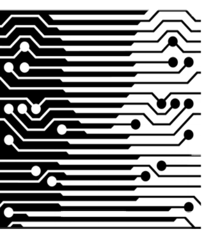
An Automatic Marker for Vector Graphics Drawing Tasks

University of Cape Town

A dissertation submitted to the department of Computer Science, Faculty of Science at the University of Cape Town in partial fulfilment of the requirements for the degree of Master of Science in Information Technology.

By
Tristan Bunn

Supervised by
Patrick Marais



August 2016

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

I know the meaning of plagiarism and declare that all of the work in this document, save for that which is properly acknowledged, is my own.

Signed by candidate

Signature Removed

Acknowledgements

I would like to express my heartfelt thanks to the following people:

- My supervisor, Patrick Marais, for taking on a multimedia design undergraduate and providing such invaluable guidance in my new field of study (and career-space).
- Springlab for taking me on as a front-end developer – mid-thesis – and allowing me the opportunity to learn so much more, in addition to providing me time off to complete this project.
- Vega Multimedia Design students for inspiring me to study further myself.

This document was created and compiled using free and open source software on a GNU/Linux operating system. The LibreOffice productivity suite was used to edit and compile the final document, with the DejaVu font family providing the typefaces used throughout. The vector graphics – which constitute the vast majority of the illustrations, graphs, and diagrams provided – were created using Inkscape; while all of the raster graphic requirements were handled by GIMP.

Abstract

In recent years, the SVG file format has grown increasingly popular, largely due to its widespread adoption as the standard image format for vector graphics on the World Wide Web. However, vector graphics predate the modern Web, having served an important role in graphic and computer-aided design for decades prior to SVG's adoption as a web standard. Vector graphics are just as – if not more – relevant than ever today. As a result, training in vector graphics software, particularly in graphic and other creative design fields, forms an important part of the skills development necessary to enter the industry.

This study explored the feasibility of a web application that can automatically mark/assess drawing tasks completed in popular vector graphics editors such as Adobe Illustrator, CorelDRAW, and Inkscape. This prototype has been developed using a collection of front-end and back-end web technologies, requiring that users need only a standards-compliant, modern web browser to submit tasks for assessment.

Testing was carried out to assess how the application handled SVG markup produced by different users and vector graphics drawing software; and whether the assessment/scoring of submitted tasks was inline with that of a human marker. While some refinement is required, the application assessed six different tasks, submitted eleven times over by as many individuals, and for the greater part was successful in reporting scores in line with that of the researcher.

As a prototype, serving as a proof of concept, the project proved the automatic marker a feasible concept. Exactly how marks should be assigned, for which criteria, and how much instruction should be provided are aspects for further study; along with support for curved path segments, and automatic task generation.

Table of Contents

Chapter 1: Introduction **1**

1.1 Motivation ... 1

1.2 Assumptions ... 2

1.3 Scope and Limitations ... 2

1.4 Goals ... 3

1.4.1 Research Questions ... 3

1.5 Approach and Solutions ... 4

1.6 Organisation of this Dissertation ... 4

Chapter 2: Background **6**

2.1 Vector Graphics Formats ... 6

2.1.1 SVG ... 8

2.1.2 Proprietary Vector Graphic File Formats ... 8

2.2 Parsing XML ... 9

2.2.1 Parsing: A Basic Introduction ... 10

2.3 Parsing an SVG ... 11

2.4 Shape Recognition ... 12

2.4.1 Detecting Shapes in SVG Markup ... 13

2.4.2 Two-Dimensional Object Recognition Algorithms ... 16

2.4.3 Computer Vision 19
2.5 Shape Comparison 19
2.5.1 Congruence and Similarity 19
2.5.2 Shape Similarity Algorithms 20
2.5.3 Automatic Marker Approach 21
2.6 Comparing Colours 23
2.6.1 Unspecified Colour Properties 25
2.7 Matching Non-Linear Data Structures 25
2.8 Web Development Languages and Frameworks 27
2.8.1 Client- and Server-Side 27
2.8.2 Front-End Development 28
2.8.3 Back-End Development 28
2.8.4 Web Development Frameworks 29
2.9 Online Assessments 32
Summary 32

Chapter 3: Framework **34**

3.1 Application Design 34
3.1.1 Goals 34
3.2 Development and Constraints 35
3.2.1 Hardware 35
3.2.2 Software 36
3.2.3 Browser Testing Details 36
3.2.4 Client- and Server-Side Architecture 37
3.2.5 Development Methodology 38
3.3 User Interface 38
3.3.1 Interface Themes 40

3.4 Features 40
3.4.1 Tolerance 40
3.4.2 Scoring 41
3.4.3 Absolute and Relative Value Comparison 42
3.4.4 Relative to Absolute Coordinates 42
3.4.5 Rules for Dealing with Extra Attributes and Shapes 43
3.4.6 Font Family 45
3.5 Evaluation 45
Summary 45

Chapter 4: Evaluation **47**

4.1 Vector Graphic Drawing Tasks 47
4.1.1 SVG Elements 48
4.1.2 SVG Attributes 50
4.2 SVG Features Testing 52
4.2.1 Recreated Drawing Task Files 52
4.2.2 Document Settings 53
4.3 Marker Capabilities and Limitations Evaluation 55
4.4 Prescribed Tasks Survey 55
4.4.1 Survey Preparation 56
4.4.2 Prescribed Tasks 57
4.4.3 Questionnaire 58
4.5 Browser Compatibility 58
4.6 Measurements 59
4.7 Results 61
4.7.1 SVG Features Testing Results 61
4.7.2 Marker Capabilities and Limitations Evaluation Results 63

4.7.3 Prescribed Tasks Survey Results 65

4.7.4 Questionnaire Results 70

Summary........ 72

Chapter 5: Conclusion **75**

5.1 The Automatic Marker for Vector Graphics Drawing Tasks 75

5.1.1 Study Goals 75

5.1.2 Shape Recognition and Comparison 76

5.1.3 Research Questions and Results 76

5.1.4 Web Application Development and Constraints 77

5.2 Future Work 77

Bibliography

Appendices

Appendix A: SVG Features Datasheet

Appendix B: Recreated Drawing Task Parameters

Appendix C: Automatic Marker Instructions

Appendix D: Prescribed Tasks Marking Rubric

Appendix E: Prescribed Task Feedback Questionnaire

Appendix F: Automatic Marker Library Source Code

Appendix G: Web Application Screenshots

List of Figures

Figure 1: Smiley and duck silhouette. 2

Figure 2: Enlarged vector (lower left) and raster (right) graphics. 7

Figure 3: Challenges detecting overlapping shapes. 13

Figure 4: Six-point and three-point triangles. 17

Figure 5: Shape similarity cases. 21

Figure 6: Application architecture. 37

Figure 7: Interface wireframe. 39

Figure 8: Vector drawing tasks for element testing. 49

Figure 9: Miter join (lower left); and bevel (right) or possible low miter-limit join. 51

Figure 10: Vector drawing tasks with attributes applied. 52

Figure 11: Graph: Comparing researcher submissions to automatic marker. 64

Figure 12: Graph: star graphic – auto-marker and researcher comparison. 66

Figure 13: Graph: triangle graphic – auto-marker and researcher comparison. 67

Figure 14: Graph: zigzag graphic – auto-marker and researcher comparison. 67

Figure 15: Graph: smiley graphic – auto-marker and researcher comparison. 68

Figure 16: Graph: text graphic – auto-marker and researcher comparison. 69

Figure 17: Graph: grid graphic – auto-marker and researcher comparison. 69

Figure 18: Duplicate curves drawn using different control points. 78

List of Tables

Table 1: Perimeter to area ratios. (source: sureshemre.wordpress.com) 18

Table 2: Calculating Scores. 59

Table 3: Summary of different editor markup. 62

Chapter 1: Introduction

Of the fundamental digital skills that are essential for any modern graphic design student to master, working with vector graphics is arguably one of the most important. Unlike raster graphics, vectors are resolution-independent, meaning that they can be scaled to any size without experiencing a loss in quality. Fonts are typically resolution independent: no matter what level of magnification is selected, characters will always render crisply without pixelation artefacts.

1.1 Motivation

Graphic design, however, is not the only domain to make extensive use of vector graphics. Design disciplines such as web design, multimedia design, architecture, and engineering all rely on software such as *Adobe Illustrator*, *CorelDRAW*, *Inkscape*, and various CAD-type applications – all of which are vector-graphic-based.

It follows that most tertiary (and often secondary) courses dealing in areas of design include some digital vector graphics training. As ongoing assessment is key to gauging and pacing learning, vector graphics tasks are set to measure a student's ability to operate such software. Although automatic markers of various types are widely utilised for marking everything from multiple-choice tests to computer programming tasks, an extensive literature review revealed no such system for vector graphics.

This project examines the feasibility of developing a basic vector graphics testing and marking system, relying on SVG files for submission, processing, and assessment. The SVG specification is an open standard, developed and maintained by the World Wide Web Consortium, based on markup/XML, and well-supported by both editors and web browsers.

1.2 Assumptions

With a plethora of devices and operating systems available today, a web application seems the most practical solution given the task at hand. Rather than investigating which platform(s) could be more/most suitable, this evaluation focuses exclusively on exploring the languages and frameworks available within the domain of *web/browser-based apps* – which can be defined as software that runs in a web browser, typically written using a combination of JavaScript, HTML, and CSS.

In explaining some of these languages, Chapter 2 (Background) introduces – but assumes some rudimentary understanding of – basic programming concepts, especially regarding XML syntax.

1.3 Scope and Limitations

As establishing just how accurately an irregular shape has been recreated requires more complex algorithms, the automatic marker tests only for criteria that can more easily be ascertained directly from XML/SVG markup – for example: regular shapes such as squares, circles, stars; the approximate positions thereof; and properties like fill-colour, and outline width and style. To clarify, in Figure 1 the proposed automatic marker can assess a basic 'smiley' (a circle for a face; two solid discs for eyes; a line for a mouth), but the silhouette of a duck is beyond this prototype's limits:

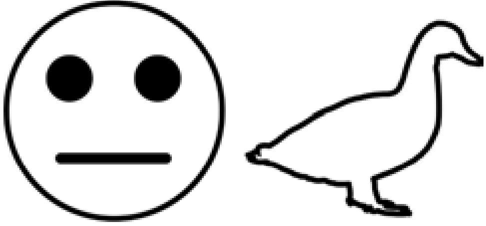


Figure 1: Smiley and duck silhouette.

The system has been developed using a combination of client- and server-side web development technologies, but relies on the client-side component for the actual assessment algorithms – whereas the server-side is responsible for serving content and storing results. In testing the automatic marker, subjects utilised both a browser and vector-graphics editor simultaneously on a typical desktop/laptop system, so it was not

necessary to develop a fully mobile-compliant web application. However, because HTML, CSS, and JavaScript – not to mention, SVG – rendering support can vary between web browsers, it has been ensured that the web application supports a broad enough range of the most popular browsers, namely: Google Chrome, Microsoft's Internet Explorer, Mozilla Firefox, and Opera. The operating systems on which the browser tests were carried out include versions of Microsoft Windows, Mac OS, and Debian Linux.

1.4 Goals

The research required that an automatic marker for vector graphics drawing tasks be developed. Given the scope and time-frame of the research, it has not been developed beyond a prototype version. The software's primary goal was to test the feasibility of such software – in this case, taking the form of a web application – that has been made possible by recent developments in browser technologies, and increasingly widespread SVG capability in vector graphics editors.

This project is exploratory in nature, taking a qualitative approach to collecting and analysing data, and serves more as a *proof of concept*, while exposing any challenges and findings that could influence further work in the area. More advanced features involving more complex shape recognition (see the duck in Figure 1, above) could be considered in further research, but remain beyond the scope of this study.

1.4.1 Research Questions

In order to research how effective and useful an automatic marker for vector-graphics drawing tasks can be, the following questions are answered:

1. Which SVG properties are exported consistently and reliably enough across popular vector graphics drawing software, in order to be assessed successfully by the automatic marker?

2. What web programming languages – and additional frameworks – will prove most effective for developing the proposed automatic marker?

3. Once developed and tested, how useful is the web application for testing and marking purposes?

1.5 Approach and Solutions

By harnessing the various new technologies available in HTML5, server- and client-side scripting languages, and development frameworks thereof, a prototype web application has been developed to run in modern, standards-compliant web browsers. This automatic marker prototype assesses basic drawing tasks, which – for the purposes of this study – have been rendered in three popular vector graphics editors, namely: Adobe Illustrator, CorelDRAW, and Inkscape.

To establish the effectiveness of the prototype, some initial testing and research helped define the SVG elements and attributes required for the automatic marker to support. Then, with the same individual subject – in this case the researcher – recreating the drawing tasks, the differences in markup produced by the vector graphics editors has been compared. Following this, the researcher's SVG files were submitted to the web application for assessment. However, further testing was required as the researcher followed the most similar drawing approach possible in all three editors.

In order to test the automatic marker against other potential approaches to the drawing tasks, multiple subjects have also submitted their attempts. Although the subjects interacted with the prototype's interface, at its state of development the interface was very basic; as this aspect was not an area of focus in the study. However, despite its primitive appearance, it was assumed intuitive enough to operate after a basic briefing, and a subsequent questionnaire allowed the opportunity for any of the subjects to reveal whether or not they were impeded or confused by it.

1.6 Organisation of this Dissertation

For an overview of how this dissertation is organised, provided below is a brief summary of the chapters to follow this one:

Chapter 2: Background

This chapter is comprised primarily of a literature review covering vector graphics formats, web development languages and frameworks, and online assessments. Special attention is paid to the SVG specification, as this forms an integral component of the automatic marking application.

Chapter 3: Framework

Beginning with the different approaches and solutions proposed for the automatic marker, this chapter discusses how the application has been designed, developed, and deployed. More specifically, the formats, languages, and frameworks reviewed in the Background chapter are expanded upon in order to explain how they have been practically implemented in the functional prototype.

Chapter 4: Evaluation

The tests used to evaluate the application – including the tasks devised for the sample group(s) – are expanded upon here, along with important considerations around test criteria and metrics. Reporting on the results and findings gathered from the testing phase, the data and findings are presented here before concluding in Chapter 5 (Conclusion).

Chapter 5: Conclusion

Overviewing the initial aims of this project, this chapter discusses to what extent these were met. Problems that have been encountered, and the implications thereof shed light on what suggestions could be made for future work in this area of research.

Chapter 2: Background

This chapter is comprised primarily of a literature review covering vector graphics formats, web development languages and frameworks, and online assessments. Special attention is paid to the SVG specification, as this forms an integral component of the automatic marking application.

2.1 Vector Graphics Formats

To explain vector graphics formats, it is best to begin with a description of *raster* display devices. To begin: most digital displays make use of pixels to render images and video. A *pixel*, or *picture element*, is the basic unit that makes up a digital image; it represents a single hue or shade. The number of pixels that constitute a display determine its *resolution*. For example, the 23-inch monitor on which this document was edited displays at a resolution of 1920 × 1080 pixels; these small squares of colour (discernible up-close) in turn make up the vivid images on screen. As it is a full-colour display, the colour of every individual pixel is mixed from red, green, and blue sub-pixels illuminated at varied intensities. This particular device makes use of an array of LEDs (light emitting diodes) for pixels, whereas older CRT (cathode ray tube) technologies relied on electron guns to stimulate phosphorescent dots [Shirley et al. 2010]. A raster graphic is an image represented by a grid of such pixels.

Figure 2 contrasts vector graphics with raster graphics. The small five-pointed star at the top left has been enlarged to illustrate the difference. Notice that the scaled-up graphic on the left is crisp and displays no loss in quality; this is vector version. In the raster version on the right, the pixels along the star's outlines are clearly visible:

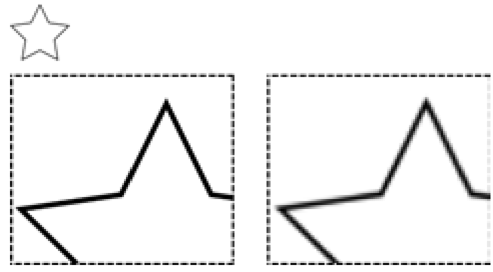


Figure 2: Enlarged vector (lower left) and raster (right) graphics.

Each vector graphic is comprised of a series of points that in turn can be connected to form lines, shapes, and other polygons. In the case of the star, there are five outer-points and five inner-points. Although the points are connected by straight-lines, if desired, curved lines could be used instead – thus creating a flower/petal-like design. Each graphical object is thus represented by mathematical expressions, allowing for them to be scaled with no loss in quality [Dawber 2013].

As a raster (or bitmap) graphic is scaled up, its pixels increase in size and as a result become more apparent squares of colour. Of course, computer displays also rely on a high resolution grid of pixels, so ultimately a vector graphic must be displayed as pixels – however this happens only once the final recomputed, scaled vector image has been processed [Adobe Systems Incorporated 2010].

Adobe Illustrator and CorelDRAW are both vector graphic editors, popular with graphic designers. Each application utilises proprietary file formats, easily identified by their AI and CDR file extensions. *SVG* and *VML* vector graphic file formats are unlike Adobe Illustrator or CorelDRAW files in that they are intended for direct rendering in a web browser – much like JPG, GIF, or PNG files. Dawber [2013] introduces the *W3C*[1] (an abbreviation for *World Wide Web Consortium)* SVG standard, drafted in 2001. The W3C is responsible for the specification and adoption of new web standards, and SVG support has since been implemented in all major modern web browsers. VML actually pre-dates SVG, having been submitted to the W3C by Microsoft in 1998, but has since been deprecated in favour of the latter. Although SVG struggled to gain traction initially – with some experts still sceptical in 2004 [Udell 2004] – it appears that the

1 http://www.w3.org/

format is now more popular than ever, with proprietary alternatives like *Flash* continuing to lose market share [McMillan 2015].

It is important at this point to differentiate between two-dimensional (2D) and three-dimensional (3D) vector formats. 3D file formats including *COLLADA* – along with other popular proprietary formats such as Autodesk 3D Studio's *3DS* – all plot coordinates in a virtual three-dimensional space. This additional axis allows for lighting, perspective, and other three-dimensional characteristics to be processed before a final rendering is displayed on-screen as pixels [Durand 2010]. This project does not deal with such graphics, instead focusing exclusively on the 2-dimensional variety.

2.1.1 SVG

A primer adapted from the W3C's SVG drafts [Eisenberg and Bellamy-Royds 2014] provides a thorough breakdown of this specification and how it is to be used. The first thing to note is that SVG is based on XML. Anyone versed is XML should recognise the syntax of the following SVG code snippet:

```
<circle cx="5" cy="6" r="4" stroke="tan" stroke-width="3" fill="red" />
```

The “circle” tag, and its accompanying attributes, will render a red circle with: a centre x-y coordinate of 5 and 6 pixels respectively; a radius of 4 pixels; and a tan-coloured stroke that is 3 pixels thick.

Aside from experiencing no loss in quality through scaling, vector graphics have a number of other advantages when deployed in the Web environment. Firstly, vector graphics are usually far smaller in file-size than raster graphics. Secondly, the individual elements in SVG files can be manipulated using *SMIL* and *JavaScript* – allowing for animation and interactive control. For example, one can create games using 'pure' SVG vector graphics and JavaScript – something that until fairly recently was only possible with 3rd party web browser plug-ins, such as Flash. Such techniques also allow developers to harness the client-side for image processing, which reduces server and network demands [Duce et al. 2002].

2.1.2 Proprietary Vector Graphic File Formats

Early releases of vector graphic editors, including *Macromedia Freehand* and Adobe Illustrator, focused largely on desktop publishing for print output [Seddon and

Waterhouse 2009]. But with the growth of the Internet, vector graphics have now entered the World Wide Web through formats such as Flash and SVG.

In a comprehensive overview of the SVG format, Dailey [Dailey et al. 2012] reviews various editors in use by designers, including industry leading software like Adobe Illustrator[2]. While Illustrator maintains its proprietary AI file format, it is capable of exporting and editing SVG as well. Inkscape[3] (a popular open source Illustrator alternative) uses SVG natively, albeit an extended version of that specified by the W3C. After initially testing a few examples, in most cases Illustrator and Inkscape designs appeared to export successfully to the standard SVG specification. Some features, like stylised strokes (which were reinterpreted as normal, solid outlines), are translated by the export/save function, and if desired, retained in the SVG markup. As XML is extensible, unrecognised attributes and tags are simply ignored – provided the markup is well-formed. These limitations are explored in further detail in Chapters 3 (Framework) and 4 (Evaluation).

2.2 Parsing XML

A *Markup language* provides a means of annotating a document, enabling a computer to manipulate the encapsulated content using the annotations provided – essentially distinguishing instructions from the text (or content). There are many such languages, of which XML is one and they all vary somewhat in rules, style, and syntax. However, tags are probably one of their most recognisable features. To provide a concrete example, consider the following XML *element*:

```
<book>The Catcher in the Rye</book>
```

Notice that the book's title (in this case *The Catcher in the Rye*) is surrounded by an opening and closing `book` *tag*. As illustrated, in XML the closing tag should be differentiated using an additional slash (`/`) character.

Attributes can be added to opening tags; for example, the book's ISBN number:

2 http://www.adobe.com/products/illustrator/

3 https://inkscape.org/

```
<book isbn="9780316769310">The Catcher in the Rye</book>
```

However, as data grows more complex, tags may be nested within other tags. As white-space is not significant to the parser, the developer is welcome to make their code more legible using tabs, spaces, and new lines:

```
<book isbn="9780316769310">
  <title>The Catcher in the Rye<title>
  <author>J. D. Salinger</author>
</book>
```

One can define/devise any tag names – for example `<writer>` instead of `<author>` – and accompanying attributes provided they adhere to some basic rules (for example, the tag name cannot contain space characters). But using an existing specification/schema means that the markup can immediately be interpreted for some predefined purpose, such as a web-page (HTML) or graphic (SVG) [Evjen et al. P6–27].

