, Sen et al., 2003, Sen, 2004, Ramanarayanan et al., 2004, Ray et al., 2005, Lefebvre and Hoppe, 2006, Qin et al., 2006, Nehab and Hoppe, 2008, Qin et al., 2008, Parilov and Zorin, 2008, Rougier, 2013]. Vector textures combine the resolution independence of vector graphics with the random-access sampling from images. This enables a range of new applications, such as direct mapping of vector graphics onto 3D surfaces and creative warping effects. Unfortunately, most of these methods restrict the complexity of the input, which is not acceptable in a general-purpose rendering scenario. More importantly, these methods build acceleration data structures during expensive preprocessing stages that are sequential in nature, and this precludes their use with dynamic content. We review the work of Nehab and Hoppe [2008], since it is the most influential method to our work.

2.2.1 Nehab and Hoppe [2008]

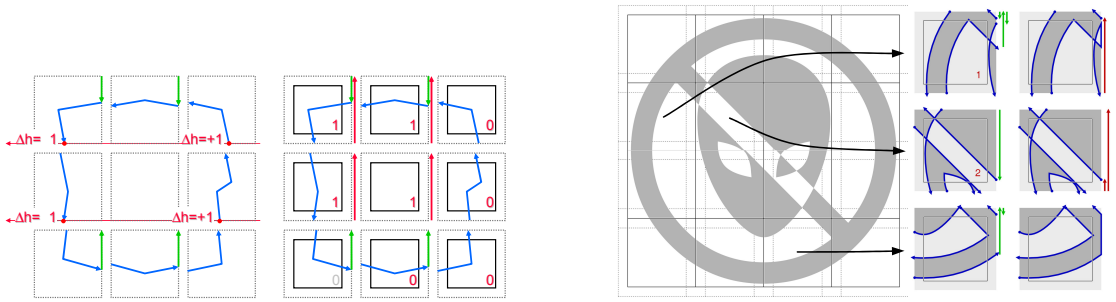
Nehab and Hoppe [2008] introduce a novel method for simplifying and encoding a vector graphics scene into a coarse grid of cells. These encoded cells are interpreted

at runtime within a pixel shader in order to evaluate the fragment's color. Anti-aliasing is done using analytical prefiltering or supersampling.

Each pixel color in the image is computed independently by a fragment shader in the GPU. The fragment shader uses the fragment coordinates in image space to find which cell in the grid contains the fragment. This is done in constant time, since both the image and the grid size are known. Then, the data corresponding the cell is loaded from a GPU buffer. This data contains a representation of the scene inside the boundary of the cell, and is coded in a special form: it is the result of the lattice-clipping algorithm. The scene data is traversed to compute the pixel color.

If only a few shapes are inside the cell, an analytic approximation to anti-aliasing is used. Prefiltering is performed independently for each shape. The filter weight is computed as a function of the distance between the pixel center and the shape. If many shapes cross the cell, those simplifications break down and supersampling is used. In this case, the colors of many samples inside the pixels area are computed and the pixel color is determined by a combination of the colors and weights given by an anti-aliasing filter.

Fast sampling is possible due to the simplicity of the scene inside each cell of the grid. To compute the content of each cell, one could employ a clipping algorithm, and clip each shape against each grid cell. This is very inefficient. Instead, Nehab and Hoppe show an elegant method to construct the grid.



(a) Winding number increments. *(b) Method applied to a scene.*

Figure 2.6: Fast lattice-clipping algorithm.
Images from Nehab and Hoppe [2008].

They begin by making the following assumption: sampling operations only use horizontal rays the right to compute intersections. The sequence of operations to compute the content of the cells is shown in figure 2.6a, and is applied to each path in the scene. Each segment in the path is clipped against the cells borders, but no new geometry is created at this stage. Then, for each segment that crosses the

right side of a cell, a shortcut segment is added. If the segment is entering the cell, a shortcut coming from the top of the cell and ending at the segment's beginning vertex is added. If the segment is leaving the cell, then the shortcut begins at the ending vertex and ends at the top of the cell. By doing so, the winding numbers of the points inside the cell may be incorrect, but the difference between the new winding number and the original winding number is constant through the cell. The difference between the new and old winding numbers can be computed by finding the winding number of the lower right corner of the cell and storing it as the *initial winding number* of the path inside the cell. This procedure will be explained in depth in chapter 4, along with a proof of its correctness. Figure 2.6b illustrates the method. Sampling with this new representation is straightforward. Besides the normal intersection between the ray and segments (shortcuts are just regular segments), one just adds the initial winding number before applying the inside/outside rule of choice.

Chapter 3

Optimizations

We start this chapter introducing a few optimizations to the algorithm given in chapter 1. These optimizations reorganize the process of sampling and integration of colors. When using antialiasing kernels with large support, the computation of neighbor pixels will collect samples in a shared area. We take advantage of the overlapping between the support of antialiasing kernels to restructure the rendering loop: each sample is evaluated once and their colors are shared among many pixels. To speed up sampling, we use an acceleration data structure that exploits geometric coherence to split and simplify the scene. The ideas presented in this chapter will be translated to the GPU in chapter 4.

3.1 Rendering Loop

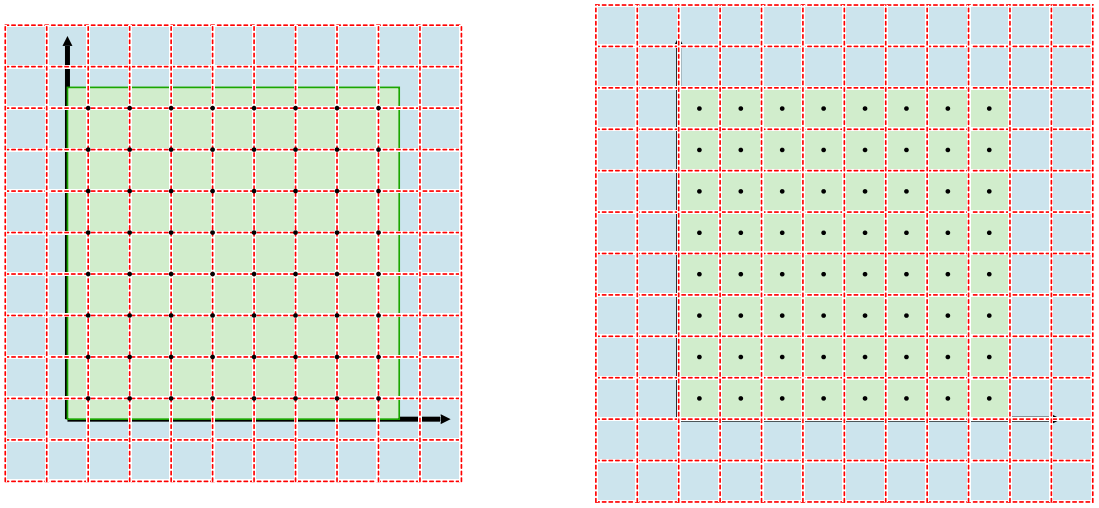
In this section, we show that, by rearranging the rasterization loop, we can exploit the overlap between the support of anti-aliasing kernels to speed up the integration of samples. We focus our attention on symmetric piecewise polynomial kernels. Elements of this family are simple to evaluate and give an uniform structure to the problem of integrating the color of samples.

3.1.1 Kernels

If ψ is symmetric, that is $\psi(-x) = \psi(x)$, for all $x \in \mathbb{R}$, so is Ψ. The symmetry of ψ makes its Fourier transform a real valued function ($\mathcal{F}[\psi](w) \in \mathbb{R}$, for all $w \in \mathbb{R}$), which means that using ψ as a kernel does not change the signal's *phase*.

A piecewise polynomial function is defined by multiple polynomials, each one applied to an interval of the function's domain. These polynomials are carefully chosen to ensure desirable properties, such as continuity and differentiability. One additional constraint that we impose is that each polynomial part should be defined over an interval whose length is an integer number.

These properties induce a grid structure in the plane. We fix the position of the samples as defined in the set of equations 1.41. These positions are called *pixel centers*, as they are the position of the samples we take from the convolution of the image with the anti-aliasing kernel. Kernels with even support induce a primal grid, while kernels with odd support sizes induce a dual grid. Examples of both grids are shown in Figure 3.1. The areas delimited by the dashed lines are the areas of the plane in which the anti-aliasing kernel is defined by a single polynomial function. We call these areas *unit areas* of the image.



(a) Primal grid for a kernel with support width 4.

(b) Dual grid for a kernel with support width 5.

Figure 3.1: Examples of grids.

3.1.2 Computation

Suppose we want to compute an image with $w \times h$ pixels, with n samples per pixel, using an anti-aliasing filter $\Psi(x, y) = \psi(x) \cdot \psi(y)$ in which $supp(\psi) = k$. In algorithm 1.33, the number of evaluations of the scene will be given by $w \cdot h \cdot n$.

We can rearrange the computation, iterating over the unit areas. The samples inside each unit area will affect k^2 pixels, and can be computed only once. Suppose we have chosen n, such that $n = sk^2$. That is, the n samples can be evenly distributed over the k^2 unit areas around a pixel. The algorithm is straightforward: for each unit area compute the s samples inside it. Then, for each pixel affected by this unit area, compute the color contribution of the samples to that pixel and add to the pixel's color. This is shown in Figure 3.2. This time, the number of evaluations is given by $w \cdot h \cdot s$, which is exactly k^2 times smaller than before.

```
function rasterize( ... )
    -- iterate over the unit areas of the image
    for ua in unit_areas_iterator() do
        -- allocate a buffer to hold the
        -- colors of the 's' samples in this
        -- unit area
        local colors = { }
        -- iterate over the samples computing
        -- their colors and storing them.
        for s in sample_pattern_iterator() do
            color[s] = evaluate( ua, s )
        end
        -- iterate over the pixels affected by this
        -- unit area
        for p in pixel_affected( ua ) do
            -- for each pixel, one polynomial part of
            -- the filter is selected
            local filter = select_filter_polynomial( ua, p )
            -- empty color
            local pcolor = {}
            -- loop over the samples again computing the
            -- the color
            for s in sample_pattern_iterator() do
                pcolor = pcolor + filter( s, ua )*color[s]
            end
            -- update the output image, pixel p
            -- adding the color from this unit area
            image.add( p, pcolor )
        end
    end
end
```

Figure 3.2: Rendering the scene iterating over the unit areas.

3.2 Acceleration Data Structures

The algorithm presented above is faster than the initial version, but it has not changed the asymptotic behavior of the method. Most of its effort is put on computing the color of the samples. This is exactly the operation we wish to optimize (Figure 1.9).

3.2.1 Splitting the scene

We know that the cost of sampling lies in the procedure that computes the intersection between the ray emanating from the point and the Bézier segments that comprise the boundary of the paths. We can speed up computations by avoiding those tests altogether. To do that, we will exploit the geometry of the scene and create an acceleration data structure that partitions the space, separating the complex regions from the simpler ones. Our choice is a quadtree [Warnock, 1969].

To build a quadtree that fits into our purposes, we will suppose the existence of two functions. The first one, which we shall call *simplify*, takes as input two arguments: a vector graphics scene and a rectangular area in the plane, called a *cell*. The return of *simplify* is a new scene in which the winding number of each path, when evaluated at points inside the cell, remains the same as in the original scene. Hopefully, *simplify* will change the scene so that the computation of those winding numbers is simpler than before. We try to exploit the geometric coherence of the scene: segments that are far from a sample might not affect the computation of the winding number. The second function, or *predicate*, is called *is_complex* and: given the same arguments, should answer whether the cell is worth being simplified. We are going to show concrete examples of these function later.

We start with the original scene and the area of interest represented by a bounding box. This bounding box is the root of the quadtree and usually corresponds to the viewport or to an area containing it. We apply *simplify* to the root in order to guarantee that the representation of the scene is simple and correct inside its boundary (see figure 3.3a). Then, we recursively apply a procedure that goes through the following steps:

1. Apply *is_complex* to test whether the node is worth subdividing. If true then;
2. Split the cell at the middle point into four parts;
3. For each sub cell, apply *simplify* and recursively call the procedure.

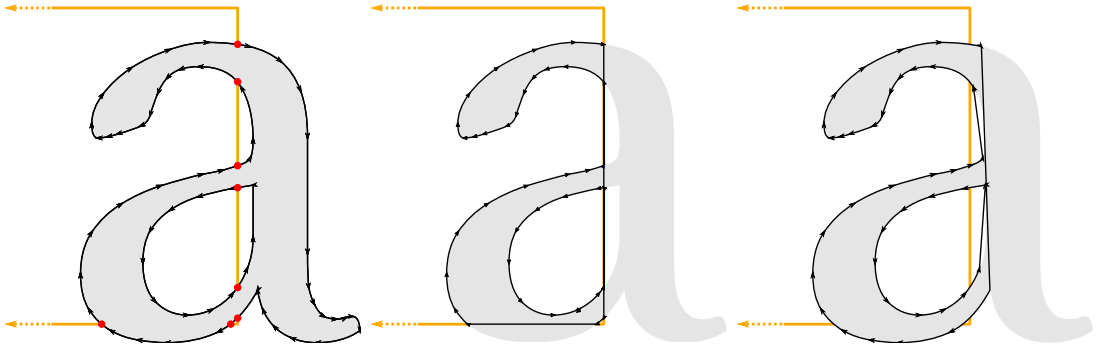
This is depicted in figure 3.3b.

(a) Root node. *(b) After simplification.*

Figure 3.3: The root node is simplified to guarantee consistency and then split into child cells.

Now, we show two ways of defining the function *simplify*. The first one is built upon the algorithm for cutting polygons create by Sutherland and Hodgman [1974]. The second simplifies the paths without cutting segments. Both methods yield useful *simplify* functions, with slight different results; the second is less efficient.

The first method simplifies paths by clipping its segments against the lines defined by the cell's borders, one line at a time. Each line divides the plane in two disjoint semi-spaces. One of them containing the cell and the line. The method iterates over all segments in the path, testing whether the segment is completely inside the semi-space that contains the cell. If so, this segment must be kept intact to avoid changing the geometry of the path inside the cell. A segment that is not completely inside the semi-space presents an opportunity for the simplification of the path. This segment is broken into pieces that are either completely inside, or completely outside the semi-space. The pieces that are inside are kept. The pieces that are outside are replaced by line segments to keep the winding number computations correct. These segments will necessarily lie over the cell's boundary. Figure 3.4a shows the intersection between the path and the cell's boundaries while figure 3.4b shows the result of the method.

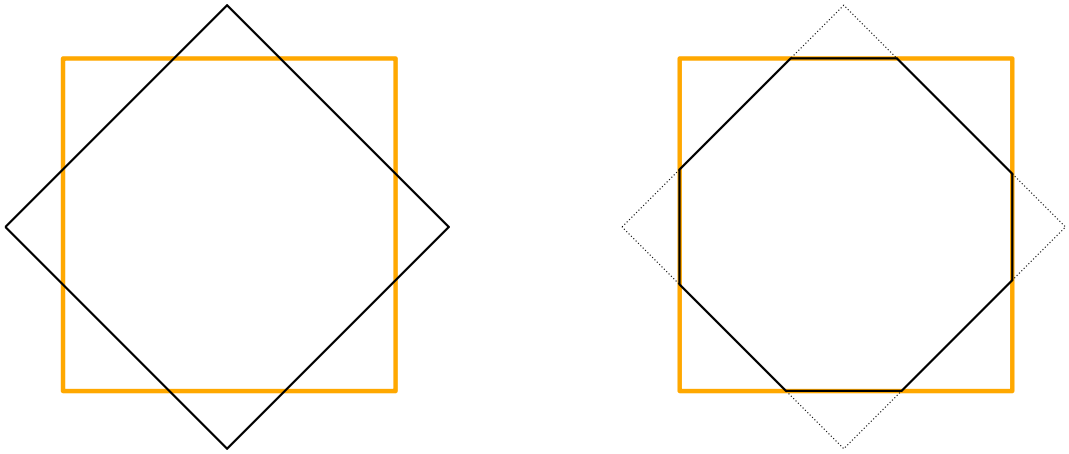


(a) Intersection points. *(b) Cutting.* *(c) Without cutting.*

Figure 3.4: Different methods of simplifying a complex path.

