Masaryk University

Faculty of Informatics

A Manipulation Library for Scalable Vector Graphics

Bachelor's Thesis

Tomáš Režňák

Brno, May 2025

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

In the course of preparing this thesis, I utilized the following tools based on Artificial Intelligence (AI):

- Grammarly and Writefull to enhance my writing style,
- Github Copilot to improve my programming efficiency.

I declare that I used these tools under the principles of academic integrity. I have reviewed their output and assume full responsibility for it.

Tomáš Režňák

Advisor: Mgr. Marek Kadlčík

Acknowledgments

I would like to express my sincere gratitude to my advisor, Mgr. Marek Kadlčík, for his excellent guidance, invaluable feedback, and, perhaps most importantly, his extraordinary patience throughout the course of my work on this thesis. In addition, I would like to thank those among my friends and family who supported, encouraged, and tolerated me during my studies.

Abstract

In this thesis, we present SVGLAB, an open-source Python toolkit for the programmatic manipulation of vector images in the Scalable Vector Graphics (SVG) format. The library provides robust functionality for parsing, modifying, and writing SVG documents, offering comprehensive support for features defined by the SVG specification. Next, we conduct a survey of existing software in this domain and demonstrate that our implementation matches or exceeds the capabilities found in even the most feature-rich libraries within the Python ecosystem while addressing some of the common limitations found therein. Finally, we suggest several possible directions for future work in this area.

Keywords

SVG, SVG manipulation, image processing, vector graphics, affine transformations, parsing, parser, computer graphics, XML

Contents

1 Introduction **1**

2 Preliminaries **3**
2.1 Extensible Markup Language (XML) 3
2.2 Scalable Vector Graphics (SVG) 5
2.3 Bounding boxes and segmentation masks 13

3 Requirements **16**
3.1 Programming language 16
3.2 Representation and I/O 16
3.3 Robust typing 16
3.4 Configurable formatting 17
3.5 Compliance with SVG specifications 17
3.6 Lossless parsing and serialization 17
3.7 Manipulation 18
3.8 Reification 18
3.9 Coordinate system rescaling 18
3.10 Bounding boxes 18
3.11 Masks 18

4 Implementation **19**
4.1 Overview 19
4.2 Representation 20
4.3 Elements 29
4.4 Parsing 29
4.5 Serialization and output formatting 31
4.6 Reification 33
4.7 Rescaling the coordinate system 38
4.8 Computing an element's bounding box and mask 39

5 Evaluation **42**
5.1 Testing 42
5.2 Comparison with existing software 42

6 Future work **46**
6.1 CSS support 46
6.2 Optimization support 46

7 Conclusion **48**

A Thesis archive contents **49**

B Installation manual **50**
B.1 Prerequisites 50
B.2 Installation 50

C Proofs and derivations **51**

References **53**

List of Figures

2.1 The difference between stroking and filling in SVG. 7
2.2 A comparison between closing a path using the closepath and lineto commands. Notice the difference in the stroke near point B. 10
2.3 A path defined by the path data `M0,0 H10 Q 15,7.5 10,10 Z`. 11
2.4 A 2x2 square inside a 5x5 image (left) and its segmentation mask (right). 14
2.5 An image with two overlapping shapes (left), the square's mask (middle), and the square's visible mask (right). 14
2.6 A composite Bézier curve (solid blue) and its bounding box (dashed black). 15

4.1 An example SVG and its SVGLAB representation. 21
4.2 The entity class hierarchy. 22
4.3 The composition of attribute mixin classes into an element class. 28
4.4 The parsing and serialization processes. 30
4.5 The effect of non-uniform scaling on a shape's stroke. 34

List of Tables

2.1 An overview of the fundamental attribute types in SVG. 6
2.2 Path commands in SVG. 9

4.1 Example element and attribute names and their normalized variants. 22
4.2 Representation of SVG transformations in SVGLAB. 25
4.3 Path commands and their representations in SVGLAB. 26
4.4 The possibility of reification among different types of elements and transformations. 35

5.1 An overview of Python SVG manipulation libraries. 43
5.2 A comparison between SVGELEMENTS and SVGLAB. 44

Chapter 1

Introduction

Scalable Vector Graphics (SVG) has become the de facto standard for vector graphics on the Web and beyond, providing a convenient way to create images that maintain their quality at any size and are often smaller in file size than their raster counterparts.

As with any other format, the need naturally arises to manipulate SVG documents programmatically, enabling the use of vector graphics in fields such as computer vision and data visualization.

However, despite its widespread use, the Python ecosystem lacks a comprehensive and well-maintained library to manipulate SVG documents. In Section 5.2, we provide a brief overview of existing software and show that only one of the surveyed libraries offers the functionality required to manipulate SVG documents.

Our contribution is threefold:

1. we analyze existing implementations and identify their shortcomings,
2. we design several novel approaches to solve these shortcomings, and, finally,
3. we present a new Python library for manipulating SVG documents, designed to focus on usability, correctness, and customizability.

This paper assumes no prior experience with SVG. However, we presume that the reader has elementary knowledge of linear algebra, as well as the Python programming language and its ecosystem.

The thesis is structured as follows. Chapter 2 covers the preliminaries, including a brief introduction to SVG and relevant concepts from computer graphics and computer vision. Chapter 3 provides an overview of the requirements for our library. In Chapter 4, we describe the implementation of the library and its

architecture. In Chapter 5, we present our efforts to verify the implementation and ensure its correctness, and compare our implementation with existing software in the field. In Chapter 6, we outline several directions for future work, and finally, we discuss the results in Chapter 7.

See Appendix A for a summary of the contents of the digital thesis archive.

We release all our code under the permissive MIT license for the benefit of the community. See Appendix B for more details and installation instructions.

Chapter 2

Preliminaries

This chapter introduces the essential concepts and terminology referenced in the rest of the thesis, providing the necessary background to understand the arguments developed in subsequent chapters.

2.1 Extensible Markup Language (XML)

Extensible Markup Language (XML) is a markup language and file format designed to store structured data. It is regarded as an *application* (a subset) of Standard Generalized Markup Language (SGML), an earlier markup language [1, Sec. 1.1].

It was designed to facilitate the straightforward transfer of documents over the Internet [1, Sec. 1.1]. The format features a human-readable structure, with relatively little effort required to implement XML parsers [1, Sec. 1.1].

An XML document is made up of *entities*, which form an *entity tree* [1, Sec. 2].

In the rest of this section, we will cover the most important concepts found in XML.

2.1.1 CDATA

CDATA sections behave similarly to text. They are parsed, but their contents are interpreted verbatim. This makes it possible to include text in XML that would otherwise be interpreted as markup (for example, text containing the characters < and &). A CDATA section is initiated by `<![CDATA[` and terminated by `]]>`. For instance:

```
<![CDATA[We can use < and & here.]]>
```

CDATA sections can be useful for embedding Cascading Style Sheets (CSS) or JavaScript source code in SVG:

```
<style type="text/css">
<![CDATA[
    rect {
        color: red;
    }
]]>
</style>
```

2.1.2 Comments

Comments are strings of text that do not affect the document semantics [1, Sec. 2.5]. They are typically used for documentation purposes. A comment starts with `<!--` and ends with `-->` [1, Sec. 2.5]. Comments cannot be nested and may be discarded by the parser [1, Sec. 2.5]. As an example, consider the following XML comment:

```
<!-- This is a comment. -->
```

2.1.3 Elements

Elements are perhaps the most essential building blocks of XML. An element is characterized by its name, which specifies its type [1, Sec. 3]. An element may be *empty*, in which case we call it an *empty element*, or may contain other elements as *children* [1, Sec. 3]. An element may also optionally contain a set of *attributes* [1, Sec. 3]. Elements are denoted by *tags* [1, Sec. 3]. As an example, we provide the following code snippet:

```
<g>
    <rect width="10" height="20" />
</g>
```

2.1.4 Tags

Tags encompass elements [1, Sec. 3]. There are three types of tags [1, Sec. 3]:

- *start-tag*: `<name>`,
- *end-tag*: `</name>`,
- *empty-element-tag*: `<name />`.

2.1.5 Attributes

Attributes are key-value pairs associated with an element [1, Sec. 3.3]. Each attribute name has a corresponding data type and, optionally, a default value [1,

Sec. 3.3]. Attributes carry the form name="value" [1, Sec. 3.3].

2.2 Scalable Vector Graphics (SVG)

SVG is an XML-based file format for describing two-dimensional graphics [2, Sec. 1.1]. Although the typical use of SVG is to create vector graphics, SVG documents can also include text and raster images [2, Sec. 2]. SVG was designed and is currently maintained by the World Wide Web Consortium (W3C) [2–4]. A recent analysis indicates that more than 60 % of all websites on the Internet use SVG to store image content, showing its widespread use [5].

SVG is an application of XML and, as such, consists of the same entities as described in Section 2.1.

In SVG, elements are used to represent graphical objects. The list of elements defined in the SVG specification [2] includes, but is not limited to:

- <g>: used to group other elements,
- <rect>, <circle>, <ellipse>, and other: elementary geometric shapes,
- <svg>: the root element of every SVG document,
- <defs> and <use>: used to create a single definition of an element (in <defs>) and use it in several places at once (with <use>),
- <linearGradient>, <radialGradient>, and <pattern>: used to create complex gradients and patterns that may be subsequently used, for instance, to fill a shape,
- <path>: used to define a path,
- <style> and <script>: used to embed CSS and JavaScript code in the document,
- <text>: used to include textual content,
- <image>: used for embedding raster images,
- <a>: used to create hyperlinks.

Attributes are used to modify the properties of graphical objects[1]. As an example, consider the following element:

```
<rect x="10" y="10" width="50" height="25" />
```

This code defines a rectangle positioned at the coordinate $\left[10, 10\right]$ with a width of 50 units and a height of 25 units.

[1] Additionally, elements may be styled using CSS.

Multiple versions of SVG are available. SVG 1.1 [2] remains the latest official SVG *Recommendation* published by the W3C. Although SVG 2, designed to replace SVG 1.1, exists as *Candidate Recommendation* [3] and a more recent *Editor's Draft* [4], neither has reached the full status of Recommendation. This thesis will primarily refer to the SVG 1.1 specification [2].

The remainder of this section introduces some of the most fundamental concepts and terminology of SVG.

2.2.1 Attribute types

In SVG, attributes can be of several types. Table 2.1 shows an overview of some of the attribute types defined in the SVG specification [2].

Name	**Description**	**Example**
`number`	A real number. May be specified using scientific notation.	2e+05
`integer`	An integer.	42
`length`	A distance measurement. Optionally followed by a unit.	1cm
`coordinate`	A position in the coordinate system (single axis). Semantically equivalent to `length`.	1.5e-02in
`angle`	An angle. Optionally followed by a unit.	45rad
`points`	A list of vertex coordinates.	0,0 10,10
`color`	A color. May be specified in RGB, HSL, hexadecimal, or named format.	#00ff00
`path-data`	Instructions that define a path.	M 0,0 L 5,1 Z
`transform-list`	A list of transformations to be applied to an element.	rotate(45) scale(2)

Table 2.1: An overview of the fundamental attribute types in SVG.