2.2.1 Parsing: A Basic Introduction

Parsing, in a computing context, involves analysing files in order to identify and derive the component parts within [Fawcett and Danny 2012]. For example, CSV files contain a string of comma-separated values:

```
Smith,Thomas,21,Cape Town
Matema,Sally,36,Pretoria
Pilay,George,52,Durban
Godfrey,Horatio,23,East London
```

In this example, to extract an individual's details the parser is first prompted by a new line-feed to locate, firstly, a surname; and then by the commas to derive in sequence a: first name; age; and finally, a city. Parsers follow different rules for different file-types. As a case in point, the XML equivalent of such a CSV could read as follows:

```
<person>
  <surname>Smith</surname>
  <name>Thomas</name>
  <age>21</age>
  <city>Cape Town</city>
<person>
```

```
<person>
  <surname>Matema</surname>
  <name>Sally</name>
  <age>36</age>
  ...
```

To keep things simple, these examples use simple text-based content – but parsers do exist for all types of files, as any application that reads input must have a parser of some kind. In the XML example the parser relies on opening and closing tag pairs (as well as quoted attributes and a few other delimiters) to extract the relevant data. As listed by Evjen [Evjen et al. 2007], numerous XML parsers exist: Microsoft's *XML Parser* and its accompanying API, *SAX2*; Mozilla's *XML Parser*; Apache's *Xerces*; IBM's *Xml4j*; and *Expat*. This means that writing one's own parser is not always necessary, especially for XML derivatives.

2.3 Parsing an SVG

When dealing with HTML (another markup language), the various elements in the markup are visually rendered as a web-page [Duckett 2011]. For example, the `h1` tag encapsulates a heading – and its contents are thus rendered bolder and larger than any surrounding text. Raster images rely on the `img` tag – however, as the content is not comprised of plain text, the image tag instead points to a binary image file (typically a JPG, GIF, or PNG):

```
<html>
<body>
  <h1>Here is a circle</h1>
  <img src="circle.jpg" />
</body>
</html>
```

A significant difference with an SVG file is that it need not be linked; rather, the SVG markup (contrasted in bold) can be embedded within the HTML [Gasston 2013]:

```
<html>
<body>
  <h1>Here is a circle</h1>
```

```
  <svg width="100" height="100">
    <circle r="40" stroke="red" stroke-width="4" fill="blue" />
  </svg>
</body>
</html>
```

As Gasston explains further, this means that the SVG now becomes part of the HTML *Document Object Model* (or DOM). The DOM can be used to address any part of a web-page. For example, to extract the text within the first `h1` tag of the document, the following DOM path can be used:

```
document.getElementsByTagName('h1')[0].textContent;
```

To change the text colour of the heading to red, this could instead be written as:

```
document.getElementsByTagName('h1')[0].style.color='red';
```

As a result of including the SVG markup within the DOM, a web-browser's built-in features can be utilised for parsing and manipulating the SVG code. For example, the circle markup (with its red stroke and blue fill) can have the stroke colour switched to red using the following line (bolded) of JavaScript:

```
<svg width="100" height="100">
  <circle r="40" stroke="red" stroke-width="4" fill="blue" />
</svg>
...
document.querySelector('svg circle').setAttribute('stroke', 'red');
```

While it may be necessary to parse other XML documents in order to format them as part of an HTML web-page, the browser needs no extra instruction on how to render the SVG specification and its various elements. Furthermore, like raster images, SVG images can also be linked using HTML's `<img>` tag [Niederst 2012].

2.4 Shape Recognition

As explained in section 2.1 (Vector Graphics Formats), the automatic marker deals only with two-dimensional vector graphics. Since 2D object recognition is complex and

broad field, the prototype presented in this project will only deal with a small subset of these concepts.

Ashbrook and Thacker [1998] cover a range of techniques to represent shape properties. After reviewing these, it becomes apparent that much of the processing ordinarily required to detect the edges of objects – which is a key step before any shape recognition and comparison can take place – is made far simpler through the SVG markup.

2.4.1 Detecting Shapes in SVG Markup

As a concrete example, consider testing whether a shape is a circle or not. Firstly, the shape should to be extracted from the surrounding area. Where there is sufficient contrast and separation between a shape and background, this a relatively straight forward task as the edges of shape are not obscured by any other shapes, nor are they lost due to colours blending together.

However, for the overlapping and varied circles shown in Figure 3, extracting the correct circle boundary information from a raster graphic is a complex task since one cannot simply trace edge pixels to extract each circle. So, simply counting the number of circles becomes challenging:

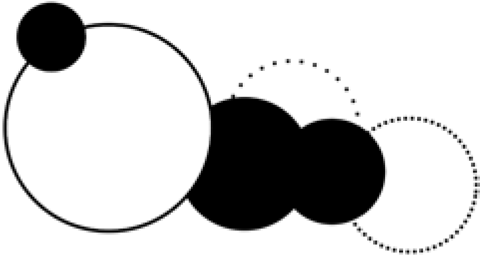


Figure 3: Challenges detecting overlapping shapes.

The SVG markup, however, contains a `<circle/>` tag for every circle rendered, along with all of its applied attributes. Using Inkscape to create the Figure 3 graphic, the following (abridged here) code is produced:

```
<circle
  style="fill:#000000;
```

```
        fill-opacity:1;
        stroke:none;
        stroke-width:3;
        stroke-miterlimit:4;
        stroke-dasharray:none;
        stroke-opacity:1"
      id="path4149-5-2-1"
      cx="303.38303"
      cy="267.10797"
      r="50.964397"
    />

    <circle
      style="fill:#ffffff;
        fill-opacity:1;
        stroke:#000000;
        stroke-width:2.59443474;
        stroke-miterlimit:4;
        stroke-dasharray:none;
        stroke-opacity:1"
      id="path4149"
      cx="197.01315"
      cy="255.00342"
      r="78.919327"
    />

    <circle
      ...
```

Note that simply by counting the number circle *tags* (the markup begins with the top-left circle, and progresses rightward), one can establish exactly how many there are.

Furthermore, two-dimensional collision detection can be used to detect overlapping shapes. The simplest approach is the *Axis-Aligned Bounding Box* method, which works by ensuring that no gap exists between any sides of two rectangles, as demonstrated in the following JavaScript example [Schwarzi 2012]:

```
var rectangle1 = {x:10, y:10, width:60, height:60}
var rectangle2 = {x:15, y:15, width:15, height:15}

if (rectangle1.x < rectangle2.x + rectangle2.width &&
   rectangle1.x + rectangle1.width > rectangle2.x &&
   rectangle1.y < rectangle2.y + rectangle2.height &&
   rectangle1.height + rectangle1.y > rectangle2.y) {
    // overlap detected
}
```

The method has its limitations as it does not precisely cater for anything other than rectangles. Although rectangles (or even rotated rectangles) could arguably be wrapped around any shape, this will result in a poor estimate of any overlaps in *complex* shapes.

Schwarzi covers another simple detection test: *Circle Collision*. As this technique is used for circles, rotation is not a factor. However it is limited to circles only and works by determining whether the distance between the centre points is less than the sum of the radii:

```
var circle1 = {x:10, y:10, radius:60};
var circle2 = {x:15, y:15, radius:15};

var distancex = circle1.x - circle2.x;
var distancey = circle1.y - circle2.y;

var distance = Math.sqrt(distancex*distancex + distancey*distancey);

if (distance < circle1.radius + circle2.radius) {
  // overlap detected
}
```

Mention of the more complex *Separating Axis Theorem* is also made [Schwarzi 2012], and this is capable of detecting collisions between any two convex polygons – shapes in which all interior angles are less than or equal to 180 degrees, e.g.: triangles, squares, and other convex polyhedral. To reduce processing overhead, broad phase algorithms such as a *sweep and prune* can be used to limit the number of checks for colliding pairs of shapes – so that: only if the bounding edges of any complex shapes happen to

overlap on a given axis, they can then be tested using more accurate yet expensive algorithms [Kelly 2012].

With no shortage of collision detection and physics libraries, such functions are easy to include in the proposed prototype. Shankar [2012] explains how to integrate the JavaScript implementation of the popular *Box2D* physics library – available for many other languages, including (originally) C++, Java, ActionScript, Haxe, C#, Lua, and D.

2.4.2 Two-Dimensional Object Recognition Algorithms

Using the *AForge.NET* library – a C# framework designed for developers and researchers in the fields of Computer Vision – Kirillov [2010] summarises some of the basic techniques for detecting different types of shapes. As his examples deal with raster graphics, the first step in the process utilises AForge to extract 'blobs' (or shapes) from the background for analysis. As explained in the previous section (2.4.1), this is not necessary for SVG files as the shape information can be extracted directly from the markup. Once extracted, one can apply algorithms for detecting circles, quadrilaterals, triangles, rectangles, squares, and equilateral triangles – for which a brief description of each technique is provided:

Circle Detection

To detect a circle, the algorithm verifies that all points are equidistant from the shape's centre. This recurring distance is effectively equal to the circle's radius, and provided all of the available points share a similar enough measurement for this, one can assume the shape is a circle.

Triangle Detection

Detecting triangles relies on first establishing where a shape's corners are located. Once three have been found a check is run to see if they fall within the edges of an assumed triangle.

However, as a shape's SVG markup can be utilised to count how many points exist, one can immediately establish if there are three. But checking the size of the angles is important as a single face of any triangle could be comprised of numerous points, provided they are arranged in a straight line (see Figure 4). This could cause problems for shape detection, so within a specified tolerance – say, less than 2 degrees – any near-straight angles will be disregarded 'corners'.

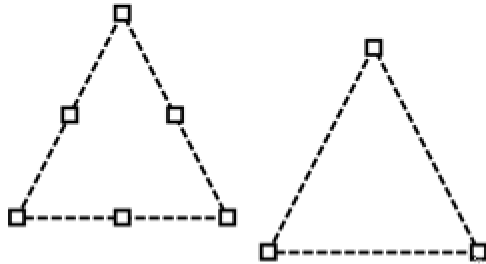


Figure 4: Six-point and three-point triangles.

JavaScript provides an arctangent method along with its Math class. Arctangent is the inverse trigonometric function of tangent (tan): so provided an angle, tangent will find the ratio of two sides of a right-angled triangle; arctangent, inversely, will find an angle given the ratio. This can in turn be used to create a function for checking the angle between any two lines [Stefanov 2014]:

```
function getAngle(ax, ay, bx, by) {
  var cy = by - ay;
  var cx = bx - ax;
  var theta = Math.atan2(cy, cx);
  theta *= 180/Math.PI;
  return theta;
}
```

Note that in this example radians are converted to degrees.

Quadrilateral Detection

As Kirillov [2010] explains further, detecting quadrilaterals involves, essentially, the same approach as triangles – the only difference being how the assumed shape's parameters are estimated. As before, the shape's SVG markup can again be used to count corners.

Once it has been established that a shape is a quadrilateral, further checks can be used to detect the sub-type: a rectangle; square; parallelogram; rhombus; or trapezoid. Moreover, as Table 1 illustrates, there is a specific relationship between the area and perimeter of any geometric shape, and one could even detect a shape on these measurements alone [Greig 2012].

Perimeter to area ratios considered, for quadrilaterals Kirillov suggests the following approach:

1. Check any two opposite sides to see if they are parallel; if so, the shape is at the very least a trapezoid;
2. if the remaining two sides are parallel, the shape is a parallelogram;
3. if these two sides are also perpendicular to their adjacent sides, the shape is a rectangle;
4. finally, check if any two adjacent sides are of equal length; if so, the parallelogram is a rhombus, or the rectangle is a square.

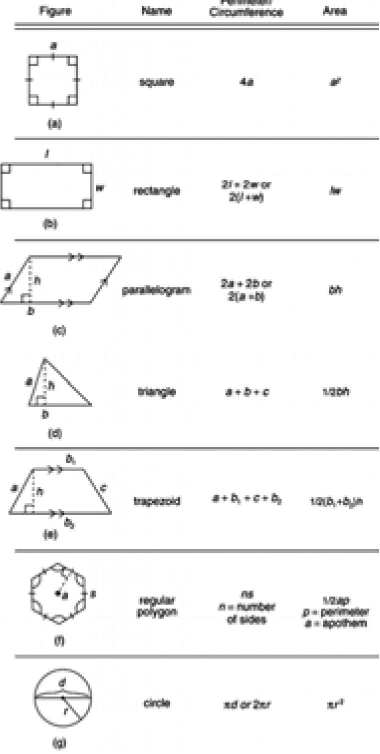


Figure	Name	Perimeter/ Circumference	Area
(a)	square	$4a$	a^2
(b)	rectangle	$2l + 2w$ or $2(l + w)$	lw
(c)	parallelogram	$2a + 2b$ or $2(a + b)$	bh
(d)	triangle	$a + b + c$	$1/2bh$
(e)	trapezoid	$a + b_1 + c + b_2$	$1/2(b_1 + b_2)h$
(f)	regular polygon	ns n = number of sides	$1/2ap$ p = perimeter a = apothem
(g)	circle	πd or $2\pi r$	πr^2

Table 1: Perimeter to area ratios.
(source: sureshemre.wordpress.com)

It should be added that, for checking straight lines and parallelism, some degree of inaccuracy should be accommodated as drawing such shapes manually will produce slightly distorted results.

2.4.3 Computer Vision

Bradski and Kaehler [2008] describe computer vision as “the transformation of data from a still or video camera into either a decision or a new representation”. There are many applications in use currently – from inspecting mass-produced goods to navigating unmanned vehicles, and even recognising and surveying people. This is more difficult than it sounds, probably because it something so inherently intuitive for human beings. Fortunately programming libraries like OpenCV – to which authors devote an entire book – provide developers with a collection of programming functions for such tasks.

Given that the requirements of the prototype are to process coordinates and attributes already defined by markup – and recognition is limited to basic shapes – implementing a computer vision library is not necessary at this stage.

2.5 Shape Comparison

With the coordinates of a shape acquired (section 2.4.1: Detecting Shapes in SVG Markup) the submitted result can be compared to the original. The accuracy is established using the shape comparison criteria programmed into the automatic marker. As there are numerous ways in which 'accuracy' – an application-dependant and often subjective matter – can be measured, the following concepts have been considered:

2.5.1 Congruence and Similarity

In geometry, *congruence* refers to two shapes being exactly equal; whereas *similarity* indicates that two shapes share the same proportions yet are rendered at different scales. For example, the architectural plans for a home are *similar* to the building they represent. However – in ascertaining whether or not shapes are congruent or similar – any combination of *reflection* (flipping the shape to create a mirror image), *translation* (moving the entire shape in a given direction), and *rotation* operations are permitted. It is important to note that while the lengths of any corresponding sides in similar shapes vary by a consistent factor, the corner angles will remain exactly the same; consider an equilateral triangle where all three angles are equal to 60 degrees no matter how large or small one draws the shape [Rock 2005].

Reflection operations can produce a result that appears substantially more 'inaccurate' (again, subjective and application-dependant) than something with some degree of rotation or translation. For example, consider a reflected triangle that points up rather than down. An inversion like this represents a poor effort for any subject attempting to recreate such a simple shape using vector graphics drawing software. Of course, a reflected hexagon could be indistinguishable from the original – so it is not a rule that translation or rotation will always be considered less 'inaccurate' than reflection.

Rock describes approaches for proving congruency and similarity, but these rely on coordinates far more precise than those dealt with by the automatic marker in this study. However, the concepts of reflection, rotation, and translation appear again in other areas, including the SVG specification itself.

2.5.2 Shape Similarity Algorithms

Shape similarity algorithms are less rigid in approach than the geometry rules discussed above. As has been alluded to already, the concept of similarity is application-dependent and often subjective. Potential algorithms to gauge similarity include: Hamming distance; Hausdorff distance; comparing skeletons; and Support Vector Machines [Skiena 2008].

To calculate a Hamming distance, the original and recreated shapes are placed as accurately as possible above one another. The Hamming distance is the size of the symmetric difference, where the symmetric difference is the difference between the union and the intersection of the covered image pixels – where a value of zero equates to identical. The most difficult part of this process is aligning the shapes – but in applications such as *optical character recognition* (OCR) this challenge is mitigated by harnessing the inherent grid formed by horizontal lines of text.

Calculating a Hausdorff distance requires finding a point along the recreated shape's boundary that is the maximum distance from a point on the original's. Using a dashed line to represent the recreated attempt, the star to the right in Figure 5 would have its Hausdorff value measured using the distance between the inner (solid line) and outer (dashed) corners of the lower-left point:

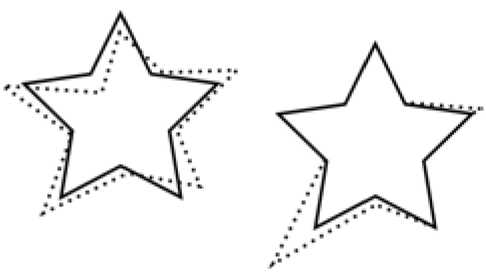


Figure 5: Shape similarity cases.

To contrast the difference between Hamming and Hausdorff distances: the left star would register as more accurate using a Hausdorff distance; whereas the right star registers as more accurate using a Hamming distance. Essentially the Hausdorff algorithm handles cases with boundary noise more effectively; while the Hamming algorithm is more effective in instances where there may be an odd spike or protrusion.

Comparing skeletons is another option. In this approach, both shapes are thinned until only narrow, branch-like lines remain of each. For example, a heart shape would be reduced to something resembling a “Y”. From here, the skeletons of each shape can be compared using the lengths and slopes of the constituent lines.

Support Vector Machines (SVM) rely on a large set of initial data – in this case shape comparisons – to train the program. With a subject comparing shapes and classifying them as accurate or inaccurate, the SVM creates its own classifications based on the incoming data. Of course, the success of such an approach is dependant on the amount of training and tweaking of the SVM. Presenting a case study involving a military application that attempted to distinguish cars from tanks, Skiena [2008] elucidates how programmers can grow unaware of the rules governing the classification system. When trained and tested using photographs and videos, the 'car-or-tank?' SVM proved highly successful in correctly identifying its targets, yet failed miserably in real-world testing. Initially vexed by the results, it was only after somebody pointed our the difference in lighting conditions (sun/cloud) that the developers could address the issue.

2.5.3 Automatic Marker Approach

However, with the aforementioned concepts considered, the automatic marker opts for a different approach: comparing the distance between the corresponding points in original and recreated shapes. This approach is preferable for a number of reasons:

1. Shape elements defined using `<circle>`, `<ellipse>`, and `<rect>` tags have attributes for height/width, and x/y coordinates. As such, they can be compared using these values – so no elaborate shape similarity algorithms are necessary.

2. For the other shapes defined using a series of points, consider how the subject interacts with the vector graphics drawing software: using a pen/Bézier tool, the shape is drawn in a dot-to-dot-like fashion; with the lines between each *anchor point* automatically connected in the process. As the automatic marker has access to these anchor point coordinates, they can be compared with those of the original. This is performed using the following steps: first, pairing-off shapes for comparison by detecting which are closest according to their centre-points; then, for each comparison pairing, checking whether the Hausdorff distance value falls within a given tolerance; if the Hausdorff distance is indeed less than the tolerance value, the shape is considered accurate enough.

3. At this point, the automatic marker is only assessing basic shapes. While including other comparison algorithms (Hamming distance; comparing skeletons; Support Vector Machines) could prove useful in time, this level of ability is beyond the scope of this study.

By comparing the original and recreated shapes, each property is assessed and a score assigned – the marking metrics of which are discussed in further detail in Table 2. Gauging roughly how closely related two shape elements are in terms of 'size' is relatively intuitive: calculate the bounding-box area of each and subtract the results to find the difference. Even more intuitive is comparing the thickness of a stroke: simply subtract the stroke-width of one shape from that of another. However, for any distances between the original and recreated anchor points, one must measure between two sets of x- and y-coordinates as the corners of the hypotenuse on a right-angled triangle. This in accordance to the Pythagorean theorem where the hypotenuse (the longest side) – or this case, the distance – is represented as “c” in the following formula [Slavin 1999]:

$$c=\sqrt{a^2+b^2}$$

2.6 Comparing Colours

When comparing how 'similar' two colours are, one must consider the system interpreting them – in this case, the *Human Visual System* (or HVS). For example, if a series of alternating black and white stripes are sufficiently narrow, the human eye simply blends them into some flat shade of grey. However, with a superior *visual acuity*, a computer will distinguish the individual stripes. As a case in point: taking such thresholds into consideration, programmers have reduced overhead in graphics processing by eliminating features that fall outside of human perception, i.e.: rather than processing the exemplar stripes, the computer simply deals with a single shape filled with an averaged colour [McNamara 2001]. Of course, when setting tasks to be marked by the automatic marker, anything involving a level of accuracy so high as to have to zoom-in to trace such delicate stripes/shapes can be avoided altogether. However, something like comparing colours is an essential feature and demands that the quirks of the HVS be taken into consideration.

To begin, colours of SVG elements are defined using an *RGB* (Red, Green, Blue) value. As discussed in Chapter 1 (Introduction), the colour of each pixel in a digital display is mixed using these three primary colours. As an example the colour red would be represented as the following RGB value: #FF0000 – with each *pair* of digits representing a red, green, and blue value respectively. As these are hexadecimal values, the FF pair indicates that the red value is set to maximum; while green is 00 – or simply 0 – and so is blue [Shirley et al. 2010]. Hexadecimal values can be converted to decimal values, and in this case [FF][00][00] would equal [255][0][0].

To compare the difference between two colours, one might assume that subtracting one RGB value from another would be an indicator of the distance between them. As there are three values in play, one could measure the euclidean distance in a three-dimensional space using the following formula [Slavin 1999]:

$$distance = \sqrt{R^2 + G^2 + B^2}$$

The measure of change in visual perception between two colours referred to as Delta E – usually represented as ΔE (Δ representing distance; and E, *Empfindung*[4]) – and is expressed on a scale, typically between zero (exact match) and one hundred (complete

4 *empfindung* is German for “sensation”

opposite). However, the RGB colour space is not uniform and this makes measurements using the euclidean formula inaccurate. The HVS is also more affected by certain values – with green being more dominant than red or blue – but one can compensate by adjusting the formula. Here the correct ratio has been factored in by multiplying the green value (G) by four making its effect the most prominent over the red (R) – which has been multiplied by three; and blue (B) – which has only been multiplied by two:

$$distance = \sqrt{3(R)^2 + 4(G)^2 + 2(B)^2}$$

However, even further adjustments must be made to accommodate for the fact that changes in darker colours are less noticeable than those in lighter colours [Mokrzycki and Tatol 2012]. The *International Commission on Illumination* has spent years developing algorithms to better match human perception. These are now available in a JavaScript library, named *DeltaE*[5], ideal for use in this project.