Another option is to simplify the path without clipping the segments. This method iterates over the path's segments keeping those that intersect the cell. When a segment is found that begins inside the semi-space that contains the cell but ends outside, all the following segments that are completely outside the semi-space are replaced by linear segments. The result is a simplified path that is not completely inside the cell, but whose parts that are outside are comprised by just a few line segments. This is shown in figure 3.4c.

These methods have similar behavior but they may produce different outputs. Cutting may not always produce a simpler path. For instance, Figure 3.5 shows a situation where it may add segments to the path, making the evaluation more costly. This may be avoided by not cutting segments.



(a) Path with 4 segments. *(b) Path with 8 segments.*

Figure 3.5: Cutting paths may create more complex paths.

The interplay between *simplify* and *is_complex* is an important factor to consider when designing a quadtree algorithm. It is important that *is_complex* is able to detect situations where the subdivision of a cell would not improve the simplification. Such situations occur when the path covers the whole cell. In this case the path could be replaced by a rectangle covering the cell, for example. It is very difficult to detect this situation when we are not cutting segments, since the test would be very costly. On the other hand, cutting segments would produce a result where all output segments lie on the border of the cell, which is an easy configuration to detect.

In figure 3.6 we show an example of quadtree.

3.2.2 Sampling with the quadtree

To compute the color of a sample using the quadtree, we need to find the leaf node of the quadtree that contains the sample, i.e., we need to *traverse* the quadtree. This can be done using a simple procedure. First we test whether the sample is inside the root node. If so, we compare the sample's coordinates with the root's middle point to find which child cell contains the sample. This is done recursively for each node in the tree until the leaf node is found. Then, the part of the scene corresponding to the leaf can be used to compute the color of the sample.

At first, the quadtree traversal needs to be done for each sample in order to produce the final image. Every traversal followed by loading the scene has costs, both in computation time and I/O. We can mitigate this cost by aligning the quadtree to the primal or dual grid associated with the anti-aliasing filter. By aligning we mean that the boundaries between adjacent cells in the quadtree are over the grid's lines (figure 3.7). This has two consequences. First, the cells in the quadtree are made of the union of unit areas in the grid. Second, the smallest cells in the quadtree are no smaller than one unit area.

To create an aligned quadtree, we perform the following steps:

1. From the viewport and anti-aliasing kernel, compute the grid of unit areas: $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$;
2. Compute the smallest number $i \in \mathbb{N}$ such that the grid's width and height are smaller than or equal to 2^i;
3. Use $[x_{min}, x_{min} + 2^i] \times [y_{min}, y_{min} + 2^i]$ as bounding box of the root cell.

Since every subdivision step in the creation of the quadtree implies a division by 2 in both x and y dimension, choosing the root's bounding box as above ensures

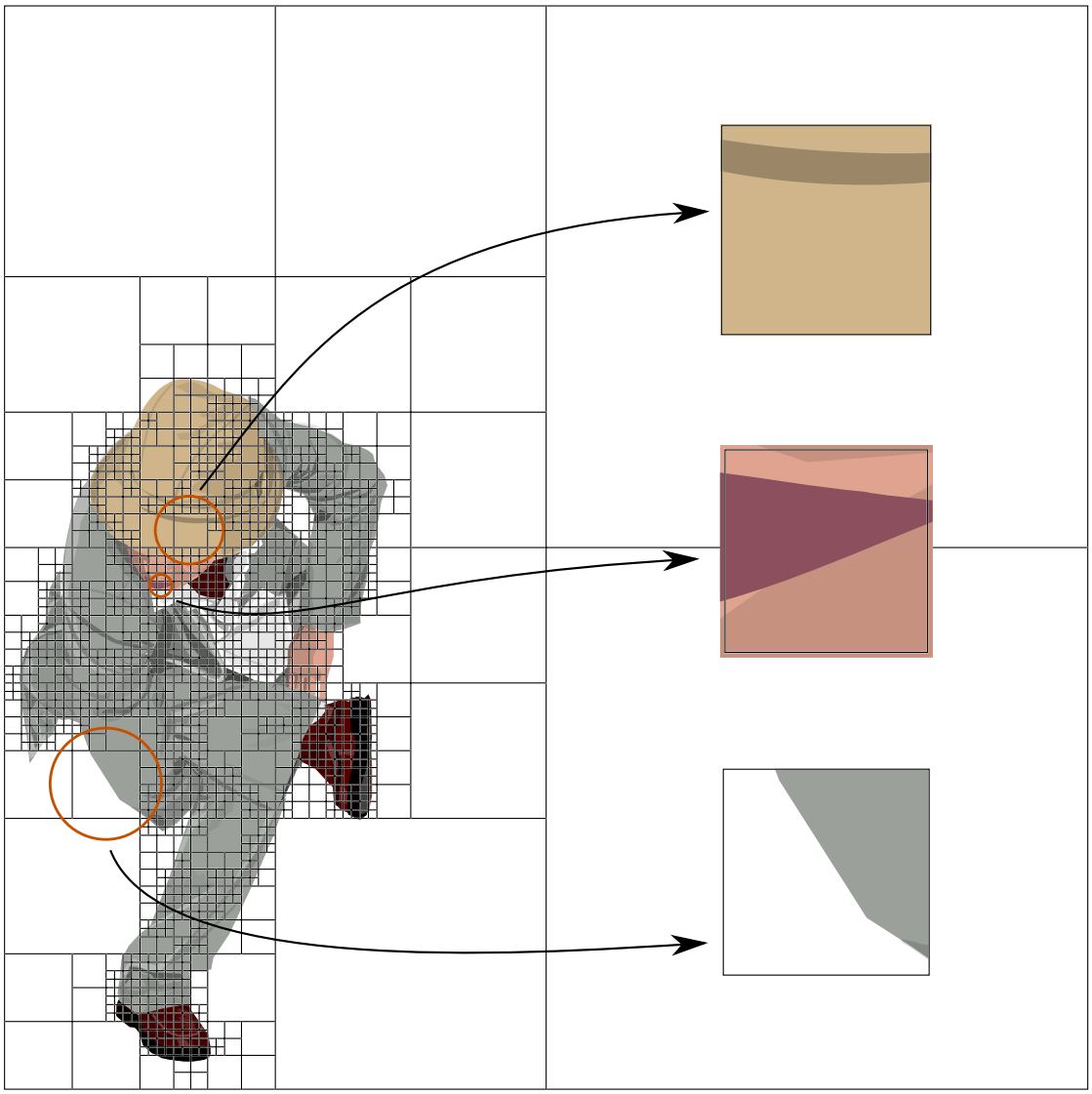


Figure 3.6: Dancer subdivided to level 7 using the method of cutting segments.

that cell's borders always fall over the grids lines. We also limit the number of subdivisions of our tree to i in order to ensure the second condition. The steps above are illustrated in figure 3.7.

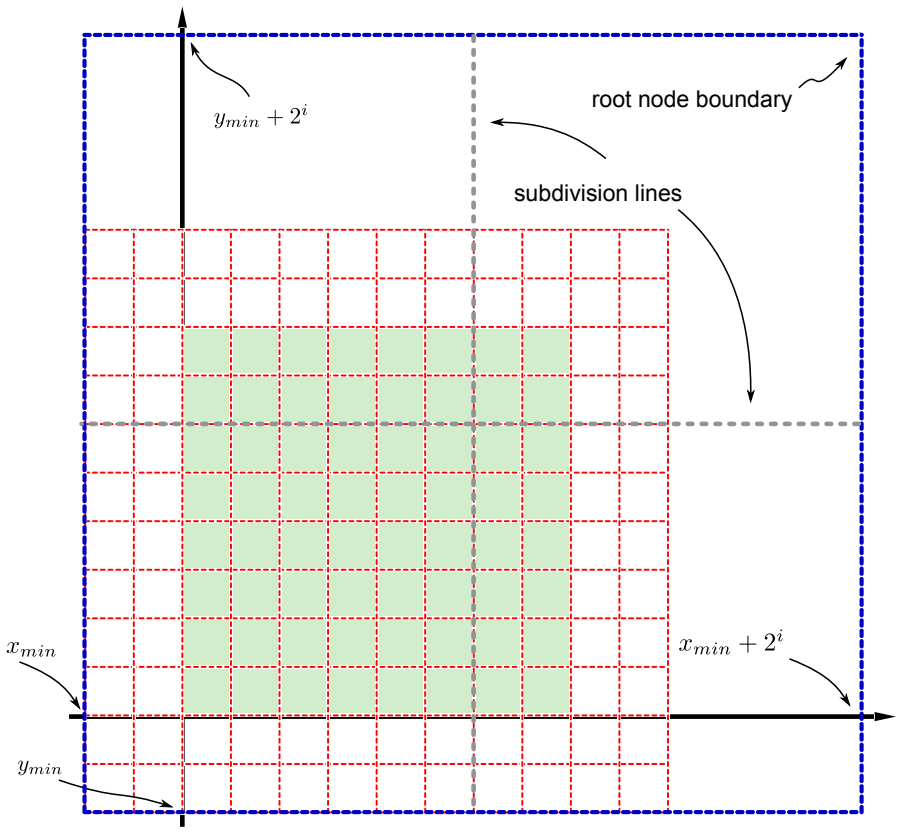


Figure 3.7: The root node is the smallest square containing the unit areas' grid, whose side is a power of 2. Its lower left corner is at (x_{min}, y_{min}).

With the aligned quadtree, computing the colors of all samples in a unit area can be done with one traversal followed by one operation of loading the cell's content. We usually choose the center of the unit area as the point used to traverse the quadtree, since all samples in that unit area are computed as offsets from the center. These offsets are given by the sampling pattern (figure 1.29). This scheme fits perfectly with the algorithm shown in figure 3.2. We can rewrite it to take the unit area being processed and compute its center, then traverse the quadtree to fetch the part of the scene relative to the leaf. This part of the scene can be used exactly as shown before (figure 1.23), given that its content is a well formed scene.

3.2.3 Experiments

To validate the ideas in this chapter we designed an experiment. We compared the rendering time of the algorithm when processing benchmark scenes. These scenes were constructed using randomly placed polygons, the number of polygons varies such that the total number of segments in the scene is in the interval from 4 to 1024. Each benchmark scene was created having either small or large polygons, and each polygon in the scene has either 4 or 8 sides. We show examples of these scenes in table 3.1.

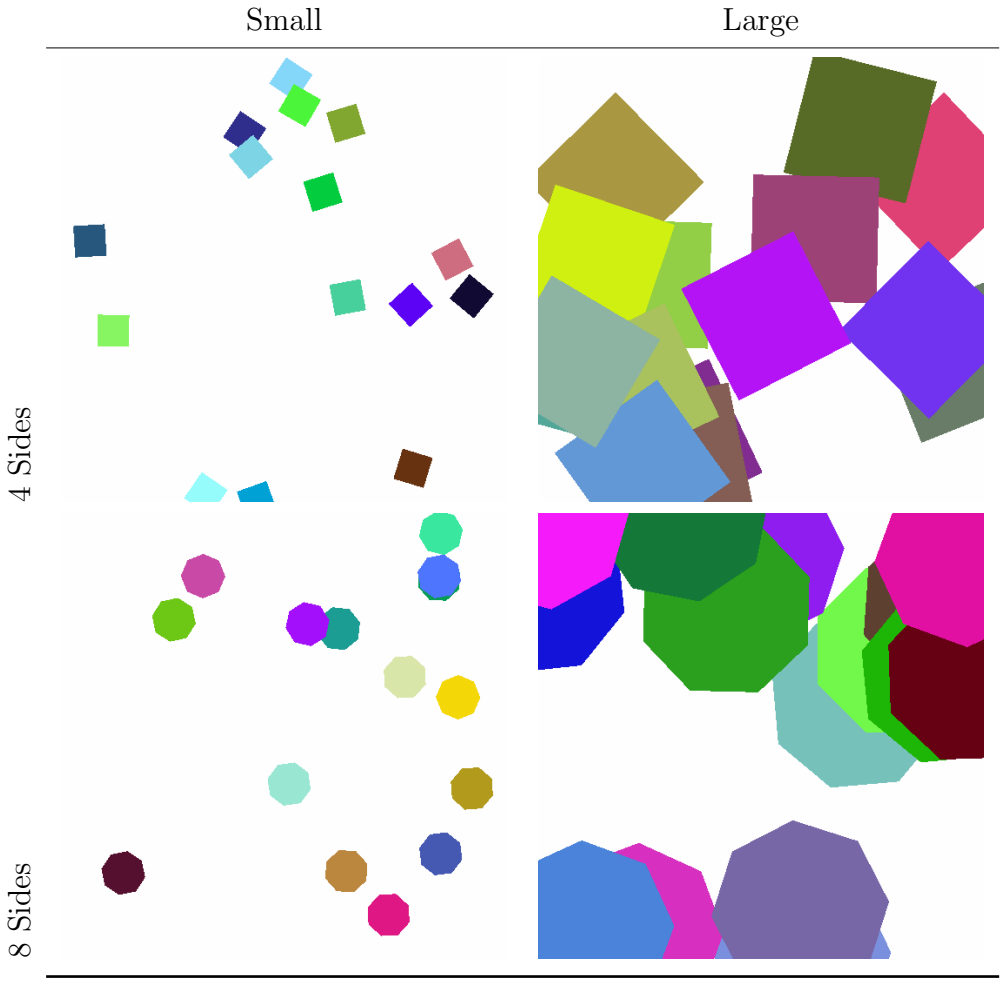


Table 3.1: Examples of the scenes used as benchmark.

Figure 3.8 shows the result. Along with the times for this experiment we provide for comparison the results shown in section 1.3.1 for randomly generated scenes

rendered with no acceleration data structure. Looking at the results, two distinct groups are evident: one for scenes with large polygons and the other for small polygons. The gap between these two groups can be explained by the fact that the subdivision process of the quadtree easily isolates the small polygons. Since they are small and uniformly distributed in the scene, there are very little overlapping between then. This configuration results in leaf cells whose geometry are simple and that can be sampled fast. On the other hand, large polygons will overlap. After subdivision, the quadtree will generate leaf cells whose complexity is larger than the cells generated before. They may contain many different paths. Again, looking at the curves for large polygons, we see that the performance for polygons of 4 sides is worse than the performance for polygons of 8 sides. This is the effect of cutting the polygons against the cells' borders. Polygons with 8 sides are more heavily simplified, resulting in paths with fewer segments. The same effect can be seen for scenes with small polygons, but with less impact.

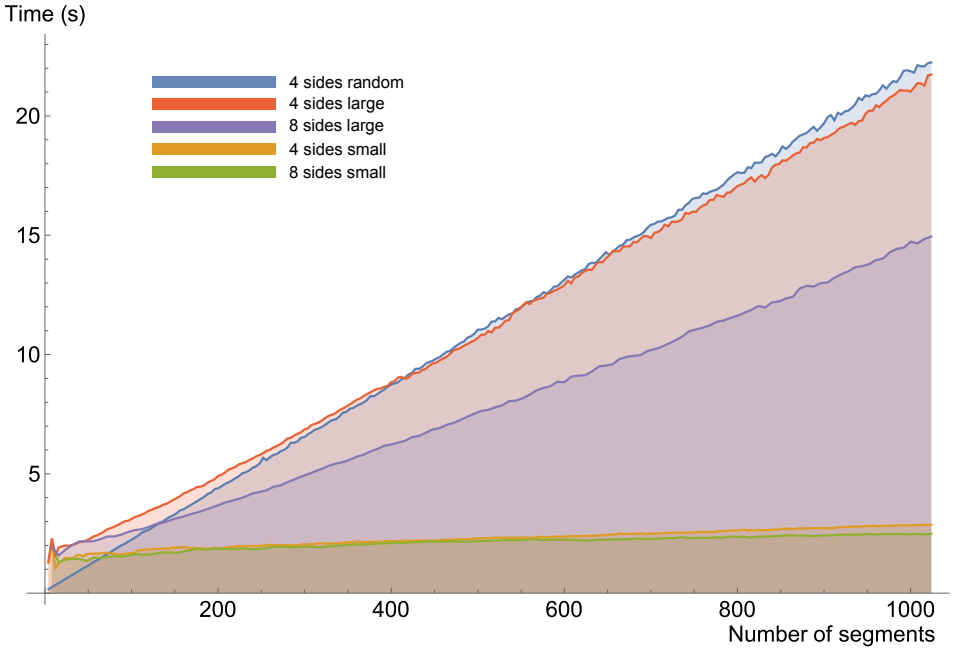


Figure 3.8: Rendering times for benchmark scenes.

