Vectorization and Compression of Animated Cartoon Videos

Bassam Helal

September 2020

Bassam Helal
809293

Abstract

Table of Contents

1 Introduction

2 Background

This document makes no assumption of the reader's knowledge in computer graphics and its related fields, thus, this section will introduce the reader to some of the required knowledge needed for understanding this project. Firstly, we will introduce raster graphics, the current de facto standard for graphics data representation. We will then introduce its counterpart vector graphics, a different way to store graphics data that addresses some of the issues with raster graphics. We will then give a brief primer on image compression and some common methods of reducing the overall size of an image. Finally, we will explain how image graphics tie into video graphics and describe some of the few differences between the two.

2.1 Raster Graphics

Raster graphics is the current most popular way of representing and storing graphics data. This representation format stores the image data as a grid of pixels, a pixel (picture element) being the color of the image at a particular point in the grid. Colors can be represented in a variety of ways but the most common is RGB with RGBA also being found in some formats. Each channel has a bit depth which details the number of possible colors that the channel can contain with 8 bits per channel being the current most common but higher values exist as well such as 10 and 12 bits per channel. Thus the image data is a 2 dimensional array of pixels which could be 3 or 4 channels of 8 bit values, thus an image can also be represented as a 3 dimensional array with the last dimension being 3 or 4 values in size representing the 3 or 4 channels of that pixel. The resolution of an image is the number of pixels it contains, which is the image's width multiplied by its height, higher resolution images have more detail but are also more expensive to store as there is more data. Raster graphics are the most popular form of image and generally graphics representation, they allow for easy representation and computationally cheap as well since all the data is stored with no extra work needed. The main disadvantages of raster graphics have to with image resolution and detail, namely that images will show pixelation artifacts when they are displayed at a higher resolution than they are actually stored at, thus the image appears to show the raw pixels. To compensate for this, higher and higher resolution images are needed but those come at the cost of size.

2.2 Vector Graphics

2.3 Image Compression

2.4 Video Graphics

2.5 Toolchain

This project uses a variety of popular tools and libraries to achieve its aims and objectives both efficiently and quickly, this section will briefly go over the most important tools used. This project uses Node.js as its runtime platform, Node.js is a cross-platform runtime environment that executes JavaScript code on a machine instead of the usual JavaScript runtime, the browser. The choice for Node.js was made paradoxically because of both a familiarity with the runtime and the JavaScript ecosystem as well as a strong desire to further enhance our knowledge and experience in both the platform and ecosystem. Node.js allows us to access the ever-growing JavaScript ecosystem which includes very powerful libraries and frameworks for both front-end designs and back-end computationally heavy workloads. Node.js uses JavaScript as its runtime language, while JavaScript is a powerful and excellent language for rapid-prototyping, it is also a very error-prone language as it has no typing checks in place. TypeScript is a programming language that is a strict superset of JavaScript that adds typing features onto JavaScript and transpiles to performant, cross-platform compatible JavaScript by means of a TypeScript compiler. TypeScript was chosen as the programming language of choice for the project for its many benefits not limited to its added type safety. The JavaScript ecosystem provides us with some powerful libraries and frameworks for application development, this project makes use of some excellent noteworthy tools such as Electron, GPU.js and OpenCV among other tools. Electron allows developers to write desktop applications using web technologies, it essentially is a Chromium browser with two processes, a front-end process(es) called the "renderer" which is similar to a Chromium web page that displays HTML, and a back-end process called the main process which is a Node.js process. Electron combines Node.js and Chromium to allow a developer to write a native desktop application using the same technologies used on the web, this has its own advantages and disadvantages but in our case, it means we can quickly develop a GUI desktop application that can access our Node.js code without resorting to web servers and other more complicated means. GPU.js is a JavaScript library that allows developers to run code on the host machine's GPU making use of the GPGPU (General-Purpose computing on Graphics Processing Units) paradigmGeneral-purpose computing on graphics processing units. This can allow for very large performance improvements and speedups for computationally expensive mathematical computations. In summary, the project uses the Node.js runtime for its large ecosystem, the TypeScript programming language for its added type safety benefits, Electron to easily display a GUI and GPU.js to run code on the GPU for additional performance when needed.