The DeltaE library makes available functions for implementing the International Commission on Illumination's (or CIE after its french name: *Commission Internationale de l'Éclairage*) dE- 76, 94, and 2000 formulas – produced in the years 1976, 1994, and 2000 respectively. dE00 is the most recent and considered the most accurate – although not without its faults. However, it is a vast improvement over dE76, which was not much more than a straight-forward euclidean distance calculation – that does not cater well for colour saturation. For example, a very-dark / almost-black red registers as “perceptibly different at a glance” (which it is not) to a very-dark / almost-black blue. It is important to note that the formula makes use of the *CIELAB* colour space, devised in 1976 along with dE76. In order to include all colours visible to the human eye, the CIELAB colour gamut is wider than that of RGB (and CMYK). Also written as *CIE L*a*b*, the model relies on three values: *L* for lightness; *a* for a red-green channel; and *b* for a yellow-blue channel. The automatic marker converts the RGB values to L*a*b values, and compares colours using the `getDeltaE00` function [Schuessler 2015]:

```
var delta_e_red_blue = DeltaE.getDeltaE00(
  {L: 54, A: 81, B: 70},   // red
  {L: 30, A: 68, B: -112}  // blue
);
```

5 https://zschuessler.github.io/DeltaE/

However, as Schuessler's documentation of the Javascript library explains: correctly speaking, the L*a*b values are converted to *L*C*h°* values for comparison purposes – a new development in dE94, where the hue is represented as an angle on a colour wheel rather than on L*a*b's red-green / yellow-blue apposed axes. In the above example, the `delta_e_red_blue` variable (the comparison result) is assigned a value of 55.57. Exactly matching colours would return a value of 0, and as a general guideline the ΔE values represent the following:

<= 1.0	Not perceptible by human eyes.
1 – 2	Perceptible through close observation.
2 – 10	Perceptible at a glance.
11 – 49	Colours are more similar than opposite.
100	Colours are exact opposite.

The automatic marker tolerance can be adjusted, so that, if for example, red and blue were close enough to be considered a match, anything below a value of 60 would be considered correct.

2.6.1 Unspecified Colour Properties

Shape attributes with no specified colour property, default to black. So the following line produces a black circle:

```
<circle r="40" stroke="red" stroke-width="4" />
```

as do these lines of markup:

```
<circle r="40" stroke="red" stroke-width="4" fill="#000000" />
<circle r="40" stroke="red" stroke-width="4" fill="black" />
<circle r="40" stroke="red" stroke-width="4" fill="rgb(0,0,0)" />
```

2.7 Matching Non-Linear Data Structures

To begin, consider *linear data structures*. In such arrangements, data is stored in a linear fashion and is traversed sequentially. This concept can be visualised as a table of entries, consisting of columns and rows, such as a list of names – or possibly a list of names with corresponding phone numbers and addresses. Examples include arrays, stacks, and queues.

HTML, XML, and SVG, however, represent *non-linear data structures* that can be visualised as a tree – with information stored at the many forks and end-points along it's 'branches'. For example, the HTML DOM, with a top-level `html` element, contains, in-turn, two child elements – `head` and `body` – which each possess multiple descendants of their own, which in-turn contain further child elements, and so forth [Miller and Ranum 2013]:

```
<html>
  <head>
    <title>...</title>
  </head>
  <body>
    <h1>...</h1>
    <p>
      <img />
      ...
    </p>
  </body>
</html>
```

Non-linear data structures, therefore, do not form a sequential or linear series, but a hierarchical model. *Graphs* are different to trees in that they resemble network (as opposed to hierarchical) models, with branches/paths that can be uni- or bi-directional. [Koutra et al. 2011]

Applying these concepts to the automatic marker, the general form of the problem takes that of matching non-linear structures, and the process of assessing an SVG file can abstracted in the following sequence of steps:

1. The objects – in this case the vector graphic paths and shapes – are extracted and defined (represented as a circle, rectangle, path, etc.) in order to be assessed.

2. This object data – complete with corresponding attributes – is then pre-processed and any spurious diversity is reduced, and if necessary, data disregarded – for example: extra shapes, corners, or proprietary extensions in the SVG markup.

3. Finally, some form of topology matching or graph similarity between the closest/matching candidates is measured to derive an appropriate score.

2.8 Web Development Languages and Frameworks

With a plethora of web development frameworks available for the task, it is best to begin by establishing which languages are suitable for this project and consider frameworks thereafter.

2.8.1 Client- and Server-Side

Website scripts can run on a user's device (client-side) or the server delivering the content (server-side). To program client-side scripts, developers rely on front-end languages like JavaScript or ActionScript. This means that the client's web browser must support the scripting language, which may sometimes require installing a plug-in – such as Flash to run ActionScript programs. For server-side scripting languages – like PHP, Python, Ruby or ASP – it is the server that runs the script, so no additional plug-ins need be installed client-side [Connolly and Hoar 2014 p16–18].

There are good reasons for using one category of technology over the other – take a browser-based game for example. In a 2D-platform style game (i.e.: Nintendo's iconic *Super Mario Bros.*) the player is relies their reflexes to move a character towards a goal while dodging enemies and collecting bonus items. As soon as the player hits, say, the “jump” key, a *client-side* script instructs the player's computer to process the next frame, immediately displaying Mario leaping upward. Conversely, if one were to attempt the same using a server-side script, the new frame would need to be processed by the server in response to a key input request, and then downloaded before it could even be displayed. In a game involving timing and reflexes this lag would be problematic. Moreover, if the game is being played by many users concurrently, this could place a very demanding load on the web server.

Continuing with the platform game example, consider a high scores table. Ideally players could compete for high scores with other players from around the world. A central database stored on a server and accessed via a *server-side* script would be well-suited to the task. Conversely, a client-side database could only contain the scores recorded on the device itself. Advantages and disadvantages aside, one can harness the client-side for gameplay and the server-side for high scores logging. As this example illustrates: combining client- and server-side technologies is often the best approach.

2.8.2 Front-End Development

HTML, CSS and JavaScript make up the standard languages available for front-end web development. Improvements introduced in HTML5 and CSS3 have made accommodating mobile devices far easier, along with improving HTML's web application capabilities. It is also important to note that HTML5 can be viewed as a collection of technologies, rather than just a markup language. For example, HTML5 includes new `<video>` and `<audio>` tags that support various new media file formats; while the addition of SVG support, and the `<canvas>` element, allow for new ways of displaying graphics [Gasston 2013].

The new *HTML5 file API* is particularly interesting. Whereas a server-side process of some sort would ordinarily be required for the uploading and storing of files, this can now be handled by the web browser itself in a purely client-side application [Bidelman 2011]. The implications for the proposed automatic marker include the ability for a task to be uploaded, displayed, and assessed using a browser-cached version. Bidelman, of course, is quick to point out potential security loopholes in this feature. At the risk of assuming too much, it would be sensible to include some server-side features in a production release of the automatic marker to prevent these exploits. For example, with no copy of the assessed file uploaded to the server, the JavaScript (client-side) logic of the automatic marker could be manipulated to report illegitimate results for tasks.

2.8.3 Back-End Development

As back-end programming languages rely on a web-server's configuration, no browser constraints exist – provided that the application serves up standards-compliant HTML, CSS, and JavaScript code [Connolly and Hoar 2014]. Although it is difficult to establish exactly which back-end programming languages are the most popular, books on *PHP*, *ASP.NET*, *Ruby*, *Java*, and *Python* appear to be among the most popular in on- and-offline book-stores currently.

All five of the server-side languages listed are capable of accomplishing what is required for the project, so additional criteria to consider are cost, complexity, web-hosting, and database compatibility. As ASP.NET (requiring a Microsoft Windows server) is not free, and Java seems overly-complex for the application at hand, PHP, Ruby (on Rails), and Python take preference [Joshi et al. 2012].

Open source database candidates for the project – such as *MySQL*, *PostgreSQL*, and *SQLite* – are well supported by PHP, Python, and Ruby. Therefore, any database requirements should not be an issue [Beaulieu 2009]. Furthermore, the database is not very complex, so any proprietary database features over and above what is available in the latest SQL standard make little difference [Kriegel 2011]. SQL:2011 happens to be the current standard, and the seventh revision of the Structured Query Language. It was formally adopted in 2011, with the first standard dating back to 1986/7 [Zemke 2012].

2.8.4 Web Development Frameworks

In Moffitt and Daoud's [2014] account of seven different web frameworks, the preface explains how such frameworks are designed to support the development of websites, web applications, and other web services by providing libraries that cater for tasks developers would otherwise have to frequently repeat in different projects. These frameworks vary in purpose and complexity, from simple templating engines to more involved model-view-controller architectures.

Every back- and front-end language provides many frameworks to choose from – take for example the popular *Ruby on Rails* framework, arguably synonymous with Ruby-based web development, but certainly not the only option available to Ruby developers [Cooper 2009].

Python also provides a myriad of frameworks varying from light and minimal, to heavy and more 'fully-stacked' [Moore et al. 2007]. A quick experiment creating a Python application using the *Django* framework automatically created the following directory structure:

```
mysite/
  manage.py
  mysite/
    __init__.py
    settings.py
    urls.py
    wsgi.py
```

Attempting something similar using Python's (far lighter) *Flask* framework created a simpler structure – which, furthermore, would otherwise not have included a `templates` directory had the feature not been (optionally) included [Grinberg 2014]:

```
mysite/
  __init__.py
  templates/
    demo.html
```

This comparison, admittedly, is over-simplified, but does help to illustrate how much decision-making is left to the developer. As Younker's comparison of several Python frameworks highlights [Younker 2008]: user load; future scope; budget; current knowledge; and time constraints, must all be addressed when weighing up one's options.

Unlike Python and Ruby, *PHP* is a purpose-built back-end web language that can more comfortably do away with frameworks altogether. Its popularity can be attributed, in part, to the wild success of *Wordpress* – an open source blogging tool driven by a versatile CMS (Content Management System) powering 23.1% of all the world's websites [W3Techs 2014]. Although these figures may be debatable, the broader picture is clear enough.

The frameworks mentioned thus far are all server-side solutions. Client-side frameworks can also prove useful for the development of the vector graphic testing and marking system.

AngularJS is a client-side framework that aims to simplify the development of web applications by using a single HTML page coupled with CSS and JavaScript. This means that JavaScript is responsible for rendering the page templates and dynamic content – not unlike many server-side approaches to web application design, although in the AngularJS scenario the client-side handles the bulk of the processing [Green and Seshadri 2013]. Similar such JavaScript frameworks include *Backbone.js* and *Ember.js*.

To illustrate this concept, three simple code samples follow. Each displays "Hello World!" in a web browser, but the important differences have been highlighted in bold text. It should be noted how the examples are similar enough, except that the AngularJS code includes links to two JavaScript files, along with a few extra attributes (easily identified by the use of an `ng-` prefix) to control the content. Whereas the

Django version's source code will ultimately appear no different to the static HTML *once* it has been downloaded by the web browser, it remains for the client-side to process the AngularJS version in order to display the greeting:

Static HTML:

```
<html><head>
</head></html>
<body>
  <h1>Hello World!</h1>
</body>
</html>
```

Python Django (Server-Side) Template:

```
<html><head>
</head></html>
<body>
  <h1>{{greeting}} World!</h1>
</body>
</html>
```

AngularJS:

```
<html ng-app><head>
  <script src="angular.js"></script>
  <script src="controller.js"></script>
</head></html>
<body ng-controller="HelloController">
  <h1>{{greeting}} World!</h1>
</body>
</html>
```

There are advantages and disadvantages to each approach, and the choice to take a more heavily server- or client-side oriented approach should be governed by the application's requirements [Freedman 2014]. However, to reiterate: as Freedman explains further, combining the approaches is often the best solution. In the case of AngularJS (or anything JavaScript-driven), *JSON* can be used to exchange information between the client and server. JSON (JavaScript Object Notation) thus serves as a standard notation for the transmission of attribute-value pairs between servers and web applications.

2.9 Online Assessments

A large amount of literature exists around online testing and examination systems. Papers covering the development of such systems often deal with the application's architecture – usually a combination of client-side programming, server-side programming, and a database(s) [Qiao–fang and Yong–fei 2012]. This supports the technologies proposed in section 2.8 (Web Development Languages and Frameworks), under points 2.8.2, 2.8.3, and 2.8.4, but does not assist with direction for a proven interface design. Although there are many examples of multiple-choice and other more orthodox online examination systems, there are fewer interfaces to provide inspiration for the project.

This is not surprising as no existing automatic marker for vector graphics drawing tasks appears to exist. However, there is extensive research in the area of Human Computer Interaction (HCI) which can provide direction for an effective design. However, this project is focused on building a simple *technical proof of concept*. As such, this study does not consider this area much further, but the prototype interface follows some basic guidelines layed-out in Chapter 3 (Framework).

Summary

SVG is now a widely-supported vector-graphics format – for both web browsers and editors. With most popular proprietary and open source vector graphics software now providing SVG export capability, the format provides an ideal solution for a web-browser-based testing and marking system. Although some limitations are revealed, especially with style attributes that do not conform to the SVG standard, essential vector graphics drawing tasks performed in most editors can be parsed and assessed. However, with the limitations identified, the tasks set for the automatic marker can avoid such problem areas. Chapters 3 (Framework) and 4 (Evaluation) further explore which SVG attributes are most reliable for testing purposes.

Shape recognition is a complex domain, and at that, in many ways beyond the scope of the prototype system. This could, however, be an area to investigate more thoroughly in further development. Some basic shape detection techniques have been reviewed, and it was also noted that SVG tags allow one to skip the edge-detection steps ordinarily required prior to recognising shapes. Shape detection is included in the prototype – that is, for shape types other than those which can be gleaned directly from `circle`,

`ellipse`, or `rect` SVG elements – for example a triangle rendered using the coordinates in a `path` tag.

For comparing the accuracy of recreated shapes with those of the originals, Hausdorff distance has been favoured over Hamming distance, comparing skeletons, and Support Vector Machine algorithms. With the exception of shape (`circle`, `ellipse`, `rect`) tags, the shapes assessed by the automatic marker will have been created by laying out anchor points dot-to-dot style using as pen/Bézier tool. By comparing the (two-dimensional) euclidean distance between each corresponding point in the recreated and original element, the level of accuracy with which the shape has been recreated is gauged using the Hausdorff distance algorithm.

The SVG markup can be represented as a non-linear data structure – most conveniently visualised as a tree. The general form of the problem, therefore, takes that of matching non-linear structures by: extracting the vector graphics elements; then pre-processing them to reduce spurious diversity; and, finally, computing the similarities between the them to derive an appropriate score.

It is important to take into account the Human Visual System (HVS) when comparing and assessing the accuracy of tasks for submission to the automatic marker. While in some cases these may be relatively straight-forward comparisons – provided the tasks avoid the caveats described – the tricky matter of comparing colours can been performed utilising existing algorithms, available as conveniently packaged libraries in JavaScript, Python, or other programming languages. For the purposes the of the automatic marker prototype, the DeltaE JavaScript library has been integrated to provide a ΔE value for colour comparison. This value represents the measure of change in visual perception between two colours.

There are multiple web development languages and frameworks suitable for the task of developing the automatic marker, many of which are conducive to the design of existing online assessment applications. Chapter 3 (Framework) covers the prototype's use of programming languages and frameworks in further detail.

Chapter 3: Framework

Beginning with the different approaches and solutions proposed for the automatic marker, this chapter discusses how the application has been designed, developed, and deployed. More specifically, the formats, languages, and frameworks reviewed in the Background chapter are expanded upon in order to explain how they have been practically implemented in the functioning prototype.

3.1 Application Design

Recent developments and support for SVG in HTML5, and increasingly widespread support for SVG in vector graphics editors, make a web application a sensible solution. Web applications run within a web browser and are programmed using a combination of client- and server-side languages. The application's primary goal was to test the feasibility of an automatic marker solution, and it is not only the algorithms for comparing shapes that have been evaluated, but also the suitability and capabilities of these new browser technologies within the context of the study. Given the scope and time-frame of the project, the system has been developed no further than a prototype.

3.1.1 Goals

In order to satisfy the goals of this research, the prototype has been developed to:

- function in modern, standards-compliant browsers;
- support standards-compliant SVG markup, and potentially a few additional proprietary extensions;

- provide a rudimentary user interface that – while not very refined – does not impede the subject's ability to operate the web application;
- display basic instructions regarding the tasks to be completed by the user;
- present a series of drawing tasks, to be completed in sequence, with a feedback form/questionnaire as a final step;
- store the results of the subject attempts, in this case the SVG markup and corresponding scores, in some form – preferably a database – in order for analysis;

Further detail around the specification of the interface is presented in section 3.3 (User Interface). The qualitative approach employed to gauge success in addressing these goals, and the accompanying result data, is reviewed and analysed in Chapter 4 (Evaluation).

3.2 Development and Constraints

With so many device and platform configurations available, it has been necessary to define certain constraints and parameters up-front in order to guide the development and testing phases of the project. By electing to create a web application, operating-system compatibility issues are negated and a modern web browser becomes the only requirement – but, as informed by Chapter 2 (Background), further hardware and software constraints are necessary to consider.

3.2.1 Hardware

Hardware considerations are largely concerned with types of devices that warrant support. As it is a browser-based solution that has been opted for, the question around whether or not to support mobile devices will need to be explored in further study. On one hand, catering for mobile devices can increase market penetration; while on the other, it is unlikely one would use a mobile phone for such a task. In practice, both a browser and vector-graphics editor were utilised simultaneously on a typical desktop/laptop system – but if sufficiently useful vector-graphics applications exist for tablets and mobile phones, these devices could be considered in future.

3.2.2 Software

The initial concerns around software were related to the various different vector-graphics editors available today, but SVG export functionality has been confirmed in three of the most popular, namely: Adobe Illustrator, CorelDRAW, and Inkscape. However, because HTML, CSS, and JavaScript – not to mention, SVG – rendering support can vary between web browsers, it has been ensured that the web application supports a broad enough range of the most popular browsers, namely: Google Chrome, Microsoft's Internet Explorer, Mozilla Firefox, and Opera. Browser compatibility, however, is not an uncommon challenge in web development, and extensive cross-browser testing is crucial to ensure consistency and support.

The automatic marker's algorithms deal largely with parsing XML and assessing how well a subject has completed a given vector graphics drawing task. The positions, sizes, and approximate accuracy of elements are established through their coordinate and style attributes. More complex features have been added where development progress was exceptionally smooth, but more 'organic' shape recognition was avoided altogether (see the duck in Figure 1, page 2).

3.2.3 Browser Testing Details

While developing the automatic marker web application, careful attention was paid to standards guidelines, and testing was carried out to ensure browser compatibility. However, with a plethora of browsers (and releases thereof) available, the development testing was restricted to:

Google Chromium – ver. 47.0.2526.80; running on Debian 9 Linux AMD64

Mozilla Firefox – ver. 46.0a2; running on Debian 9 Linux AMD64

Microsoft IE – ver. 11.0.26; running on Windows 7 64-bit

Microsoft Edge – ver. 25.10586.0.0; running on Windows 10 64-bit

Opera – ver. 35.0.2066.37; running on Debian 9 Linux AMD64

The operating system in each instance has been listed, but Google Chromium/Chrome, Mozilla Firefox, and Opera should produce the same results on Microsoft Windows,

Mac OS and Linux versions. Microsoft IE and Edge, however, are only available for Windows.

3.2.4 Client- and Server-Side Architecture

Server-side aspects were not the primary focus of development, as a heavily front-end reliant and intuitive testing/marking system took priority. Moreover, the server-side component did not deal with much more than performing some simple database transactions to record the results. This helped avoid some obvious exploits in the application: as the submissions are entered into the database, in the event of any suspected manipulation of the client-side script, the results can be verified against the stored SVG markup. Currently, however, this would involve manually submitting the SVG file back through the client-side application – but an automated, secondary server-side round of marking is a more complex solution that can be added in future. It should also be added that JavaScript can be processed on the server-side, and the shape-recognition and assessment component of the automatic marker has been modularised to allow for this, as per Appendix C (Automatic Marker Library Source Code). However, as explained in section 4.4 (Prescribed Tasks Survey), the test subjects had little to no inclination or motivation to behave dishonestly in testing the prototype. Figure 6 provides a conceptual digram of the application architecture to help illustrate the responsibilities of the client-side, server-side, and database components:

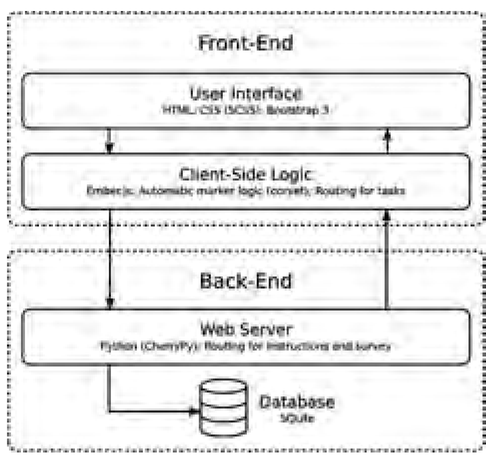


Figure 6: Application architecture.

Initial background research (section 2.8.4: Web Development Frameworks) revealed that front-end frameworks could potentially prove very useful, especially where mobile

device support is considered. The *Ember.js* JavaScript framework has been used to implement a front-end MVC (Model-View-Controller) solution.

On a more technical aside, it should be noted that Ember relies on Bower[1] for package management, and also includes the popular jQuery[2] JavaScript library. Both jQuery, and the *DeltaE* library used to compare how similar colours are, comprise just some of the packages managed by Bower.

As the back-end does not rely on the user's browser and operating system, most server-side languages are an option. However, there were a number of factors to weigh-up when selecting a back-end language – for example, cost, web hosting support, database connectivity, and image processing features. PHP, Python, and Ruby were short-listed, but the *CherryPy* Python framework was selected to build a rudimentary API to interface with an *SQLite* database. It could be argued whether or not this is an ideal solution, but given the developer's experience and the capabilities of CherryPy, it was more than adequate.