It is disappointing that the performance for large polygons is close to the performance of the naive method from chapter 1. The solution, in this case, is to use front-to-back rendering. Figure 3.9 shows the experiment using front-to-back rendering with large polygons with 4 sides. It is clear that performance has improved. The reason is that most of the leaf cells yielded by the quadtree have paths that completely cover them. This makes the front-to-back sampling very efficient.

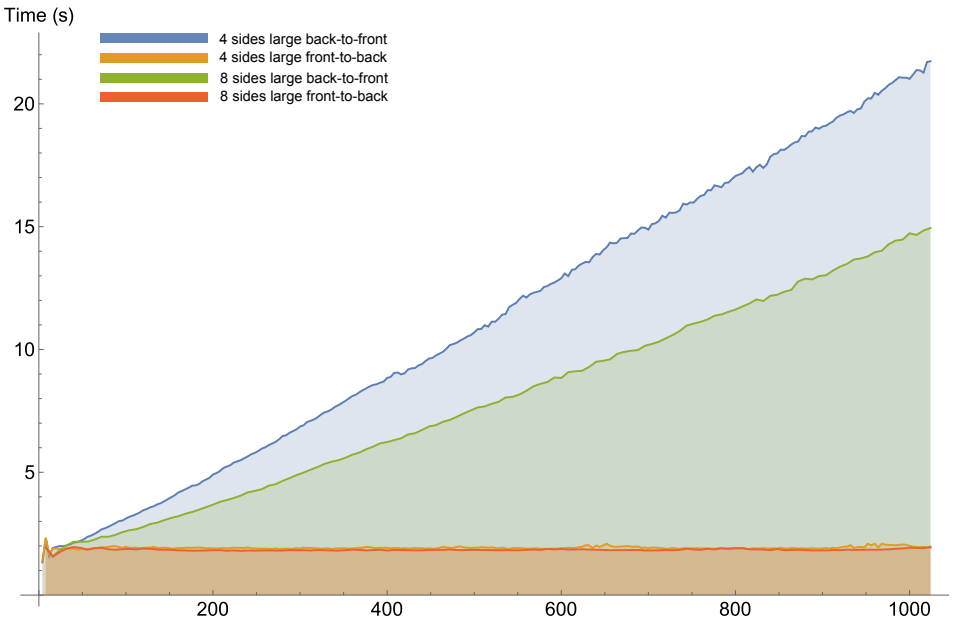


Figure 3.9: Rendering times with front-to-back rendering.

One last question that remains is whether the naive algorithm from chapter 1, using front-to-back rendering, would be as effective. The answer is no. The reason is that without the elimination of paths from the cells performed by the quadtree, many point-in-shape tests would have to be performed before the first hit was found.

We can justify the way we constructed the scenes for these experiments by pointing out that scenes with these same properties are common: pages of text have many small paths, figures such as Lion and Drops (see 1.3) have many large overlapping paths.

3.3 Moving to the GPU

In the next chapter we describe our pipeline for vector graphics rendering. It was designed as a extension of the ideas presented in chapter 1 and this chapter. We briefly describe it as follows.

The window and viewport, as defined by the user, are used to define the image transformation (section 1.1.2). As the scene enters the pipeline, all segments are transformed to image coordinates and then converted into monotonic abstract segments. Rendering occurs in image coordinates. The chosen antialiasing kernel

defines the grid of unit areas. This grid defines the root node of our acceleration data structure: the shortcut tree. The shortcut tree is computed in breadth first order, as a sequence of subdivisions and pruning steps. The tree is sampled in order to compute the final image. Each sample collected affects as many pixels as the size of the support of the antialiasing kernel. Samples are shared in order to maximize performance while achieving great quality.

The conversion of segments into abstract segments is performed in parallel at the segment level. Each segment is broken into monotonic pieces. The resulting segments are implicitized using the method of Loop and Blinn [2005], and an abstract segment is created. Abstract segments can be queried efficiently for intersections with horizontal rays.

The computation of the shortcut tree is done by a novel algorithm that encodes the scene using a method inspired by the lattice-clipping from Nehab and Hoppe [2008]. Instead of clipping, we simply detect intersections between the segments and the cells of the quadtree. The local nature of the decision taken by the algorithm allow us to implement it in parallel. We process each abstract segment independently. After subdivision, pruning is done. It simplifies the tree's cells in order to reduce the complexity of the newly created nodes.

When rendering, all samples are computed in parallel. For that, we created a scheduler that maps each sample to be evaluated to its containing quadtree node. This map is then transposed, that is, we create a list of samples for each node. This list is broken into chunks, that are consumed by working groups of threads. Grouping samples in this way reduces I/O and computation. The contribution of color of each unit area is computed in local memory and then committed to global memory.

Our pipeline delivers high quality images with high performance.

Chapter 4

Massively Parallel Vector Graphics

As we have seen in previous chapters, previous works have at least one sequential stage. Nehab and Hoppe [2008] construct their acceleration data structure sequentially before rendering. Kilgard and Bolz [2012] have to process each path sequentially in order to compute the sample's colors. We designed our vector graphics pipeline to be massively parallel at every stage. As shown in figure 4.1, it is divided into a preprocessing component that is parallel at the input segment level, and a rendering component that is parallel at output sample and pixel levels.

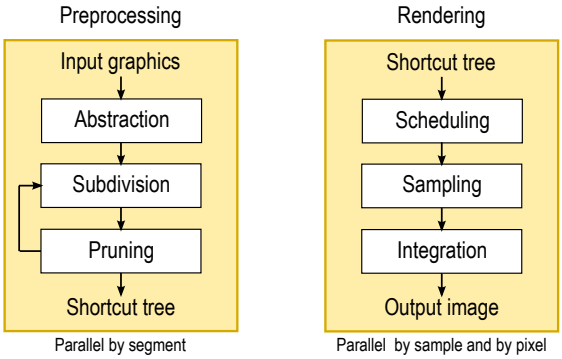


Figure 4.1: Structure of the pipeline.

At preprocessing, every Bézier segment in the scene is converted into an *abstract segment* using monotonization and implicitization. Monotonic segments can be intersected only once by horizontal rays, and can be efficiently queried for the

existence of such intersections. Abstract segments are used to create the *shortcut tree*. A shortcut tree is a quadtree adapted to the scene that allows the color of each point in the illustration to be evaluated very efficiently. Each of its leaves contains the minimal amount of geometry necessary to correctly evaluate the winding number of its internal points. The shortcut tree is constructed in a breath-first fashion. At each subdivision level, all segments in all cells marked for subdivision are processed in parallel and routed to the appropriated child cells. As subdivision progresses, *pruning* stage considers all segments in parallel and eliminates those that have been clipped or entirely occluded by other paths.

The rendering component finds the leaf cell that contains each sample, loads the appropriate cell contents, and performs the required computations to evaluate the sample colors. To enable efficient support for user-defined warps, the samples are *scheduled* so that those falling within the same leaf cell are grouped together. This allows the *sampler* to evaluate these samples in parallel, without control-flow divergence, while reusing the bandwidth required to load cell contents. Moreover, samples falling under the overlapping supports of antialiasing filters associated with neighboring pixels are evaluated only once and shared between them. *Integration* of sample colors happens in fast local memory, before pixels are written to global memory. This setup enables antialiasing filters with wide support (e.g., 4×4) and large sampling rates (e.g., 512 samples/pixel) for sharp, noise-free renderings.

4.1 Abstraction

The color of a sample is computed by selectively blending the paints of all paths for which the sample passes the inside-outside test, in addition to the inside-outside test of all active clip-paths. The fundamental inside-outside test consists of applying the path's fill rule to the winding number of the path about the sample. The winding number is computed by counting the number of intersections between a horizontal ray, shot from the sample to infinity in the $+x$ direction, and all segments in the path, incrementing or decrementing depending on whether segments are going up or down at each intersection.

Nehab and Hoppe [2008] compute winding numbers by solving for the parameter values corresponding to the intersections between each segment and the ray (by solving linear, quadratic, or cubic equations), keeping those in the interval $[0, 1]$, substituting into the parametric equation of the segment to find the intersection point, and accepting only the intersections to the right of the sample point.

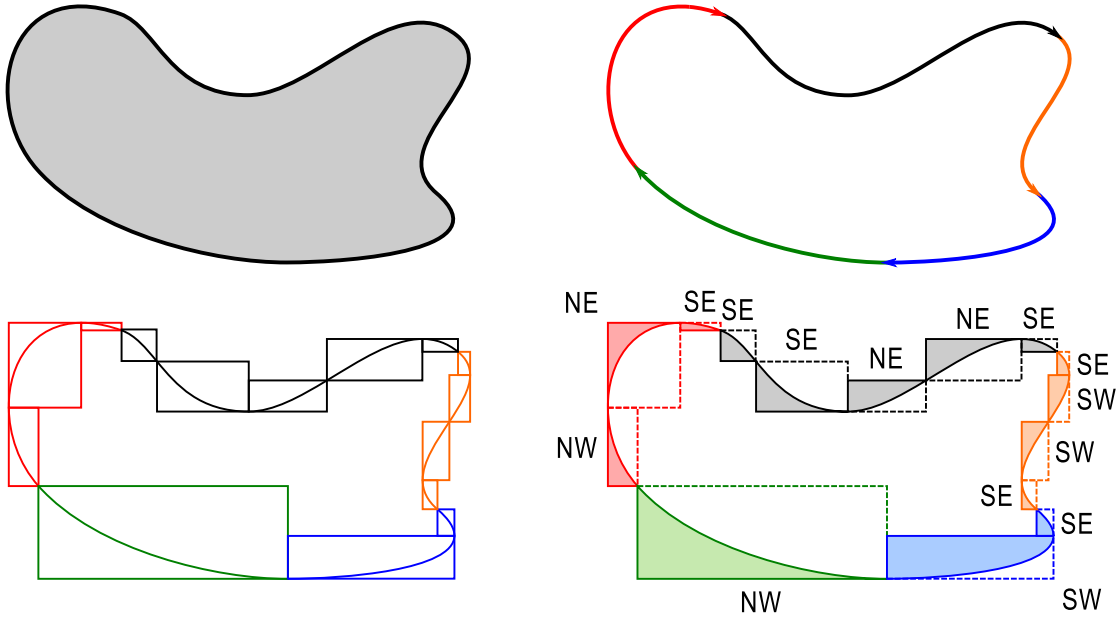


Figure 4.2: The segments in each path are classified and decomposed into monotonic abstract segments*. Abstract segments can be queried for their bounding box, their orientation* (NE,NW,SE,SW)*, and for the side on which a sample lies.*

We use *abstract segments* (using monotonization and implicitization) to greatly simplify this process. Abstract segments can be queried for a bounding box, for an orientation (NE, NW, SE, SW), and for the side of the segment on which a given sample lies. Figure 4.2 illustrates the decomposition of a contour into abstract segments.

Since abstract segments are monotonic, they can be intersected only once by horizontal rays. As seen in section 1.1.3, we can identify whether there is an intersection by comparing the sample's position with the segment's bounding box. The difference in this case is that instead of solving the system of equations 1.1 we use the implicit form of the segment. By testing the sign of the implicit form of the parametric segment at the sample position, we can determine the side of the segment on which the sample lies. If it is on the left, there is an intersection, otherwise there isn't. The intersection itself need not be computed. Similar reasoning applies to vertical rays, which are used during the construction of the shortcut tree. This is shown in figure 4.3.

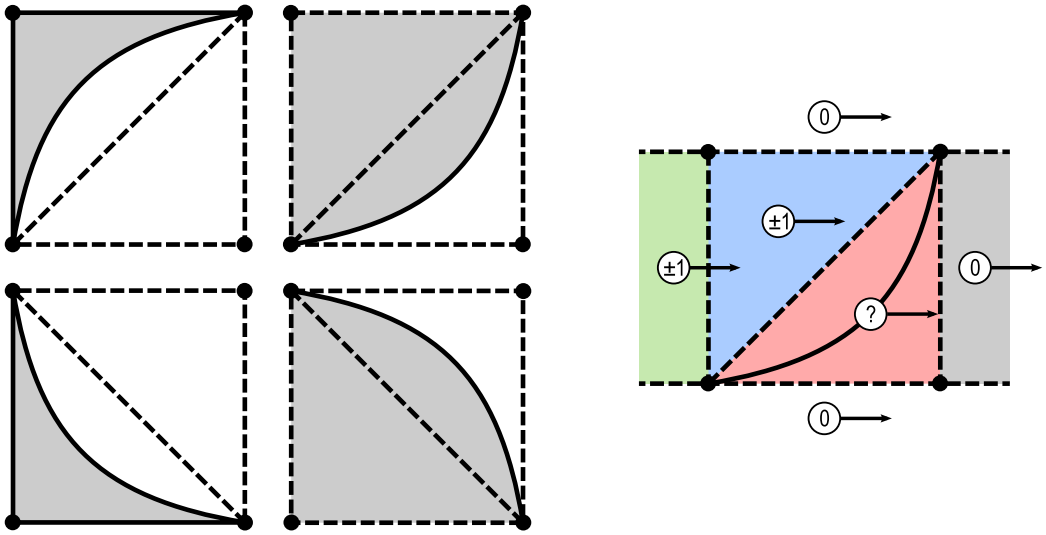


Figure 4.3: Monotonic segments can appear in one of four configurations. Intersections with horizontal rays can be ruled out or confirmed trivially unless the sample and the segment are in the same side of the dashed bounding box diagonal. In that case, an implicit test is used. The intersection itself need not be computed.

4.1.1 Monotonization and Implicitization

Monotonization of segments is done, as shown in section 1.1.3, by splitting the segments at the extreme points of its derivatives: $x'(t) = 0$ or $y(t) = 0$.

The implicit representation of the segments is found using the method of Loop and Blinn [2005] (section 2.1.1) Each abstract segment stores the row vectors corresponding to the required affine functions, and we use them to quickly perform implicit tests on the required samples.

There is one important caveat. Within the region where implicit tests are used, we must ensure that the implicit function changes sign only once along axis-aligned rays. This is to prevent the situations depicted in figure 4.5, which would cause an incorrect number of detected intersections and ultimately to incorrect rendering. Since segments have been monotonized, it suffices to prove that the parametric curve is outside the test region for all parameters outside of $[0, 1]$. The diagrams in figure 4.4 illustrate the proofs that follow.

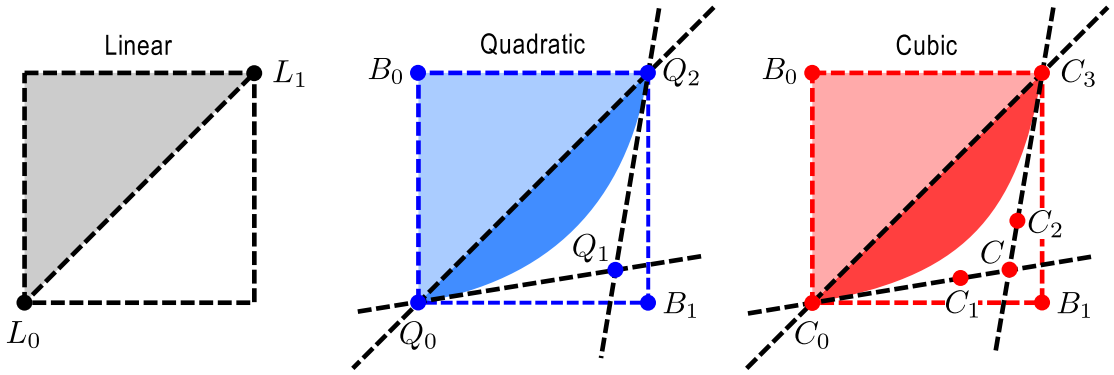


Figure 4.4: Linear segments pose no problems. Implicit tests must be restricted to triangle $Q_0B_1Q_2$ in the case of quadratics, and to triangle C_0CC_3 in the case of cubics.