We present some of these types in more detail in subsequent sections.

2.2.2 Canvas, viewport, and viewbox

The SVG canvas can be thought of as an infinite plane on which the content is drawn [2, Sec. 7.1]. To represent points in the plane, SVG uses a Cartesian coordinate system where the ordinate axis is oriented downward, which means the coordinate values increase as we move to the right and down [2, Sec. 7.3].

Although the canvas is, in theory, infinite, the content is rendered with respect to a finite rectangular part of the canvas called the *viewport* [2, Sec. 7.1]. The viewport can be thought of as a "window" through which we observe the content on the canvas [6, 7]. If the window is shrunk or enlarged, the apparent size of the content remains unchanged, but the visible portion of the content changes accordingly. The size of the viewport is defined by the attributes `width` and `height` on the root `<svg>` element [2, Sec. 7.2].

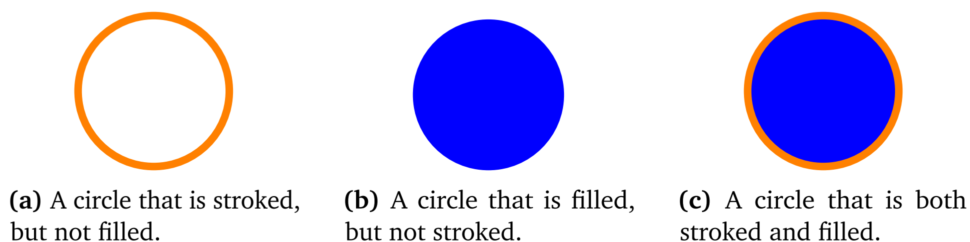
Furthermore, SVG defines the concept of *viewbox*. The viewbox can be thought of as a telescope [7, 8]. Similarly to the viewport, the viewbox restricts the portion of the canvas that is visible to the observer. However, unlike the viewport, the viewbox enables panning and zooming, similar to how a telescope may be repositioned to allow one to see a different area of the observed object or magnify a specific section of it [7, 8]. The viewbox is specified by the `viewBox` attribute on the root `<svg>` element in the form of a 4-tuple

$$(x_{min}, y_{min}, w, h),$$

where x_{min} and y_{min} define the position of the upper left corner of the viewbox, and w and h define the size of the viewbox [2, Sec. 7.7].

2.2.3 Fill and stroke

In SVG, an element may be *filled* and *stroked* [2, Sec. 11.1]. Filling means applying *paint* (a solid color, gradient, or pattern) to the inside of the element, while stroking refers to painting along the edge of the element [2, Sec. 11.1]. The difference between stroking and filling is demonstrated in Figure 2.1.



(a) A circle that is stroked, but not filled.

(b) A circle that is filled, but not stroked.

(c) A circle that is both stroked and filled.

Figure 2.1: The difference between stroking and filling in SVG.

The stroke and fill colors are controlled by the `stroke` and `fill` attributes, respectively [2, Sec. 11.2]. The width (thickness) of the stroke can be set using `stroke-width`; the default value is the width of one pixel [2, Sec. 11.4].

2.2.4 Basic shape

A common and simple *shape* that is predefined for convenience as an element [2, Sec. 1.6]. The basic shapes are [2, Sec. 1.6]:

- `<rect>`: a rectangle,
- `<circle>`: a circle,
- `<ellipse>`: an ellipse,
- `<line>`: a line segment,
- `<polyline>`: a series of connected line segments,
- `<polygon>` a series of connected line segments that form a closed shape.

2.2.5 Shape

A shape is a *graphics element* that is "defined by some combination of straight lines and curves" [2, Sec. 1.6]. Shapes include all *basic shapes* as well as the `<path>` element [2, Sec. 1.6].

2.2.6 Graphics element

A graphics element in SVG is any element that causes graphics to be drawn on the canvas [2, Sec. 1.6]. Graphics elements include all *shapes* as well as the `<text>` and `<use>` elements.

2.2.7 Paths

Paths are perhaps the most important building block of SVG. They allow one to create arbitrarily complex shapes using a simple set of commands. All basic shapes in SVG can be converted into equivalent path elements [3, Chap. 10, 9, 10].

Paths in SVG are based on the concept of a virtual "pen" that moves around the canvas and draws lines and curves to produce the desired shape [2, Sec. 8.1]. A path is defined by its *path data*: a sequence of elementary drawing operations called *path commands* that describe how the pen moves [2, Sec. 8.1]. The path commands available in SVG are listed in Table 2.2 [2, Sec. 8.3].

Character	Name	Description
m/M	moveto	Move the pen without drawing.
z/Z	closepath	Close the current subpath by drawing a line segment to the starting point of the subpath.
l/L	lineto	Draw a line segment.
h/H	horizontal lineto	Draw a horizontal line segment.
v/V	vertical lineto	Draw a vertical line segment.
q/Q	quadratic Bézier curveto	Draw a quadratic Bézier curve.
t/T	shorthand/smooth quadratic Bézier curveto	Draw a quadratic Bézier curve.
c/C	curveto	Draw a cubic Bézier curve.
s/S	shorthand/smooth curveto	Draw a cubic Bézier curve.
a/A	elliptical arc	Draw an elliptical arc.

Table 2.2: Path commands in SVG.

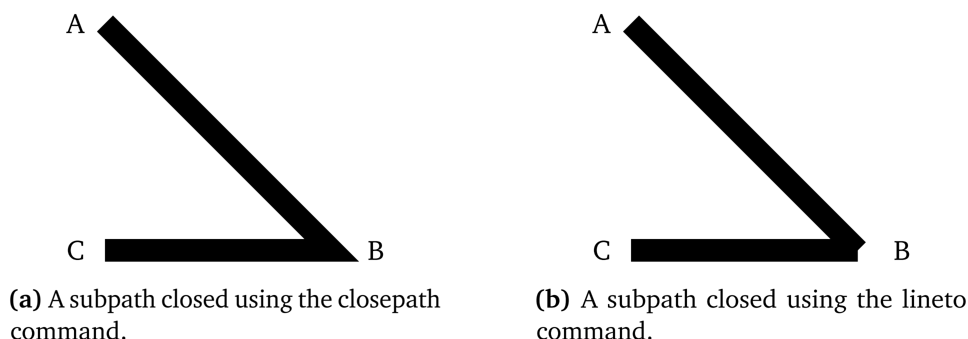
Each path command is expressed by its character and a set of parameters. When the path command character is uppercase, the parameters are interpreted as absolute coordinates [2, Sec. 8.3]. For example, the path command `M 10,20` moves the pen to the coordinate $[10, 20]$. On the other hand, path commands with a lowercase command character indicate the use of coordinates that are relative to the current point [2, Sec. 8.3]. For instance, given a current point $[x, y]$, the command `m 10,20` moves the pen to $[x + 10, y + 20]$. We use the terms *absolute path command* and *relative path command* to refer to the uppercase and lowercase versions of the commands, respectively.

If a path command is used several times in a row, subsequent command characters may be omitted [2, Sec. 8.3.1]. As an example, consider the path data `L 10 20 30 40`, which is semantically equivalent to `L 10 20 L 30 40`. We use the terms *implicit path command* and *explicit path command* to refer to these two scenarios.

The moveto and closepath commands mark the start and end of a *subpath*, respectively [2, Sec. 8.3]. A subpath is thus a subset of the path data that forms a continuous curve. A path (potentially discontinuous) may contain more than one subpath, in which case we call it a *compound path* [2, Sec. 8.3.1]. A subpath (and, by extension, a path) must always start with a moveto command that establishes the initial position of the pen [2, Sec. 8.3.2].

Note that closing a path with a closepath command is not necessarily equivalent to “manually” closing the subpath by drawing a line segment to the starting point of the subpath using the lineto command [2, Sec. 8.3.3]. In the first case, the starting and ending points will be automatically connected in a way that

smoothly connects the stroke around the connection point. However, in the latter case, the ends of the subpath will not be connected smoothly, and the stroke near the end points may appear jagged as a result. We demonstrate this subtle, yet important difference in Figure 2.2.



(a) A subpath closed using the closepath command.

(b) A subpath closed using the lineto command.

Figure 2.2: A comparison between closing a path using the closepath and lineto commands. Notice the difference in the stroke near point B.

We refer to the commands **H**, **V**, **S**, and **T** as *shorthand commands*. Shorthand commands provide functionality identical to their regular counterparts while requiring fewer parameters [2, Sec. 8.3]. The values of the missing parameters are inferred from the context[2] [2, Sec. 8.3.2].

A path is specified using the `<path>` element, with path data defined by the `d` attribute [2, Sec. 8.2]. As an example, consider the following element:

```
<path d="M0,0 H10 Q 15,7.5 10,10 Z" />
```

This path element

- starts a new subpath at the coordinate $[0, 0]$,
- draws a horizontal line segment to $[10, 0]$,
- draws a quadratic Bézier curve to $[10, 10]$ with a control point at $[15, 7.5]$,
- closes the subpath by drawing a straight line back to $[0, 0]$.

The visual appearance of this path is shown in Figure 2.3.

[2] The method employed varies based on the specific command utilized; in our thesis, we will not go into the specifics of this inference process. See [2, Sec. 8.3] for more details.

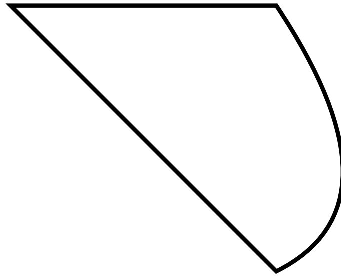


Figure 2.3: A path defined by the path data `M0,0 H10 Q 15,7.5 10,10 Z`.

Formally, we can think of paths as *parametric curves*[3]. A parametric curve is a continuous function

$$\gamma\colon [0,1] \to \mathbb{R}^2$$

that maps the values of a parameter t to points in $\mathbb{R}^2$. We call the points $\gamma(0)$ and $\gamma(1)$ the *starting* and *ending* points of the curve, respectively. If $\gamma(0) = \gamma(1)$, we say that the curve is *closed*. If a curve is not closed, it is said to be *open*.

2.2.8 Transformations

In SVG, elements can be transformed using arbitrary affine transformations [2, Sec. 7].

Definition 1 (Affine transformation)**.** An affine transformation is a mapping of the form

$$T(x) = Ax + \vec{b},$$

where

$$A = \begin{pmatrix} a & c \\ b & d \end{pmatrix}$$

is the matrix of a linear transformation and

$$\vec{b} = \begin{pmatrix} e \\ f \end{pmatrix}$$

is a translation vector.