3.2.5 Development Methodology

With a single developer (the researcher), an iterative design methodology seemed most appropriate. Given that the developer was required to adopt many new skills, gauging exact time-frames and milestones proved challenging; but with this in mind, the development process was kept manageable by using realistic goals along with sufficient provision for potential skill-related shortfalls.

An incremental style approach also ensured that, at the very least, a simple yet functioning system was available for evaluation. More complex features were added where development progress was exceptionally smooth.

3.3 User Interface

The web application made use of a questionnaire to establish whether or not the user interface hindered the test subject's ability to perform the prescribed tasks. Ideally, users should not require any guidance on how to use the system. More intensive

---

1 https://bower.io/

2 https://jquery.com/

usability studies and refinements are beyond the scope of this project. Figure 7 presents a wireframe design upon which the application's front-end has been developed, and screenshots of the live interface can be found in Appendix G (Web Application Screenshots):

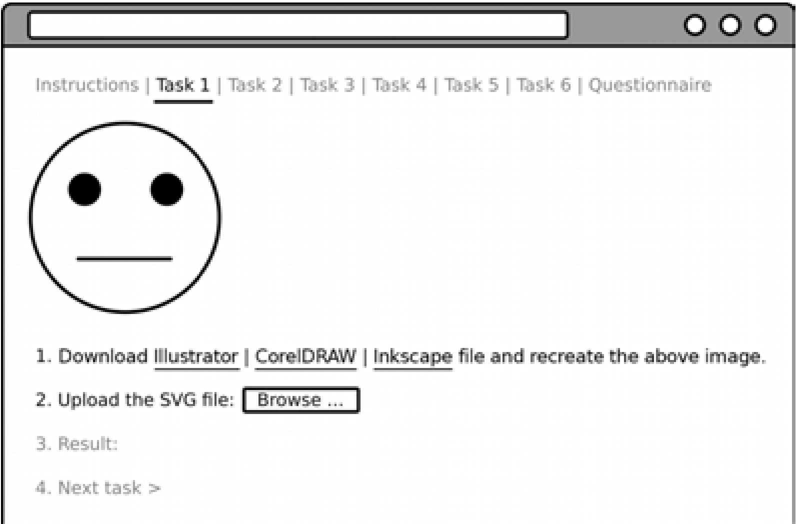


Figure 7: Interface wireframe.

Interface Description

The header section of the layout provides an indication that there are six tasks to be completed, preceded by some initial instructions in Appendix C (Automatic Marker Instructions) and the closing questionnaire in Appendix E (Prescribed Task Feedback Questionnaire).

The current task link is highlighted so that the user is aware of how far along they have progressed and how much of the test remains. More information about the instructions step, including the required documents settings and export options, is described in section 4.2.2 (Document Settings).

Once the raster image – in the case, the face/'smiley' graphic – has been recreated using the appropriate template (Illustrator | CorelDRAW | Inkscape), the resulting SVG is uploaded using the "Browse ..." button. It should be noted that the raster image can be saved 'out' of the browser using the right-click-activated context menu, and then traced using the subject's vector graphics editor of choice.

However, the template solution was provided instead, as after some initial testing during the development phases of the project, it was found that leaving the subject to import and trace the raster (PNG) graphic led to complications assessing the uploaded result. This was due to SVG files setup as either: an improperly cropped canvas; a document of different resolution; or a set of incorrectly-scaled elements.

Once the subject has uploaded the SVG result, it is displayed, and in his or her own time, the subject can proceed to the next step by clicking “Next task >”. Screenshots of these steps are presented in Appendix G (Web Application Screenshots).

3.3.1 Interface Themes

CSS is the language used to style HTML content. The selected front-end framework for the automatic marker, Ember.js, includes: SASS – a popular extension of the CSS language; and Bootstrap – a popular CSS framework, complete with fully predefined interface components and responsive widgets. As a result, the automatic marker adopts the default Bootstrap theme, but could be easily restyled using Bootstrap's theming functionality. This may be useful for styling the system to match an existing website or brand, or perhaps creating something more aesthetically pleasing, but exploring such options has not been necessary in this study.

3.4 Features

Given the requirements of the automatic marker, important features were listed and specced before the development phase could begin. Those warranting special mention have been described under the headings that follow.

3.4.1 Tolerance

As the automatic marker is required to assess the subject's recreation of a raster graphic – originally a vector graphic rendering with beyond-pixel-perfect (decimal) coordinate and measurement markup – some tolerance is required in order to avoid the marker behaving in an absurdly strict fashion. A set of variables in the shape comparison (JavaScript) module can be configured to determine just how 'lenient' the application is in assessing submissions. As seen in the source code below, thirteen variables are available to control various tolerances, so that, if desired, the application can be can stricter with certain aspects and more lenient with others:

```
corner:     2,   // disregard corners under this value (degrees)
points:     5,   // total % of difference between polygon points
position:   10,  // shapes distance apart
deltae:     10,  // color difference in fills & stokes
rectdetect: 2.5, // the degrees any path/poly-to-rect corner can be out by

// for the following: 0 is an exact match; 1 is way off
area:             0.3,
fillopacity:      0.2,
fontsize:         0.2,
strokemiterlimit: 0.2,
strokeopacity:    0.2,
strokewidth:      0.2,
rx:               0.2,
ry:               0.2
```

3.4.2 Scoring

Two essential functions have been programmed for scoring purposes. Presented respectively below: one gauges whether the difference between two (the original and submission) shape attributes falls within the required tolerance; the other simply checks if both shapes possess, or do not possess, the required attribute regardless of the value. In the examples of each below, the *final* parameter just before the closing parenthesis – in both cases, a `1` – represents the mark awarded:

```
scoreWithinTolerance(area, 100, 1, 0.2, 1);
...
scoreTrueFalse(strokedasharray, true, true, 1);
...
```

The first parameters represent: the attribute to assess (`area`/`strokedasharray`/etc.); the second is the value assigned to the preceding parameter (e.g. an `area` of `100` pixels / a `strokedasharray` of `true`); and the third is the correct answer (as a perfect area calculation returns `1`). The `scoreWithinTolerance` function, however, includes an additional fourth parameter as a tolerance threshold in which the value is considered correct – in this case, anything between `1` and 0.8 (1 minus `0.2`).

This means that certain criteria can be more heavily-weighted than others. For example: perhaps the assessor believes that recreating areas correctly is the most

important and challenging part of the drawing task, and should therefore be weighted four times more heavily than correctly applying a stroke-dash-array; and as such, she sets correct areas to count for four points; while stroke-dash-arrays are worth just a single point. Section 4.6 (Measurements) discusses how the marks have been allocated in the testing phase of this study, and also includes a table with the breakdown (see Table 2). Currently the marker does not subtract marks for incorrect results, although this could be configured with some minor amendment to the code.

> *Note* that for the stroke-dash-array attribute, multiple values representing the lengths of dashes and gaps can be compared, but in the current version of the automatic marker, a `scoreTrueFalse()` is used – so provided a stroke-dash-array exists, a mark is awarded regardless of whether or not the actual dash/gap values match. There were problems in how some vector graphics editor settings generated stroke-dash-array SVG markup, so this provided a more reliable approach.

3.4.3 Absolute and Relative Value Comparison

Two methods have been programmed for comparing numbers. Presented respectively below: `compareNumber` simply ascertains the difference between two values by subtracting one from the other; whereas `compareProportional` calculates what percentage of difference there is between each:

```
this.compareNumber = function(i1, i2) {
...
this.compareProportional = function(i1, i2) {
...
```

As an example scenario of the latter: if a 1cm × 1cm square were compared to a 2cm × 2cm square, one would judge this to be a rather poor recreation. However, when comparing a 101cm × 101cm square and 102cm × 102cm square, one would consider this a far more accurate attempt, despite there still being a whole centimetre of difference between the two. For this reason, comparing certain attributes relative to their size is necessary, rather than simply comparing an absolute difference in value.

3.4.4 Relative to Absolute Coordinates

Depending on how a vector graphics editor handles the adjustment of shapes, transform attributes often appear. For example, a subject may recreate a circle as best they can, only to find the circle's radius and/or position is a little off from what he or

she desired. Vector graphics editors offer various tools for moving, rotating, and scaling, which are particularly useful for amending such issues. However these may introduce `transform` attributes in the process. In the example below, each circle has been placed within a group (`<g>...</g>`) element. Group elements are usually used to gather multiple elements together – making it easier to deal with collections of shapes while visually editing – but can also contain just a single element. The automatic marker simply ignores group elements altogether – as how the subject has grouped items is unimportant for assessment purposes – and this causes no issues *unless* a transform attribute has been included. Although both of the circles below are identical in terms of their x, y, radius, and fill attributes, the second one will appear 10 pixels to the left of the first, and 20 pixels below it:

```
<g>
  <circle cx="5" cy="5" r="4" fill="red" />
</g>

<g transform="translate(10 20)">
  <circle cx="5" cy="5" r="4" fill="red" />
</g>
```

To avoid translate issues, the automatic marker converts all of the coordinates of translated child elements, so effectively the exemplar markup is interpreted as:

```
  <circle cx="5" cy="5" r="4" fill="red" />
  <circle cx="15" cy="25" r="4" fill="red" />
```

It should be noted that translate is not the only type of transform available. However, in its current state of development, the automatic marker does not cater for the others, namely: `matrix()`; `scale()`; `rotate()`; `skewX()`; and `skewY()`. However, it appears the editors tested in this study do not make use of the matrix, scale, and rotate transforms when generating SVG markup (section 4.2: SVG Features Testing) - but potential instances of skewX and skewY would need to be properly tested, as it is highly unlikely these would appear in any subject's attempts, because given the types of tasks there was no need to distort or skew such shapes.

3.4.5 Rules for Dealing with Extra Attributes and Shapes

When a subject recreates a graphic, they may add attributes or other shapes that are not present in the original. This section explains how the automatic marker deals with

such situations. Another potential issue presents itself in the ways that certain shapes can be created using different approaches – for example, a rectangle can be created as a `path`, `polygon`, `polyline`, or `rect` element.

Extra attributes

The automatic marker only searches for attributes it seeks to assess. Any attributes that the marker has not been programmed to process are simply ignored. For example, the application does does not deal with SVG filter effects (like drop-shadows).

Extra shapes

If a subject draws seven shapes – for the sake of this example: seven circles – where only two are required, the automatic maker compares the closest possible matches by comparing the centre-points of any contesting shapes with one another; the closest are paired-off for comparison, and the remainder are disregarded. In the unlikely event that two shapes are exactly the same distance from the correct coordinates, the one which appears first in the markup takes precedence. However, most editors use floating-point co-ordinates, so, to reiterate, this is unlikely. Arguably, taking more than just position into account could improve the correct detection of which shapes are most similar, but this leads to questions such as: are shapes more similar because of their position, or their fill? And if one believes the position is indeed a better benchmark, then what about a more correct position versus a correct fill-and-stroke-and-area-combination? Presumably, this would be a topic for further research, and will remain beyond the scope this study. This could potentially be an area to consider for SVM (see section 2.5: Shape ComparisonShape Comparison, under the heading Shape Similarity Algorithms, for more on State Vector Machines).

Polygons to rectangles

Firstly, it should be noted that all `paths`, `polyines`, and `lines` are converted to `polygons`. It has also been noted that different editors seem to prefer different elements (see section 4.2: SVG Features Testing).

Should a `path` possess attributes that constitute a rectangle – four corners and opposing sides that are parallel – then it is converted to a `rect` element for comparison purposes.

It should also be noted that unspecified fills/colours default to black (see section 2.6.1: Unspecified Colour Properties).

3.4.6 Font Family

After some initial research, it became clear that typefaces would prove tricky. Without explicitly requesting the use of, say, *Times New Roman* – or *Helvetica*, or some other font – it is left to the subject's discretion and fonts available on his or her device to select something appropriate. Although typefaces can be classified in various ways – such as serif or sans-serif – with so many available, even comparing the submission with an extensive list of similar typefaces could prove that no list is complete enough. Variances in weight and obliqueness complicate matters even further as fonts are usually redesigned and renamed to describe each variant – yet one can still bold and oblique-ify a non-bold, non-oblique font, regardless. Furthermore, while visually similar, oblique fonts are separate to italic fonts. For these reasons, font-family detection has been omitted as a feature in the automatic marker. However, `fontsize` is assessed, as is the position, fill, and fill-opacity.

3.5 Evaluation

To establish what features to include, some research into existing testing and marking systems has been be conducted in Chapter 2 (section 2.9: Online Assessments) – although it has been noted that the system is rather unique in its requirements. To evaluate the success of the automatic marker's features, prescribed tasks and user surveys have been be carried out. These provide qualitative data for analysis in the chapters to come.

Summary

The automatic marker for vector graphics drawing tasks has been developed as a web application. As a result, all that the user requires is a modern, standards-compliant web browser. However, given that the drawing tasks require desktop/laptop vector graphics editors – namely Adobe Illustrator, CorelDRAW, and Inkscape – mobile device support has been considered unnecessary in this stage of the prototype's development. After all, why would one carry out tasks on a desktop computer, then transfer them to a mobile device in order to submit them?

The algorithms involved deal largely with parsing XML – or more correctly, the SVG variant thereof – by assessing how well a subject has completed a given vector-graphics task based on the markup produced. The positions, sizes, and approximate accuracy of the recreated elements are established using coordinates and style attributes to be found within the SVG elements.

The client-side scripting component deals with the shape comparison, routing, templating, and assessment. Ember.js has been implemented as a client-side MVC framework, which in turn harnesses SASS, Bootstrap, and jQuery. Overall, most of the application code resides in the front-end. The back-end – built using CherrPy, a minimalist Python web framework – manages the results capture into a SQLite database.

Using the Bootstrap CSS framework also allows for easier theming of the interface, if desired. While the application's interface has not been a focus area of this research, there has been some consideration around what could be most intuitive. To ensure that the interface and application were functional for users, testing was carried out across multiple platforms and web browsers. A simple survey, in the form of a questionnaire, has been included as a final step after completing the drawing tasks, to establish if any unforeseen code bugs or interface issues had impeded the test subjects.

Numerous features where highlighted for development at the outset of the project. This included functionality for handling: *tolerance*; *scoring*; *absolute/relative value comparison*; *conversion of relative to absolute* coordinates; and rules for *dealing with additional attributes and shapes*. The tolerance and scoring functions deal with how strict or lenient the application ultimately is. The absolute/relative value comparison determines whether shape attributes are compared proportionately or not – for example: should a shape area be considered correct enough if it is within 10 pixels of the original, or 10 percent? Converting relative coordinates to absolute coordinates is necessary for any shapes affected by translate attributes. The automatic marker is also able to convert rectangle (or very close to rectangle) type paths to actual `rect` elements. This is useful for situations where a subject renders a rectangle by drawing it manually with bezier tool – or possibly where some vector editing software has converted a `rect` to `path` for some or other reason. However, font-family detection proved too complicated at this stage of development, largely due to the plethora of similar looking typefaces in existence, so this feature was omitted.

Chapter 4: Evaluation

The tests used to evaluate the application – including the tasks devised for the sample group(s) – are expanded upon here, along with important considerations around test criteria and metrics. Reporting on the results and findings gathered from the testing phase, the data and findings are presented here before concluding in Chapter 5 (Conclusion).

In establishing the feasibility of the automatic marking application, data has been gathered from:

- tests (without test subjects) to assess which *SVG features* are reliably and compliantly generated by different vector graphics design software;
- testing (also without users) to measure the automatic *marker capabilities and limitations* assessing SVG markup;
- and tests using human subjects interacting with the application by completing and submitting *prescribed tasks* – followed by a survey – which also serve to test browser compatibility.

4.1 Vector Graphic Drawing Tasks

The tests devised to address the above-listed points rely on a series of vector graphic drawing tasks. These tasks are designed to investigate the SVG and assessment capabilities of the automatic marker.

The official SVG specification is developed and documented online by the W3C's *SVG Working Group*. However, the references available in the *Mozilla Developer Network*[1] (MDN) resources are particularly useful and well-organised, especially for the purposes of this project.

4.1.1 SVG Elements

MDN have arranged and listed SVG elements under roughly a dozen category headings. As explained, the project is very much a proof of concept exercise at this moment – so only features for marking the most basic and essential elements have been programmed into the application – leaving out the need to address the animation, filter, structural, and the other more complex categories documented. Although there is some content overlap under the pertinent headings, the following encapsulate the relevant groups of elements:

Basic shapes

`<circle>`, `<ellipse>`, `<line>`, `<polygon>`, `<polyline>`, `<rect>`

Font elements

`<font>`, `<font-face>`, `<font-face-format>`, `<font-face-name>`, `<font-face-src>`, `<font-face-uri>`, `<hkern>`, `<vkern>`

Note that fonts are a complicated area for the automatic marker (see section 3.4.6: Font Family), so only the `<font>` tag is supported for now.

Graphics elements

`<circle>`, `<ellipse>`, `<image>`, `<line>`, `<path>`, `<polygon>`, `<polyline>`, `<rect>`, `<text>`, `<use>`

Shape elements

`<circle>`, `<ellipse>`, `<line>`, `<path>`, `<polygon>`, `<polyline>`, `<rect>`

With these identified, the six drawing tasks depicted in Figure 8 have been devised to target them for the testing:

1 https://developer.mozilla.org/en-US/docs/Web/SVG

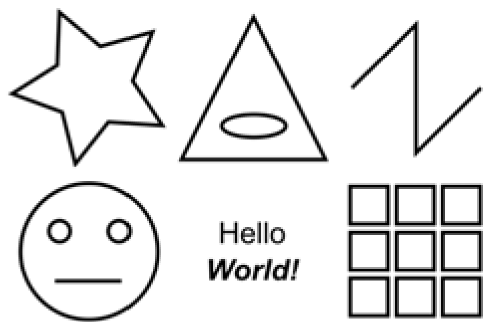


Figure 8: Vector drawing tasks for element testing.

Beginning clockwise from the top-left: the star should register as a `<polygon>`; the triangle-and-ellipse combination as a `<path>` and `<ellipse>`; zigzag as a `<polyline>`; grid as a group of `<rect>`'s; "Hello World!" as `<text>`; and the smiley as a combination of `<circle>`'s and a `<line>`.

> *Note* that `<image>` and some of the font elements (see section 3.4.6: Font Family) have been ignored. The `image` tag allows for raster graphics to be displayed in an SVG file – useful for tracing purposes, but not necessary to assess. Another listed, yet unaddressed element is `<use>` – employed for cloning elements – that could conceivably be added during copy-paste operations while using an editor such as Adobe Illustrator, CorelDRAW, or Inkscape. However, instances of `<use>` did not appear in the SVG Features Testing Results (section 4.7.1).

These test shapes were chosen for a number of reasons – namely, they:

- embody the essential drawing tools used to create shapes with vector graphics drawing software (including `paths`, but with the exception of *curved* path segments);
- do not present the user with too many different shapes to recreate in a single task, so as to keep the tasks focused along with the results data for interpretation.
- encompass all of the SVG elements the automatic marker has been designed to assess in its current phase of development (prototype) –
- and can also accommodate the additional attributes required in section 4.1.2 (SVG Attributes) below.

4.1.2 SVG Attributes

As with the elements, there is no need to address all SVG attributes but rather just those applicable to the visual appearance of static (as opposed to animated or interactive) graphics.

> *Note* that attributes are inserted into an opening element tag. For example, the following circle has cx, cy, r, and fill attributes to define the x- and y-coordinates, radius, and fill colour:

```
<circle cx="5" cy="5" r="4" fill="red" />
```

Standard to all elements are coordinate and dimension attributes – although these vary somewhat depending on the shape; for example, take the code above: `r` (radius) is only available for circles. MDN also lists a further sixty attributes under the heading *Presentation*, which has been necessary to reduce for this study. As of its current release, the automatic marker supports the most essential, namely:

Presentation attributes

`fill`, `fill-opacity`, `stroke`, `stroke-dasharray`, `stroke-linecap`, `stroke-linejoin`, `stroke-miterlimit`, `stroke-opacity`, `stroke-width`

Note that vector graphics editors may apply presentation attributes using a *presentation attribute(s)* or *style attribute* – so:

```
<circle cx="5" cy="5" r="4" fill="red" stroke="green" />
```

has the same effect as:

```
<circle cx="5" cy="5" r="4" style="fill:red; stroke:green" />
```

Coordinates & Dimensions

`cx`, `cy`, `d`, `height`, `points`, `r`, `rx`, `ry`, `transform`, `width`, `x`, `x1`, `y`, `y2`

On a more technical aside, the JavaScript `getAttribute()` works well for x, y, r, and similar presentation attributes; but `getComputedStyle()` has been utilised as it supports both presentation- and style-type attributes.

Ideally a subject using the application should be able to simply copy/draw a graphic without any special instruction, but some presentation attributes are open to interpretation. For example (refer to Figure 9): bevelled-joins on stroke corners are

applied by setting the `stroke-linejoin` attribute to `"bevel"`; however, they can also be induced by reducing the `stroke-miterlimit` until the corner is 'blunt':

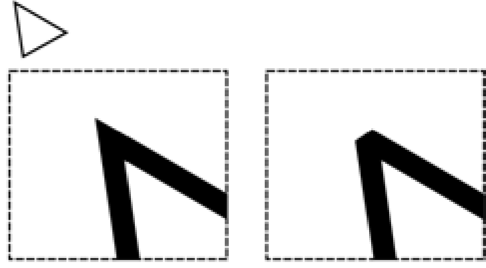


Figure 9: Miter join (lower left); and bevel (right) or possible low miter-limit join.

> *Note* that, however, to avoid complications and, even more so, further development time, the `stroke-miterlimit` was removed as an assessment criteria. The subjects favoured the bevel option and did not appear to make use of the editor miter-limit parameters, probably because this requires a more roundabout approach when using a visual editor (see section 4.7 on Results). Ideally this feature could be added in a future revision of the software.

Subjects carrying out the prescribed tasks shed more light on issues such as stroke miter-limit/bevel complications (also discussed in section 4.7 on Results), as this is where subjective interpretations are best exposed and surveyed.