3 Related Work

3.1 Current Research

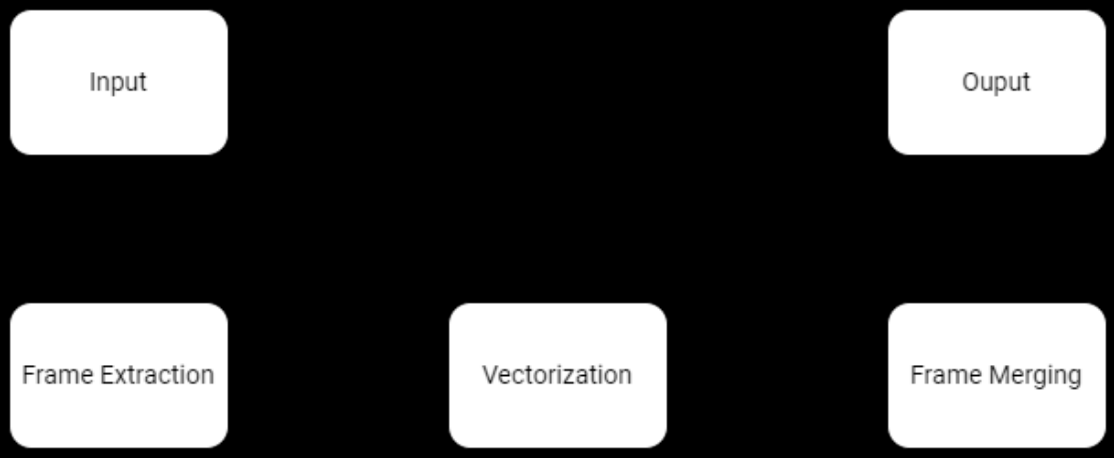
3.2 Current Applications

3.3 Existing Limitations

4 Implementation

The following section will go into high level detail describing the implementation details of the project. While this section may seem rather long, it is in fact made dense to ensure conciseness while keeping all the necessary details explained. This is because the project explores many different ideas and implementations to try to achieve its goals as will be evident by the multiple vectorization methods used. Firstly, we introduce the high level process pipeline of the project from input to output. Then, we explore in depth some of the many methods of vectorization used in the project. Finally, we describe the limitations and drawbacks that were found during the implementation.

4.1 Pipeline



The above image shows the high level process pipeline of the project from input to output with each phase of the pipeline further detailed in its own section below. Firstly, we explain the expected input video and its ideal characteristics that will result in a higher accuracy output. Then, we establish the expected and desired output and detail its expected and ideal characteristics before further discussing any of the processing. Then, the first phase of the processing begins, the frame extraction, where the frames of the video are extracted to be used further down the pipeline. Next, the main phase of the processing, the vectorization of a single frame, we describe and detail the multiple methods and approaches that were used in order to gain the highest level of accuracy and compression rate on each frame. Finally, the final stage of the processing, frame merging combines the vectorized frames back into a playable video form with additional computation executed to add favorable traits such as further reduced file sizes.

4.1.1 Input

Firstly, it is important to understand the input that the process will be expecting and detail what makes an ideal input file that will produce the best results. The minimum requirement is that the input file be any video, but more specifically, this project aims to work on animated cartoons, so while any video will technically work, for the purpose of this project, a video of an animated cartoon is expected. However, not all animated cartoons are equal, in fact there is a great deal of variation between cartoons in terms of art style, animation method, and many other artistic effects that can heavily influence the output of the process. Thus, we must define the "ideal" input file, or more precisely, the input file that will produce the best results because the process was designed with those files in mind for reasons to do with the limitations of vectorization. The ideal input video is one that is most friendly towards vectorization, meaning it has the most vectorizable qualities and characteristics with as few incompatibilities as possible, these were mentioned in depth in the Existing Limitations section but will be briefly repeated within this new context.