Compositions of linear transformations can be conveniently represented using matrix multiplication. However, as translation is non-linear, we cannot use this approach for affine transformations [11, Sec. 1.2.1]. To overcome this

[3] In a strict sense, only *subpaths* can be represented as parametric curves because parametric curves must be continuous. However, we can bypass this limitation by assuming that all paths are made up of exactly one subpath.

limitation, we can represent affine transformations using an augmented matrix in *homogeneous coordinates* [11]:

$$H = \begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix}.$$

The composition of affine transformations $T_1, T_2, \dots, T_n$ given by the matrices $H_1, H_2, \dots, H_n$ in homogeneous coordinates is then simply obtained by post-multiplication of the corresponding transformation matrices:

$$T_n \circ T_{n-1} \circ \dots \circ T_1 = \prod_{i=1}^{n} H_i.$$

Given a point $\left[x, y\right]$, the transformed point $\left[x', y'\right]$ is obtained using the following [2, Sec. 7.4, 11, Sec. 1.2.1]:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

Each SVG element may have an associated *transformation list* specified by the `transform` attribute [2, 3]. The transformation list contains a sequence of transformations that are applied to the element in the order of left-to-right [2, Sec. 7.4]. The transformation may be specified as an affine transformation matrix:

$$\text{matrix}(a, b, c, d, e, f),$$

or using one of the predefined *transformation functions*; they are [2, Sec. 7.4]:

- $\text{translate}\left(t_x, t_y\right)$: translation by t_x on the x-axis and t_y on the y-axis,
- $\text{scale}\left(s_x, s_y\right)$: scaling by s_x along the x-axis and s_y along the y-axis; if $s_x = s_y$, we say that the scaling is *uniform* (otherwise, we say it is *non-uniform*),
- $\text{rotate}\left(\alpha, c_x, c_y\right)$: couter-clockwise rotation by α degrees around the point $\left[c_x, c_y\right]$,
- $\text{skewX}(\alpha)$: skewing by the angle α along the x-axis,
- $\text{skewY}(\alpha)$: skewing by the angle α along the y-axis.

For instance, the transformation list

$$\text{translate}(10, 0)\,\text{scale}(2)$$

first horizontally translates the element by 10 units and subsequently scales the element uniformly in both directions by a factor of 2.

To maintain consistency with the syntax of the `transform` attribute, we will adopt an alternative notation for function composition throughout this thesis. In addition to the traditional notation for function composition

$$f \circ g = f(g(x)),$$

we will use the notation

$$f\ g = g(f(x)),$$

in which functions are composed in the same order as in the `transform` attribute.

2.3 Bounding boxes and segmentation masks

Bounding boxes and segmentation masks are key concepts in the field of computer vision [12–14]. They enable a straightforward representation of the bounds of a shape and its interaction with other shapes in an image. In this section, we define these two terms.

2.3.1 Segmentation mask

A segmentation mask divides an image into distinct regions based on certain properties [13, 14]. Segmentation is performed by assigning each pixel a *label* such as "apple" or "orange" [14].

In this work, we are particularly interested in binary segmentation masks (or simply *masks*) that assign to each pixel of a shape the logical value of 1 and to all other pixels in the image the logical value of 0 [15].

Definition 2 (Full mask)**.** Given an element and two $m, n \in \mathbb{N}$, a (full) mask is an $m \times n$ matrix:

$$\begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,n} \end{pmatrix},$$

where

$$b_{i,j} = \begin{cases} 1, & \text{if the point } [i, j] \text{ lies \textit{inside} the element,} \\ 0, & \text{otherwise.} \end{cases}$$

The concept is shown in Figure 2.4. The choice of m and n is arbitrary.

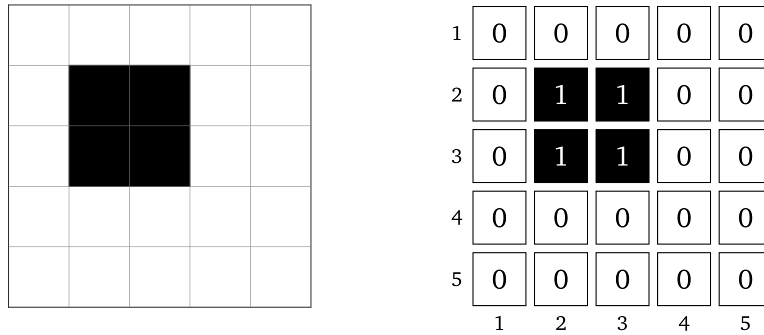


Figure 2.4: A 2x2 square inside a 5x5 image (left) and its segmentation mask (right).

Definition 3 (Visible mask)**.** A visible mask is almost identical to the full mask. The only distinction is that every pixel labeled with the value of 1 must belong to the chosen shape *and* remain unobscured by other shapes in the image. The difference between the full mask and the visible mask is illustrated in Figure 2.5.

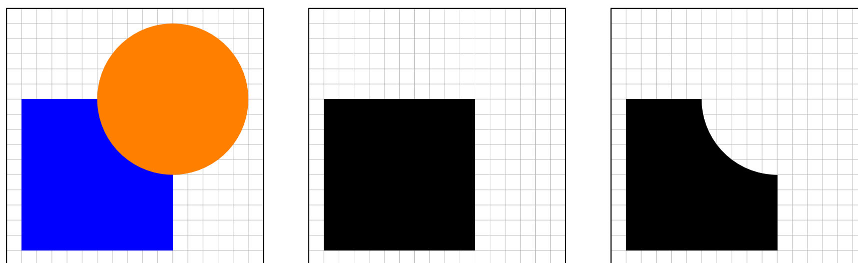


Figure 2.5: An image with two overlapping shapes (left), the square's mask (middle), and the square's visible mask (right).

2.3.2 Bounding box

A shape's bounding box is the smallest possible rectangle that contains the entire shape.

Definition 4 (Bounding box)**.** Given a shape

$$\gamma(t) = (x(t), y(t)), \quad t \in [0, 1],$$

the shape's (full) bounding box is a 4-tuple

$$(x_{min}, y_{min}, x_{max}, y_{max}),$$

such that

$$x_{min} = \min_{t \in [0,1]} x(t), \tag{2.1}$$

$$y_{min} = \min_{t \in [0,1]} y(t), \tag{2.2}$$

$$x_{max} = \max_{t \in [0,1]} x(t), \tag{2.3}$$

$$y_{max} = \max_{t \in [0,1]} y(t). \tag{2.4}$$

Notice that by our definition, the bounding box is always *axis-aligned*, meaning its sides are parallel to the axes of the Cartesian coordinate system. Such bounding boxes are also called Axis-Aligned Bounding Boxes (AABBs) [12].

Figure 2.6 illustrates the concept of the bounding box on an example shape.

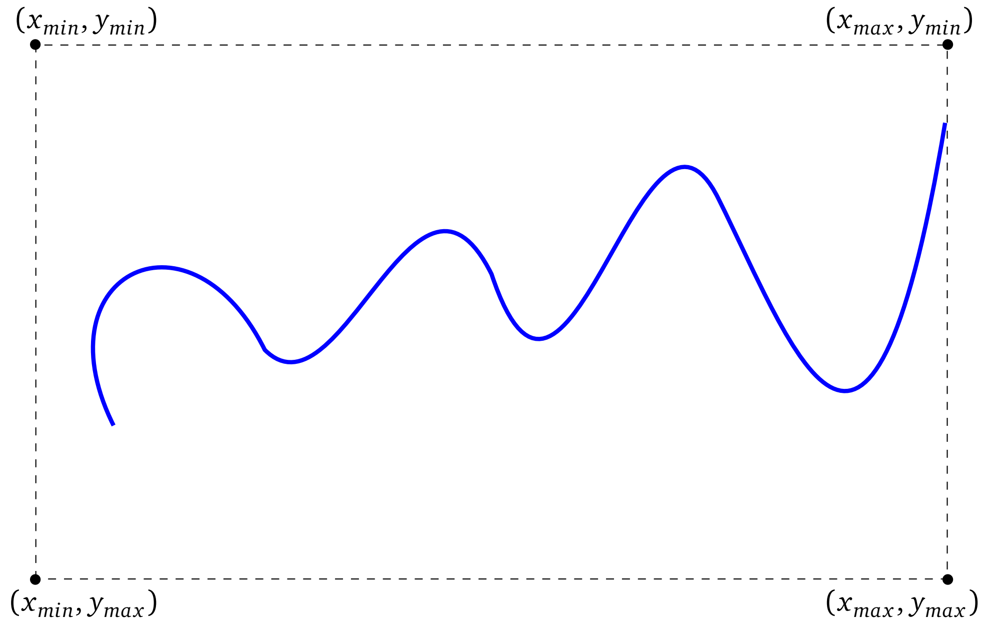


Figure 2.6: A composite Bézier curve (solid blue) and its bounding box (dashed black).

Definition 5 (Visible bounding box)**.** The difference between the visible bounding box and the full bounding box is analogous to the difference between the visible mask and the full mask. The visible bounding box is the bounding box of the visible parts of the element. If a part of the element is obscured by other elements, that part will not contribute to the dimensions of the visible bounding box.

Chapter 3

Requirements

This chapter lists some of the most significant requirements for implementing the library, most of which stem directly from the thesis assignment.

We generally prioritize features related to transformations or computer vision (for instance, reification, computing bounding boxes and masks, and applying transformations to elements). On the other hand, little emphasis is put on performance or dependency count.

3.1 Programming language

The library is developed using Python and is compatible with CPython [16] versions 3.12 and above.

3.2 Representation and I/O

The library provides robust parsing support for SVG documents. Documents created programmatically can be serialized back to the original XML format. The library can parse any *well-formed* (syntactically valid) SVG document.

The SVG document is represented using a tree data structure. The library parses attribute values into Python objects where appropriate and feasible to facilitate convenient and straightforward use. These objects are designed to express the values of the specified attribute type in an idiomatic way.

3.3 Robust typing

The library is fully typed using Python type hints according to Python Enhancement Proposal (PEP) 484 [17]. In particular, the library provides robust typing for all supported SVG elements and attributes.

3.4 Configurable formatting

The library provides several methods for modifying the format of serialized SVG code.

3.4.1 Color format

Various formats are available in SVG for defining colors. The library allows the user to choose one format to apply consistently for all color values in the output.

3.4.2 Floating-point precision

The library supports configuring the maximum number of digits for serializing numbers.

3.4.3 Path data

The library allows for configuring whether path data commands appear implicitly (where possible) or explicitly in the serialized output. In addition, users can specify whether absolute or relative path commands are used.

3.4.4 Whitespace

The library supports configuring how whitespace occurs in the output. At a minimum, the library supports configuring the number of spaces used for indentation.

3.5 Compliance with SVG specifications

The library supports all elements and attributes defined in SVG 1.1 [2] and can correctly and without errors manipulate all documents that conform to that specification.

3.6 Lossless parsing and serialization

Semantically significant data is not lost when a document is parsed and subsequently serialized. However, there may be modifications in syntax, such as whitespace changes or the use of alternative syntax to represent identical information.