So that both SVG elements and attributes can be assessed, Figure 10 is an iteration of Figure 8 more fully resolved to include presentation attributes:

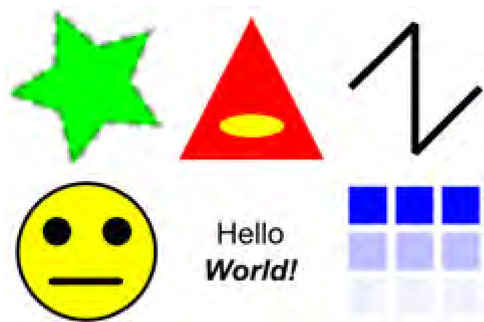


Figure 10: Vector drawing tasks with attributes applied.

Beginning clockwise from the top-left: the star uses a round `stroke-linecap` as well as `stroke-dasharray` and `stroke-opacity` attributes; the triangle-and-ellipse combo rely on only `fill`; the zigzag uses a bevel `stroke-linejoin` and butt `stroke-linecap`; the grid applies `fill-opacity` in various increments; “Hello World!” is coloured using `fill`; and the smiley's mouth uses a round `stroke-linecap`.

These graphics are utilised in all the testing, evaluation, and surveys that follow.

4.2 SVG Features Testing

The SVG features discussed have been listed – alongside each individual drawing task – on the SVG Features Datasheet (Appendix A) so that their values may be captured. This is an electronic spreadsheet, complete with formulas, that has been reformatted slightly to fit the page layout of this document. By recreating the graphics in different applications the markup produced is compared. The more consistent the markup, the easier it will be for the automatic marker to work effectively.

The datasheet, with its six individual drawing tasks, was completed three times by the researcher – in each instance using different vector graphics drawing software.

4.2.1 Recreated Drawing Task Files

The drawing tasks were recreated using the following vector graphics editors:

- Adobe Illustrator CS6 – ver. 16.0.0 64-bit; running on Windows 8.1 Pro 64-bit

- CorelDRAW X7 – ver. 17.1.0.572; running on Windows 8.1 Pro 64-bit
- Inkscape 0.91 – ver. r13725; running on Debian 9 Linux AMD64

The operating system in each instance has been listed, but Adobe Illustrator should produce the same results on Microsoft Windows or Mac OS (64- or 32-bit); and the same applies for Inkscape's Windows, Mac, and Linux versions. CorelDRAW, however, is only available for Windows.

Although a number of other vector graphics editors exist, these three are: probably the most popular; together cover the most prominent desktop operating systems; and all support SVG.

4.2.2 Document Settings

When creating new files or exporting to an SVG format, each of the above-listed applications provide various options affecting the final markup – with two of the most significant being colour space and styling.

The automatic marker – as per the W3C SVG specification[2] – uses the CSS *sRGB* colour space. This is fitting for digital displays that mix colour using Red, Green, and Blue primaries – unlike, say, printers that rely on Cyan, Magenta, Yellow, Key/Black (CMYK).

Regarding styling options, take an earlier example: a `circle` with `cx`, `cy`, `r`, and `fill` attributes; and in this instance include an additional stroke. The markup can be represented using three variations that all produce the same result:

Presentation attributes:

```
<circle cx="5" cy="5" r="4" fill="red" stroke="blue" />
```

Style attributes:

```
<circle cx="5" cy="5" r="4" style="fill:red; stroke:blue" />
```

Internal stylesheet:

```
<style>
  .redcircle {fill:red; stroke:blue}
```

2 https://www.w3.org/TR/SVG/types.html#DataTypeColor

```
</style>
<circle cx="5" cy="5" r="4" class="redcircle" />
```

The reader will notice that the bold text highlights the differences in the above examples, and should infer how the variation works. Rather than going into more detail or weighing up the merits of each, the purpose of presenting these examples is instead to make one aware of the challenges in assessing the markup, and, more importantly, which SVG export options should work best with the prototype.

As the automatic marker is programmed to accommodate the *style attributes* convention (as opposed to the *presentation attribute* or *internal stylesheet* variants, as described above), the file settings listed in Appendix B (Recreated Drawing Task Parameters) table have been selected upon exporting the six recreated drawing tasks to better accommodate the automatic marker by providing the styling attributes convention it has been programmed to parse. However, as the DOM addresses *style attributes* and *internal stylesheet class* properties using the same path, both variants should work in the automatic marker:

Presentation attributes DOM path:

```
document.getElementsByTagName('circle')[0].getAttribute('fill');
```

Style attributes DOM path:

```
getComputedStyle(document.getElementsByTagName('circle')[0], null).fill;
```

Internal stylesheet DOM path:

```
getComputedStyle(document.getElementsByTagName('circle')[0], null).fill;
```

For CorelDRAW, *internal stylesheets* have been selected, as CorelDRAW – as of its current version – does support style attributes for SVG export; Inkscape provides no style options, and implicitly relies on style attributes; and while Illustrator has export options for all three variants, style attributes have been favoured. This better suited the manual data collection process required for parts this study where the researcher was gleaning attributes directly from the SVG source code. Whereas internal stylesheets lend to themselves to more DRY (don't repeat yourself) code, a human has to scan back to the top of the document to match an element's class with its corresponding properties each time.

4.3 Marker Capabilities and Limitations Evaluation

To evaluate the capabilities of the automatic marker, each of the drawing tasks has been completed by the researcher and submitted to the web application for assessment, and then the results reviewed. The scorecard spreadsheet (Appendix D: Recreated Drawing Task Parameters sans the “Researcher” mark entry field) provides a column for the output of Adobe Illustrator, CorelDRAW, and Inkscape renditions of the graphics.

Given that the researcher has recreated the drawing correctly in all instances, ideally, the automatic marker should report consistent grades for SVG files produced by different editors, even if the markup produced varies somewhat – thus ensuring that the application accommodates Illustrator, CorelDRAW, and Inkscape. Unlike the tests using multiple subjects, it can be assured that the researcher has used the same / or similar as possible an approach with each editor.

This evaluation makes use of the same six tasks as those in the features test (refer to section 4.2.1: Recreated Drawing Task Files) and Prescribed Tasks Survey (section 4.4) below.

4.4 Prescribed Tasks Survey

This research is *qualitative* in nature, relying on eleven test subjects in all. By carefully reviewing the results of each task, along with the questionnaire data, the researcher has explored how the subjects and vector graphics editors have handled the tasks. By identifying challenges and problems, these can be alleviated through adjustments in the prescribed tasks, web application logic, interface, and marking criteria of the automatic marker.

In this survey, the test subjects were required to complete a series of prescribed tasks: namely to recreate the six test graphics (section 4.2.1: Recreated Drawing Task Files). The data capture spreadsheet – found in Appendix C (Automatic Marker Instructions) – includes two fields: one for a mark assigned by the automatic marker; and another by the researcher. Any additional information that may not have been directly gleaned from the captured element, attribute, and property fields, has also been noted. With the six drawing tasks submitted-to and assessed by the automatic marker, the submission results and corresponding markup have then been further reviewed, task-by-task, by

the researcher and combined with the post-tasks-completion survey data to provide insights into the reason for any anomalous or incorrect results – which could then be attributed to idiosyncratic or unexpected drawing approaches, interface hindrances, application bugs, or some other potentially unforeseen source.

Each column in Appendix D (Prescribed Tasks Marking Rubric) represents the result of a single test subject's six submissions. Ideally, the “Auto-marker” should assign the same mark as the “Researcher” (human marker); so that any discrepancies highlight shortfalls the application's capabilities.

4.4.1 Survey Preparation

After some casual discussion offline with around twenty students, a formal email briefing was sent to thirteen potentially interested participants explaining:

- the purpose of the research;
- what they were expected to do;
- how long it would roughly take;
- that they would receive no compensation for their time;
- that the information may be made publicly accessible – with their names to be kept confidential;
- that participation was completely voluntary;
- and that should they have any questions or concerns they were free to contact the researcher using the details provided.

Approaching students seemed the most sensible option, being that the automatic marker's purpose is to be used as an assessment tool for training vector graphics software.

Eleven of the thirteen recipients committed to participate. Given the number of prospective participants the researcher was able to approach, and those who initially expressed interest, this number was deemed sufficient for the purpose of the survey – which was to qualitatively highlight any major differences in the approaching of the

drawing tasks, or the potentially incorrect assumption that the tasks were not too complicated, even for even a novice user.

The final group of participants was comprised of seven women and four men – all between the ages of nineteen and twenty-six years. All of these students were enrolled in creative degrees, and each possessed over a semester (six months) of experience using vector graphic drawing software. All received training in Adobe Illustrator, as per the institution's curriculum, but four also had experience using Inkscape – which, in every instance, happened to predate their exposure to Illustrator.

The automatic marker runs from a web server in the client's browser, requiring only that the computer have the desired vector graphics drawing software installed and an internet connection. The candidates completed the test remotely in their own time as the web application imposes no location or time limits. Of course, this means they may have cheated – but given that their performance in the tests was confidential, and of no consequence to their studies, there was little incentive to be dishonest. Furthermore, the tasks should not have posed much of a challenge for students of their level and experience.

4.4.2 Prescribed Tasks

Using the automatic marker application developed for this project, the participants recreated the six drawing tasks using a vector graphics editor of their choice. The SVG Features Testing (section 4.2) – and Marker Capabilities and Limitations Evaluation (section 4.3) in a smaller extent – identified differences in how editors produced markup. These prescribed tasks aimed to establish how potentially different interpretations and approaches to recreating the graphics could pose challenges for the automatic marker.

As described in section 3.3 (User Interface) of the Framework chapter, the web interface provides a raster image of the graphic to be redrawn – embedded within a downloadable “Illustrator | CorelDRAW | Inkscape” template file – along with a field to upload the completed task. The header section of the layout displays links to the six different tasks and questionnaire, highlighting the active task and indicating the ones awaiting completion, all preceded with some instructions on the required documents settings and export options (see section 4.2.2: Document Settings).

4.4.3 Questionnaire

The purpose of the questionnaire was to provide additional data to accompany the automatic marker's SVG input and subsequent assessment data. By establishing the following, any potential differences in markup, and errors could be identified. With the help of the questionnaire, possible issues could be attributed to:

1. differences in markup, over and above the subjects approach to the tasks, that may have resulted from SVG files produced using different editors;

2. possible confusion or inability to interpret what was required;

3. problems that may have been induced by browser quirks, or unforeseen browser-compatibility issues.

The questionnaire also highlights any bugs or errors that the subjects may have noted while carrying out the prescribed tasks, and confirms whether or not the tasks were quick and easy to complete.

The automatic marker presents this feedback questionnaire (see Appendix E: Prescribed Task Feedback Questionnaire) once the final task has been submitted. It is a web-based form, complete with a basic input validator and submit button. The input fields in the appendix have been omitted, and the formatting varies slightly in the live version, as this has been adjusted to suit the format of this document. Chapter 3 (Framework) describes the 'live' web application features in further detail under section 3.3 (User Interface).

4.5 Browser Compatibility

No specific web browser was prescribed for the survey, and although not explicitly stated, the application functioned correctly in all of the standards-compliant modern browsers used for testing during the development phase of the project – as noted in the User Interface section of the chapter 3 (Framework).

The questionnaire includes specific questions about the browser and vector graphics editor the subject elected to use, so that any (unlikely) browser-induced issues could be identified.

4.6 Measurements

'Tolerance' values are defined in the automatic marker's configuration variables, essentially controlling how 'strictly' – or 'leniently' – the system marks. That certain attributes should count more heavily than others, and exactly what tolerance settings are appropriate, are questions demanding more inquiry and investigation that ultimately fall beyond the scope of this study. Every attribute and element used to score accuracy in the automatic marker has been assigned one mark. So, using the first task (the star) as an example:

The properties of a recreated star drawing – intentionally containing a few errors for example purposes – captured in the spreadsheets presented in this chapter and the appendices, would have its score calculated by the researcher as per Table 2:

Criteria	Value	Tolerance	Result	Mark
`<polygon>`	`<polygon>`	`Exact`	`<polygon>`	*1*
`points=`	`m 232 …`	`← 5% →`	`m 224 …`	*1*
`fill=`	`#00FF00`	`< 10`	`#01FC00`	*1*
`stroke=`	`#000000`	`< 10`	`#FF0000`	*0*
`-dasharray=`	`> 0`		`-`	*0*
`-linecap=`	`round`	`exact`	`round`	*1*
`-linejoin=`	`miter`	`exact`	`miter`	*1*
`-opacity=`	`0.25`	`← 0.2 →`	`0.25`	*1*
`-width=`	`10`	`← 5% →`	`1`	*0*
			TOTAL	*6 / 9*

Table 2: Calculating Scores.

The automatic marker also calculates scores according to the table above. To reiterate, further permutations of tolerances, tasks, and marking criteria, are reserved for possible further study – and, importantly, would be accompanied by further development of the automatic marking application itself. Using the web application, each shape, along with its corresponding attributes and scores, is outputted – using the JavaScript `console.log()` function – to the web browser's development console. In the case of the star example above, the console output for a perfect score reads as follows:

```
* no problem elements removed

scoresheet:
---
shape (polygons[0]): 1
points: 1
fill: 1
fillopacity: 1
stroke: 1
strokedasharray: 1
strokelinecap: 1
strokelinejoin: 1
strokeopacity: 1
strokewidth: 1

comparison data: Object { circles: Array[0], ellipses: Array[0], polygons:
Array[1], rects: Array[0], texts: Array[0] }

total score: 10
```

After submitting any given task, the developer can review this information for debugging purposes. In the case of the above example, the:

```
* no problem elements removed
```

indicates that it has not been necessary to remove any elements that are likely to cause problems for the automatic marker – such as those with `clipPath` attributes. The information under the *scoresheet* heading indicates where each mark has been awarded. The *comparison data* object can be interactively explored further for deeper

investigation into the exact values of every attribute comparison, by clicking on any of the underlined “Array” items:

```
comparison data: Object { circles: Array[0], ellipses: Array[1], ...
```

The total score, along with all of the submission SVG markup is submitted to the results database for further analysis and record-keeping purposes.

Further explanation of the various tolerance settings can be found in sections 3.4.1 and 3.4.2 on Tolerance and Scoring.

4.7 Results

The results presented here begin with the SVG Features Testing Results (section 4.7.1), some of which testing was implicitly carried out while developing the application, but formally tested once the application's development was complete and any outstanding bugs had been eliminated.

With the supported features established through the SVG features testing, versions of the SVG graphics recreated by the researcher using Adobe Illustrator, CorelDRAW, and Inkscape were submitted and assessed using the automatic marker web application. Some minor bugs were revealed here, but amended a short time into the testing process, so that this step could be restarted. The results are documented in the Marker Capabilities and Limitations Evaluation Results (section 4.7.2).

Section 4.7.3 (Prescribed Tasks Survey Results) covers the eleven subject attempts using the system, and section 4.7.4 (Questionnaire Results) summarises their questionnaire feedback after completing the final task.

4.7.1 SVG Features Testing Results

Table 3 (below) presents a summary of the three SVG Features Datasheets (Appendix A) containing the markup produced by each of the vector graphics editors tested in this study. Notable differences include CorelDRAW and Inkscape's use of group elements – even within documents containing just a single shape element.

The difference between the three editor's preferences for paths, polygons, polylines, or lines, poses no issue for the automatic marker as it converts all three latter elements to

paths, regardless. CorelDRAW's use of polygons in the grid graphic seems particularly odd, as the top row of squares still relied on `rect` elements – however, this is insignificant as the automatic marker also performs polygon-to-rectangle conversion where necessary. It is also important to note that Inkscape utilises a completely different approach to the Text task, relying on `flow`-type type elements.

Graphic	Illustrator	CorelDRAW	Inkscape
Star	1 × `polygon`	2 × `polygon` 2 × `group`	1 × `path` 1 × `group`
Triangle	1 × `polygon` 1 × `ellipse`	1 × `polygon` 1 × `ellipse` 1 × `group`	1 × `path` 1 × `ellipse` 1 × `group`
Zigzag	1 × `polyline`	1 × `polyline` 1 × `group`	1 × `path` 1 × `group`
Smiley	3 × `circle` 1 × `line`	3 × `circle` 1 × `line` 1 × `group`	3 × `circle` 1 × `line` 1 × `group`
Text	2 × `text` 1 × `rect`	2 × `text` 1 × `group`	1 × `flowroot` 1 × `flowregion` 1 × `rect` 2 × `flowpara` 1 × `group`
Grid	9 × `rect`	3 × `rect` 6 × `polygon` 1 × `group`	9 × `rect` 2 × `group`

Table 3: Summary of different editor markup.

Many of these differences were highlighted during development, and as such, influenced the final prototype.

4.7.2 Marker Capabilities and Limitations Evaluation Results

Governed by the SVG Features Testing Results (section 4.7.1), the automatic marker was programmed to support all of the necessary elements and attributes (barring typeface-related tags other than `font`). The answer SVG files were hand-coded, and the files produced by the researcher using Abobe Illustrator, CorelDRAW, and Inkscape were submitted to the automatic marker for comparison with these answer files; this is the basis of how the assessment feature functions. A 'perfect' score served as a benchmark with which to gauge the application's performance. Perfect scores were calculated and verified by:

- manually reading though the markup and assigning a mark for each of the assessed criteria;
- submitting the hand-coded SVG back into the marker so that it was compared with itself, thus verifying that the automatic marker was detecting all of the necessary elements and attributes, and henceforth assigning a perfect score.

Following this, the markup produced by the different vector graphics editors was submitted for automatic marking to establish the capabilities and limitations of the application in dealing with Illustrator, CorelDRAW, and Inkscape files.

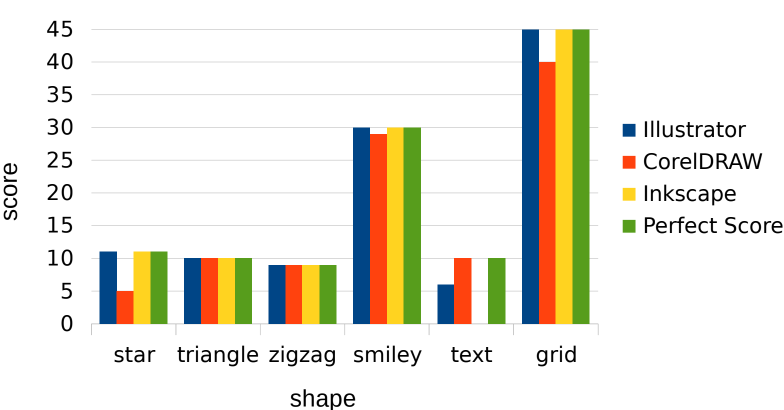


Figure 11: Graph: Comparing researcher submissions to automatic marker.

Figure 11 (above) displays the results of the testing. The green bars represents a perfect score for each respective task, so that the researcher's as-correctly-reproduced-as-possible SVG file scores could be compared to these. Ideally, the Illustrator, CorelDRAW, and Inkscape files should all produced bars equal in height to the applicable green bar. However, there were instances where this was not the case, and upon further investigation, the shortfalls were ascribed following issues:

Star

CorelDRAW created two star graphics: one was a green-filled star; the other, a semi-opaque dashed outline of a star. It was revealed that this is due to how CorelDRAW handles outline/stroke opacity when converting files to SVG.

Triangle

No issues. The red triangle graphic with the yellow oval inside/above it produced a perfect score in all instances.

Zigzag

No issues. The zigzag graphic produced a perfect score in all instances.

Smiley

With the exception of a `stroke-linejoin` issue in CorelDRAW, the smiley graphic produced perfect scores in all instances.

Text

CorelDRAW produced a perfect score for the text graphic. However, as highlighted during the development phases of the project, fonts have been acknowledged as a complicated area for the automatic marker (see section 3.4.6: Font Family).

Illustrator lost marks for `font-size` and `position` in both lines of the text. Inkscape uses a completely separate set of typeface elements (`flow-...`) and scored a zero as a result.

Grid

CorelDRAW created `rect` elements (as one would expect) for the top row of the grid arrangement, but polygons for the subsequent rows. This may have been because of the copy-paste approach used to create the second and third rows as duplicates of the first. The automatic marker did, however, correctly convert the polygons to `rects`, but somehow confused which rectangles to compare with which, that resulted in marks lost for `position` and `fill-opacity` attributes.

Illustrator and Inkscape produced perfect scores for the grid graphic.

4.7.3 Prescribed Tasks Survey Results

With the Marker Capabilities and Limitations Evaluation Results (section 4.7.2) completed, testing with subjects commenced. Completing the same tasks, using the same six graphics from the previous tests, each subject used his or her preferred vector graphics software to test the system. A graph for each task (Figures 12 to 17) presents the results of each subject's attempts, with a red bar indicating the mark the submission received when compared *visually* side-by-side the with original (according to the researcher), and a blue bar indicating the score the application awarded the subject.

The reasons for any discrepancies are discussed below each of the graphs that follow, and section 4.7.4 (Questionnaire Results) helps shed further light on the likely causes.

Star task

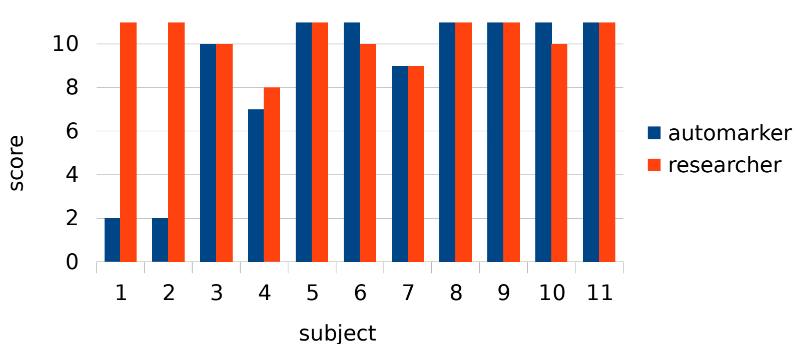


Figure 12: Graph: star graphic – auto-marker and researcher comparison.

Subjects 1 and 2 both used Illustrator to complete the tasks, but for reasons unknown Illustrator used an outline drawn with multiple line & polyline segments, which threw off the automatic marker. This may have been due to how the subjects created the graphic, but is inconclusive. However, this same issue did not occur in the other Illustrator submissions (subjects 6 through 11).