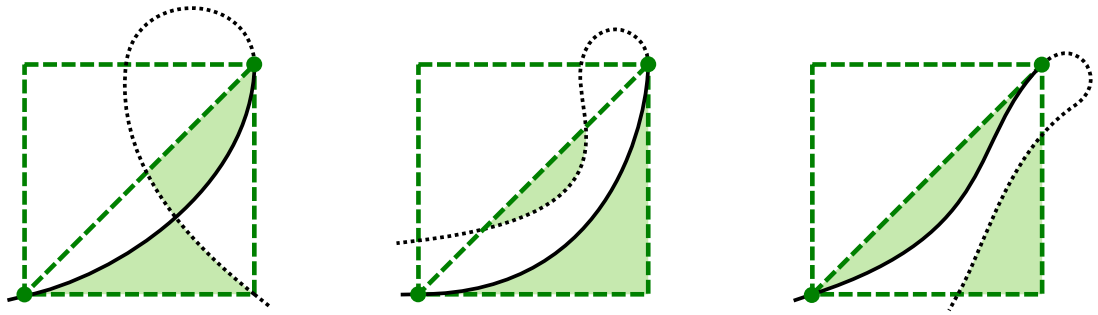


Figure 4.5: Within the implicit test region, the implicit function must only change sign once along any horizontal rays, or the algorithm would report incorrect intersection counts as in the examples.

Quadratic segments It is sufficient to restrict the test to triangle $Q_0B_1Q_2$. This requires one implicit test against segment Q_0Q_2 and two comparisons against bounding box coordinates.

Proof: The quadratic curve cannot cross segment Q_0Q_2 outside of points Q_0 and Q_1 since it can intersect a straight line at most twice. Similarly, it cannot cross segments Q_0Q_1 and Q_1Q_2. Indeed, since the curve is tangent at both Q_0 and Q_2, these points count as double intersections. Finally, note the quadratic cannot intersect segments Q_0B_1 and B_1Q_2 without first incurring forbidden additional intersections with segment Q_0Q_1 or Q_1Q_2, respectively. □

Loop and Blinn [2005] use the GPU rasterizer to generate fragments only inside triangle $Q_0Q_1Q_2$. Using their solution would require us to perform three implicit line tests.

Cubic segments Cubics are more demanding. To prevent the curve from looping back and intersecting segment C_0C_3 (and the curve itself), Loop and Blinn [2005] split cubics at a double point whenever one is found for a parameter t_d with $0 < t_d < 1$. This requires solving a quadratic equation, and we do the same. We go one step further and split the cubics at an inflection point whenever one is found for t_i with $0 < t_i < 1$. This requires solving another quadratic, but ensures the intersection of lines C_0C_1 and C_2C_3 happens at a point C inside the bounding box. Then, it is sufficient to restrict the implicit test to the triangle C_0CC_3. Note that these quadratics must be solved during the implicitization process anyway.

Proof: We again use root-counting arguments. First, we show that the curve cannot intersect segment C_0C_3. If it did, it would either have to exit triangle C_0CC_3 again through segment C_0C_3 (but it cannot have four intersections with line C_0C_3), or it would have to self-intersect (but by assumption it has no double point for $t \in (0, 1)$). The arguments for why the curve cannot intersect segment C_0C and CC_3 are analogous, so consider segment C_0C. We start from the part of the curve that exits triangle C_0CC_3 at C_0. If it is below C_0C, then C_0 is an inflection and exhausts all three possible intersections with line C_0C. If it is above C_0C, then C_0 is only a tangent. Now recall the curve cannot intersect C_0C_3. Therefore, in order to intersect C_0C a third time, it would have to either go up around triangle C_0CC_3, thereby intersecting CC_3 four times (twice at the tangent C_3 and twice before it can reach C_0C), or go down back into C_0C (wasting the third and last intersection with line C_0C at a point outside of segment C_0C). Now consider the part of the curve exiting at C_3. If it exits to the right of CC_3, then C_3 is an inflection and precludes the fourth intersection with line CC_3, needed to reach segment C_0C. If it exits to the left, it would intersect line C_0C the third time outside of segment C_0C, since it cannot intersect segment C_0C_3. □

Loop and Blinn [2005] use the GPU rasterizer to generate fragments only inside the two triangles that form the convex hull of the cubic control polygon. Adopting this approach would require us to perform at least four implicit line tests and maintain some bookkeeping.

4.1.2 Scene Abstraction

We store a vector graphics scene as a stream that contains the geometry, as well as auxiliary information, such as paint data and delimiters for clipping operations. The structure of the stream is best described by a context-free grammar, whose production rules are:

$$\begin{aligned}
\textit{scene} &\rightarrow \texttt{(}\ \textit{fill}^*\ \texttt{)} && (4.1)\\
\textit{fill} &\rightarrow \texttt{F} && (4.2)\\
\textit{fill} &\rightarrow \texttt{(}\ \textit{clip-path}^*\ \texttt{|}\ \textit{fill}^*\ \texttt{)} && (4.3)\\
\textit{clip-path} &\rightarrow \texttt{C} && (4.4)\\
\textit{clip-path} &\rightarrow \texttt{(}\ \textit{clip-path}^*\ \texttt{|}\ \textit{clip-path}^*\ \texttt{)} && (4.5)
\end{aligned}$$

Here, terminal production (4.2) stands for a filled path. Terminal production (4.4) stands for a clip test. The geometry of filled paths and clip tests is given by a list of segments that form the outlines of the shape. Filled paths contain additional paint information enabling the computation of sample colors (e.g., gradient and texture paints).

The remaining terminals '`(`', '`|`', and '`)`' are short for *push*, *activate*, and *pop*, respectively, and are used to delimit clipping operations. The clip-path area starts empty with a *push*, and is given by the union of an arbitrary number of clip tests appearing before its matching *activate*. An union of clip-paths can be seen as an *or* operator, that is, if a sample lies inside one of the clip-paths then it is considered inside the union of all clip-paths. These clip-paths can themselves be clipped by other clip-paths, so that nesting is equivalent to intersection. The clip-path is active between the *activate* and its matching *pop*. The entire scene is delimited by a dummy *push–pop* pair. Production (4.5) can be seen as an *and* operator, that is, a clipping test succeeds if it succeeds at both sides of the expression.

Scenes can be represented in two ways: back-to-front or front-to-back order. In back-to-front representation, the first path (clipping or fill) to appear in the stream is the *bottommost* path in the scene. In front-to-back representation, the first path to appear in the stream is the *topmost* path in the scene. Back-to-front is the most commonly used representation. It fits directly into the vector graphics model given so far, and allow the scene to be evaluated easily. Front-to-back makes more convenient to blend samples from front to back and abort the computation as soon as the sample color becomes opaque (see section 1.4.2).

4.2 The shortcut tree

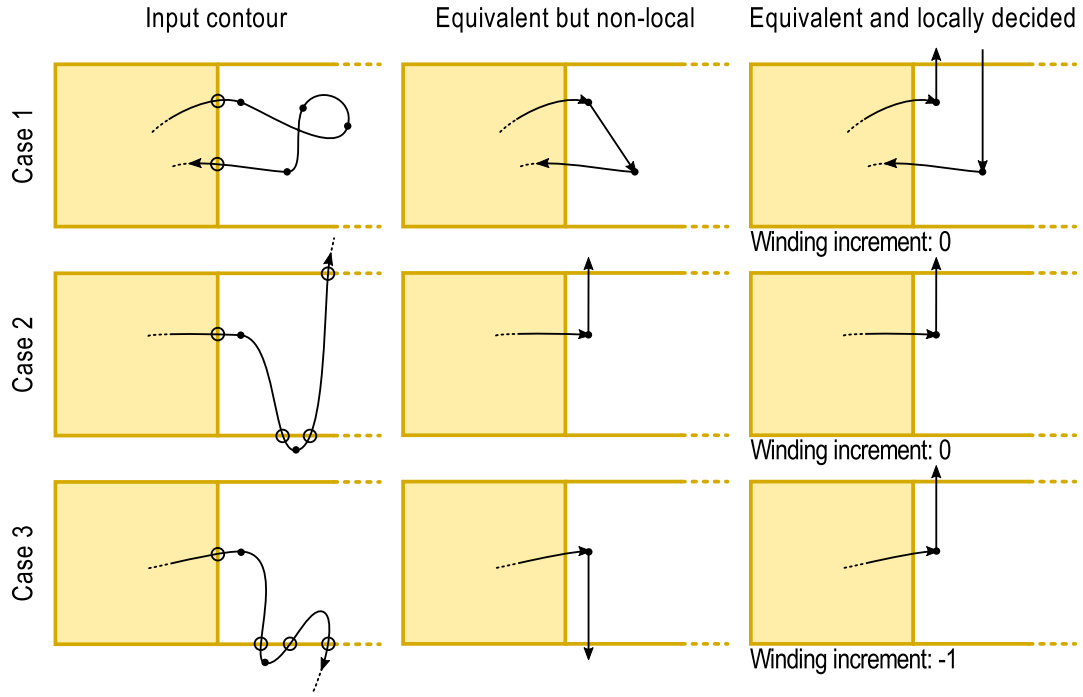


Figure 4.6: Contours on the left column are equivalent to contours in the central and right columns. The intuitive representation in the center requires knowledge about segments that can be far apart in the input. The equivalent representation on the right can be generated locally by inspecting segments independently.

Assume that we have partitioned the illustration area into a union of small cells. The key strategy for speeding up the inside-outside test within each cell is to reduce the number of segments that must be tested for intersection during the computation of winding numbers. We will specialize the representation of the illustration within each cell while maintaining the following invariant: the winding number of any path about any sample in a cell, computed by shooting a horizontal ray from the sample to infinity in the $+x$ direction, is the same as in the original illustration. It is clear that we can eliminate a segment whenever its bounding box is completely above, below, or to the left of the cell. The difficulty is what to do otherwise. The breakthrough in the *lattice clipping* algorithm of Nehab and Hoppe [2008] is to include in each cell only the parts of segments that overlap with them, with the addition of *winding increments* and *shortcut segments* that restore the invariant. We describe an improved version of the idea that does not clip segments to the interior of cells and thus eliminates all intersection computations.

The examples in figure 4.6 show how a contour's behavior to the right of a cell boundary can be summarized with shortcut segments. Since all input contours are closed, any contour that leaves a cell by crossing its right boundary must later return to the cell. If the contour does not return to the cell via the right boundary, it must return from a different side. In order to do that, the contour must first leave the row of cells, and this must happen in the region to the right of the cell. Therefore, there are only three possibilities: (1) the contour comes back inside by crossing the right boundary again; (2) it exits to the row above; or (3) it exits to the row below. In case 1, we can represent all omitted segments by a shortcut connecting the end of the exiting segment and the beginning of the entering segment. In case 2, we can add a shortcut going up from the end of the segment that intersects the right boundary. Finally, in case 3, we can add a shortcut going down from the end of the segment that intersects the right boundary. The reasoning is similar for contours entering the cell from the right boundary. In figure 4.6, note how the winding numbers obtained from the input contours in the left column are the same as those obtained with the compressed representation in the central column, no matter where the sample lies inside the cell area.

Unfortunately, no local procedure can distinguish between these cases by inspecting one segment at a time. Nehab and Hoppe solve this problem by always assuming case 2 and adding a shortcut going up. This change in the shortcut direction will induce an error in the computation of the winding number for all samples in the cell. Fortunately, this error is consistent throughout the cell: it sums $+1$ to the winding number of all samples about this path. All errors combined add to a value k. The invariant–that is, the winding number of each sample–can be restored by modifying the initial winding number of all cells to the left of the violating intersection by $-k$. The value $-k$ can be found by explicit computing the *correct* winding number of the bottom-right corner of the cell. Its winding number is given by the number of intersection between segments of the path and the horizontal line emanating from the point.

The result of the original lattice clipping algorithm is a regular grid of cells. We now proceed to the description of the shortcut tree, our hierarchical data structure based on the same ideas and, more importantly, how to build it efficiently and in parallel.

4.2.1 Subdivision

The key operation when building the shortcut tree is *cell subdivision*. Assuming that a parent cell respects the invariant, our task is to find subdivision rules that

produce child cells that also respect it. Then, by induction, the resulting tree will satisfy the invariant everywhere.

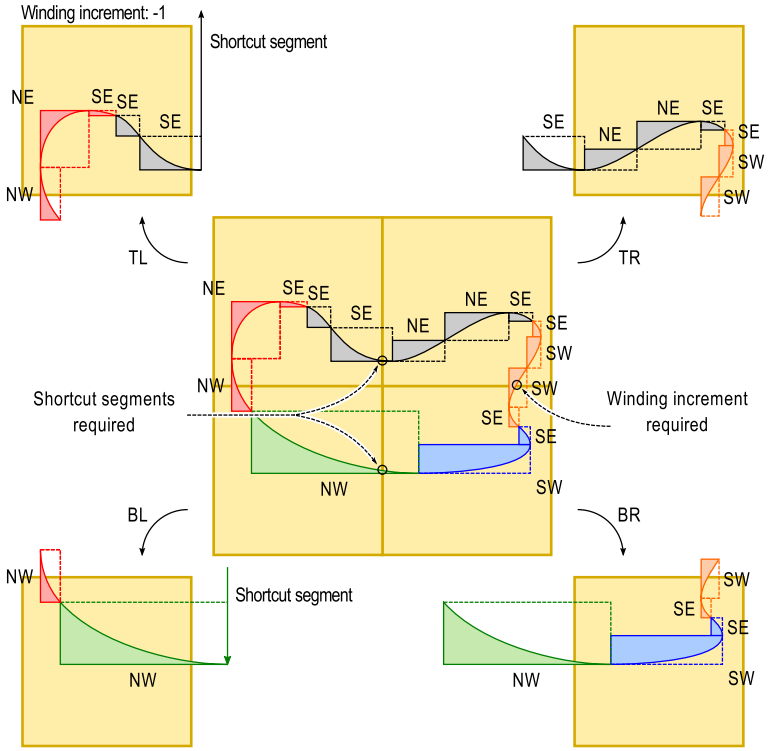


Figure 4.7: Example subdivision of a shortcut tree cell. Segments are included in a child cell if and only if they intersect its area. Marked intersections generate shortcut segments and winding increments that restore the invariant within each child cell.

Figure 4.7 shows an example of cell subdivision. A child cell includes a segment (in its original form, *including the parts outside the cell*) if and only if the segment intersects its area. The intersections with the TL / TR and the BL / BR boundaries respectively generate shortcut segments in child cells TL and BL. The intersection with the TR / BR boundary generates a winding increment in child cell TL that corrects the misclassification of child cell TL from case 2 to case 3.

In general, for each segment in the parent cell, we must decide in which child cells to include it, and whether it generates shortcut segments and we minding increments. The inclusion test is particularly simple: a segment is included in a cell if one of its endpoints is inside the cell or if it intersects one of the cell's boundaries. Figure 4.8 illustrates the procedure used to identify shortcut segments and winding increments. During tree creation, the root of the shortcut tree is generated first. Shortcut

segments are generated for all segments crossing boundary A. Winding increments are generated for all segments crossing the half-line boundary B (which extends to infinity). Cell subdivision is performed in a similar way. Shortcut segments are generated in child cells TL and BL for all segments intersecting boundaries C and D, respectively. Segments intersecting boundary E and G generate winding increments in child cells TL and BL, respectively. Segments intersecting the half-line boundary F generate winding increments in both child cells TL and TR. Likewise, segments intersecting the half-line boundary H generate winding increments in both child cells BL and BR. We can detect intersections by testing whether cell boundary vertices lie on different sides of the abstract segment. To distinguish between intersections with C and D, with E and F, or with G and H, we test the side of their shared vertex.