The library accommodates additional elements and attributes beyond those specified in [2] to enhance flexibility. Nevertheless, the level of support for these custom elements and attributes is limited; the library does not contain typed

definitions for said elements and attributes, and the attribute values are not parsed into native types.

3.7 Manipulation

It is possible to manipulate parsed SVG documents. This includes modifying attributes, adding and removing elements, and applying affine transformations to shapes where possible.

3.8 Reification

As we have described in Section 2.2.8, arbitrary affine transformations can be applied to elements using the `transform` attribute. However, relying heavily on the use of the `transform` attribute can lead to complex and lengthy transformation lists, resulting in large file sizes and difficulty in editing.

A potential optimization involves directly applying the transformations inside the `transform` attribute to the element's attributes, eliminating the need for the attribute. This process is known as *reification* [18].

The library supports reifying translation and scaling transformations defined by the `transform` attribute. The library may allow the reification of other transformations if possible.

3.9 Coordinate system rescaling

Coordinates and lengths are commonly expressed in *user units* (without a unit) in SVG. When done so, their values depend on the size of the viewbox.

If the size of the viewbox is selected poorly, the coordinate and length values used in the document may have unnecessarily low or high magnitudes. This can contribute to large file sizes and complexity.

The library supports rescaling the coordinate system to more manageable dimensions.

3.10 Bounding boxes

The library can calculate an element's bounding box and visible bounding box.

3.11 Masks

The library can calculate an element's mask and visible mask.

Chapter 4

Implementation

In this chapter, we describe the main output of this thesis: SVGLAB, a comprehensive Python library for programmatic manipulation of SVG documents.

The chapter is structured as follows. In Section 4.1, we provide a brief general overview of the implementation. In Section 4.2, we describe how the SVG entities are represented in the library. The parsing and serialization processes are described in Section 4.4 and Section 4.5, respectively. In Section 4.6, we present our solution to the problem of reification. In Section 4.7, we briefly talk about how we support the rescaling of the coordinate system. Finally, we outline our approach to the calculation of bounding boxes and masks in Section 4.8.

4.1 Overview

The SVGLAB library is designed as a single Python package of the same name. The package has a flat import structure; all symbols intended for public consumption can be imported directly from the `svglab` namespace:

```
from svglab import <symbol>
```

The package is compatible with CPYTHON [16] versions 3.10 and higher.

The library is designed to comply with the SVG 1.1 specification [2], supporting all elements and attributes defined thereby. Furthermore, we have incorporated various advantageous features from SVG 2 [3] and the SVG 2 Editor's Draft [4], such as:

- the `pathLength` attribute, present on all basic shapes,
- the `transform-origin` attribute[4].

[4] See Section 4.6.4 for more details.

In total, the library contains the definitions of 80 SVG elements and 295 attributes.

4.2 Representation

In this section, we describe how our library represents SVG entities.

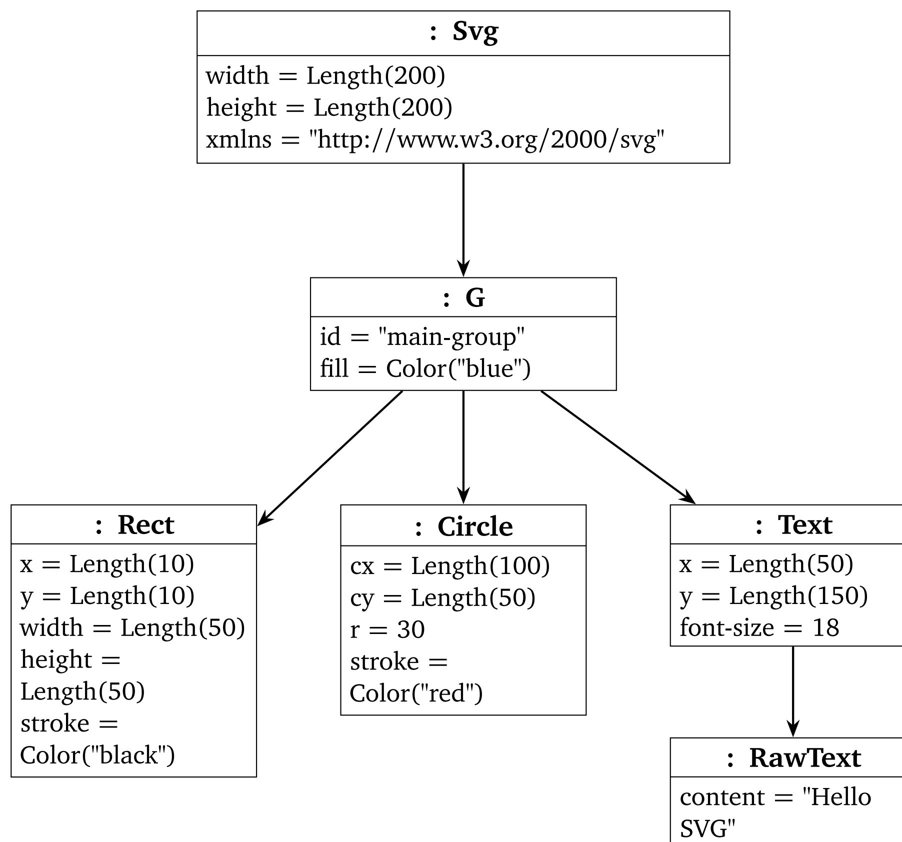
Our implementation is capable of representing all types of SVG entities, ensuring the ability to manipulate SVG documents in their entirety and that no data is lost between parsing and serialization.

We utilize an object-oriented approach, where each entity type is represented as a class and each entity as an object. We use the PYDANTIC [19] validation library to construct these classes. This design choice allows us to take advantage of PYDANTIC's parsing and validation capabilities.

In Figure 4.1, we show example SVG code and its SVGLAB representation.

```
<svg width="200" height="200" xmlns="http://www.w3.org/2000/svg">
  <g id="main-group" fill="blue">
    <rect x="10" y="10" width="50" height="50" stroke="black"/>
    <circle cx="100" cy="50" r="30" stroke="red"/>
    <text x="50" y="150" font-size="18">Hello SVG</text>
  </g>
</svg>
```

(a) SVG source code.



(b) SVGLAB representation. Arrows indicate parent-child relationships.

Figure 4.1: An example SVG and its SVGLAB representation.

4.2.1 Naming

Our implementation adheres to the naming conventions established by the SVG specification while adapting them to ensure that all names are valid Python iden-

tifiers and that names are better aligned with the Python code style guide [20]. We process the names using the following rules:

- all element names are converted to *PascalCase*,
- attribute names in *kebab-case* are converted to *snake_case*,
- attribute names in *camelCase* are kept as-is.

If, after this normalization, the resulting name is not a valid Python identifier, we suffix it with an underscore (`_`). Table 4.1 shows several example SVG names and their normalized versions for use in SVGLAB.

Type	**SVG specification** (original)	**SVGLAB** (normalized)
Element	`svg` `rect`	`Svg` `Rect`

Attribute	`stroke-width`
	`viewBox`
	`class`
		`stroke_width`
		`viewBox`
		`class_`

Table 4.1: Example element and attribute names and their normalized variants.

4.2.2 Entity base classes

In this section, we present the Abstract Base Classes (ABCs) we use to represent SVG entities. The hierarchy of these classes is shown in Figure 4.2.

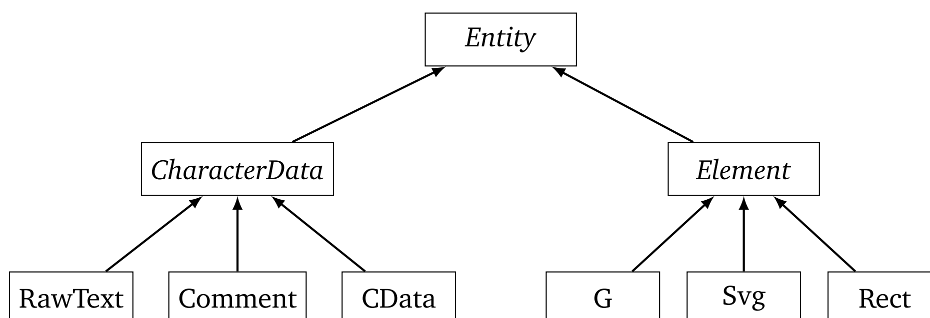


Figure 4.2: The entity class hierarchy.

Entity

The base class of the entity hierarchy. This class contains basic attributes and methods common to all entities, such as `parent`, a reference to the parent element, or `to_xml()`, a method that serializes the entity to its SVG representation.

Element

The base class for all SVG elements, which contains additional attributes and methods specific to elements in SVG. These include `children`, `descendants`, `siblings`, `find()`, and `find_all()`, which are attributes and methods for navigating the entity tree. It also includes `add_child()`, `remove_child()`, and `clear_children()`, which are methods to manipulate the entity tree.

For each concrete SVG element, we create an *element class* (such as `Svg` or `Rect`), which inherits from the `Element` class.

CharacterData

The base class for textual entities in SVG. It contains only one attribute: `content`, which stores the raw textual content of the entity.

For each concrete SVG textual entity, we create a class that inherits from `CharacterData`: the classes `Comment`, `CData`, and `RawText` represent comments, CDATA sections, and text, respectively.

4.2.3 Attributes

For the purposes of our implementation, we divide the element attributes into two categories:

- *standard attributes*: all attributes defined in any of the SVG specifications,
- *extra attributes*: all other, user-defined attributes.

We offer first-class support for standard attributes: they are represented as native Python objects for convenient manipulation, and they are automatically parsed and validated. In contrast, we store extra attributes in their original string form. By storing non-standard attributes, we ensure that our implementation complies with the requirement that no data is lost between parsing and serialization.

In this section, we will describe our approach to representing standard attributes. We represent simple attributes such as numbers, strings, and booleans using the appropriate standard Python types. For more complex attributes such as lengths, colors, transformations, and path data, we define custom dataclasses.

Lengths and angles

For the `length` and `angle` types, we provide the `Length` and `Angle` classes, respectively. Both classes provide comparable functionality and have a similar interface. The classes contain the following attributes:

- `value`: the floating-point value of the quantity (length or angle), e.g. `45`,
- `unit`: the unit associated with the quantity, e.g. `"deg"`.

The classes also define a `to(unit)` method which returns a new object of the same type, but with the value converted to the specified unit. If conversion is not possible, an exception is raised.

We use operator overloading to define the operations of addition (+) and subtraction (−) on two lengths (or two angles):

```
a = Length(1, "cm")
b = Length(5, "mm")
c = a + b  # Length(1.5, "cm")
```

If the units of the operands differ, the result is returned in the units of the left operand.

We also support the multiplication (*) and division (/) of lengths and units by scalars:

```
a = Angle(45, "deg")
b = 2 * a  # Angle(90, "deg")
```

Points

To represent the `points` type, we have created the `Point` class to represent a point in the plane[5]. The class contains two attributes:

- `x`: the x-coordinate of the point,
- `y`: the y-coordinate of the point.