Subject 4 used an incorrect outline/stroke, which duly resulted in marks deducted, but also the had the knock-on effect of an additional mark lost as another stroke attribute was omitted.

By the researcher's reckoning, subject 6 should have lost a mark for the fill colour, but the automatic marker was, arguably, overly-lenient with its comparison tolerance. However, how strict or lenient the colour comparison truly was probably requires a larger sample than a single individual's (researcher) opinion.

Triangle task

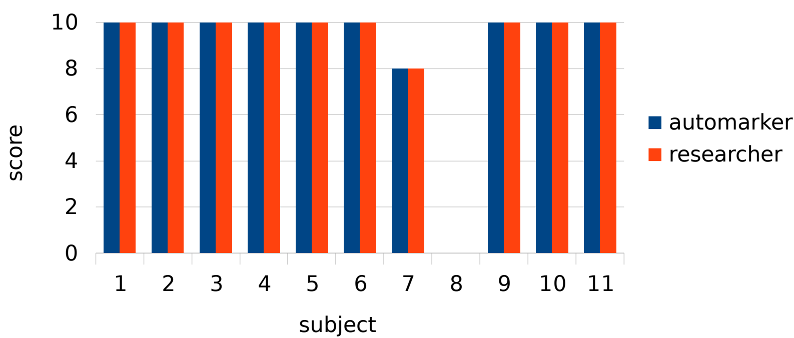


Figure 13: Graph: triangle graphic – auto-marker and researcher comparison.

The triangle task assessment marks were entirely aligned in comparing the researcher's and automatic marker's results. To explain the submissions that received less than a perfect score: subject 7 used a fill very different to that of the original (possibly because the colours were mixed in CMYK); and subject 8 submitted the wrong graphic entirely (later revealed to be a browser-related issue).

Zigzag task

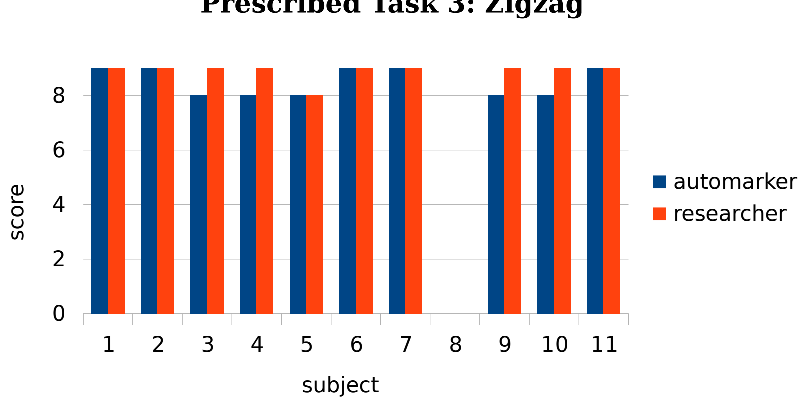


Figure 14: Graph: zigzag graphic – auto-marker and researcher comparison.

The zigzag results were close, but there were a number of single-mark discrepancies: subjects 3, 4, and 10 used the wrong `stroke-linecap` (although it is near impossible to tell the difference when looking at the graphic); subject 5 and 9's stroke colour was a slightly-off black mixture, which the marker colour comparison regarded outside of the tolerance range.

Smiley task

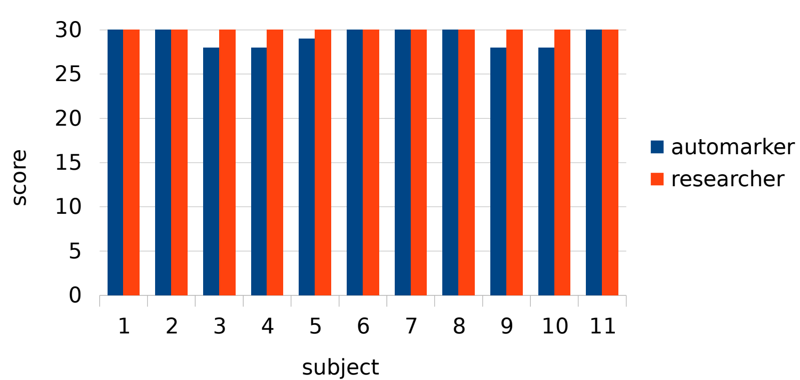


Figure 15: Graph: smiley graphic – auto-marker and researcher comparison.

The smiley task results were close again, but issues from the previous tasks reappeared, namely: subjects 3, 4, and 10 used the wrong `stroke-linecap` (probably an editor setting carried over from the previous task); subject 5's fill colour was off (although, arguably, the colour comparison may have been overly-strict in this instance); and subject 9's stroke colour was a slightly-off black mixture, which the marker colour comparison regarded outside of the tolerance range.

Text task

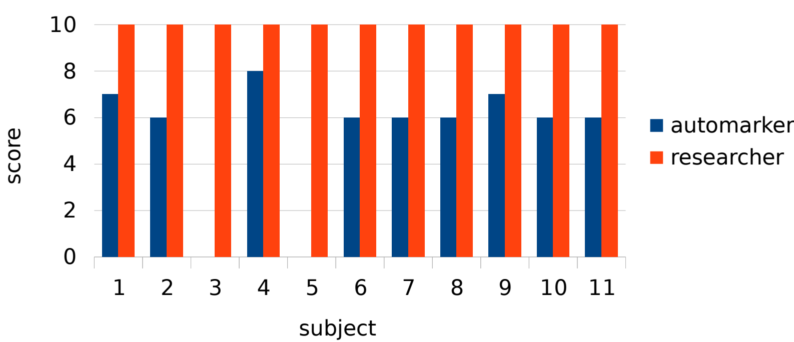


Figure 16: Graph: text graphic – auto-marker and researcher comparison.

It was established prior to this prescribed tasks testing that fonts were going to be problematic. This was further confirmed here, with every subject producing accurate recreations whereas the automatic marker dealt out low scores. Subject 5 used Inkscape and received a zero; the best result was subject 4's, who used CorelDRAW, but lost out on a perfect score due to odd font metrics in their typeface.

Grid task

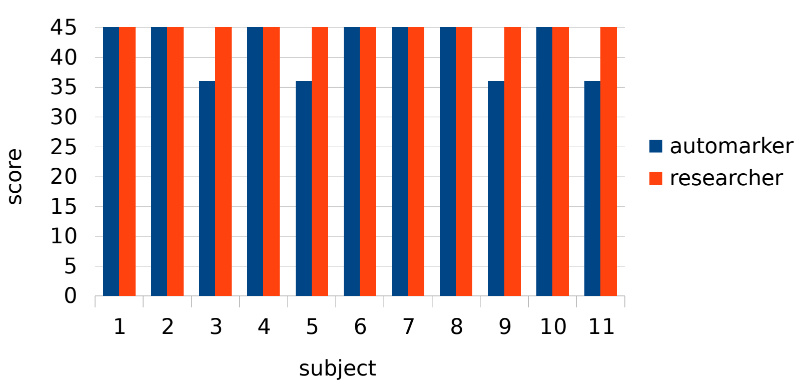


Figure 17: Graph: grid graphic – auto-marker and researcher comparison.

The grid task results were relatively accurate, but subjects 3, 5, 9, and 11 all made use of pale blue fills rather than semi-opaque fills – which appeared indistinguishable from the original, but relied on a different technique.

Interestingly, the issue in the CorelDRAW SVG Features Testing Results (see section 4.7.1) and Marker Capabilities and Limitations Evaluation Results (section 4.7.2) did not occur in subject 4's submission – possibly because she did not utilise a copy/paste approach for the second and third rows of squares, as the researcher did.

The main issues encountered in the prescribed tasks test can therefore be listed as follows:

- CorelDRAW creates a duplicate shape to apply semi-opaque strokes;
- colour comparison can be a bit inconsistent, with near black colours being assessed very strictly, and other colours being assessed somewhat over-leniently. It would seem it depends in which part of of the spectrum the colours lie, but further testing would be required to determine what role subjectivity plays in this criteria;
- `stroke-lincaps` can be impossible to determine by looking at certain graphics, particularly where `square` and `butt` caps are concerned – a possible solution would be to program multiple correct values, so that both square and butt caps will reflect as correct answers;
- semi-opaque colours can also be interpreted as a paler shade of the same colour, or possibly a new colour if overlapping another fill – but this could be accommodated by programming the automatic with a function to calculate what the colour will be as a semi-opaque value, or a mixture of the background and foreground colours.

4.7.4 Questionnaire Results

Upon completing the final task the subject was presented with the Prescribed Task Feedback Questionnaire (Appendix E). The responses to the questions follow, along with an explanation and interpretation of the data.

1. What vector graphics software did you use to complete the tasks?

With 9 subjects, Illustrator was by far the most popular editor. This was unsurprising, as the software was prescribed and taught in every subject's current studies curriculum. One subject used CorelDRAW; one subject used Inkscape.

2. Did you have any difficulties interpreting the instructions?

Subjects had no difficulty interpreting the instructions, but one commented that a “back” button would have been useful, as the web application offered no means of returning to a previous step once completed.

3. Which web browser did you use?

Firefox (5 subjects) and Chrome (4 subjects) were the most popular browsers. One subject made use of Apple's Safari, but switched after experiencing troubles.

4. Did you experience any issues or bugs with the system?

Of the eleven respondents, 5 mentioned that the preview of the uploaded SVG appeared very large. This is a result of some additional markup – that differs between editors, which can often be configured when exporting an SVG – but does not affect the automatic marker results.

Four of the respondents also pointed out issues with an incorrect font being rendered in the uploaded SVG preview. Firefox appears to have some problems rendering different fonts in SVG, but once again, this did not affect the automatic marker results.

One subject used Safari (Apple Mac browser), which did not function properly with the web application, and was forced to restart with another browser. This was the same subject who submitted the incorrect files.

5. On a scale of 1(easy) to 5(difficult), how difficult did you find the tasks?

The tasks were designed to be easy. The questionnaire confirms they were, as, with the exception of subjects 8 and 10 who selected “2”, every subject selected the minimum difficulty value of “1” (out of a possible 5). Subject 8 was the same individual who used Safari and submitted the incorrect graphics.

6. Did you trace the image after downloading the graphic, or simply estimate the proportions?

Every respondent selected “traced” – which confirms the template downloads (Illustrator | CorelDRAW | Inkscape) were self-explanatory enough, and a successful starting platform to completing each task.

7. Approximately how many minutes did it take for you to complete all 6 tasks?

With the exception of subject 8, every respondent selected 15-30 minutes. Subject 8 – the same individual who used Safari and submitted the incorrect graphics – selected 30-45 minutes.

Summary

This chapter commenced with a review of the SVG attributes and elements accommodated by the automatic marker. Given that it has only been developed as far as an early prototype, a restricted list of the SVG feature-set has been targeted. This list is constrained to the attributes and elements that focus primarily on essential drawing features – as opposed animation, interactive, filter, structural, and the other more complex categories.

The same six, simple graphics have been used for the testing, evaluation, and surveying – from which the data was provided for the analysis and findings. When recreating these files, some important guidelines have been necessary to follow, specifically around document setup and export settings. These parameters have a marked influence on colour and styling in the outputted markup; and as a result, instructions were provided for the subjects in the first step of the automatic marking web application.

The SVG Features Testing (section 4.2) required manually reviewing the markup produced by Adobe Illustrator, CorelDRAW, and Inkscape, for all six tasks. There were differences, notably: the inclinations of different editors in opting for paths, polylines, lines, or polygons; and CorelDRAW and Inkscape's propensity to nest even one-off shapes within group (`<g>`) elements. Provision for handling such differences has been programmed into the application, but Inkscape's `flow`-type text elements approach has not been accommodated in any way in the current version of the automatic marker.

The Marker Capabilities and Limitations Evaluation (section 4.3) relies on submitting one individual's (the researcher's) correctly recreated six test graphics using three different vector graphics editors – as well as submitting the answer markup back into the web application for comparison against itself – in order to establish how well the automatic marker dealt with the varied markup. This also provided further insight into just how much the markup really varied. Most tasks were well handled, but CorelDRAW produced a few issues with semi-opaque strokes and `stroke-linjoins`. As anticipated, text elements did not compare and process well, with the exception of those created in CorelDRAW.

The Prescribed Tasks Survey (section 4.4) has been used to measure the efforts of the eleven test subjects to recreate the six graphics. The submissions have been marked by both a human (the researcher) and the automatic marker to compare how well it performs. A simple one-mark-per-feature scoring system has been used throughout the testing, so as to avoid overcomplicating the study. The test subjects have also completed a questionnaire to provide any feedback on issues that may have impeded their ability to complete the tasks – which, at their level, should have, and did, provide little challenge. The key findings in the Prescribed Tasks Survey Results (section 4.7.3) revealed: CorelDRAW creates a duplicate shape to apply semi-opaque strokes; colour comparison can be a bit inconsistent, particularly in certain areas of the colour spectrum; `stroke-lincaps` can be impossible to determine by looking at certain types of graphics; and colour opacity can also be interpreted by users as a paler or mixed shade of the same colour. Solutions to these challenges can be fairly easily addressed with further development to the automatic marker, but also demand further research in key areas.

As a case in point: while the DeltaE JavaScript library did indeed prove useful in assessing the accuracy of colours, this, it must be reiterated, is according to the observations of an individual researcher. Drawing any conclusions as to just how inaccurately it performs in certain areas of the colour spectrum requires a larger sample and more specific data.

The Questionnaire Results (section 4.7.4) around the subject experiences using the web application reported that:

- Adobe Illustrator was the most prominent vector graphic editor used in these tests;

- subjects generally had little trouble interpreting the instructions;
- Firefox and Chrome worked well with the web application, but Safari did not;
- the only bugs observed were related to previews of uploaded SVGs, that did not affect the assessment process;
- the tasks were easy for the subjects to recreate;
- and, all except one of the students took between 15 and 30 minutes to complete and upload all six tasks.

Chapter 5: Conclusion

Overviewing the initial aims of this project, this chapter discusses to what extent these were met. Problems that have been encountered, and the implications thereof shed light on what suggestions could be made for future work in this area of research.

5.1 The Automatic Marker for Vector Graphics Drawing Tasks

5.1.1 Study Goals

Vector graphics are an essential component of the modern creative's digital skill-set, be they an illustrator; graphic, multimedia, or web designer. Vector graphics offer a resolution-independent means of creating graphic artwork, and their coordinate- and attribute-based workings have recently found an established standard for Web usage in the form of the SVG format. The popularity of web applications – coupled with modern web browser abilities to parse, display, and manipulate SVG markup – presented an opportunity to create an automatic marking system by combining these technologies.

To help gauge a novice users skills operating vector graphics editing software – such as Adobe Illustrator, CorelDRAW, or Inkscape – this study explored the feasibility of employing an automatic marker, created using Web technologies, to compare and assess an individual's attempts at recreating vector drawing tasks. Using a basic scoring system, the automatic marker processes an uploaded SVG file and compares it to an answer file, then returning a result in the form of a score. As the system was only developed as far as a prototype, there are many limitations to what it can effectively assess, but the progress made here indicates that further development will produce a useful product.

5.1.2 Shape Recognition and Comparison

SVG graphics present numerous advantages over raster graphics when attempting to detect shapes and their associated attributes. SVG markup not only provides a more easily parsable set of data – especially considering that no edge detection is required and overlapping shapes are more immediately distinguishable – but also an accurate set of coordinate, fill, stroke, and other styling parameters.

Initial research in this area (Chapter 2: Background) introduced established algorithms for comparing and measuring the similarity of colours, shapes, and distances – many of which were programmed into the automatic marking application. This literature review, furthermore, provides direction for future developments and features.

5.1.3 Research Questions and Results

In gauging the effectiveness of the automatic marker, the following research questions were addressed:

1. Which SVG properties are exported consistently and reliably enough across popular vector graphics drawing software, in order to be assessed successfully by the automatic marker?

2. What programming languages – and additional frameworks – will prove most effective for developing the proposed automatic marker?

3. Once developed and tested, was the web application useful for testing and marking purposes?

Beginning with question 1: in testing, the automatic marker handled Adobe Illustrator, CorelDRAW, and Inkscape SVG markup convincingly enough to warrant further development. There were a few issues, particularly regarding text in SVG files, but this aside, nothing too complicated to resolve with some relatively straightforward programming.

The web application was built using the client-side framework, Ember.js, which itself is built on HTML, CSS (SASS and Bootstrap), and Javascript (with jQuery). However, the actual shape comparison and scoring component was developed as a modularised piece of JavaScript code that can be 'plugged-in' to a wide range of front/back-end solutions. The current back-end runs on CherryPy (Python) with an SQLite database. Overall,

these technologies formed a solid foundation for the automatic marker and would comfortably accommodate further development and feature additions.

Although the automatic marker was only able to assess basic graphics, it has already proved useful for beginner-level tasks, as well as shown potential to cater for more advanced tasks. The early testing and evaluation phases of this study experimented with markup produced by three different editors. Following this, the application was programmed to handle these permutations – and then tested using six different graphics tasks that were recreated and submitted to the web application by eleven different test subjects. The results showed that the application required a few more features and tweaks, and confirmed that with further development it will become capable of handling increasingly more complex drawing tasks.

5.1.4 Web Application Development and Constraints

The automatic marker was designed for modern, standards-based browsers running on desktop/laptop systems, and not mobile devices; therefore there was no need to accommodate smart phones and tablets, as industry-standard, production-level vector graphics editors are currently not available for such systems.

The interface design was simple and intuitive enough, as proved in the results of the post-tasks-completion questionnaire, but not an area of focus in this study. However, due the application's front-end framework, the interface component can accommodate further development to include a different design, theme, and layout.

5.2 Future Work

During the testing and development phases a number of small, useful features were included in the automatic marker; but adding support for curved path segments – as currently the system only handles straight line connections between points – would be the next major feature to include. But this fell beyond the scope of the project, as it will involve dealing with the far more complex area of organic (non-geometric) shape types. This undertaking, however, could be greatly simplified if the task templates dictate exactly where anchor points are to be laid, so that the automatic marker will assess just the curve(s) control point values. For example, Figure 18 presents two curves constructed using: 3 anchor points (represented using squares) in left instance; and 2

in the right. Although the curves appear identical, control points a1/a2 are plotted using different values to control points b1/b2:

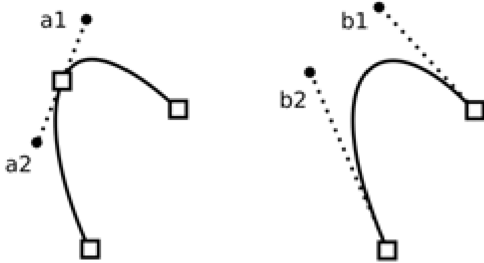


Figure 18: Duplicate curves drawn using different control points.

> *Note* that the dashed lines and solid dots are used to visually manipulate curves in vector graphics editors, and will not appear in the test graphic itself.

By indicating to users where the anchor points are to be placed – using something like the square markers – the automatic marker can simply assess the control point coordinates. Support for curved segments should expand the markers capabilities more than any other single feature, and the success of the element and attribute support currently implemented indicate this would be a feasible prospect – although users would require additional instruction on how to interpret the markers.

An automatic task generator would also prove useful, so that individuals are presented with unique tasks. For example: within certain parameters – i.e. 3 circles in random positions with random fills – generate a unique graphic for each subject. This could be one means of ensuring that users do not copy from one another, while still allowing for equally difficult test scenarios.

Colour is an area that would ultimately require additional surveying – preferably using a sizeable group of subjects, in order to accomodate any physiological – and possibly psychological – differences in human colour interpretation. While the DeltaE JavaScript library did indeed prove useful in assessing the accuracy of colours, it was revealed that there are certain areas of the colour spectrum where it performs in either an overly-strict- or lenient manner – this, it must be restated, is according to the observations of an individual researcher.

The aforementioned features are concerned with aspects of how the automatic marker can be developed further, and ultimately require further programmer/developer time; but another aspect for further study is around what makes shapes more similar? Psychologically, subjectively, and according to the professionals and teachers who use vector graphics editing software. This would lead to more, what could be considered 'accurate' marking criteria, and in turn, assessment results from the automatic marker.

Bibliography

Adobe Systems Incorporated. 2010. Chapter 3: Opening and importing images. Using Adobe Photoshop CS5, Software manual. Adobe Systems Incorporated. California, USA, 50–54.

Beaulieu, A. 2009. A Little Background. Learning SQL, Treseler, M. O'Reilly Media. California, USA, 1–12.

Bidelman, E. 2011. 1: Introduction. Using the HTML5 Filesytem API, Loukides, M and Blanchette, M. O'Reilly Media. California, USA, 1–3.

Bradski, G. and Kaehler, A. 2008. Overview. OpenCV: Computer Vision with the OpenCV Library, Loukides, M. O'Reilly Media. California, USA, 1–15.

Connolly, R. and Hoar, R. 2014. Introduction to Server-Side Development with PHP. Fundamentals of Web Development, Horton, M. Pearson Education, Inc. New Jersey, USA, 323–326.

Cooper, P. 2009. Web Application Frameworks: Rails, Sinatra, and Ramaze. Beginning Ruby: from Novice to Professional, Lowman, M. Apress. New York, USA, 349–410.

Dailey, D., Frost, J., and Strazzullo, D. 2012. Drawing Tools and Utilities. Building Web Applications with SVG, Jones, R. O'Reilly Media. California, USA, 201–207.

Dawber, D. 2013. Chapter 1: The Graphical Web. Learning Raphaël JS Vector Graphics, Thomas, C. and Tuver, L. Packt Publishing. Birmingham, UK, 5–9.

Duce, D, Herman, I., and Hopgood, B. 2002. Web 2D Graphics File Formats. Computer Graphics Forum, 21 (Mar.), 43–64.

Duckett, J. 2011. Structure. , Stone, E. John Wiley & Sons, Inc. Indianapolis, USA, 12–40.

Durand, F. 2010. A Short Introduction to Computer Graphics. MIT Laboratory for Computer Science, 1 (Feb.), 1–7.