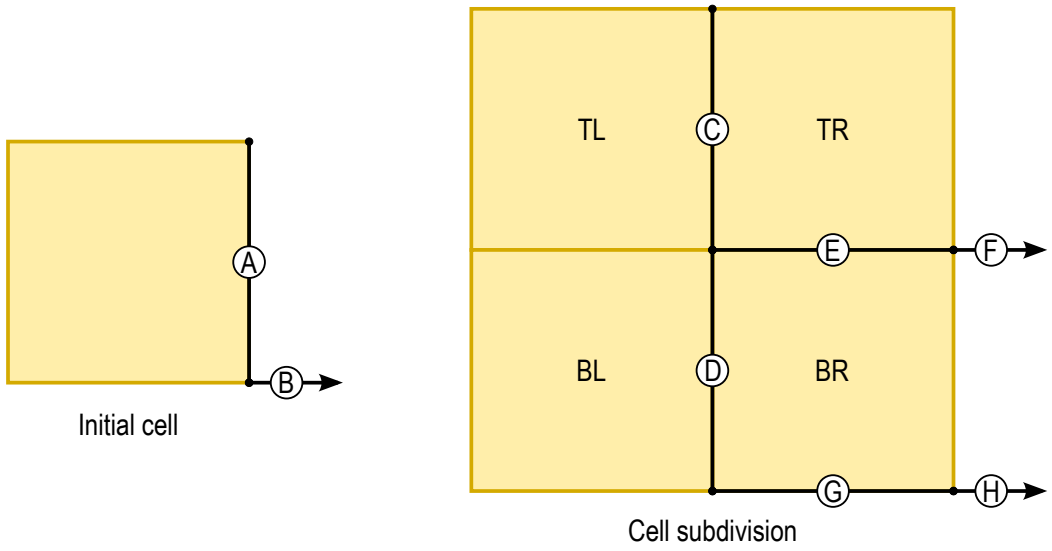


Figure 4.8: During subdivision, segments are classified based on their endpoints, and on intersections with the marked boundaries.

Stopping criteria We stop cell subdivision when: (1) the amount of memory taken by the shortcut tree reaches a maximum threshold; (2) the number of segments in a cell is smaller than a minimum threshold. Criterion 1 allows users to limit memory consumption; criterion 2 prevents futile subdivisions. More sophisticated criteria will be investigated as future work.

4.2.2 Parallel Subdivision

The shortcut tree is generated in parallel, in breadth-first order, with each step subdividing all cells at progressively deeper tree levels. The cells to be subdivided are laid out contiguously in memory. Each cell contains a specialized description of the scene that is correct within the cell's boundaries. The contents of each path in the specialized scene are also laid out contiguously. Paths are represented as sequences of lightweight *segment entries*. Each segment entry uses a few bits to distinguish between clipping control terminals, abstract segments, shortcut segments, and winding increments. (The initial winding number for a path is the sum of all its winding increments.) There is also room for references to the corresponding abstract segment, the originating path, and the originating cell.

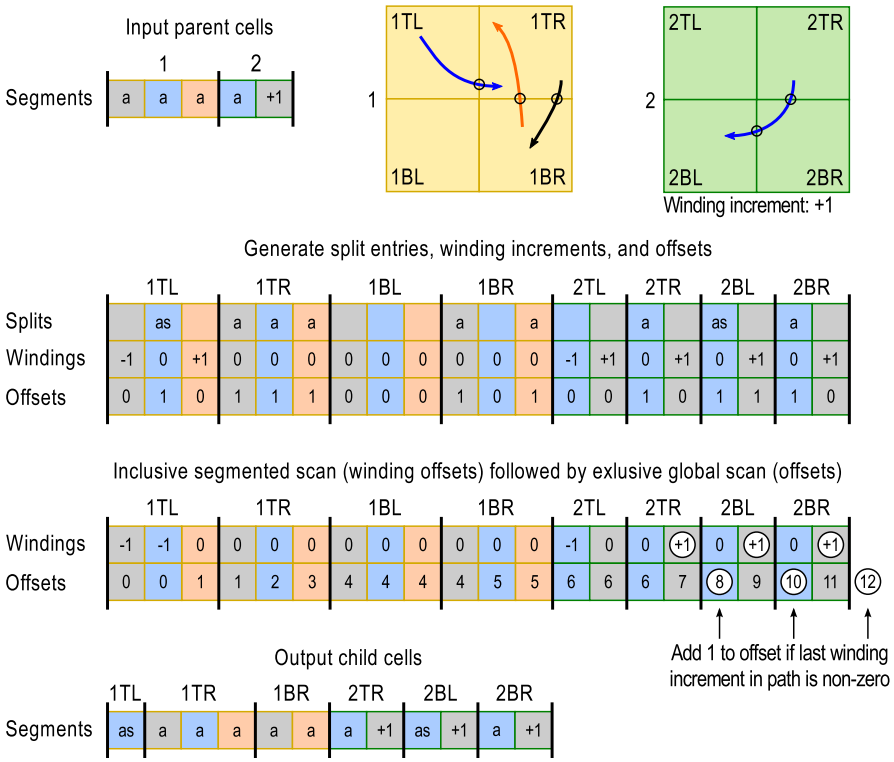


Figure 4.9: Parallel shortcut tree subdivision. Entries for each segment are shaded with its hue. Abstract segments, shortcut segments, and winding increments are respectively marked with a, s, and ±1.

Each round of subdivision independently processes every entry, of every path, of every cell at the current subdivision level. According to the subdivision rules, each parent entry may generate, in each of the four child cells: a reference to an abstract segment, a shortcut segment, and a winding increment. In other words, the number

of outputs varies by input entry. This requires us to compute an *output offset* before we can write the child cells, which in turn forces us to split the process into four computational CUDA kernels [Kirk and Hwu, 2012]. Figure 4.9 shows an example in which two cells are subdivided in parallel. For simplicity, each cell contains a single path and only a few entries.

The first kernel is the most important. It computes an array of *splits*, with one slot for each parent entry in each of the four child cells. Each thread inspects a single parent entry and generates one split for that entry in each child cell. These splits specify whether to include a parent abstract segment in the child cell and whether to add a shortcut segment. The kernel also computes two additional values per entry: a winding increment, if any, and an offset that counts the number of output entries generated by the parent split entry. For layout reasons in figure 4.9, these are shown as two independent arrays. In reality, we represent them as an array of pairs.

The purpose of the second kernel is to consolidate the winding increments into a single entry per path, per cell. Consolidation is necessary because the first kernel can produce multiple winding increments within each path in a child cell, and we need to prevent their uncontrolled proliferation. Consolidation is achieved with an inclusive segmented scan on the winding increment array. This scan adds together the winding increments that belong to the same path, leaving the result as the last entry for that path in the windings array.

The role of the third kernel is to compute the global offset for the output of each parent entry into each child cell. This is accomplished with exclusive scan on the windings/offset array. (Recall they are stored as pairs in a single array.) Besides adding the values in the offset entry, the operator used for the scan checks the corresponding value in the windings entry. If the entry is the last in a path, and if the windings entry is non-zero (as consolidated by the second kernel), the scan operator adds an extra 1 to the value to make room for a single winding increment for the path.

The fourth and final kernel inspects all split entries in parallel. Each thread loads the corresponding offset and writes the appropriate segments to their final positions in the child cell. If an entry is the last in a path, the kernel also inspects the windings array, and generates the appropriate winding increment if needed. The offsets are such that the order in which the paths appear in the child cells is the same as their order in the parent cell. In fact, even the order of entries within each path is preserved. Note that all kernels involved in the subdivision process are parallel at the segment level.

4.2.3 Pruning

After every subdivision step, the total number of entries in the child cells is likely to be greater than the original number of segments in the parent cell. After all, segments that cross the subdivision boundaries are replicated and may generate additional shortcut segments. Certain optimizations can be performed locally per segment and help attenuate this growth. For example, shortcut segments that cannot be intersected by any ray emanating from a cell need not even be generated. Conversely, shortcut segments that are intersected by *all* rays can be converted to the equivalent winding increment.

The most powerful strategies for keeping this growth in check, however, involve path interactions at the cell level. For example, when a path is opaque and covers the entire cell, all paths underneath it can be pruned. Clip-paths complicate the pruning algorithm, but also provide us with additional opportunities for optimization. For example, if a clip-path is empty when restricted to a cell, it can be pruned along with all paths under its influence.

Pruning is easiest to understand by means of *stream rewriting rules*. Each rewrite rule simplifies the stream while maintaining the following invariant: within the cell, the output stream is *valid and equivalent* to the input stream. Pruning is performed by the repeated application of rewrite rules, until no rule can be applied.

We introduce new terminals $\texttt{F}_0$ and $\texttt{C}_0$ to represent filled paths and clip tests, respectively, that fail all inside-outside tests for samples in the cell area. Conversely, $\texttt{F}_1$ and $\texttt{C}_1$ refer to paths that pass the inside-outside test for all samples in the cell area, and in addition the paint associated to $\texttt{F}_1$ is fully opaque. Since our shortcut tree is tight, such paths are easy to identify: simply apply the inside-outside test to the initial winding number of paths that contain no segments. Finally, non-terminals A, B, and C represent well-formed streams, while ϵ represents the empty stream. Given these definitions, the stream rewrite rules are as follows:

$$\texttt{F}_0 \rightarrow \epsilon \tag{4.6}$$

$$\texttt{C}_0 \rightarrow \epsilon \tag{4.7}$$

$$\texttt{(}\ A\ \texttt{|}\ B\ \texttt{F}_1\ C\ \texttt{)} \rightarrow \texttt{(}\ A\ \texttt{|}\ B\ \texttt{F}_1\ \texttt{)} \tag{4.8}$$

$$\texttt{(}\ \texttt{|}\ A\ \texttt{)} \rightarrow \epsilon \tag{4.9}$$

$$\texttt{(}\ A\ \texttt{|}\ \texttt{)} \rightarrow \epsilon \tag{4.10}$$

$$\texttt{(}\ A\ \texttt{C}_1\ B\ \texttt{|}\ C\ \texttt{)} \rightarrow C \tag{4.11}$$

$$\texttt{(}\ A\ \texttt{|}\ B\ \texttt{C}_1\ C\ \texttt{)} \rightarrow \texttt{(}\ A\ \texttt{|}\ \texttt{C}_1\ \texttt{)} \tag{4.12}$$

Two key properties have guided the selection of rewrite rules: (1) there is no reordering, only elimination of elements, and (2) the rules can be applied in parallel since they do not interfere with each other.

Rules (4.6) and (4.7) state that empty paths can be summarily eliminated from the stream. Rule (4.8) states that a fully opaque path covering the entire cell occludes all content that comes behind it at the same clipping nesting depth. Rule (4.9) states that a clip-path that always fails can be eliminated along with all content under its influence. Rule (4.10) states that a clip-path that has no content under its influence can also be eliminated. Rule (4.11) short-circuits the evaluation of a clip-path that always succeeds within the cell, leaving behind only the content that was under its influence. Since the inside/outside test of $\mathtt{C}_1$ always return inside, testing against $A\mathtt{C}_1 B$ will always return inside as well. The whole expression will evaluate as inside if, and only if, it evaluates as inside when tested against C.

Rule (4.12) is more subtle. It implements short-circuiting in the evaluation of a nested clip-path. When rules are evaluated one at a time, we could prune more aggressively:

$$\texttt{(}\ A\ \texttt{|}\ \ B\ \ \mathtt{C}_1\ C\ \texttt{)}\ \rightarrow A \tag{4.13}$$

Which is analogous to rule (4.11), but applied to the right side of the expression. However, in the next section we will parallelize the pruning and rule (4.11) will be applied simultaneously in a single step. The aggressive rule (4.13) combines with (4.11) to produce incorrect results:

$$\texttt{(}\ A\ \mathtt{C}_1\ B\ \texttt{|}\ \ C\ \mathtt{C}_1\ D\ \texttt{)}\ \overset{4.11,\ 4.13}{\longrightarrow} \epsilon \neq \mathtt{C}_1 \tag{4.14}$$

Rule (4.12) does not interact with rule (4.11), and results are correct:

$$\texttt{(}\ A\ \mathtt{C}_1\ B\ \texttt{|}\ \ C\ \mathtt{C}_1\ D\ \texttt{)}\ \overset{4.11,\ 4.12}{\longrightarrow} \mathtt{C}_1 \tag{4.15}$$

4.2.4 Parallel Pruning

Pruning is a challenging operation to perform efficiently and in parallel. The key is to split the computation into simple massively parallel tasks, and to ensure our invariant is preserved at each step. We proceed with multiple iterations of *mark-and-sweep*. During each iteration, we mark the elements in the stream that each rewrite rule wants to eliminate. We then sweep the marked elements away by compacting the stream. The mark-and-sweep process is repeated until no element can be eliminated. Naturally, the difficult part is marking the correct elements for elimination.

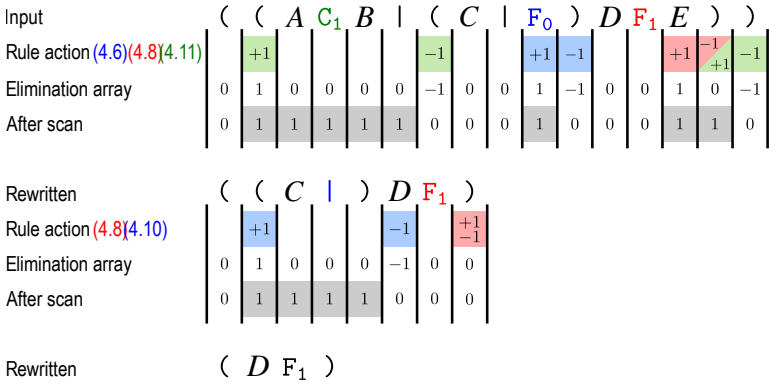


Figure 4.10: Parallel pruning example. The first pruning round engages three different rules. The segment entries that trigger action in each rule are colored. The second and final round engages only two rules. Elements selected for elimination are marked in gray.

Other than rules (4.6) and (4.7), all rules require the matching of delimiters `(`, `|`, and `)`. We start by cross-linking them so we can freely move from one to the other in constant time. To do so, we first obtain the clip nesting depth of each element. This is a simple matter of initializing to 0 a linear array with one element per segment entry. Values associated to `(` are then set to +1, and values immediately to the right of `)` are set to −1. An inclusive-scan of this array produces the required result. For a maximum nesting depth n, we then perform n segmented scans. Each segmented scan produces the links from all elements at depth d to their matching `(`. To do so, the initialization sets all entries to 1, and marks as boundaries all `(` elements at depth d. The segmented scan results in spans that start at 1 and progressively increase, but that restart for every `(` at depth d. In other words, all elements at depth d are associated to their distance to the matching `(`, which we convert to absolute pointers. To complete the process, we use this information to cross-link the matching `(`, `|`, and `)` at depth d between them. Recall we already could reach `(` from the matching `|` and `)`. After n segmented scans, all matching delimiters are properly cross-linked.

Given this information, we can finally describe the procedure that marks segment entries for elimination. Marking starts with the allocation of an *elimination array* with one element associated to each segment entry, all initially set to 0. Then, all segment entries are inspected in parallel, and each rule is given a chance to conditionally modify the elimination array by atomic increments or decrements to appropriates elements. When all rules have been executed, the elimination array is subjected to an inclusive scan. The segment entries to be eliminated are the ones associated to a *positive* element in the elimination array. Figure 4.10

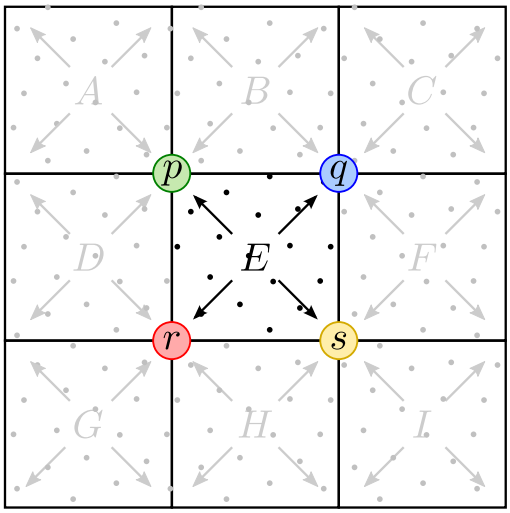


Figure 4.11: Integration with sample sharing for 2 × 2 anti-aliasing filters. Samples in each unit area are evaluated only once. Unit area E sends appropriately weighted sums to pixels p, q, r, and s. Pixel p receives contributions from unit areas A, B, D, and E.

shows an example that includes two rounds of pruning with the execution of multiple simultaneous rules. We describe rules (4.10) and (4.11). Other rules are analogous.