We support the operations of addition (+) and subtraction (−) of two points and multiplication (*) and division (/) by a scalar:

```
a = Point(1, 1)
b = Point(1, 0)
c = a + 2 * b  # Point(3, 1)
```

The `Points` type is then constructed simply as a list of points (that is, type `list[Point]`).

Transformations

We represent each type of SVG transformation using an associated class. The classes are listed in Table 4.2.

[5] We also use this class to represent vectors in two-dimensional space. Although we acknowledge the formal distinction between points and vectors, we do not provide a dedicated class for manipulating vectors, as its implementation would be virtually identical to the `Point` class.

SVG syntax	SVGLAB class	Attributes
translate(t_x, t_y)	`Translate`	`tx`, `ty`
scale(s_x, s_y)	`Scale`	`sx`, `sy`
rotate(α, c_x, c_y)	`Rotate`	`angle`, `cx`, `cy`
skewX(α)	`SkewX`	`angle`
skewY(α)	`SkewY`	`angle`
matrix(a, b, c, d, e, f)	`Matrix`	`a`, `b`, `c`, `d`, `e`, `f`

Table 4.2: Representation of SVG transformations in SVGLAB.

Instances of each class can be converted to equivalent instances of type `Matrix` using the `to_matrix()` method:

```
t = Translate(1, 2)
m = t.to_matrix()  # Matrix(0, 0, 0, 0, 1, 2)
```

Transformations may be composed using the matrix multiplication operator (@):

```
a = Translate(1, 2)
b = Scale(3, 4)
c = a @ b  # Matrix(3, 0, 0, 4, 1, 2)
```

Similarly, transformations can be applied to points:

```
a = Point(1, 0)
b = Translate(0, 2)
c = b @ a  # Point(1, 2)
```

The `transform-list` type is then a list of any of the transformation classes, that is,

```
list[Translate | Scale | Rotate | SkewX | SkewY | Matrix]
```

Colors

For representing colors, we provide a `Color` class[6]. Objects of type `Color` are initialized using the string representation of the color: `"rgb(255, 0, 0)"`, for instance. The class provides methods such as `as_hsl()` and `as_named()` that return the string representation of the color in the desired format.

Path data

Our approach to representing path data is similar to that used to represent transformations. For each path command, we provide an associated class. Table 4.3 shows each type of command and the class that we use to represent it.

[6] Most of the functionality is provided by the PYDANTIC-EXTRA-TYPES [21] package.

Cmd.	Name	SVGLAB class
M	moveto	`MoveTo`
Z	closepath	`ClosePath`
L	lineto	`LineTo`
H	horizontal lineto	`HorizontalLineTo`
V	vertical lineto	`VerticalLineTo`
C	curveto	`CubicBezierTo`
S	shorthand/smooth curveto	`SmoothCubicBezierTo`
Q	quadratic Bézier curveto	`QuadraticBezierTo`
T	shorthand/smooth quadratic Bézier curveto	`SmoothQuadraticBezierTo`
A	elliptical arc	`ArcTo`

Table 4.3: Path commands and their representations in SVGLAB.

As mentioned in Section 2.2, the commands **H**, **V**, **S**, and **T** are *shorthand commands*, and the values of some of their parameters are inferred from the context. In our implementation, this calculation is performed solely on an as-needed basis. Notably, no Python package appears to support shorthand commands at the time of writing; instead, existing implementations resolve shorthand commands to their full versions upon parsing [18, 22, 23].

The path data as a whole is represented by the `PathData` class. It is a mutable sequence[7] of path commands and can be used as the value of the `d` attribute for the `<path>` element. The `PathData` class contains sanity checks to ensure that the path is always valid. For example, the first command of the path must be a `MoveTo` command.

Similarly to points, transformations may be applied to path data:

```
path = PathData()
        .line_to(Point(1, 1))
        .quadratic_bezier_to(Point(5, 0), Point(10, 10))
        .close()
SkewX(30) @ path  # PathData(...)
```

Path commands can be incorporated into the path data via methods from the `MutableSequence` ABC, such as `append()`, or using convenience methods defined in the `PathData` class, such as `PathData.line_to()`.

By default, command coordinates are interpreted as absolute. However, the user may also wish to specify commands using relative coordinates. This can be done using the aforementioned methods with `relative=True`.

[7] In other words, the class inherits from `collections.abc.MutableSequence`.

The `ClosePath` (**Z**) command deserves particular attention—it is the only command that requires no parameters and thus requires specialized handling.

Existing libraries (such as SVGPATHTOOLS [22]) sometimes implement this feature by treating the command as if it were a line. When the user manipulates the path containing the said command and subsequently attempts to serialize it, the library outputs **Z** if the endpoint of the `LineTo` (**L**) command coincides with the starting point of the path, **L** otherwise. This approach produces incorrect results (see Figure 2.2b) because the information on whether the path is closed is irrevocably lost during this conversion.

To avoid this issue, we instead represent the command as is, using a dedicated class. We can use this approach because no matter which affine transformations we apply to our path, the transformed path will be closed precisely when the original path was closed (see Theorem 2 for proof).

Attribute composition

We use primitive Python types and the dataclasses described above to define attributes. Each attribute is defined using a mixin class; for instance:

```
class StrokeWidthAttr:
    stroke_width: Length | Literal[
        "none",
        "inherit"
    ] | None = None
```

These mixin classes, along with the `Element` ABC, are then combined to form a final concrete element class, such as `Rect`:

```
class Rect(StrokeWidthAttr, Element):
    pass
```

This approach has the following benefits:

- simple composition of attribute classes into element classes without redundancy,
- the possibility to share attribute documentation,
- the possibility to determine the presence of an attribute on an element in a type-safe manner via `isinstance()`:

```
elem = ...
elem.stroke_width = Length(10)  # type checker error

if isinstance(elem, StrokeWidthAttr):
    elem.stroke_width = Length(10)  # OK
```

The concept is depicted in Figure 4.3.

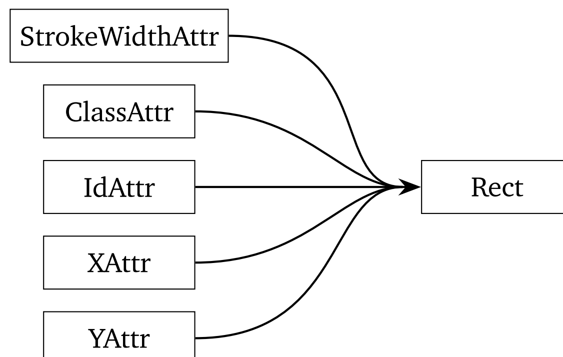


Figure 4.3: The composition of attribute mixin classes into an element class.

The element classes themselves typically do not contain any additional logic or attribute definitions. Where appropriate, we group several attributes often used together into a single mixin class.

4.2.4 Traits

In the SVG specification [2], elements that are related in some way are often referred to by a special term. Such groups of elements include *graphics elements*, *shapes*, and *basic shapes*, which we have introduced in Section 2.2. Another example is the *container elements*: elements that may contain other elements as children [2, Sec. 1.6].

For each of the groups defined in [2, Sec. 1.6], we create an ABC; collectively, we call these ABCs *traits*. Elements pertaining to a particular group inherit the associated trait.

We do this because elements belonging to the same group can often share common attributes or functionality; for instance, the operation of calculating a bounding box is defined on all graphics elements:

```
class GraphicsElement(...):
    def get_bbox() -> Bbox:
        ...
```

This approach allows us to further reduce code duplication when defining elements. In addition, it allows users to conveniently check elements against familiar terminology used in the SVG specification:

```
elem = Rect()
isinstance(elem, BasicShape)        # True
isinstance(elem, ContainerElement)  # False
```

4.3 Elements

We represent SVG elements using PYDANTIC [19] models. As described in Section 7, we compose previously defined attribute mixin classes into *element classes*, where each distinct SVG element has a corresponding class. Examples include `Svg`, `Rect`, `Circle`, or `Path`.

All *standard attributes* can be accessed and modified using dot notation and passed to the constructor:

```
rect = Rect(x=Length(10), color=Color("black"))
rect.color = Color("#ff00ff")
rect.x  # Length(10)
rect.y  # None
```

By default, all attribute values are initialized to `None`. The value of `None` signifies the absence of an attribute; attributes with this value will not be included in the serialized output:

```
rect = Rect()
rect.to_xml()  # <rect />
```

Non-standard attributes (or *extra attributes*) may also be passed to the constructor, but their type must be `str`. Instead of using the dot notation, they are available as a dictionary of type `dict[str, str]`:

```
Circle(foo=1)              # error, must be str
circ = Circle(foo="bar")   # OK
circ.extra_attrs()         # {"foo": "bar"}
```

4.4 Parsing

In this section, we describe how we parse SVG documents from their textual representation into their SVGLAB representation we described in the previous section.

Our mechanism for parsing an SVG document is fundamentally divided into the following two stages:

1. parsing the SVG document with an XML parser to obtain an entity tree,
2. parsing the attributes of each element in the tree into native Python types.

We delegate the initial phase to BEAUTIFUL SOUP [24], a Python toolkit designed to parse Hypertext Markup Language (HTML) and XML documents. First, we use BEAUTIFUL SOUP to parse the SVG document into a tree of BEAUTIFUL SOUP objects. In stage two, we transform these objects into instances of SVGLAB classes

such as Svg and Rect, parsing the attributes along the way. The entire process is depicted in Figure 4.4.

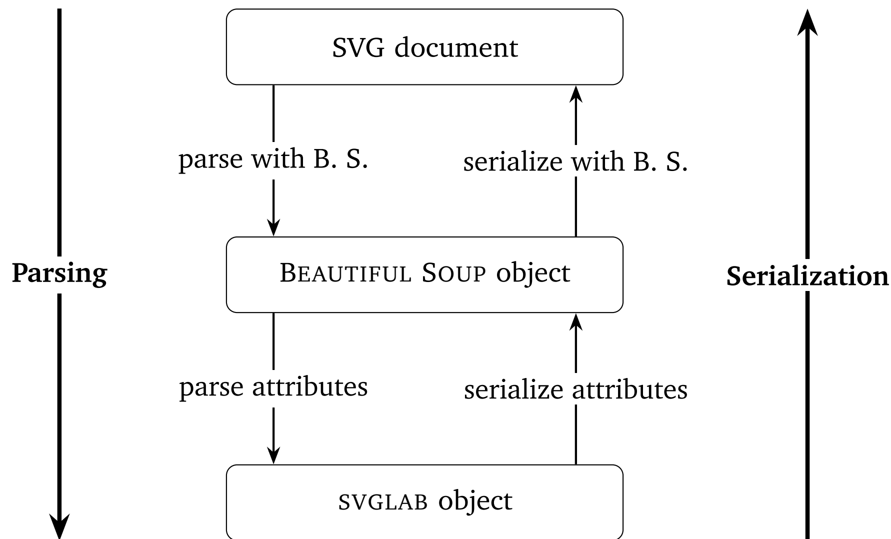


Figure 4.4: The parsing and serialization processes.