Since vectorization is heavily dependent on the color variation of the original raster image, an ideal input image would be one with discrete color regions made up of a single color, with no noise, gradients, shadows, lighting or any other effects, essentially an image made of a set of color blocks that are very discrete from one another. In addition to this, it is important to note that we are dealing with moving pictures and it is also ideal to have as little noise as possible *between* frames, this can be an artifact of compression or rasterization but can also be seen as an artistic style choice as shown in Cartoon Network's *Ed, Edd n Eddy*. Obviously very few real world media satisfies these conditions, but a surprising number of cartoon videos can come very close to this description, with very few complications such as shadows and light gradients that are usually in the background or are less important elements of the image. An example of a real world cartoon that has these ideal qualities is Nickelodeon's *The Fairly OddParents* which uses discrete color regions, very few (if any) gradients and no complex shadows or lighting. On the other extreme end cartoons with heavy use of photographs or heavy noise are the least ideal as well as any 3D cartoons, ie those that make use of 3D effects such as shadows, lighting and many complex gradients. Examples include Hanna Barbera cartoons prominent from the 1960s to the 1980s such as *Top Cat* and *The Flinstones* and modern 3D animations such as Pixar Animation Studios' *Toy Story* and *The Incredibles*. In summary, the minimum required input is simply any video but the expectation is that it will be an animated cartoon video and the ideal input being an animated cartoon video that has highly vectorizable qualities.

4.1.2 Output

Before discussing any of the processing, it is important to first understand the expected output that the process will generate and detail the expected and ideal

characteristics of the output. In essence, the output is a playable video that uses vector graphics to store the same data as the input file but with the added benefits of vector graphics, namely improved scalability and reduced size. Ideally, the output will be visually indistinguishable from the original but with a smaller size essentially achieving so called *Visually Lossless Compression*. However, since the video is using vector graphics, it will be much more scalable in comparison and essentially void of the pixelation problem that plagues traditional raster graphics. The size difference should especially be evident when taking the upscaling ability into account since the new video can be effective at any resolution display whilst the original will show pixelation once it is displayed on high enough resolution displays. Thus, an input video in 480p could likely produce an output video of larger size, and an input video in 720p could likely produce an output of similar size to the original, but the new vector graphics videos will be scalable to any resolution such as 4K or higher where the originals will show major pixelation at those resolutions. One important thing to note is that the output will still be frame based, this means the video data is frames of images moving at some given framerate, as is the case with raster video formats. This is in contrast to what vector animations are often represented in (usually for UI animations) and that is using some form of interpolation making the playback framerate independent, that is not the case here as that would add much more complexity. This frame-based characteristic of the output is important because it means we can leverage existing theories and concepts from raster videos mainly with regard to compression as is made clearer in the Frame Merging section. This frame based vector graphics video output makes the process as simple as possible while still achieving the desired benefits of vector graphics into the original raster video, namely improved scalability and reduced size.

4.1.3 Frame Extraction

The frame extraction phase has a single purpose and that is to convert the input video into a some form of medium that can be vectorized, namely images. Since the input video will always be frame based, we can thus convert a video into a set of vectorizable units that accurately represent its data, frames. Each frame of the video will be represented in an image which we can then vectorize in the next step. To do this, we use the industry standard tool, FFmpeg to create a directory of PNG images for each frame of the video using the command:

```
ffmpeg -i input.mp4 outputDirectory/%d.png -y
```

This is so that the next step (vectorization) can be done on each frame can be independently and can more easily be parallelized, since the vectorization of a single frame does not depend on any other frames around it. While in theory one can skip this step and read each frame of the video performing the vectorization step in serial for each frame, it is much more efficient to split the workload first by first dividing the video into frames stored on disk and then perform the vectorization step, now

with the additional benefit of being able to run the vectorization step on multiple frames at once.