Rule (4.10) only requires action when inspecting a `|` segment entry. In that case, it checks the segment entry to its right, in search of a matching `)`. If it finds one, it atomically adds +1 to the element associated to its matching `(` in the elimination array, and atomically adds −1 to the element to the right of that associated to the matching `)`. After the scan, the effect is to add +1 to all elements between `(` and `)` (including the terminals themselves), thereby marking them for elimination as the rule dictates.

Rule (4.11) is a bit more involved. It acts unconditionally when inspecting a $\texttt{C}_1$ segment entry. It atomically adds +1 to the elimination array element associated to the `(` that is reachable from $\texttt{C}_1$. From the `(`, the matching `|` is also accessible. The rule atomically adds −1 to the elimination array element immediately past it. The rule can also reach `)`. There, it atomically adds +1, and adds −1 to element immediately past it. After the scan, only the region matched by the C is preserved. Note that this works even in the presence of multiple elements $\texttt{C}_1$ between the `|` and `)`. The only side effect is that certain elements that are marked for elimination may end up associated to numbers larger than 1 after the scan.

4.3 Rendering

In antialiased rendering, the color of each pixel is given by the convolution between the illustration and an antialiasing filter (section 1.2.1). To compute the color we use a Monte-Carlo estimator (section 1.2.2). Choosing box as the anti-aliasing kernel simplifies the computation since its support has one *unit area* in terms of the inter-pixel spacing. In that case, the integral can be computed independently for each pixel. In contrast, higher-quality filters can have support larger than 4×4 unit areas, where at least 16 filters overlap each sample in the illustration. Since computing sample colors dominates the cost of rendering, we cannot afford to recompute them so many times. The sampling and integration phases of our pipeline are based on the method presented in section 3.1: each sample is computed once and their values are shared among all pixels whose support cover them.

A key property of this method is that it uses a fixed amount of memory per pixel, regardless of the number of samples per unit area. Our pipeline supports a variety of different filters and sample distributions. The default high-quality setting uses a blue-noise pattern generated with the method of Balzer et al. [2009] with 32 samples per unit area, weighted by the 4×4 cubic B-spline. The combination is equivalent to $32 \times 16 = 512$ samples per pixel. The B-spline filters are then reshaped to cardinal cubic B-splines with a post-processing parallel recursive-filter [Nehab et al., 2011]. This explains the high quality of our results.

4.3.1 Sampling

When clip-paths are absent, the sampling algorithm is straightforward: it iterates over the shapes in front-to-back order blending the colors until a opaque color is computed. This is shown in section 1.1.7.

In the presence of clip-paths things get more complicated. Since general polygon clipping is out of the question due to its complexity, we decide to perform clipping operations per sample and with object precision. For that, we add clip-paths to the shortcut tree like any other path geometry, and maintain in each shortcut tree cell a stream that matches the scene grammar described in section 4.1. Clipping operations are performed per sample and with object precision.

When evaluating the color of each sample, the decision of whether or not to blend the paint of a filled path is based on a Boolean expression that involves the results of the inside-outside tests for the path and all currently active clip-paths. Since this expression can be arbitrarily nested, its evaluation seems to require one independent stack per sample (or recursion). This is undesirable in code that runs on GPUs.

Fortunately, as discussed in section 4.2.3, certain conditions (see the pruning rules) allow us to skip the evaluation of large parts of the scene. These conditions are closely related to the short-circuit evaluation of Boolean expressions. Once we include these optimizations, it becomes apparent that the value at the top of the stack is never referenced. The successive simplifications that come from this key observation lead to the *flat clipping* algorithm, which does not require a stack (or recursion).

Flat clipping The intuition is that, during a union operation, the first inside-outside test that succeeds allows the algorithm to skip all remaining tests at that nesting level. The same happens during an intersection when the first failed inside-outside test is found. Values on the stack can therefore be replaced by knowledge of whether or not we are currently skipping the tests, and where to stop skipping. The required context can be maintained with a finite-state machine.

The machine has three states: *processing* (P), *skipping* (S), and *skipping by activate* (SA). Inside-outside tests and color computations are only performed when the machine is in state P. The S and SA states are used to skip over entire swaths of elements in the stream.

In addition to the machine state, the algorithm maintains the sample color currently under computation and three state variables that control the short-circuit evaluation. The first two state variables keep track of the current clipping nesting depth d and the number u of nested clip-paths that have not yet been activated. These variables are updated when the machine comes across terminals `(` , `|` , and `)` :

$$\texttt{(} \;\Rightarrow\; d \leftarrow d+1,\; u \leftarrow u+1 \tag{4.16}$$

$$\texttt{|} \;\Rightarrow\; u \leftarrow u-1 \tag{4.17}$$

$$\texttt{)} \;\Rightarrow\; d \leftarrow d-1 \tag{4.18}$$

Skipping is interrupted when one of terminals `|` or `)` is found at a depth at least as shallow as the current *stopping depth* s. The stopping depth is set right before any transition to a skipping state, and is the third and last state variable needed by the algorithm.

Figure 4.12 shows the state transition diagram. Each transition is marked by an annotated arrow. Arrow annotations can have one or two rows. The first row specifies the conditions that trigger the transition. The first condition is the *triggering terminal*. Besides the clipping operators `|` and `)` , terminals $\texttt{f}_1$ and $\texttt{c}_1$ can also trigger transitions. These terminals denote, respectively, a filled path and a clip test for which the *current sample* has passed the inside-outside test. (This is in contrast to terminals $\texttt{F}_1$ and $\texttt{C}_1$ from section 4.2.3, which denoted paths

that pass inside-outside tests for *all samples* in the cell area.) After the triggering terminal, additional required conditions can be specified. The second row in arrow annotations is optionally used to update the stopping depth d.

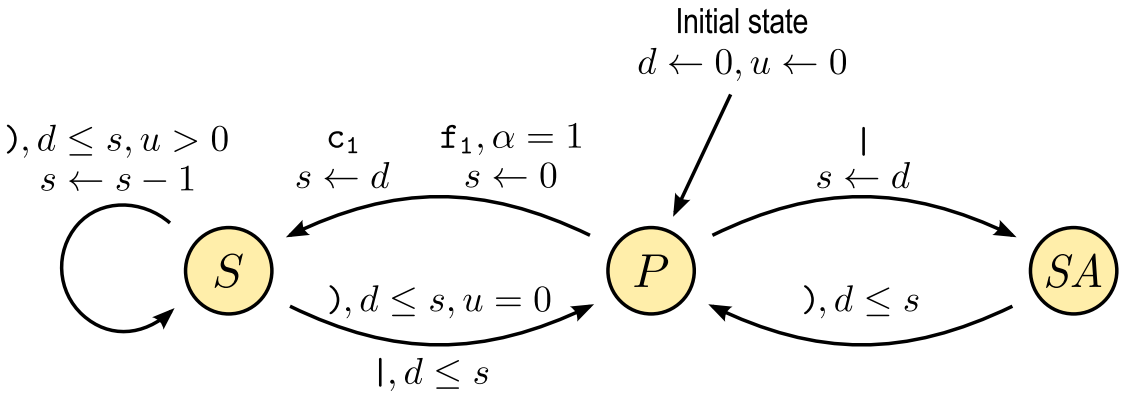


Figure 4.12: State transition diagram for the finite-state machine of the flat-clipping algorithm.

The machine starts in P with $d \leftarrow 0, u \leftarrow 0$. Consider the transitions between P and S. In the transition triggered by $\mathtt{f}_1$, the additional condition $\alpha = 1$ tests if the sample color is now opaque. In that case, since we render primitives front-to-back, there is no point in continuing. The machine transitions to S, and sets s to 0. Condition $\mathtt{c}_1$ means a clip test has succeeded. The remaining clip tests in the clip-path can therefore be skipped by short-circuit. The machine transitions to S and sets the stop depth to d. There are two transitions away from S. The first transition happens when an *activate* operation is found. Looking at the scene grammar, we see that this can only happen if the machine arrived at S due to a $\mathtt{c}_1$ transition from P. In other words, an entire clip-path test has succeeded, and therefore we transition unconditionally back to P. The second transition happens when a matching) is found. The condition $u = 0$ means the machine is not inside a nested clip-path test, so it simply transitions back to P. If the machine is skipping *inside* a nested clip-path test, one of the inner clip tests must have passed, and therefore the outer test can be short-circuited as well. The machine simply resets the stop depth to the outer level and continues in state S.

The remaining transitions are between P and SA. If the machine finds a | while in state P, it must have been performing a clip-path test that failed. Otherwise, it would have been in state S. Since the test failed, it can skip until the matching) . This is what motivates the name *skipping by activate*.

4.3.2 Scheduling

The pipeline allows a user to specify a 3×3 projective transformation to be applied to the sample coordinates. Experienced users can design arbitrary warping functions in CUDA.[1] Since the pipeline remaps individual samples, and not the rendered image, results are exactly the same as if the illustration had been warped in object space by the inverse of the warp function, and only then rendered.

With the integration and sampling algorithms in place, we can complete the rendering algorithm. The sample positions from each unit area must be warped by the user-supplied function and pushed down the shortcut tree until the appropriate leaf cell is found. With the cell contents and warped sample positions, the sampling algorithm computes their colors. These colors are weighted, added together, and routed to all the appropriate pixels by the integration algorithm.

The scheduler plays two key roles: it minimizes the global memory bandwidth requirements by allowing cell contents to be loaded once and reused by multiple unit areas, and it minimizes control-flow divergence by grouping together samples that fall in the same cell. To do so, we use three computational kernels.

The first kernel goes over each unit area and identifies the set of leaf cells that contain at least one of the warped sample positions. This information is obtained by descending with each warped sample position down the shortcut tree until a leaf cell is found. The resulting list of cell IDs is compressed within shared memory to eliminate repetitions. A list with pairs containing the originating unit area ID and the required cell ID is stored into global memory.

The role of the second kernel is to transpose the results of the first kernel, which come naturally sorted by unit area ID, so that they are instead sorted by cell ID. This is accomplished with a simple parallel sort.

The third and last kernel performs the actual rendering. Each computational block is responsible for a batch of U unit areas from the list produced by the second kernel. The different unit areas in each batch send at least one warped sample position to the same given cell. There is enough shared memory to load I input segments. The context for the S samples that will be evaluated simultaneously is stored in the registers of independent threads. While there are unit areas to be processed, the algorithm warps their samples and eliminates those that fall outside the cell. This process is repeated until S samples are found (potentially originating from distinct unit areas). Then, it loops over the cell stream, loading chunks of I segments to shared memory. For each chunk, it advances the sampling algorithm

[1] This feature currently requires recompiling the scheduler. Changing the API to support warps defined at runtime is a simple if tedious task.

in parallel for the S samples over these I input segments. When the entire cell contents have been processed, the algorithm computes independent weighted sums for the samples originating from each unit area, and atomically adds them to the appropriate pixels. It then goes back for more unit areas in the batch until they have all been processed. We use $S = 128$, $I = 32$, and $U = 32$ in all our tests. With this setup, we are able to process 4 unit areas in the same cell, with 32 samples each, without reloading the input.

A specialized version of the scheduler handles the common case when there is no user-defined warp. We align the shortcut tree cell boundaries with the unit areas, so that all samples originating from a given unit area fall within the same cell. This greatly simplifies the generation of the list of unit areas per cell: it suffices to descend on the shortcut tree with the unit area center. It also simplifies the integration step: there is no need to keep track of which samples belong to each unit area since no sample is ever eliminated.

Chapter 5

Results

Comparisons between our work and other vector graphics renderers is done in two different axis: quality of the rendering and performance.

5.1 Quality Comparison

5.1.1 Conflation

Figure 5.1 shows renderings of a contour plot (exported as SVG by the Mathematica software), in which areas of constant color are unions of precisely abutting triangles. When independently rendered triangles are blended together, as many renderers do (e.g., Cairo, Adobe Reader, Apple's Quartz, etc.), the correlated mattes lead to incorrect results and the underlying mesh appears. Our pipeline renders these areas as intended.

5.1.2 Integration in Linear RGB

Another common problem is with renderers that evaluate the anti-aliasing integral (1.36) in gamma space. This leads to dark regions that look wider than intended. Our renderer can transform colors to linear space before integration and reapply gamma in the end. The difference is obvious on the right of figure 5.2, which renders text with the correct weight. Unfortunately, many users have grown accustomed to incorrect rendering. As a compromise, we support both alternatives.

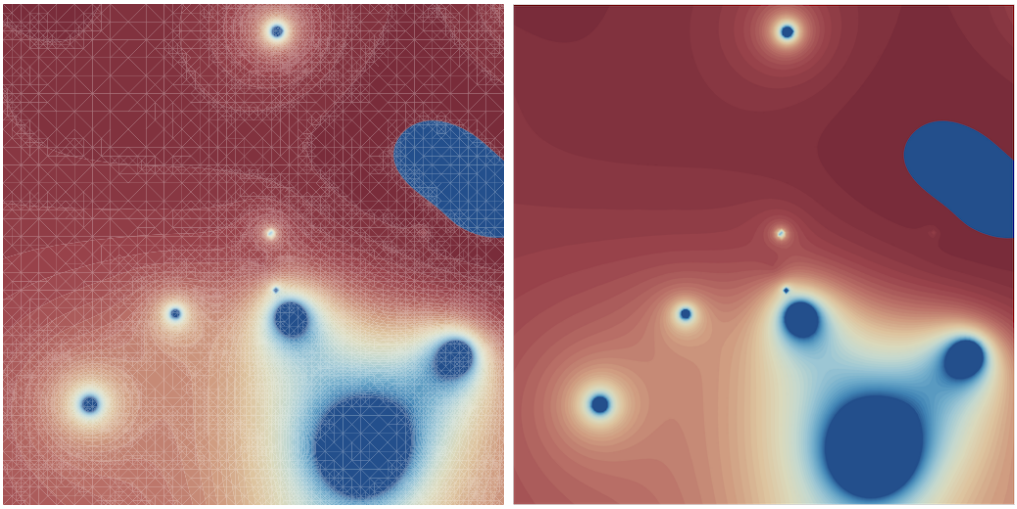


Figure 5.1: (Left) Artifacts appear when polygons that share an edge are independently resolved to pixels before blending. (Right) Our renderer blends colors independently per sample and resolve later.

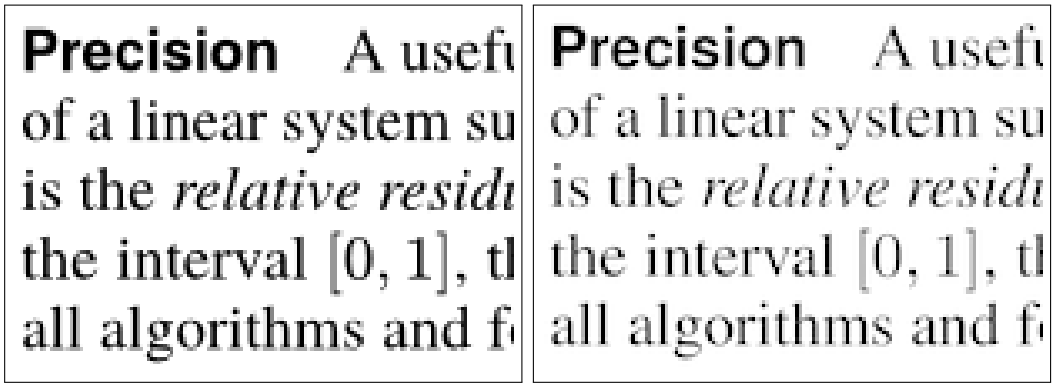


Figure 5.2: (Left) Integration in gamma space incorrectly widens dark regions and produces heavier text. (Right) Our renderer integrates in linear space to produce text with the intended weight.