Most attribute values represent basic types[8] and therefore can be parsed directly by PYDANTIC.

However, SVG contains several more elaborate attribute types, such as `length` or `angle`. We must parse those attributes manually.

To accomplish this task, we use the LARK [25] parser generator. With a context-free grammar specified using LARK's grammar language, similar to Extended Backus–Naur Form (EBNF), LARK can parse arbitrary input into Python objects [26].

A parser generator enables robust and maintainable attribute parsing compared to traditional methods, such as manually constructing a recursive-descent parser.

4.4.1 BEAUTIFUL SOUP parsers

The BEAUTIFUL SOUP library offers the option to use several different parsers [24]:

- `html.parser`: an HTML parser with medium performance that is a part of the Python standard library,
- `lxml`: an HTML parser with high performance; requires an external dependency (LXML [27]),

[8] For instance, string literals, numbers, boolean values, and unions and sequences thereof.

- `lxml-xml`: an XML parser with high performance; requires an external dependency (LXML),
- `html5lib`: an extremely lenient HTML parser with very low performance; requires an external dependency (HTML5LIB [28]).

We use the `lxml-xml` parser because it is purpose-built for XML parsing, offers superior performance, and is recommended for general use by the BEAUTIFUL SOUP documentation [24]. Users may opt for an alternative parser if preferred—our implementation is designed to ensure uniform behavior across different parsers.

4.4.2 Using HTML parsers

Although HTML parsers generally work well for parsing SVG documents, they automatically enclose the document in varying HTML elements such as `<html>` and `<body>`.

To overcome this limitation, we perform a Breadth-First Search (BFS) of the parsed tree to locate the outermost `<svg>` element. If found, the element is considered the root SVG document fragment and designated the root of the SVG element tree. If there is no such element or if one of the element's siblings is an `<svg>` element (and so there is more than one root `<svg>` element), the input is assumed invalid.

As a corollary, this approach enables the parsing of SVG embedded in HTML documents.

4.5 Serialization and output formatting

In this section, we describe serialization, the process of converting library objects into their string representation.

Recall Figure 4.4 and notice that the serialization process is precisely the opposite of the parsing method we have described in the previous section.

The library supports a wide range of configurable formatting rules that can be applied during serialization. We describe some of those as well.

4.5.1 The formatter

To facilitate the user's ability to customize the formatting options, we have created a dataclass called `Formatter`.

The user can utilize the formatter to specify their desired formatting preferences. Subsequently, the formatter can be used in several ways (from lowest to highest priority):

- globally using the `set_formatter()` function,
- temporarily using a context manager,
- as a direct argument supplied to functions that output XML (such as `to_xml()`).

The formatter contains, among others, the following options:

- `indent: int`
  the number of spaces to use for indentation,
- `path_data_commands: "implicit" | "explicit"`
  whether to use implicit or explicit commands in path data,
- `path_data_coordinates: "absolute" | "relative"`
  whether to use absolute or relative coordinates in path data,
- `alpha_channel: "percentage" | "float"`
  whether to represent the alpha channel of colors as a percentage (e.g., 50%) or a decimal number from the interval $[0, 1]$ (e.g., 0.5).

In total, the formatter supports 24 formatting directives.

4.5.2 Serializable objects

Not all Python objects are serializable. Serializable objects include a small set of predefined primitives and objects that implement the `CustomSerializable` protocol. Moreover, iterables of serializable objects also form a serializable object.

4.5.3 Serialization of primitives

Certain primitive types are serialized directly in the `serialize()` function due to their inability to implement the `CustomSerializable` protocol. They are:

- `int`, `float` } serialized using several user-defined formatting rules,
- `str` serialized as-is,
- `bool` serialized as `"false"` and `"true"` or `"0"` and `"1"`, depending on the context.

Serialization is performed by a set of mutually recursive functions. For example, the `serialize()` function serializes iterables by recursively calling itself on each item in the iterable and joining the results.

4.5.4 The `CustomSerializable` protocol

The `CustomSerializable` protocol is a simple protocol that requires the implementation of a single method:

```
class CustomSerializable(Protocol):
    def serialize(self) -> str:
        ...
```

The protocol allows arbitrary objects to define their own serialization logic. As an example, consider the `Point` class:

```
class Point(CustomSerializable, ...):
    ...

    @override
    def serialize(self) -> str:
        formatter = serialize.get_current_formatter()
        x, y = serialize.serialize(self.x, self.y)

        return f"{x}{formatter.point_separator}{y}"
```

In this example, `get_current_formatter()` retrieves the current formatting settings. Then, the `serialize()` function serializes two floats, the x and y coordinates of the point. Subsequently, these coordinates are combined according to user preferences to produce the final string representation. This example illustrates how predefined and custom serialization logic is combined to form a coherent serialization model.

4.5.5 The serialization process

With these principles established, we can analyze the logic of `serialize()`, the core of the serialization process. The function takes an arbitrary object and, using the current formatter settings, returns the object's string representation.

Initially, the function ensures that the object is serializable. Should the value be one of the predefined primitives, it is serialized directly using the appropriate function. In instances where the object implements the `CustomSerializable` protocol, its `serialize()` method is invoked, and the result is returned. As a special case, if the object is iterable, its values are serialized using a recursive call to `serialize()` and thereafter combined.

4.6 Reification

In this section, we present our implementation of the reification algorithm. Our objective is to systematically process the transformation lists of elements and

apply the transformations found therein to the elements' attributes. In doing so, the transformations will effectively become *baked into* the attributes of the elements. At the end of the process, the `transform` attributes can be safely removed. As an example, consider the following SVG code:

```
<svg width="100" height="100">
  <g transform="translate(25, 50) scale(2)">
    <rect x="5" y="5" width="10" height="5"
     ↪  transform="translate(5, 10)" />
  </g>
</svg>
```

In this SVG document, we have two elements with associated transformation lists. We wish to obtain the following (visually equivalent) SVG where all transformations have been reified:

```
<svg width="100" height="100">
  <g>
    <rect x="45" y="80" width="20" height="10"
     ↪  stroke-width="2" />
  </g>
</svg>
```

Let us begin with the observation that not all transformations can be reified.

4.6.1 Non-reifiable transformations

Not all transformations can be reified because of their effect on the element's stroke. Consider, for instance, an element with the non-uniform scaling transformation

$$\text{scale}(2, 1)$$

applied to it. This transformation creates a stroke of unequal width along the x and y axes, as demonstrated in Figure 4.5.

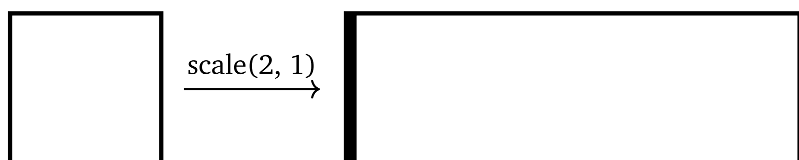


Figure 4.5: The effect of non-uniform scaling on a shape's stroke.

Recall now that in SVG, the width of the stroke is specified by the `stroke-width` attribute, in which a single value determines the thickness of the stroke along the entire outline of the element. Clearly, the `stroke-width` attribute does not offer a means to represent the non-uniform stroke width on our transformed

element. Therefore, it is not possible to represent the transformed element using the syntax of SVG[9]. Similarly, skewing transformations may not be reified for the same reason.

Furthermore, the reified shape may not always be representable using the original SVG element. For example, reifying rotation on a rectangle without converting it to an equivalent path is not possible. Table 4.4 shows which transformations are reifiable on different SVG shapes.

	translate()	scale()	rotate()	skewX() and skewY()
`<line>`	✓	✓	✓	✗
`<rect>`	✓	✓	†	✗
`<circle>`	✓	✓	✓	✗
`<ellipse>`	✓	✓	†	✗
`<polygon>`	✓	✓	✓	✗
`<polyline>`	✓	✓	✓	✗
`<path>`	✓	✓	✓	✗

✓ Possible to reify.
† Possible to reify only after converting the element to a path.
✗ Not possible to reify.

Table 4.4: The possibility of reification among different types of elements and transformations.

4.6.2 Reordering transformations

Affine transformations are inherently non-commutative; the order in which they are applied significantly impacts the resulting shape. Given a transformation list

$$T_1\, T_2\, \ldots\, T_n,$$

only the last transformation T_n can be applied directly to the element. Consequently, encountering a non-reifiable transformation during reification requires the reification process to halt, which leaves that transformation and any subsequent transformations in the list.

To address this limitation, we propose to reorder the transformations such that all non-reifiable transformations occur first in the list, followed by transformations that can be reified. This method enables the reification of all but the non-reifiable transformations—regardless of their position within the original list. That is,

[9] Technically, it may be possible to represent this shape using paths. However, determining the correct path data to represent the transformed element would be far from trivial (especially when a skewing transformation is applied). Furthermore, this approach would typically increase the overall size and complexity of the document, which would largely defeat the purpose of reification.

given a transformation list L, consisting of reifiable transformations $T_1 \ldots T_n$ and non-reifiable transformations $N_1 \ldots N_k$, we want to find a new list L' such that

$$L' = N_1 N_2 \ldots N_k T_1 T_2 \ldots T_n$$

and

$$\prod_{t \in L} t = \prod_{s \in L'} s.$$

To tackle the task of reordering affine transformations, we have derived the following identities (see Appendix C for the derivations):

$$\text{translate}\left(t_x, t_y\right) \text{scale}\left(s_x, s_y\right) = \text{scale}\left(s_x, s_y\right) \text{translate}\left(\frac{t_x}{s_x}, \frac{t_y}{s_y}\right) \tag{4.1}$$

$$\text{translate}\left(t_x, t_y\right) \text{skewX}(\alpha) = \text{skewX}(\alpha) \text{translate}\left(t_x - t_y \tan\alpha, t_y\right) \tag{4.2}$$

$$\text{translate}\left(t_x, t_y\right) \text{skewY}(\alpha) = \text{skewY}(\alpha) \text{translate}\left(t_x, t_y - t_x \tan\alpha\right) \tag{4.3}$$

$$\text{scale}\left(s_x, s_y\right) \text{skewX}(\alpha) = \text{skewX}\left(\arctan\left(\frac{s_x}{s_y} \tan\alpha\right)\right) \text{scale}\left(s_x, s_y\right) \tag{4.4}$$

$$\text{scale}\left(s_x, s_y\right) \text{skewY}(\alpha) = \text{skewY}\left(\arctan\left(\frac{s_y}{s_x} \tan\alpha\right)\right) \text{scale}\left(s_x, s_y\right) \tag{4.5}$$

$$\text{translate}\left(t_x, t_y\right) \text{rotate}\left(\alpha, c_x, c_y\right) = \text{rotate}\left(\alpha, c_x + t_x, c_y + t_y\right) \tag{4.6}$$

$$\text{scale}\left(s_x, s_y\right) \text{rotate}\left(\alpha, c_x, c_y\right) = \text{rotate}\left(\alpha, s_x \cdot c_x, s_y \cdot c_y\right) \text{scale}\left(s_x, s_y\right) \tag{4.7}$$

In each formula, the transformations are swapped and their parameters are adjusted so that the effect of composing the transformations remains unaltered.