Eisenberg, J and Bellamy-Royds, A. 2014. Basic Shapes. SVG Essentials, St. Laurent, S. O'Reilly Media. California, USA, 39–47.

Evjen, B., Sharkey, K., Thangarathinam, T., Kay, M., Vernet, A., and Ferguson, S. 2007. XML Parsers. Professional XML, Barton, W. Wiley Publishing, Inc. Indianapolis, USA, 7–8.

Fawcett, J. and Ayers, D. 2012. Well-Formed XML. Beginning XML, Long, C. John Wiley & Sons, Inc. Indianapolis, USA, 25–42.

Freedman, A. 2014. AngularJS Services. Pro AngularJS, DeWolf, J. Apress. New York, USA, 523–578.

Gasston, P. 2013. 1: The Web Platform. The Modern Web, Yang, S. No Starch Press. San Francisco, USA, 12–20.

Green, B. and Seshadri, S. 2013. Chapter 1: Introduction to AngularJS. AngularJS, St. Laurent, S., and Blanchette, M. O'Reilly Media. San Francisco, USA, 1–9.

Greig, J. 2012. Area. Tutor in a Book's Geometry, Shiletto J. O'Reilly Media. California, USA, 165–199.

Grinberg, M. 2014. Basic Application Structure. Flask Web Development: Developing Web Applications with Python, Blanchette, M and Roumeliotis, R. O'Reilly Media. California, USA, 7–17.

Joshi, O., Gundale, V., and Jagdale, S. 2012. Guidelines in Selecting a Programming Language and a Database Management System. International Journal of Advances in Engineering & Technology, 3 (Mar.), 137–144.

Kelly, C. 2012. Collisions and Entities. Programming 2D Games, Davis, T. CRC Press. Florida, USA, 145–177.

Kirillov, A. 2010. AForge.NET Framework. Detecting some simple shapes in images. http://www.aforgenet.com/articles/shape_checker/.

Koutra, D., Parikh, A., Xiang, J., and Ramdas, A. 2011. Algorithms for Graph Similarityand Subgraph Matching. Carnegie Mellon University: Machine Learning Department Technical Report, (Dec.), 3–8.

Kriegel, A. 2011. Drowning in Data, Dying of Thirst for Knowledge. Discovering SQL: A Hands-On Guide for Beginners, Trukhnov, B. Wiley Publishing, Inc. Indianapolis, USA, 1–11.

McMillan, R. 2015. Tech World Prepares Obituary for Adobe Flash. The Wall Street Journal, (Jul. 20), 1.

McNamara, A. 2001. Visual Perception in Realistic Image Synthesis. Eurographics, 1 (Jan.), 1–14.

Miller, B. and Ranum, D. 2011. Trees and Tree Algorithms. Problem Solving with Algorithms and Data Structures Using Python, Leisy, J. Franklin, Beedle & Associates. Portland, USA, 185–231.

Moffitt, J. and Daoud, F. 2014. Preface. Seven Web Frameworks in Seven Weeks, Tate, B. Pragmatic Bookshelf. California, USA, i–1.

Mokrzycki, W. and Tatol, M. 2012. Colour difference ΔE - A survey. Machine Graphic & Vision, 21 (Oct.), 19–20.

Moore, D., Budd, R., and Wright, W. 2007. Introducing the Frameworks. Professional Python Frameworks: Web 2.0 Programming with Django and Turbogears, Minatel, J. Wrox Press. Birmingham, England, 92–99.

Niederst, J. 2012. Adding Images. Learning Web Design: A Beginner's Guide to HTML, CSS, JavaScript, and Web Graphics, St. Laurent, S. O'Reilly Media. California, USA, 123–125.

Qiao–fang, Z. and Yong–fei, L. 2012. Research and Development of Online Examination System. 2nd International Conference on Computer and Information Application, 2 (Mar.), 1–3.

Rock, N. 2005. Congruence & Similarity Proofs. Standards-Driven Power Geometry I, Team Rock. Team Rock Publishing. California, USA, 4.0–1.

Schuessler, Z. 2015. Delta E 101. DeltaE | Color Difference Algorithms. http://zschuessler.github.io/DeltaE/.

Schwarzi, T. 2012. Collision Detection. 2D Game Collision Detection: An introduction to clashing geometry in games, Golem, B. Amazon Digital Services, Inc. Seattle, USA, 29–61.

Seddon, T. and Waterhouse, J. 2009. Getting Started. Graphic Design for Nondesigners, Landers, R. Chronicle Books. San Francisco, USA, 24–26.

Shankar, A. 2012. Physics Engine Basics. Pro HTML5 Games, Corrigan, L. Apress. New York, USA, 39–58.

Shirley, P., Ashikhmin, M., and Marschner, A. 2010. Raster Images. Fundamentals of Computer Graphics, Peters, A K. CRC Press. Florida, USA, 54–63.

Skiena, S. 2008. Shape Similarity. The Algorithm Design Manual, Wheeler, W and Wylde, A. Springer-Verlag London Limited. London, England, 607–609.

Slavin, S. 1999. Rate, Time, and Distance Problems. All the Math You'll Ever Need: A Self-Teaching Guide, McCarthy, J. John Wiley & Sons, Inc.. New York, USA, 113–115.

Udell, J. 2004. Whatever Happened to SVG?. InfoWorld, 26 (Nov.), 32–33.

W3Techs. 2014. Content Management > Wordpress. Usage statistics and market share of WordPress for websites. http://w3techs.com/technologies/history_overview/content_management/all.

Younker, J. 2008. Web Servers and Web Applications. Foundations of Agile Python Development, Welsh, T. Apress. New York, USA, 312–316.

Zemke, F. 2012. What's new in SQL:2011. SIGMOD Record, 41 (Mar.), 67–73.

Zemke, F. 2008. Appendix C: Built-in Objects. Object-Oriented JavaScript: Create scalable, reusable high-quality JavaScript applications and libraries, Paterson, D. Packt Publishing. Birmingham, England, 318–320.

Appendices

Appendix A: SVG Features Datasheet

Graphic	Element(s)	Coordinates & Dimensions	Fill & Stroke
	<polygon>	X=10; y=10; …	fill=green; …
Hello *World!*			

Appendix B: Recreated Drawing Task Parameters

	Illustrator	CorelDRAW	Inkscape
Colour Space	New document *Profile* set to *Web*, ensuring the colour space is RGB.	New document *Primary color mode* set to *RGB*.	Inkscape provides no options. Documents are implicitly RGB.
Styling	Upon export, the *CSS Properties* field (under *Advanced Options*) set to *Style Attributes* (or *Inline Style*).	Upon export, the *Styling Options* field (under *General Options*) set to *Inline Stylesheet*.	Inkscape does not provide styling options in the export dialogue. Exporting implicitly makes use of *style attributes*.
Font	The *Fonts: Type* set to *SVG*, as opposed to *Adobe CEF*, or, *Convert to outline*.	The *Export Text* set to *As Text*, as opposed to *As Curves*. The *Embed Font in File* option checked and set to *Used Only*.	No options provided. Inkscape uses `<flow…>` elements (However, *Convert to Text* option is available in editing mode).
Size	New document setup with a width and height of 250px.	File > Document Properties… width and height set to 250px.	New document has been adjusted to a width and height of 250px.
Version	Exported as *SVG 1.1*.	Exported as *SVG 1.1*.	Exported as *Plain SVG* (SVG 1.1).

Appendix C: Automatic Marker Instructions

Automatic Marker for Vector Graphics Drawing Tasks

Instructions

In each step, you will be presented with a downloadable graphic. Recreate/trace it in a vector graphics editor of your choice (Adobe Illustrator, CorelDRAW, or Inkscape).

Once who have completed a task, save/export it as an SVG using the export guidelines listed in the column to the right

There are six graphics in all, and a brief questionnaire to complete at the end.

Adobe Illustrator

Upon export, ensure that you check the **Use Artboards** option. The CSS Properties field (under Advanced Options) must be set to **Style Attributes** (or Inline Style).

CorelDRAW

Upon export, the Styling Options field (under General Options) must be set to **Inline Stylesheet**.

Inkscape

Inkscape does not provide styling options in the export dialogue. Simply save the graphic as a **plain SVG** (as opposed to a Inkscape SVG).

Enter your details and begin:

Firstname:

Surname:

Begin

Appendix D: Prescribed Tasks Marking Rubric

Graphic	Student 1		Student 2		Student ...	
	<polygon>		<polygon>		<polygon>	
	fill=		fill=		fill=	
	-opacity=		-opacity=		-opacity=	
	points=		points=		points=	
Notes:	position=		position=		position=	
	stroke=		stroke=		stroke=	
	-dasharray=		-dasharray=		-dasharray=	
	-linecap=		-linecap=		-linecap=	
	-linejoin=		-linejoin=		-linejoin=	
	-opacity=		-opacity=		-opacity=	
	-width=		-width=		-width=	
	Auto-marker	/ 11	Auto-marker	/ 11	Auto-marker	/ 11
	Researcher	/ 11	Researcher	/ 11	Researcher	/ 11
	<ellipse>		<ellipse>		<ellipse>	
	area=		area=		area=	
	fill=		fill=		fill=	
	-opacity=		-opacity=		-opacity=	
Notes:	position=		position=		position=	
	<polygon>		<polygon>		<polygon>	
	fill=		fill=		fill=	
	-opacity=		-opacity=		-opacity=	
	points		points		points	
	position=		position=		position=	
	Auto-marker	/ 10	Auto-marker	/ 10	Auto-marker	/ 10
	Researcher	/ 10	Researcher	/ 10	Researcher	/ 10

Notes:	<polygon> points= position= stroke -opacity= -dasharray= -linecap= -linejoin= -width=		<polygon> points= position= stroke -opacity= -dasharray= -linecap= -linejoin= -width=		<polygon> points= position= stroke -opacity= -dasharray= -linecap= -linejoin= -width=	
	Auto-marker	/ 9	Auto-marker	/ 9	Auto-marker	/ 9
	Researcher	/ 9	Researcher	/ 9	Researcher	/ 9
Notes:	<rect> ×3 area= fill= -opacity= position		<rect> ×3 area= fill= -opacity= position		<rect> ×3 area= fill= -opacity= position	
	<rect> ×3 area= fill= -opacity= position		<rect> ×3 area= fill= -opacity= position		<rect> ×3 area= fill= -opacity= position	
	<rect> ×3 area= fill= -opacity= position		<rect> ×3 area= fill= -opacity= position		<rect> ×3 area= fill= -opacity= position	
	Auto-marker	/ 45	Auto-marker	/ 45	Auto-marker	/ 45
	Researcher	/ 45	Researcher	/ 45	Researcher	/ 45

Hello ***World!*** Notes:	<text> fill= -opacity= font-size position		<text> fill= -opacity= font-size position		<text> fill= -opacity= font-size position	
	<text> fill= -opacity= font-size position		<text> fill= -opacity= font-size position		<text> fill= -opacity= font-size position	
	Auto-marker	/ 10	Auto-marker	/ 10	Auto-marker	/ 10
	Researcher	/ 10	Researcher	/ 10	Researcher	/ 10
Notes:	<circle> area= fill= -opacity= position= stroke= -opacity= -dasharray= -linecap= -linejoin= -width=		<circle> area= fill= -opacity= position= stroke= -opacity= -dasharray= -linecap= -linejoin= -width=		<circle> area= fill= -opacity= position= stroke= -opacity= -dasharray= -linecap= -linejoin= -width=	
	(eyes) <circle> ×2 area= fill= -opacity= position=		(eyes) <circle> ×2 area= fill= -opacity= position=		(eyes) <circle> ×2 area= fill= -opacity= position=	
	<polygon>		<polygon>		<polygon>	

	points= stroke= -dasharray= -linecap= -linejoin= -opacity= -width=		points= stroke= -dasharray= -linecap= -linejoin= -opacity= -width=		points= stroke= -dasharray= -linecap= -linejoin= -opacity= -width=	
	Auto-marker	/ 29	Auto-marker	/ 29	Auto-marker	/ 29
	Researcher	/ 29	Researcher	/ 29	Researcher	/ 29

Appendix E: Prescribed Task Feedback Questionnaire

Thank you for completing the six drawing tasks. Please take a moment to complete this short questionnaire form.

1. What vector graphics software did you use to complete the tasks?

2. Did you have any difficulties interpreting the instructions?

   If so, please explain why:

3. Which web browser did you use?
   Chrome
   Edge
   Firefox
   Internet Explorer
   Opera
   Safari
   not listed

4. Did you experience any issues or bugs with the system?

   If so, please describe them:

5. On a scale of 1(easy) to 5(difficult), how difficult did you find the tasks?

6. Did you trace the image after downloading the graphic, or simply estimate the proportions?

7. Approximately how many minutes did it take for you to complete all 6 tasks?
   less than 10 minutes
   between 10 and 15 minutes
   between 15 and 30 minutes
   between 30 and 45 minutes
   longer than 45 minutes

Appendix F: Automatic Marker Library Source Code

This is the source code for the web application's shape comparison library. As a self-contained component, any front-end (or back-end) could interface with it. It contains the logic that forms the focus of this study, and as such, it has not been necessary to include further source code – namely: the front-end; back-end; and database components. However, a complete copy of the repository can be found at: https://github.com/tabreturn/corvet

```
/**
 * SVG auto-marker library
 * @constructor
 * @param {string} svg - SVG markup to be compared to (answer)
 * @param {string} svg - SVG markup to be compared (submission)
 */

export default {

  Libcorvet: function(answer, submission) {

    this.tolerance = {
      corner:           2,   // disregard corners under this value (degrees)
      points:           5,   // total % of difference between polygon points
      position:         10,  // shapes distance apart
      deltae:           10,  // color difference in fills & stokes
      rectdetect:       2.5, // the degrees any path/poly-to-rect corner can be out

      // for the following: 0 is an exact match; 1 is way off
      area:             0.3,
      fillopacity:      0.2,
      fontsize:         0.2,
      strokemiterlimit: 0.2,
      strokeopacity:    0.2,
      strokewidth:      0.2,
      rx:               0.2,
      ry:               0.2
    };

    // extract shapes and attributes

    this.countShapes = function(svgselector, shape) {
      return document.querySelectorAll(svgselector + ' ' + shape).length;
    };

    this.setShapeAttributes = function(svgselector, total, type, array) {
      for (let i=0; i<total; i++) {
```

```
    let shape = this.getShapeAttributes(
      document.querySelectorAll(svgselector + ' ' +type)[i], type
    );
    array.push(shape);
  }
};

this.relativeToAbsolute = function(shapesarray) {
  for (let i=0; i<shapesarray.length; i++) {
    if (shapesarray[i].x || shapesarray[i].y) {
      shapesarray[i].x =
        parseFloat(shapesarray[i].x) + parseFloat(shapesarray[i].transform.x);
      shapesarray[i].y =
        parseFloat(shapesarray[i].y) + parseFloat(shapesarray[i].transform.y);
    }

    if (shapesarray[i].points) {
      let points = shapesarray[i].points.split(' ');
      let p = '';

      for (let j=0; j<points.length; j++) {
        let xy = points[j].split(',');

        p += (parseFloat(xy[0]) + parseFloat(shapesarray[i].transform.x));
        p += ',';
        p += (parseFloat(xy[1]) + parseFloat(shapesarray[i].transform.y));

        if (j < points.length-1) {
          p += ' ';
        }
      }

      shapesarray[i].points = p;
    }
  }

  return shapesarray;
};

this.SvgShapes = function() {
  this.circles = [];
  this.ellipses = [];
  this.lines = [];
  this.paths = [];
  this.polygons = [];
  this.polylines = [];
  this.rects = [];
  this.texts = [];
};
```

```
this.getShapes = function(svgselector) {
  let sel = svgselector;
  let shapes = new this.SvgShapes();

  this.setShapeAttributes(
    sel,
    this.countShapes(sel, 'circle'),
    'circle', shapes.circles
  );
  shapes.circles = this.relativeToAbsolute(shapes.circles);

  this.setShapeAttributes(
    sel,
    this.countShapes(sel, 'ellipse'),
    'ellipse', shapes.ellipses
  );
  shapes.ellipses = this.relativeToAbsolute(shapes.ellipses);

  this.setShapeAttributes(
    sel,
    this.countShapes(sel, 'rect'),
    'rect',
    shapes.rects
  );
  shapes.rects = this.relativeToAbsolute(shapes.rects);

  this.setShapeAttributes(
    sel,
    this.countShapes(sel, 'text'),
    'text',
    shapes.texts
  );
  shapes.texts = this.relativeToAbsolute(shapes.texts);

  // path/polygon/polyline all as polygons:
  this.setShapeAttributes(
    sel, this.countShapes(sel, 'line'),
    'line',
    shapes.lines
  );
  this.setShapeAttributes(
    sel, this.countShapes(sel, 'path'),
    'path',
    shapes.paths
  );
  this.setShapeAttributes(
    sel,
    this.countShapes(sel, 'polygon'),
    'polygon',
    shapes.polygons
```

```
  );
  this.setShapeAttributes(
    sel,
    this.countShapes(sel, 'polyline'),
    'polyline',
    shapes.polylines
  );
  shapes.lines = this.linesToPolygons(shapes.lines, shapes.polygons);
  shapes.polygons = this.pathsToPolygons(shapes.paths, shapes.polygons);
  shapes.polygons = shapes.polygons.concat(shapes.polylines);
  delete shapes.lines;
  delete shapes.paths;
  delete shapes.polylines;

  let polygondelete = [];

  for (let i=0; i<shapes.polygons.length; i++) {
    if (this.checkIfPolygonIsRect(shapes.polygons[i])) {
      shapes.rects.push(this.polygonToRect(shapes.polygons[i]));
      polygondelete.push(i);
    }
  }

  if (polygondelete.length >= 1) {
    for (let i=polygondelete.length; i>=0; i--) {
      shapes.polygons.splice(polygondelete[i], 1);
    }
  }

  shapes.polygons = this.relativeToAbsolute(shapes.polygons);

  for (let i=0; i<shapes.polygons.length; i++) {
    let polygon = shapes.polygons[i];
    polygon.x = this.getPointsShapeCentre(polygon.points).x;
    polygon.y = this.getPointsShapeCentre(polygon.points).y;
  }

  return shapes;
};

this.checkIfPolygonIsRect = function(polygon) {
  let pp = this.pointsToArray(polygon.points);
  let pc = [];
  pc[0] = this.findAngle(pp[0], pp[1], pp[2], pp[3]);
  pc[1] = this.findAngle(pp[2], pp[3], pp[4], pp[5]);
  pc[2] = this.findAngle(pp[4], pp[5], pp[6], pp[7]);
  pc[3] = this.findAngle(pp[6], pp[7], pp[0], pp[1]);

  let isrect = true;
```

```
  let rt = [
    0   - this.tolerance.rectdetect/2,
    0   + this.tolerance.rectdetect/2,
    90  - this.tolerance.rectdetect/2,
    90  + this.tolerance.rectdetect/2,
    180 - this.tolerance.rectdetect/2,
    180 + this.tolerance.rectdetect/2
  ];

  for (let i=0; i<pc.length; i++) {
    let c = Math.abs(pc[i]);

    if (!(
        c >= rt[0] && c <= rt[1] ||
        c >= rt[2] && c <= rt[3] ||
        c >= rt[4] && c <= rt[5]
       )) {
      isrect = false;
    }
  }

  return isrect;
};

this.dToPoints = function(d) {
  let points = [];

  if (d) {
    let com = d.split(/(?=[mMlLzZ])/);

    for (let i=0; i<com.length; i++) {
      switch (com[i].charAt(0)) {
        case 'm':
          let cm = com[i].substring(1);
          cm = cm.trim();
          cm = cm.split(' ');

          for (let j=0; j<cm.length; j++) {
            if (!points.length) {
              points.push(parseInt(cm[j].split(',')[0]) + 0);
              points.push(parseInt(cm[j].split(',')[1]) + 0);
            }
            else {
              let pl = points.length;
              points.push(parseInt(cm[j].split(',')[0]) + points[pl-2]);
              points.push(parseInt(cm[j].split(',')[1]) + points[pl-1]);
            }
          }

          break;
```

```
        case 'L':
          let cL = com[i].substring(1);
          cL = cL.trim();
          cL = cL.split(' ');

          for (let j=0; j<cL.length; j++) {
            points.push(parseInt(cL[j].split(',')[0]));
            points.push(parseInt(cL[j].split(',')[1]));
          }

          break;
      }
    }
  }

  let p = '';

  for (let i=0; i<points.length; i+=2) {
    p += points[i] + ',';
    p += points[i+1] + ' ';
  }

  return p.slice(0, -1);
};

this.linesToPolygons = function(lines, polygons) {
  let polylines = polygons;

  for (let i=0; i<lines.length; i++) {
    lines[i].points =
      `${lines[i].x1},${lines[i].y1} ${lines[i].x2},${lines[i].x2}`;
    delete lines[i].x1;
    delete lines[i].y1;
    delete lines[i].x2;
    delete lines[i].y2;
    polylines.push(lines[i]);
  }
  return polylines;
};

this.pathsToPolygons = function(paths, polygons) {
  let polypaths = polygons;

  for (let i=0; i<paths.length; i++) {
    paths[i].points = this.dToPoints(paths[i].d);
    delete paths[i].d;
    polypaths.push(paths[i]);
  }
```

```
  return polypaths;
};

this.polygonToRect = function(polygon) {
  let p;
  p = polygon.points.replace(/\ /g, ',');
  p = p.split(",");
  let xmin = p[0],
      xmax = p[0],
      ymin = p[1],
      ymax = p[1];

  for (let i=0; i<p.length; i+=2) {
    xmin = p[i]<xmin ? p[i]:xmin;
    xmax = p[i]>xmax ? p[i]:xmax;
    ymin = p[i+1]<ymin ? p[i+1]:ymin;
    ymax = p[i+1]>ymax ? p[i+1]:ymax;
  }

  let newrect = polygon;
  newrect.type   = 'rect';
  newrect.width  = Math.abs(xmax-xmin);
  newrect.height = Math.abs(ymax-ymin);
  newrect.x      = xmin;
  newrect.y      = ymin;
  newrect.area   = newrect.width * newrect.height;
  delete newrect.points;

  return newrect;
};

this.getTransform = function(shape) {
  try {
    let t = shape.parentElement.getAttribute('transform');
    t = t.replace(/(translate\()|\)/g, '');
    let txy = t.split(",");
    return {x:txy[0], y:txy[1]};
  }
  catch(e) {
    return {x:0, y:0};
  }
};

this.pointsToArray = function(points) {
  let p = points.replace(/,/g , " ");
  p = p.split(" ");
  p = p.map(parseFloat);

  return p;
};
```

```
this.getPointsShapeCentre = function(points) {
  let p = this.pointsToArray(points);
  let xmin, xmax, ymin, ymax;

  for (let i=0; i<=p.length; i+=2) {
    if (p[i]<xmin || xmin===undefined) {
      xmin = p[i];
    }
    if (p[i]>xmax || xmax===undefined) {
      xmax = p[i];
    }
    if (p[i-1]<ymin || ymin===undefined) {
      ymin = p[i-1];
    }
    if (p[i-1]>ymax || ymax===undefined) {
      ymax = p[i-1];
    }
  }

  let midpoints = { x: xmin+xmax/2, y: ymin+ymax/2 };
  return midpoints;
};

this.getShapeAttributes = function(shape, type) {
  let attr = {};
  attr.type = type;

  switch (attr.type) {
    case 'circle':
      attr.x            = shape.getAttribute('cx');
      attr.y            = shape.getAttribute('cy');
      attr.r            = shape.getAttribute('r');
      attr.area         = attr.r*attr.r * Math.PI;
      break;
    case 'ellipse':
      attr.x            = shape.getAttribute('cx');
      attr.y            = shape.getAttribute('cy');
      attr.rx           = shape.getAttribute('rx');
      attr.ry           = shape.getAttribute('ry');
      attr.area         = attr.rx*attr.ry * Math.PI;
      break;
    case 'line':
      attr.x1           = shape.getAttribute('x1');
      attr.x2           = shape.getAttribute('x2');
      attr.y1           = shape.getAttribute('y1');
      attr.y2           = shape.getAttribute('y2');
      break;
    case 'path':
      attr.d            = shape.getAttribute('d');
```

```
      break;
    case 'polygon':
    case 'polyline':
      attr.points       = shape.getAttribute('points').trim();
      break;
    case 'rect':
      attr.width        = shape.getAttribute('width');
      attr.height       = shape.getAttribute('height');
      attr.x            = shape.getAttribute('x');
      attr.y            = shape.getAttribute('y');
      attr.area         = attr.width * attr.height;
      break;
    case 'text':
      attr.x            = shape.getAttribute('x');
      attr.y            = shape.getAttribute('y');
      attr.fontfamily   = getComputedStyle(shape, null).fontFamily;
      attr.fontsize     = parseFloat(getComputedStyle(shape, null).fontSize);
      break;
  }
  attr.fill             = getComputedStyle(shape, null).fill;
  attr.fillopacity      = getComputedStyle(shape, null).fillOpacity;
  attr.stroke           = getComputedStyle(shape, null).stroke;
  attr.strokeopacity    = getComputedStyle(shape, null).strokeOpacity;
  attr.strokewidth      = getComputedStyle(shape, null).strokeWidth;
  attr.strokelinecap    = getComputedStyle(shape, null).strokeLinecap;
  attr.strokelinejoin   = getComputedStyle(shape, null).strokeLinejoin;
  attr.strokemiterlimit = getComputedStyle(shape, null).strokeMiterlimit;
  attr.strokedasharray  = getComputedStyle(shape, null).strokeDasharray;

  attr.transform        = this.getTransform(shape);

  return attr;
};

// comparison helper functions

this.isInt = function(n) {
  return Number(n)===n && n%1===0;
};

this.isFloat = function(n) {
    return Number(n)===n && n%1!==0;
};

this.rgbToArray = function(rgb) {
  let pattern = /rgb\((\d{1,3}), (\d{1,3}), (\d{1,3})\)/;
  rgb = pattern.exec(rgb);

  if (rgb) {
    rgb.shift();
```

```
    rgb = rgb.map(Number);
    return rgb;
  }
};

this.findAngle = function(ax, ay, bx, by) {
  let cy = by - ay;
  let cx = bx - ax;
  let theta = Math.atan2(cy, cx);
  theta *= 180/Math.PI;
  return theta;
};

this.compareDistance = function(ax, ay, bx, by) {
  if (ax===undefined || ay===undefined ||
      bx===undefined || by===undefined) {
    return undefined;
  }
  return Math.sqrt((ax-bx)*(ax-bx) + (ay-by)*(ay-by));
};

this.compareNumber = function(i1, i2) {
  if (i1===undefined || i2===undefined) {
    return undefined;
  }
  return Math.abs(i1 - i2);
};

this.compareProportional = function(i1, i2) {
  if (i1===undefined || i2===undefined) {
    return undefined;
  }

  i1 = Math.abs(i1);
  i2 = Math.abs(i2);

  if (i1 < i2) {
    return i1 / i2;
  }
  else if (i2 < i1) {
    return i2 / i1;
  }
  else {
    return 1;
  }
};

this.arrayMinIndex = function(array) {
  let index = 0;
  let value = array[0];
```

```
  for (let i=1; i<array.length; i++) {
    if (array[i] < value) {
      value = array[i];
      index = i;
    }
  }

  return index;
};

// comparison functions

this.compareColor = function(color1, color2) {
  if (color1===undefined || color2===undefined) {
    return undefined;
  }

  color1 = this.rgbToArray(color1);
  color2 = this.rgbToArray(color2);

  if (color1 !== undefined && color2 !== undefined) {
    return window.DeltaE.getDeltaE00(
      {L: color1[0], A: color1[1], B: color1[2]},
      {L: color2[0], A: color2[1], B: color2[2]}
    );
  }

};

this.findCorners = function(points, tolerance) {
  points = this.pointsToArray(points);
  let corners = [];

  let lastangle = this.findAngle(
                    points[points.length-2], points[points.length-1],
                    points[0], points[1]
                  );

  for (let i=2; i<=points.length; i+=2) {
    let angle;

    angle = this.findAngle(
              points[i-2], points[i-1], points[i], points[i+1]
            );

    if (i >= points.length) {
      angle = this.findAngle(
                points[0], points[1],
                points[points.length-2], points[points.length-1]
```

```
            );
    }

    if (Math.abs(angle-lastangle) > tolerance) {
      corners.push(points[i-2], points[i-1]);
    }

    lastangle = angle;
  }

  return corners;
};

this.getHausdorffDistance = function(points1, points2) {
  let h = 0;

  for (let i=0; i<points1.length; i++) {
    let d = this.compareDistance(
              points1[i], points1[i+1],
              points2[i], points2[i+1]
            );

    if (d > h) {
      h = d;
    }
  }

  return h;
};

this.alignPolygonPoints = function(points1, points2) {
  let p2 = points2;
  let offsets = [];

  function shiftPairs(points) {
    points.splice(0, 0, points[points.length-1]); points.pop();
    points.splice(0, 0, points[points.length-1]); points.pop();
  }

  for (let i=0; i<points1.length; i+=2) {
    let sumofd = 0;

    for (let ii=0; ii<p2.length; ii+=2) {
      let d = this.compareDistance(
                p2[ii], p2[ii+1], points1[ii], points1[ii+1]
              );
      sumofd += d;
    }

    offsets.push(sumofd);
```

```
    shiftPairs(p2);
  }

  var offest = this.arrayMinIndex(offsets);
  p2 = points2;

  for (let i=0; i<offest; i++) {
    shiftPairs(p2);
  }

  return p2;
};

this.comparePolygon = function(p1, p2) {
  if (p1 && p2) {
    p1 = this.findCorners(p1, this.tolerance.corner);
    p2 = this.findCorners(p2, this.tolerance.corner);

    if (p1.length !== p2.length) {
      return `${p2.length/2} corners found (should be ${p1.length/2})`;
    }
    else {
      p2 = this.alignPolygonPoints(p1, p2);
      return this.getHausdorffDistance(p1, p2);
    }
  }
};

this.compareShape = function(shape1, shape2) {
  let comparisons = {
    // common
    position    : this.compareDistance(shape1.x,shape1.y, shape2.x,shape2.y),
    area        : this.compareProportional(shape1.area, shape2.area),
    fill        : this.compareColor(shape1.fill, shape2.fill),
    fillopacity : this.compareNumber(shape1.fillopacity, shape2.fillopacity),
    stroke      : this.compareColor(shape1.stroke, shape2.stroke),
    // ellipses
    rx          : this.compareProportional(shape1.rx, shape2.rx),
    ry          : this.compareProportional(shape1.rx, shape2.rx),
    // polygons (and paths)
    points      : this.comparePolygon(shape1.points, shape2.points),
    fontsize    : this.compareProportional(shape1.fontsize, shape2.fontsize),
  };

  if (this.isFloat(comparisons.stroke) || this.isInt(comparisons.stroke)) {
    comparisons.strokeopacity     = this.compareNumber(
                                        shape1.strokeopacity,
                                        shape2.strokeopacity);
    comparisons.strokewidth       = this.compareProportional(
                                        shape1.strokewidth,
```