5.1.3 Anti-aliasing Quality

Figure 5.3 shows an aliasing-prone resolution chart rendered with different antialiasing strategies. Nehab and Hoppe [2008] (NH) employ a very efficient 1D prefiltering approximation. Although results are very good in certain areas, aliasing and conflation artifacts are clearly visible in others. Modes 1×8 and 1×32 are box-filtered with respectively 8 and 32 samples per unit area. As shown in figure 5.3, mode 1×8 shows significant amounts of noise. It is included in our tests simply because it is the limit of what Kilgard and Bolz [2012] (NVPR) can accomplish in a single pass using multisampling in current hardware ($8\times$MS). Although the hardware supports a hybrid single-pass mode $32\times$ ($8\times$MS, $24\times$CS), it is too crude an approximation for 1×32. NVPR's demo offers a much better approximation by accumulating 4 passes with $8\times$MS. The amount of aliasing is a property of the box filter and remains the same regardless of the number of samples. Our $4\times4\times32$ mode uses a cardinal cubic B-spline with 512 samples under the 4×4 support of each pixel's filter, sharing samples across overlapping filters. The results are visibly reduced aliasing in challenging areas and renderings that are virtually free of noise.

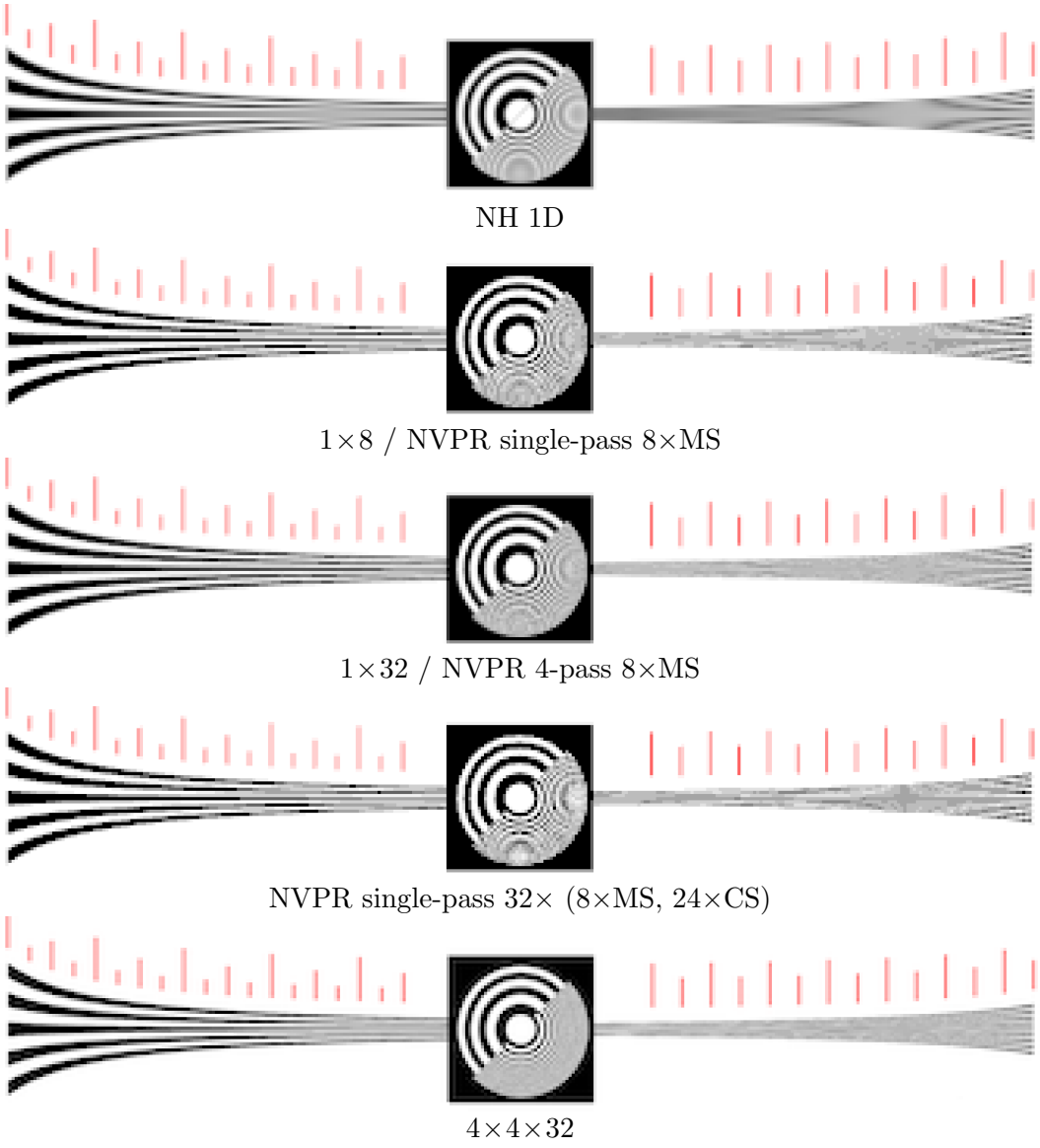


Figure 5.3: An aliasing-prone resolution chart rendered by NH's 1D mode is free of noise but shows aliasing and conflation. Mode 1×8 *shows both noise and aliasing. As expected, mode* 1×32 *reduces noise, but aliasing persists. NVPR's single pass mode* $32\times$ *(*$8\times$*MS,* $24\times$*CS) is too crude an approximation. The sharper antialiasing filter we use in mode* $4\times4\times32$ *is made possible by sample sharing.*

5.2 Performance

Figure 5.4: Different levels of detail can be obtained for the same region using OpenStreetMaps. (Left) Paris 50k. (Right) Paris 70k.

Table 5.1 shows statistics for some of the illustrations we used in our tests. These illustrations are available in the supplemental materials[1]. Drops, Car, and Embrace make heavy use of semitransparent gradient fills, but are otherwise simple illustrations. Reschart is the alias-prone resolution chart that contains the pattern of figure 5.3. Contour appears in figure 5.1, and is a dense triangulation with flat-colored triangles. Tiger is the standard PostScript sample and contains many opaque overlapping paths that simulate gradients. Paper 1 and Paper 2 are SIGGRAPH paper pages using Type 1 and TrueType fonts, respectively (i.e., cubics vs. quadratics). Boston, Paris, and Hawaii are maps. Hawaii appears in figure 5.5 and includes many overlapping semi-transparent layers. We tested maps of Paris at different scales (from OpenStreetMaps), spanning a large variation in complexity. These maps include finely dashed strokes with decorations that significantly increase the rendering complexity beyond the number of input segments. See figure 5.4.

Table 5.1 also shows a performance comparison between our work and those of Kilgard and Bolz [2012] (NVPR) and of Nehab and Hoppe [2008] (NH)[2]. Rendering times do not include preprocessing time, although we also provide preprocessing times and memory consumption for our method. Our tests were run on an NVIDIA GeForce GTX Titan (2688 CUDA cores, 6GB of global memory) hosted by an Intel

[1] `http://w3.impa.br/~diego/projects/GanEtAl14/`

[2] Nehab and Hoppe [2008] only include a subset of the inputs we tested.

Core i7 980 at 3.33GHz with 24GB of system memory. When comparing against competing algorithms, we used the original published implementation and demo programs, running on the same hardware.

Table 5.1: Description of tests and performance comparison. Images were rendered with 1024 width and proportional height, except for the Paris dataset: 70k-470×453, 50k-657×635, and 30k-1096×1060. Number of filled and stroked paths are given, with segments broken into each type. All times are expressed in milliseconds (smaller are better). "Pre." denotes the preprocessing time, "Mem." is memory usage in MiB (i.e., mebibyte, *or* 2^{20} *bytes). All rendering modes (*1×8, 1×32 *etc.) are explained in the "antialiasing quality" discussion.*

Input	Res. 1024×	Filled paths	Fill segments			Stroked paths	Stroke segments			Total segments	Our method					NVPR		NH	
			L	Q	C		L	Q	C		Pre.	Mem.	1×8	1×32	4×4×32	1×8	1×32	1D	1×32
Car	682	361	701	165	3187	59	32	12	183	4280	28.45	8.68	12.86	14.73	28.85	3.42	10.56		
Drops	1143	204	45		1359					1404	21.77	2.83	14.28	18.59	46.03	2.63	5.11	0.91	33.61
Embrace	1096	225	25		4621					4646	24.06	4.14	15.50	19.38	48.07	2.78	5.08	0.88	31.18
Reschart	625	723	7823		96	24	140			8059	24.96	3.10	8.51	11.14	32.34	2.88	10.84	0.58	19.33
Tiger	625	236	177		1988	66	16		346	2527	31.04	4.12	12.89	17.24	52.70	2.66	5.50	0.82	34.15
Boston	917	122	1818		13669	1800	137		12470	28094	128.02	46.92	37.22	41.81	71.14	8.28	31.02	2.45	66.77
Hawaii	844	1008	6312	6	43208	131	8	6	2129	51669	115.29	42.15	26.16	29.48	50.50	3.68	14.70		
Paper 1	1325	5099	39573		59708	9	23			99304	53.17	13.22	19.28	23.71	**67.65**	20.80	78.64		
Paper 2	1325	5621	42620	85216	26096	68	111		39	154082	72.51	13.86	10.80	17.50	**35.09**	24.95	95.72		
Contour	1024	53241	188340							188340	77.58	41.29	30.07	30.36	**63.21**	203.93	1025.11		
Paris 70k		32454	2983		1999	13136	987		1487	7456	101.85	45.96	22.39	21.00	**28.45**	151.67	796.34		
Paris 50k		32639	2443		4149	13157	759		1679	9030	110.12	51.03	26.82	25.22	**34.96**	155.88	737.20		
Paris 30k		34751	7186		22155	15939	5437		18109	52887	192.94	94.72	49.51	48.81	**78.59**	176.54	904.17		

Let us focus on the 1×32 rendering mode. For each input, the times for the fastest method are shown in blue, and the others in red. The key comparison is between our method and NVPR. This is because NH was optimized for the 1D mode, where it excels. In 1×32 mode, it performs its own supersampling instead of taking advantage of hardware-accelerated multisampling.

Results show that, once input complexities are sufficiently high, our pipeline has the advantage. We believe that the main reason for this behavior is that NVPR renders each individual path one after the other. Even though the hardware rasterizer can process paths much faster than our pipeline, as the number of paths increases, this sequential processing becomes a bottleneck. Our performance advantage can already be noticed while rendering a typical page of text, with subpixel positioning of characters, at 32 samples per pixel and 100 pixels per inch. Grouping shapes with the same paint into a single path would significantly improve NVPR's performance. Unfortunately, this could result in incorrect rendering where such shapes overlap spatially. An optimization along these lines could be implemented at the application level, at least for simple and common cases such as pages of text, where it would be very effective.

Our improvements are even more pronounced for inputs of higher complexity. In fact, due to sample sharing, we can render both faster *and* at a higher quality level. Such results are marked in bold in table 5.1. We would like to stress that this is not the result of extensive optimization. It is the result of new algorithms that map better to massively-parallel hardware.

User-defined warps Figure 5.5 shows three user-defined warps: a twisting warp on the Tiger, a zoom-lens effect on Paper 1, and a projective transformation on a map of Hawaii. The pipeline renders these effects as if the illustration had been warped in object space. The scheduler ensures samples are shared between all pixels with overlapping antialiasing filters while minimizing control-flow divergence as well as memory and bandwidth requirements.

Scalability to output resolution Figure 5.6 shows the behavior of our rendering stage as the number of output pixels is increased progressively from 256×256 to 2048×2048. For each sample, image dimensions were selected to maintain the original aspect ratio while matching the specified number of output pixels. Results show that the rendering algorithm scales close to linearly with image resolution. Small deviations are due to the different shortcut tree structures that result for different target resolutions.

Figure 5.5: Examples of user-defined object-space warps.

Pruning and clipping Although typical illustrations do not include deeply nested clip-paths, the pipeline supports them as specified by the standards. Figure 5.8 shows one of our test cases. Clipping (or equivalently, occlusion) is the main justification for the pruning of the shortcut tree. Enabling pruning in the scene shown in the figure leads to a 25% reduction in memory consumption and 45% improvement in rasterization time. The total time reduction for preprocessing and rasterization is 10%.

Front-to-back rendering Many vector graphics renderers draw shapes back-to-front. This is inefficient when there is substantial overdraw. We address this problem in two ways. First, we proceed front-to-back when rendering. As soon as a sample becomes opaque, the remaining scene content can be safely ignored. In scenes with high depth complexity, this optimization can significantly improve

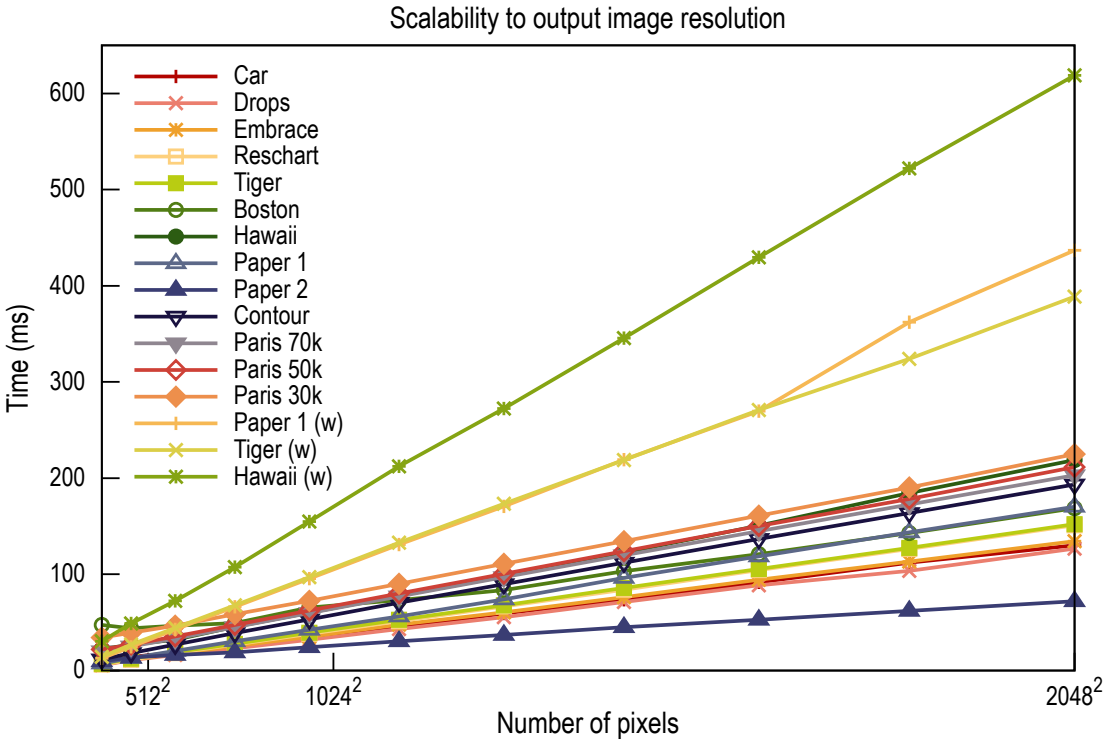


Figure 5.6: Rendering times are linear on the number of output pixels. Small variations are due to the different shortcut trees used at different target resolutions.

rendering performance (e.g., by 33% in the Paris 30k input). Second, the shortcut tree pruning algorithm eliminates from the stream all paths that would have been completely occluded by an opaque path within a cell. Pruning does not take into account the possibility of multiple semi-transparent paths combining into an opaque layer and obscuring the paths underneath. This would be difficult to accomplish, especially in the presence of gradient fills and textures.

Subpixel positioning of text Our renderer treats character glyphs as regular paths, in object precision. Many renderers pre-render glyphs in image precision. This is especially noticeable when scrolling or resizing text, which causes pre-rendered glyphs to move horizontally or vertically relative to one another. See the animations of Paper 1 available in the supplemental materials[3].