4.6.3 Decomposition of affine transformation matrices into elementary transformations

As discussed in section 4.6.2, swapping the order of two transformations is crucial in the reification process. We have shown that for pairs of *elementary transformations* (`translate`, `scale`, `rotate`, `skewX`, and `skewY`), there exist formulas that allow their orders to be swapped and their parameters adjusted in a manner that preserves the overall effect when the transformations are composed.

However, we hypothesize that there is no simple closed-form formula for arbitrary affine transformations of the form `matrix(a, b, c, d, e, f)`.

To address this limitation, we propose decomposing all matrices in a transform list as a preprocessing step before reification begins. Let T be an affine transformation given in homogeneous coordinates:

$$T = \begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix}.$$

Using the definition of an affine transformation $T(x) \coloneqq Ax + \vec{b}$ allows us to isolate the translation vector $\vec{b} = (e, f)^T$ and proceed with the decomposition of the linear transformation A:

$$T(x) = Ax + \vec{b} = \begin{pmatrix} a & c \\ b & d \end{pmatrix} x + \begin{pmatrix} e \\ f \end{pmatrix}.$$

We want to obtain a factorization of A such that each factor contains only one type of elementary transformation.

According to Wang [29], the Lower-Diagonal-Upper (LDU) and QR factorizations form promising candidates for a suitable decomposition. However, neither method is universally optimal in the sense that the decomposition may occasionally contain "undesirable" transformation types (skewing, non-uniform scaling), even though the transformation could be expressed without them (for example, using rotation). In this regard, LDU is unsuitable for matrices that involve rotation (but handles skewing well), while QR struggles with skewing (and, in contrast to LDU, handles rotation well) [29].

Therefore, we propose a hybrid approach that involves computing both factorizations and selecting the most suitable one. The selection is made using a cost function ω, defined as:

$$\omega(t) = \begin{cases} 10^6 & \text{if } t \text{ is a skew or non-uniform scale,} \\ 10^3 & \text{if } t \text{ is a rotation,} \\ 1 & \text{otherwise.} \end{cases}$$

The costs are defined in a manner that avoids non-reifiable transformations and favors simpler decompositions. The cost of a decomposition $\mathfrak{D}$ is calculated as the sum of $\omega(t)$ over all its constituent transformations t:

$$\text{cost of decomposition } \mathfrak{D} = \sum_{t \in \mathfrak{D}} \omega(t).$$

We select the decomposition with the lowest total cost. Once we have obtained this decomposition, we can apply the swapping formulas as usual to sort the transformation list so that we can reify all reifiable transformations.

4.6.4 The `transform-origin` attribute

By default, all transformations in SVG occur around the origin of the coordinate system $[0, 0]$. However, the `transform-origin` attribute allows one to alter the origin of a transformation. Therefore, it is essential to consider the value of this attribute during reification.

An element with a custom transform origin can be converted into an equivalent element with a transform origin $[0, 0]$ [30]. Consider an element with a transform origin $[o_x, o_y]$ and a transformation list $L = T_1 \, T_2 \, \ldots \, T_n$. By changing the

transformation list to

$$\text{translate}\left(o_x, o_y\right)\ T_1\ T_2\ \ldots\ T_n\ \text{translate}\left(-o_x, -o_y\right),$$

we effectively move the origin of the transformations to $\left[0,0\right]$.

Subsequently, we set the transform origin to the value $\left[0,0\right]$ by removing the `transform-origin` attribute.

This approach enables the redefinition of the `transform-origin` attribute in terms of the element transformation list. After the process is completed, we can proceed with the reification as we usually would.

4.6.5 Implementation of the reification algorithm

With the above foundation, we can implement the reification algorithm.

First, `matrix()` invocations in the `transform` attribute are decomposed into elementary transformations using the approach described in Section 4.6.3.

Second, the `transform-origin` attribute is decomposed as described in Section 4.6.4 if present.

Third, for each transformation, we verify that it is reifiable. If it is, we move it to the end of the transformation list using the method from Section 4.6.2. During this step, we obtain new parameters for the transformation; the modified transformation is then applied to the element.

When applying a transformation to an object, we must also apply it to the element's children. We accomplish this by prepending it to the transformation list of each child. Subsequently, we recursively call the reification algorithm to reify the transformation.

Finally, we remove the `transform` attribute if the transformation list is empty.

4.7 Rescaling the coordinate system

We rescale the coordinate system of the SVG document by altering the size of the viewbox. Upon modifying the viewbox, we must adjust the values in the document to counteract the change. This ensures that the visual appearance of the objects remains unaltered. To address this, we scale and translate the root element accordingly. Given a viewbox (x, y, w, h) and its replacement $\left(x', y', w', h'\right)$, we calculate

$$s_x = \frac{w'}{w} \text{ and } s_y = \frac{h'}{h}, \quad \text{the scaling parameters,} \tag{4.8}$$

$$t_x = x' - x \text{ and } t_y = y' - y, \quad \text{the translation parameters.} \tag{4.9}$$

Afterwards, we prepend the following transformations to the transformation list of the root element:

$$\text{scale}\left(s_x, s_y\right), \tag{4.10}$$

$$\text{translate}\left(t_x, t_y\right). \tag{4.11}$$

Finally, we use the reification algorithm described in Section 4.6 to reify the new transformations.

4.8 Computing an element's bounding box and mask

In this section, we present our method for computing an element's full/visible bounding box and full/visible mask.

The problem of determining an element's bounding box can be reduced to obtaining its mask and then finding its bounding box as if it were an image. Therefore, we will only consider the element's mask for now.

During the computation, we wish to consider the element in its entirety—including its stroke, stroke width, and visibility. Furthermore, in the case of a visible mask, we also want to consider the appearance of other elements in the image, including their stroke, fill, and other properties.

Consequently, the process is affected by a particularly high number of attributes in the SVG document—including, but not limited to:

- the `stroke-*` attributes, which alter the element's stroke,
- the `fill` attribute,
- the `display` and `visibility` attributes, which can disable the rendering of the element altogether.

It would be impractical to implement this operation from scratch—it is akin to creating a fully-featured SVG renderer. Instead, we propose an alternative implementation method: by leveraging an existing rendering framework, we convert the SVG document into a bitmap image. The mask can then be obtained by simply distinguishing between the element's color and the image background.

We use RESVG [31], a lightweight SVG rendering tool written in Rust, to render SVG documents into raster images. We include RESVG in our project through the RESVG_PY [32] Python package, which provides a "safe and high level binding for the resvg project", enabling straightforward use within our Python environment. Finally, we use the PILLOW [33] imaging library to represent and work with the rendered images.

We make the rendering functionality part of SVGLAB's public Application Programming Interface (API) for users' convenience; users can invoke the `render()` method on the Svg class to render the SVG document into a bitmap image.

4.8.1 Full mask

To obtain the full mask of an element in an SVG document, we first preprocess the document by setting the `visibility` attribute to `"hidden"` on all elements but the element we are interested in, thus disabling their rendering. Subsequently, we set the following attribute values on the target element:

- `display`: (unset),
- `visibility="visible"`,
- `fill="black"`,
- `fill-opacity="1"`,
- `stroke="black"`,
- `stroke-opacity="1"`,
- `opacity="1"`.

These preprocessing steps ensure that, regardless of whether the element was originally visible, filled, stroked, or obscured by other elements, the element is now visible and has a solid fill and stroke. Furthermore, the element is now the only visible element in the document.

After this step, we can easily compute the mask of the element by rendering the SVG document to a bitmap image and iterating over all pixels in the image. If the pixel is black, we set the corresponding member of the mask to 1. Otherwise, we set it to 0.

The result is returned as a binary NumPy [34] `NDArray` of shape (w, h), where w and h are the width and height of the SVG document, respectively.

4.8.2 Visible mask

To compute the visible mask of an element, we prepare two SVG documents based on the original SVG document:

1. a copy of the document with the target element's `visibility` attribute set to `"hidden"`,
2. the original document without any modifications.

We render both documents to raster images and determine the visible mask by comparing their pixels. At any position, if the pixels differ, we set the corresponding cell of the mask to 1. Otherwise, we set it to 0.

Thus, we essentially monitor the pixels that change when the target element is introduced into the SVG document. If a certain pixel's value changes when

the element is added, it means the element is visible at that position and not obscured by other elements.

Once again, the result is returned as a NUMPY array.

4.8.3 Bounding box

The full and visible bounding boxes of an element can be easily computed from its full and visible mask, respectively. We do this by converting the mask to a bitmap image where the pixels are either black (for mask values of 1) or transparent (for mask values of 0). We then use PILLOW's `getbbox()` method to obtain the bounding box as a 4-tuple.

Chapter 5

Evaluation

5.1 Testing

To ensure that our implementation meets the specified requirements and behaves as expected, we have created a test suite comprising various test cases designed to validate the functionality and prevent regression when refactoring or implementing new features. Although our test suite is not comprehensive, we aim to cover key functionality and areas with high risk of failure or complexity.

Our test suite comprises over 250 distinct test cases and achieves approximately 82% code coverage.

5.2 Comparison with existing software

In this section, we compare our library, SVGLAB, with existing software of similar functionality within the Python ecosystem.

To obtain a list of candidates for comparison, we searched GitHub[10] for projects that satisfy the following conditions[11]:

- the project is implemented in Python,
- the project has accumulated at least one hundred stars,
- the term "svg" appears in the project name, description, or list of topics.

These criteria ensure that the selected candidates are relevant. At the time of writing[12], there are 79 repositories that meet these criteria.

[10] `https://github.com/`
[11] The search string representing these criteria is `"svg language:Python stars:>100"`.
[12] 2025-04-22.

We subsequently conducted a manual inspection to filter out repositories not relevant to our work: only libraries where the primary purpose is the manipulation of SVG documents or their components were included[13]. Of the 79 repositories, we identified eight suitable candidates for comparison.

In table 5.1, we present each project along with its official description, the number of stars, and the date of the last release or tag.

Name	Description	Stars	Last Release
DRAWSVG [35]	"Programmatically generate SVG (vector) images, animations, and interactive Jupyter widgets"	633	2024-06-23
SVGPATHTOOLS [22]	"A collection of tools for manipulating and analyzing SVG Path objects and Bezier curves"	581	2023-05-20
SVGWRITE [36]	"Python Package to write SVG files"	559	2022-07-14
SVG_UTILS [37]	"Python tools to create and manipulate SVG files"	320	2021-02-10
SVG.PY [38]	"Type-safe and powerful Python library to generate SVG files"	308	2025-03-16
SVG.PATH [23]	"SVG path objects and parser"	221	2025-02-27
SVG_STACK [39]	"concatenate SVG files"	169	2021-03-13
SVGELEMENTS [18]	"SVG Parsing for Elements, Paths, and other SVG Objects"	151	2023-08-17

Table 5.1: An overview of Python SVG manipulation libraries.