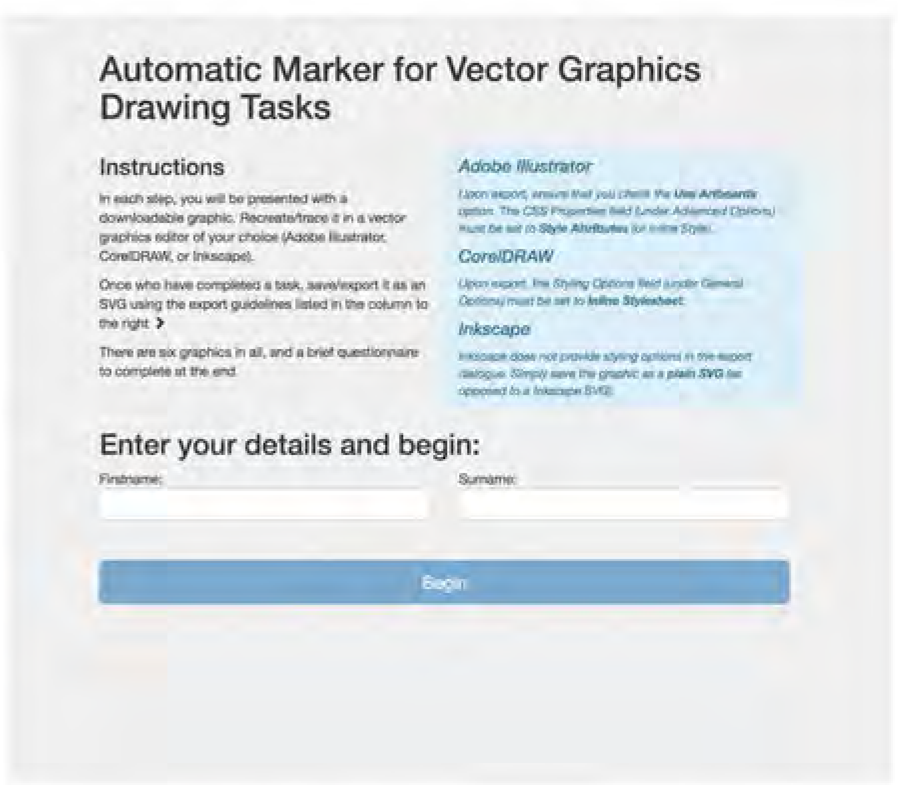
```
                                    shape2.strokewidth);
    comparisons.strokemiterlimit  = this.compareProportional(
                                    shape1.strokemiterlimit,
                                    shape2.strokemiterlimit);
    comparisons.strokelinecap     = (
      shape1.strokelinecap === shape2.strokelinecap);
    comparisons.strokelinejoin    = (
      shape1.strokelinejoin === shape2.strokelinejoin);
    comparisons.strokedasharray   = (
      Boolean(shape1.strokedasharray) && Boolean(shape2.strokedasharray));
  }

  return comparisons;
};

this.getMostSimilarShapes = function(comparisonresults, criterion) {
  let r = {};

  for (let shape in comparisonresults) {
    r[shape] = [];
    let ansid = 0;

    for (let i=0; i<comparisonresults[shape].length; i++) {

      if (comparisonresults[shape][i].id.ans >= ansid) {
        r[shape].push([]);
        ansid ++;
      }

      for (let ii=0; ii<comparisonresults[shape].length; ii++) {
        if (comparisonresults[shape][i].id.ans === ii) {
          r[shape][ii].push(comparisonresults[shape][i]);
        }
      }
    }

    let indv = [];

    for (let i=0; i<r[shape].length; i++) {
      let ii = 0;

      while (ii < r[shape][i].length) {
        let id =  r[shape][i][ii].id.ans;

        if (indv[id] === undefined ||
            r[shape][i][ii][criterion] < indv[id][criterion]) {
          indv[id] = r[shape][i][ii];
        }

        ii ++;
```

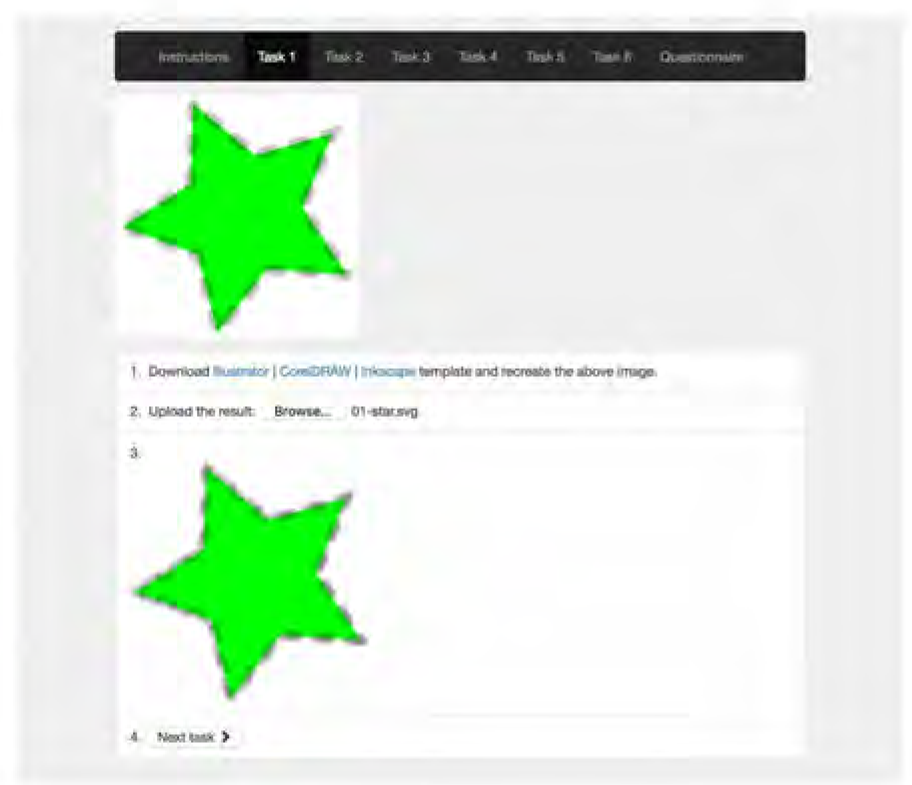
```
      }
    }

    r[shape] = indv;
  }

  return r;
};

// marker functions

this.compareAllShapes = function(ansshapes, subshapes) {
  let shapes = [];
  let candidates = {};

  for (let k in ansshapes) {
    shapes.push(k);
    candidates[k] = [];
  }

  for (let i=0; i<shapes.length; i++) {

    for (let ii=0; ii<ansshapes[shapes[i]].length; ii++) {

      for (let iii=0; iii<subshapes[shapes[i]].length; iii++) {
        let r = this.compareShape(
         ansshapes[shapes[i]][ii],
         subshapes[shapes[i]][iii]
        );
        r.id = {ans:ii, sub:iii};
        candidates[shapes[i]].push(r);
      }
    }
  }

  return this.getMostSimilarShapes(candidates, 'position');
};

this.removeProblemShapes = function() {
  try {
    var clippath = document.querySelector('defs clipPath');
    clippath.parentNode.removeChild(clippath);
  } catch(e) {
    console.log('* no problem elements removed');
  }
};

this.gatherSubmissionAnswer = function() {
  this.removeProblemShapes();
  let subshapes = this.getShapes(submission);
```

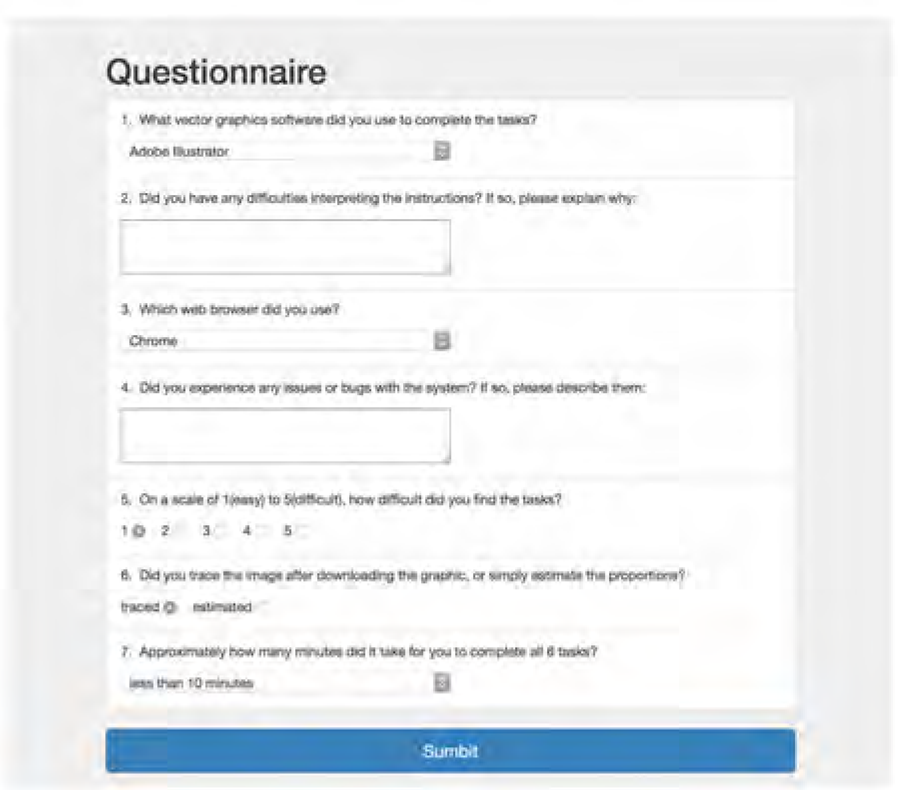
```
  let ansshapes = this.getShapes(answer);
  return this.compareAllShapes(ansshapes, subshapes);
};

this.calculateResult = function() {
  let r = this.gatherSubmissionAnswer();
  let score = 0;
  let scoresheet = 'scoresheet:\n';

  function scoreAdd(attr, mark) {
    score += mark;
    scoresheet += `${attr}: ${mark}\n`;
  }

  function scoreWithinTolerance(attr, value, correct, tolerance, mark) {
    if (value > correct-tolerance && value < correct+tolerance) {
      scoreAdd(attr, mark);
    }
  }

  function scoreTrueFalse(attr, value, correct, mark) {
    if (value===correct) {
      scoreAdd(attr, mark);
    }
  }

  for (let k in r) {

    for (let i=0; i<r[k].length; i++) {

      scoresheet += `---\nshape (${k}[${i}]): 1\n`;
      score += 1;

      for (let attr in r[k][i]) {
        let a = r[k][i][attr];

        if (attr==='position') {
          scoreWithinTolerance(attr, a, 0, this.tolerance.position, 1);
        }

        if ((a || a===0) && attr!=='id' && attr!=='position') {

          switch (attr) {
            case 'area':
              scoreWithinTolerance(attr, a, 1, this.tolerance.area, 1);
              break;
            case 'fill':
              scoreWithinTolerance(attr, a, 0, this.tolerance.deltae, 1);
              break;
            case 'fillopacity':
```

```
                  if (r[k][i].fill !== undefined) {
                    scoreWithinTolerance(attr, a, 0, this.tolerance.fillopacity, 1);
                  }
                  break;
                case 'fontsize':
                  scoreWithinTolerance(attr, a, 1, this.tolerance.fontsize, 1);
                  break;
                case 'points':
                  scoreWithinTolerance(attr, a, 0, this.tolerance.points, 1);
                  break;
                case 'stroke':
                  scoreWithinTolerance(attr, a, 0, this.tolerance.deltae, 1);
                  break;
                case 'strokedasharray':
                  scoreTrueFalse(attr, a, true, 1);
                  break;
                case 'strokelinecap':
                  scoreTrueFalse(attr, a, true, 1);
                  break;
                case 'strokelinejoin':
                  scoreTrueFalse(attr, a, true, 1);
                  break;
                case 'strokeopacity':
                  scoreWithinTolerance(attr, a, 0, this.tolerance.strokeopacity, 1);
                  break;
                case 'strokewidth':
                  scoreWithinTolerance(attr, a, 1, this.tolerance.strokewidth, 1);
                  break;
              }
            }
          }
        }
      }

      console.log(scoresheet);
      console.log('comparison data:', r);
      console.log('total score:', score);
      return score;
    };

  }

};
```

Appendix G: Web Application Screenshots




Instructions
Task 1
Task 2
Task 3
Task 4
Task 5
Task 6
Questionnaire
1. Download Illustrator | CorelDRAW | Inkscape template and recreate the above image.
2. Upload the result: Browse... 01-star.svg
3.
4. Next task


Questionnaire
1. What vector graphics software did you use to complete the tasks?
Adobe Illustrator
2. Did you have any difficulties interpreting the instructions? If so, please explain why:
3. Which web browser did you use?
Chrome
4. Did you experience any issues or bugs with the system? If so, please describe them:
5. On a scale of 1(easy) to 5(difficult), how difficult did you find the tasks?
1 2 3 4 5
6. Did you trace the image after downloading the graphic, or simply estimate the proportions?
traced estimated
7. Approximately how many minutes did it take for you to complete all 6 tasks?
less than 10 minutes
Sumbit