Shortcut tree behavior Table 5.2 shows the behavior of shortcut trees for increasing levels of subdivision. The examples were selected to span the range of

[3] `http://w3.impa.br/~diego/projects/GanEtAl14/`

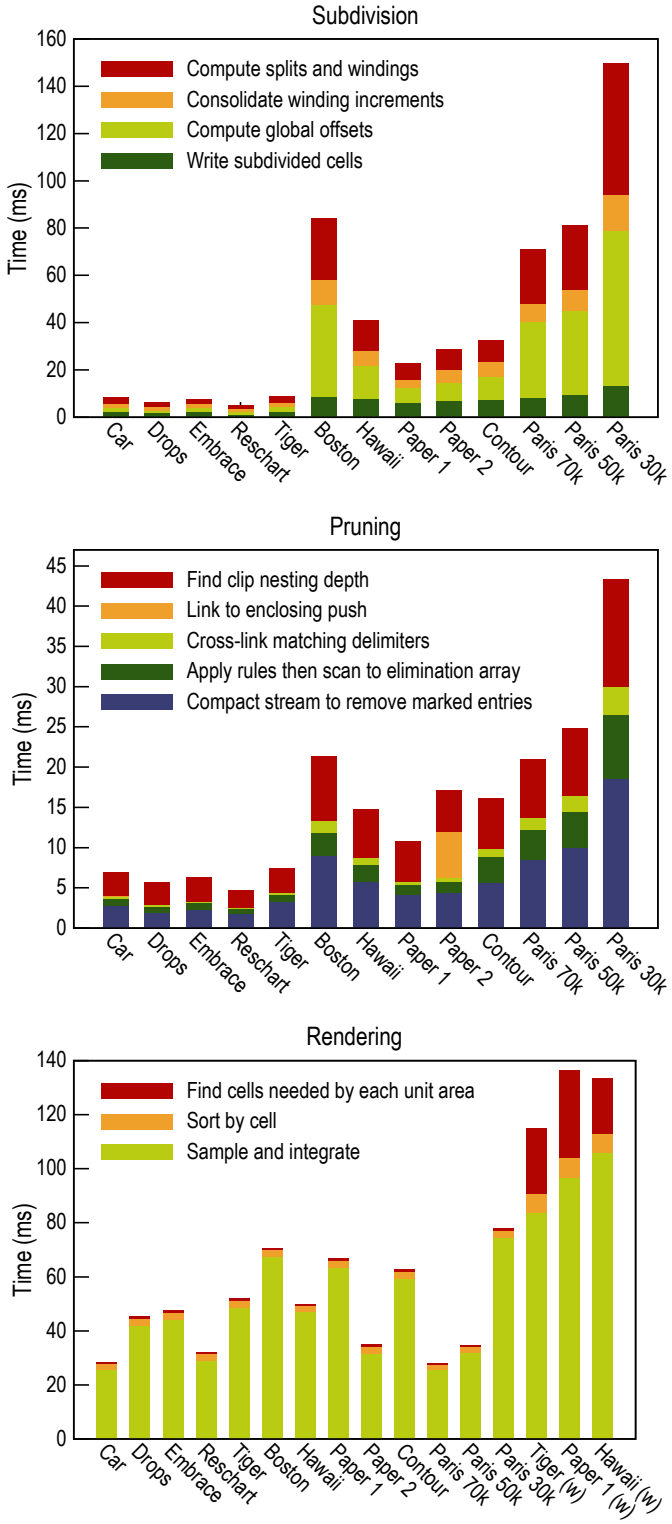
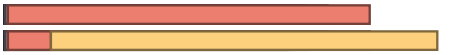
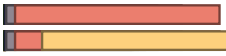


Figure 5.7: Relative costs of major steps in the rendering pipeline.

behaviors observed in practice. Most content in Paper 1 consists of relatively small characters. Once subdivision is deep enough to isolate them, it stops. The high density of detail in Paris 30k, which also includes significant overdraw, forces tree subdivision to proceed further. In general, we do not observe an explosion in the number of segments shared by different cells.

Table 5.2: Behavior of the shortcut tree subdivision. Each horizontal bar represents the number of segments after each subdivision level. The visualization shows there is no explosion in the number of segments. Each bar is further divided to represent the number of segments at each tree depth. Bar sizes are normalized by the highest subdivision level in each input and are not comparable across inputs.

	Segment distribution by tree depth	
Level	Paper 1	Paris 30k
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

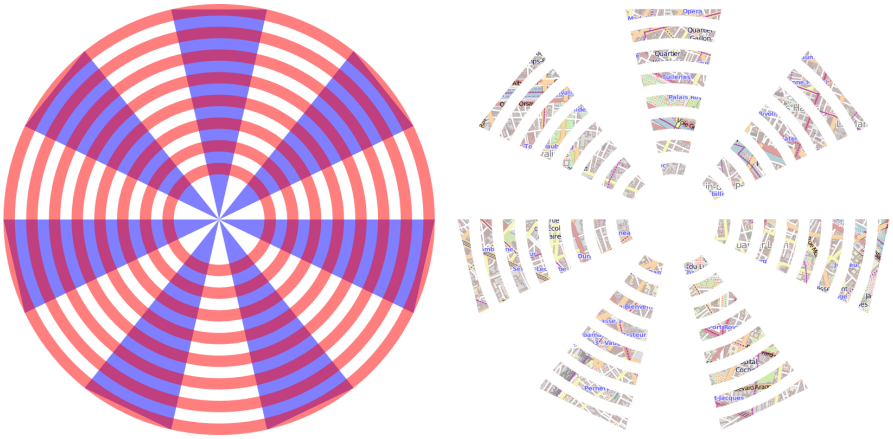


Figure 5.8: Nested clip-paths. A shape defined as the intersection a set of concentric rings (in red) and a set of triangles (in blue) is used to clip the map of Paris. Pruning speeds up the process.

Conclusion and Future Work

Rendering vector graphics is not an easy task. High-quality rendering requires huge amounts of processing power. To achieve real-time performance, many implementations compromise on quality. The results, as we have shown, suffers from many different problems. Low sampling rate and kernels with small support lead to noise and aliasing. Conflating pixel coverage with color transparency leads to rendering artifacts.

Beyond that, all prior implementations have at least one sequential component. Methods using retained mode preprocess the scene sequentially to create an acceleration data structure. In immediate mode renderers, shapes must be processed sequentially to avoid interference between different paths.

We have described a complete and fully parallel pipeline for rendering vector graphics on the GPU. We build an acceleration data structure—parallel at segment level—that allow us to render images at high sample rate—parallel at sample level—with kernels of large support.

Our work opens the door for a variety of interesting follow-up research and engineering problems.

Shortcut tree auto-tunning We would like to investigate the possibility of creating methods that control the subdivision of cells, using statistics collected from the scene, and heuristics driven by prior knowledge. We would like to train classifiers to test the necessity of subdividing a node.

Bootstrapping the tree The shortcut tree is constructed from the root node in a breadth-first order. This process could benefit of a prior stage in which the root node is subdivided into a coarse regular grid. The fast lattice-clipping of Nehab and Hoppe [2008] could be translated to the GPU using parallel primitives.

Rational cubic segments In order to make the entire pipeline closed under projective transformations, we would have to support rational cubic segments. Projective transformations such as perspective mappings may convert integral cubic Bézier segments into rational cubic Bézier segments. Although the implicitization of rational cubics is not a problem, the monotonization of these segments is. Monotonizing a rational cubic requires solving for the roots of polynomials of degree 4. There are analytic solutions to this problem, but numerical inaccuracies when solving for the roots may lead to incorrect results. We believe that iterative methods along with methods to isolate roots are necessary.

Stroked paths Stroked paths are converted into filled paths in the CPU, in a step prior to rendering. This conversion involves approximating curved segments by line segments. This is followed by a computation of the length of the newly created segments along the entire path—in order to compute dashes. This processes are sequential but can be reformulated in a more parallel fashion, and then mapped into the GPU. This reformulation would allow us to incorporate the conversion into the pipeline.

Filter effects Filter effects are graphical operations that can be applied to one or more shapes, such as blur or lighting effects. These effects may require computations in image space. Therefore, to fully support filter effects, our pipeline would have to be able to render some shapes in the scene into a different buffer, apply the effects, and then blend the result with the other shapes in the scene. This would require support for multiple shortcut trees and some modifications on the sampling algorithm.

Subpixel rendering Subpixel rendering (e.g., [Betrisey et al., 2000]) uses the geometry of the elements that form each pixel in a computer display to increase the density of points available when rendering text. This makes it possible to exhibit text with better legibility at the cost of some color distortion. To support subpixel rendering in our pipeline we would have to align the shortcut tree to the geometry of the subpixels or use cells that overlap.

Mesh-based gradients Mesh-based gradients define another way of filling paths with procedural colors. PDF [2006] defines gradients that are created over a triangulation on the plane, where each vertex has a defined color. Points inside the triangulation have their colors defined by the barycentric interpolation of the colors of the vertexes. SVG [2011] specification defines mesh gradients that are

based on an array of Coons patches. A Coons patch is defined by colors placed at the corners of an area enclosed by four Bézier curves. Supporting mesh-based gradients requires the encoding of the gradient into the scene. Sampling the mesh efficiently is also necessary.

CPU implementation We are particularly interested in investigating the implementation of our pipeline on the CPU. On the one hand, it would seem overkill to maintain the preprocessing stage parallel at the segment level and the rendering parallel at the sample and pixel levels. A more coarse division of work between fewer threads should be more appropriate. On the other hand, most of the effort we have invested in minimizing control-flow divergence on the GPU should also be effective when used with the vectorized instructions available in modern CPUs.

Bibliography

M. Balzer, T. Schlomer, and O. Deussen. Capacity-constrained point distributions: A variant of Lloyd's method. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2009)*, 28(3):86, 2009. 86

C. Betrisey, J. F Blinn, B. Dresevic, B. Hill, G. Hitchcock, B. Keely, D. P. Mitchell, J. C. Platt, and T. Whitted. Displaced filtering for patterned displays. *Society for Information Display Symposium Digest of Technical Papers*, 31(1):296–299, 2000. 106

J. F. Blinn. How to solve a quadratic equation. *IEEE Computer Graphics and Applications*, 25(6):76–79, 2005. 24

E. Dahlström, P. Dengler, A. Grasso, C. Lilley, C. McCormack, D. Schepers, and J. Watt. Scalable vector graphics (svg) 1.1. *W3C*, 2011. URL `http://www.w3.org/TR/SVG/`. 11

M. A. Z. Dippé and E. H. Wold. Antialiasing through stochastic sampling. *ACM Siggraph Computer Graphics*, 19(3):69–78, 1985. 39

G. Farin, J. Hoschek, and S. Kim. *Handbook of Computer Aided Geometric Design*. Elsevier Science, 2002. URL `http://books.google.com.br/books?id=GGXcUS5p2CIC`. 21

J.D. Foley. *Computer Graphics: Principles and Practice*. Addison-Wesley, 1996. 47

S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of ACM SIGGRAPH 2000*, pages 249–254, 2000. 52

F. Ganacim, R. S. Lima, L. H. de Figueiredo, and D. Nehab. Massively-parallel vector graphics. *ACM Transactions on Graphics (Proceedings of the ACM SIGGRAPH Asia 2014)*, 36(6):229, 2014. 10

A.S. Glassner. *Principles of Digital Image Synthesis*. The Morgan Kaufmann Series in Computer Graphics. Elsevier Science, 2014. URL `https://books.google.com.br/books?id=C2riBQAAQBAJ`. 38

K. Kerr. Introducing Direct2D. MSDN Magazine, June 2009. 47

M. Kilgard. A simple OpenGL-based API for texture mapped text. Silicon Graphics, 1997. 52

M. J. Kilgard and J. Bolz. GPU-accelerated path rendering. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2012)*, 31(6):172, 2012. 47, 49, 51, 52, 69, 93, 95

D.B. Kirk and W.W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science, 2012. ISBN 9780123914187. URL `https://books.google.com.br/books?id=E0Uaag8qicUC`. 81

S. Lefebvre and H. Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2006)*, 25(3):579–588, 2006. 52

C. Loop and J. F. Blinn. Resolution independent curve rendering using programmable graphics hardware. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2005)*, 24(3):1000–1009, 2005. 47, 49, 50, 51, 67, 72, 73, 74

D. Nehab and H. Hoppe. Random-access rendering of general vector graphics. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2008)*, 27(5):135, 2008. 52, 53, 67, 69, 70, 76, 77, 93, 95, 105

D. Nehab and H. Hoppe. A fresh look at generalized sampling. *Foundations and Trends in Computer Graphics and Vision*, 8(1):1–84, 2014. 36

D. Nehab, A. Maximo, R. S. Lima, and H. Hoppe. GPU-efficient recursive filtering and summed-area tables. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2011)*, 30(6):176, 2011. 86

A. Orzan, A. Bousseau, H. Winnemöller, P. Barla, J. Thollot, and D. Salesin. Diffusion curves: A vector representation for smooth-shaded images. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2008)*, 27(3):92, 2008. 9

K. Packard and C. Worth. A realistic 2d drawing system. 2003. 9, 47

E. Parilov and D. Zorin. Real-time rendering of textures with feature curves. *ACM Transactions on Graphics*, 27(1):3, 2008. 52

PDF. *Adobe Portable Document Format, v. 1.7*. Adobe Systems Incorporated, sixth edition, 2006. 106

T. Porter and T. Duff. Compositing digital images. *Computer Graphics (Proceedings of ACM SIGGRAPH 1984)*, 18(3):253–259, 1984. 31, 48

Z. Qin, M. McCool, and C. Kaplan. Real-time texture-mapped vector glyphs. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, pages 125–132, 2006. 52

Z. Qin, M. McCool, and C. Kaplan. Precise vector textures for real-time 3D rendering. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, pages 199–206, 2008. 52

G. Ramanarayanan, K. Bala, and B. Walter. Feature-based textures. In *15^{th} Eurographics Symposium on Rendering*, pages 265–274, 2004. 52

L. Ramshaw. Béziers and B-splines as multiaffine maps. In *Theoretical Foundations of Computer Graphics and CAD*, volume 40 of *NATO ASI Series*, pages 757–776. Springer Berlin Heidelberg, 1988. 24

N. Ray, X. Cavin, and B. Lévy. Vector texture maps on the GPU. Technical Report ALICE-TR-05-003, INRIA, 2005. 52

Daniel Rice and RJ Simpson. Openvg specification, version 1.1. *Khronos Group*, 2008. 11

N. P. Rougier. Higher quality 2D text rendering. *Journal of Computer Graphics Techniques*, 2(1):50–64, 2013. 52

G. Salmon. *A Treatise on the Higher Order Plane Curves*. Hodges & Smith, 1852. 50

P. Sen. Silhouette maps for improved texture magnification. In *Graphics Hardware*, pages 65–73, 2004. 52

P. Sen, M. Cammarano, and P. Hanrahan. Shadow silhouette maps. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 22(3):521–526, 2003. 52

Skia. Skia website, 2015. URL `http://skia.org/`. 9

Ivan E Sutherland and Gary W Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974. 59

SVG. *Scalable Vector Graphics, v. 1.1.* W3C, second edition, 2011. 106

B. A. Wallace. Merging and transformation of raster images for cartoon animation. *Computer Graphics (Proceedings of ACM SIGGRAPH 1981)*, 15(3):253–262, 1981. 30

J. Warnock. *A hidden surface algorithm for computer generated halftone pictures.* PhD thesis, University of Utah, 1969. 58

J. Warnock and D. K. Wyatt. A device independent graphics imaging model for use with raster devices. *Computer Graphics (Proceedings of ACM SIGGRAPH 1982)*, 16(3):313–319, 1982. 9, 13

C. Wylie, D. Romney, G. Evans, and A. Erdahl. Half-tone perspective drawings by computer. In *Proceedings Fall Joint Computer Conference*, pages 49–58, 1967. 47

G. Wyszecki and W. S. Stiles. *Color science*, volume 8. Wiley New York, 1982. 26

Thomas Young. The bakerian lecture: On the theory of light and colours. *Philosophical transactions of the Royal Society of London*, pages 12–48, 1802. 26