Libraries DRAWSVG, SVGWRITE, and SVG.PY only support the programmatic *creation* of SVG documents, but they do not possess SVG parsing capabilities [35, 36, 38].

The SVG.PATH project and its fork SVGPATHTOOLS are specifically designed to parse, manipulate, and serialize SVG *path data*, but lack support for other parts of SVG [22, 23]. However, compared to SVGLAB, they provide a wider range of path-related functions such as calculating the arc length of a path or obtaining the coordinate of a point in a path based on the value of the parameter t [22, 23].

[13] In particular, we excluded projects such as file format converters, renderers, machine learning projects, and projects where SVG is merely used as a format to represent output data.

The SVG_UTILS and SVG_STACK libraries offer an extremely minimalistic API for SVG manipulation: they are primarily intended to concatenate existing SVG documents [37, 39].

Finally, the SVGELEMENTS project contains functionality similar to SVGLAB, offering a wide range of functionality related to SVG parsing, manipulation, and serialization [18].

5.2.1 Comparison with SVGELEMENTS

In this section, we provide an in-depth comparison between SVGELEMENTS and SVGLAB.

To begin with, Table 5.2 briefly summarizes some of the most important similarities and differences between the two libraries.

Table 5.2: A comparison between SVGELEMENTS and SVGLAB.

	SVGELEMENTS	SVGLAB
SVG parsing	✓	✓
SVG manipulation	✓	✓
SVG writing	✓	✓
SVG optimization	✗	Basic (via formatting options)
Lossless parsing and serialization	✗	✓
Formatting options	Basic	Highly-configurable
XML entities	Text (limited support)	Text, CDATA sections, comments
SVG 1.1 support	Partial	Full
SVG 2 support	Partial	Partial
Support for other SVG versions	✗	✗
CSS support	✓	✗
Reification	✓	✓
Bounding box	✓	✓
Mask	✗	✓
Rendering	✗	via RESVG [31]
Path operations	Comprehensive	Basic
Shorthand/smooth path commands	✗	✓
Closepath command	✗	✓

Continued on the next page.

	SVGELEMENTS	SVGLAB
Gradients	✗	✓
Clipping paths	✓	✓
Animations	✗	✓
Typed	✓	✓
Runtime validation	✗	✓
Python compatibility	≥ 3	≥ 3.10
Top-level dependencies	0	14
License	MIT	MIT

Perhaps the greatest difference between the two libraries lies directly in their general focus.

SVGLAB is primarily focused on *lossless manipulation* of SVG documents: the goal is to parse an SVG document, provide the user with the means to manipulate a part of it, and then serialize the document back to its original representation. In particular, the library strives to preserve the original semantics of the document where possible (unless the user explicitly changes them by manipulation).

In contrast, SVGELEMENTS' primary focus is on *extracting* geometric information from SVG documents:

> *"The goal is to successfully and correctly process SVG for use with any scripts that may need or want to use SVG files as geometric data."*
>
> — SVGELEMENTS [18]

Instead of relaying the semantics of the document and the original intent of the document's author directly to the user, the library attempts to simplify the document by techniques such as automatic reification upon parsing or resolving embedded images. This is done to make the document easier to work with, with the assumption that the user does not care about the actual SVG content, but rather the geometric information it contains.

SVGELEMENTS' subsystem for paths is based on SVG.PATH and SVGPATHTOOLS and thus offers similar functionality [18].

SVGELEMENTS' compliance with the SVG specification is very limited, and the library supports only a handful of the elements and attributes defined by the specification.

Chapter 6

Future work

Despite our efforts described in Chapter 4, we consider our implementation to be far from feature-complete. Therefore, we suggest several possible directions for future work.

6.1 CSS support

In SVG, elements can be styled using CSS. CSS properties have the same name as the corresponding SVG attributes and typically have the same syntax, but may occasionally offer more powerful functionality [2]. For this reason, we consider it valuable to implement proper support for parsing, manipulating, and serializing CSS embedded in SVG.

6.2 Optimization support

Our implementation already provides a basic means of optimizing SVG documents using formatting options, reification, and rescaling of the coordinate system. However, these features need to be used manually, and the results strongly depend on the options used.

To address this limitation, we propose developing a clear and unified interface that would allow automatic optimization of SVG documents based on the application of various optimization techniques.

Along with this change, it is naturally desirable to extend our current set of optimization procedures with, for example, some of the following advanced techniques:

- removing unused definitions,
- removing redundant attributes,

- removing invisible elements,
- simplifying and approximating path data,
- converting between basic shapes and their equivalent path representation (whichever is shorter),
- composing transformations in a transformation list into a single `matrix()` form.

We believe that the proposed changes would bring the library in line with other SVG optimization software, such as the popular SVGO [40] project.

Chapter 7

Conclusion

In this paper, we have presented SVGLAB, a new open-source Python toolkit for the programmatic manipulation of SVG documents.

Containing the definitions of 80 SVG elements and 295 attributes, the library offers comprehensive compliance with the SVG 1.1 specification [2], supporting all elements and attributes defined therein.

The library provides robust functionality for parsing, modifying, and writing SVG documents and a wide range of formatting options that can be used to format or optimize SVG documents. Additionally, the library contains advanced functionality such as reification, coordinate system rescaling, and calculation of element bounding boxes and segmentation masks.

By leveraging type hints and the PYDANTIC [19] validation library, our implementation offers both strong type safety and dynamic validation during runtime.

We have shown that within the entire Python ecosystem, only one project comes close to matching the extensive functionality of SVGLAB. Furthermore, our implementation is unparalleled in its comprehensive compliance with SVG specifications and further distinguishes itself by introducing novel solutions to existing limitations, such as support for shorthand path commands and customizable output formatting.

Finally, we have outlined several possible directions for future work.

Appendix A

Thesis archive contents

The digital thesis archive[14] contains the following items:

- `thesis.pdf`: the text of this thesis as a PDF document,
- `svglab.zip`: the complete source code of the SVGLAB library.

The ZIP archive with the source code contains, among others, the following:

- `svglab`: the SVGLAB Python package,
- `tests`: the test suite,
- `LICENSE`: the text of the MIT license,
- `README.md`: installation and usage instructions.

[14] Available at `https://is.muni.cz/th/ddfps/`.

Appendix B

Installation manual

B.1 Prerequisites

The library requires the presence of the CPython Python interpreter in version 3.10 or higher on the host system. We have not identified any other runtime dependencies.

B.2 Installation

Tagged releases are hosted on the Python Package Index (PyPi)[15] and can be conveniently installed with pip:

```
$ pip install svglab
```

Alternatively, it is possible to install the library directly from the latest source code available in the GitHub repository[16]:

```
$ pip install git+https://github.com/reznakt/svglab.git
```

[15] https://pypi.org/project/svglab/
[16] https://github.com/reznakt/svglab/

Appendix C

Proofs and derivations

Theorem 1 (Swapping the order of scale and skewX)**.**

$$\text{scale}\left(s_x, s_y\right)\text{skewX}(\alpha) = \text{skewX}\left(\arctan\left(\frac{s_x}{s_y}\tan\alpha\right)\right)\text{scale}\left(s_x, s_y\right)$$

Proof. Consider the following two transformations:

$$\text{scale}\left(s_x, s_y\right)\text{skewX}(\alpha), \tag{C.1}$$

$$\text{skewX}(\beta)\,\text{scale}\left(s_x, s_y\right). \tag{C.2}$$

Our goal is to find the value of β for which the two transformations are equal. By the definitions of scale and skewX [2, Sec. 7.4], we have

$$x' = s_x x + s_y y\tan\alpha \qquad \text{for scale}\left(s_x, s_y\right)\text{skewX}(\alpha), \tag{C.3}$$

$$x' = s_x x + s_x y\tan\beta \qquad \text{for skewX}(\beta)\,\text{scale}\left(s_x, s_y\right). \tag{C.4}$$

In both scenarios, $y' = y$; therefore, we will focus on the x coordinate. By comparing both expressions, we obtain the following equation, which we solve for α:

$$s_x x + s_y y\tan\alpha = s_x x + s_x y\tan\beta \tag{C.5}$$

$$s_y\tan\alpha = s_x\tan\beta \tag{C.6}$$

$$\tan\alpha = \frac{s_x}{s_y}\tan\beta \tag{C.7}$$

$$\alpha = \arctan\left(\frac{s_x}{s_y}\tan\beta\right) \tag{C.8}$$

However, it is important to remember that in SVG, it is the *coordinate system*, rather than the element, that is transformed [2, Sec. 7.4]. Therefore, what we

have, in fact, calculated is the new value of β in terms of α, as desired:

$$\beta = \arctan\left(\frac{s_x}{s_y}\tan\alpha\right).$$

□

The other identities described in Section 4.6.2 can be derived in the same manner.

Lemma 1.1 (Invariance of curve closure under affine transformations)**.** Let $\gamma\colon [0,1] \to \mathbb{R}^n$ be a parametric curve, and T be an invertible affine transformation. $T \circ \gamma$ is closed if and only if γ is closed.

Proof. First, assume γ is closed. By definition, we have $\gamma(0) = \gamma(1)$. Applying T to $\gamma(0)$ yields the following:

$$T(\gamma(0)) = A\gamma(0) + \vec{b} = A\gamma(1) + \vec{b} = T(\gamma(1))$$

Because $T(\gamma(0)) = T(\gamma(1))$, $T \circ \gamma$ is closed. Therefore, γ is closed $\implies$ $T \circ \gamma$ is closed.

We will now approach the problem from the opposite direction. Suppose $T \circ \gamma$ is closed and thus $T(\gamma(0)) = T(\gamma(1))$. Since T is invertible, applying T^{-1} to $T(\gamma(0))$ gives

$$\underbrace{T^{-1}(T(\gamma(0)))}_{\gamma(0)} = CT(\gamma(0)) + \vec{d} = CT(\gamma(1)) + \vec{d} = \underbrace{T^{-1}(T(\gamma(1)))}_{\gamma(1)}$$

we see that $\gamma(0) = \gamma(1)$, and therefore γ is also closed.

We have shown that

- γ is closed $\implies$ $T \circ \gamma$ is closed, and,
- γ is closed $\impliedby$ $T \circ \gamma$ is closed.

Therefore, γ is closed $\iff$ $T \circ \gamma$ is closed, as desired. □

Theorem 2 (Invariance of curve closure under compositions of affine transformations)**.** Let $\gamma\colon [0,1] \to \mathbb{R}^n$ be a parametric curve, and $\{T_i\}^n$ be a sequence of invertible affine transformations. Then

$$T_1 \circ T_2 \circ \ldots \circ T_n \circ \gamma$$

is closed if and only if γ is closed.

Proof. Follows immediately from Lemma 1.1 and the fact that affine transformations are closed under composition [41, p. 34] □

References