4.1.4 Vectorization

The most important and most difficult step of the process is vectorization, which converts frames from raster graphics to SVG vector graphics. The expected behavior of this step is extremely simple, a raster graphics input image will be converted to a vector graphics output image ideally one with the highest possible accuracy and lowest size possible. While the expected behavior is extremely simple, the actual process is extremely difficult and complicated and there are many possible correct approaches to this problem that yield varying results. These processes are detailed more thoroughly in the Vectorization Methods subsection further below with their advantages and disadvantages. From a high level standpoint however, all the approaches aim to achieve the same result, that is to transform the data from one representation format (raster) to another representation format (vector). This inherently means that there will be some form of data loss with the goal being to achieve what is called *Visually Lossless Compression* while gaining the benefits of vector graphics such as scalability in addition to the aforementioned smaller size. To achieve this, we convert the input image into an SVG image made up only of one SVG element, the powerful `path` element. As mentioned earlier the `path` element allows us to declare a path of arbitrary shape by declaring the commands used to draw it. An image can be represented as a set of discrete connected color regions that together form the whole image, these can be done using the path element which can then be filled with a single color allowing us to create single-color regions of arbitrary shape. Across all tried approaches, this is the common method used to create a vector graphics image, with the only differences being in how these color regions are parsed and determined from the original raster image.

4.1.5 Frame Merging

The final processing step is frame merging, the step that will combine all frames (now in glorious vector graphics) back into some playable video form. This is the step where the most optimizations and improvements can be made with regards to file size. Since we are returning back to the realm of video, i.e. moving pictures, we can make use of the existing compression technology and theory that is used in raster based video to drastically reduce file size. This is true because the output is still frame based just as raster videos are, and thus the same theories can be applied, this would not be true if the video was not frame based such as using interpolation or some other advanced method. The main concept to be brought over from current raster video compression is compression through leveraging temporal redundancy in a video as mentioned in the Video Graphics section. In this case, we can join a series of similar frames into what we call a frameset which will have a root or anchor frame. Only the root frame will be completely stored, all successive frames in the frameset will only be stored as the delta between the root frame. This is currently

the only concept that is taken from raster video to reduce size but as mentioned, since the video is still frame based, many of the same ideas can be directly ported into the vector graphics world for further size reduction. Many more compression optimizations can be done during this step to further reduce the size of the final video, hence the importance of the frame merging step within the process pipeline.

4.2 Vectorization Methods

As previously mentioned, many different approaches and methods were used to tackle the vectorization step of the pipeline in order to achieve the best possible results, or sometimes even acceptable results in some cases as will be detailed below. Three main approaches were used to tackle the problem, a color quantization based approach and two connected component labelling based approaches, one using edges as components and the other using colors as components. One detail to note is that multiple approaches were not planned early on and were in fact researched and explored as the project developed and as results began to appear. This is noteworthy because each successive approach aimed to fix the issues with the previous one, only to introduce new issues of its own. We will explain all three approaches in depth in their own subsections below as well as give an introduction to the theory of connected component labelling as two of the three approaches are based on this.

4.2.1 Color Quantization Approach

The first approach is based on color quantization, or simply, reducing the total number of distinct colors in an image while still aiming for *Visually Lossless Compression*. This approach was chosen because it is the one used in the *ImageTracer.js* library that was used as an initial blueprint and foundation for the project's codebase. Color quantization can be implemented in a variety of different algorithms but essentially any 3-dimensional (or 4-dimensional if including alpha) clustering algorithm can work because it is in essence a classification problem. All algorithms have the same expected behavior though; given an image and a desired number of colors n, find the n-size set of colors that best represents this image. Each pixel in the image is then changed from its original color to the color in the reduced palette that is closest to its original color, and thus the image has been reduced since the image has only n distinct colors. Each distinct color can then be assigned a single SVG path element, this is like a layer because it need not necessarily be connected, thus we end up with an SVG image made up of n number of SVG path elements, each a single color.

Color quantization is an excellent way to reduce the size of an image drastically while making it visually similar to the original and it is usually an extremely fast and efficient process. Their are a few drawbacks though, the first being that it performs much worse when attempting to reduce images with gradients and noise and there will be a visually obvious loss of data, similar to the issues that affect vectorization. However, in our case, the larger issue is to do with the manually inputted number

of colors that the algorithm requires. The number of colors that are given to a color quantization algorithm can drastically change all of accuracy, performance and file size all at the same time. A high number such as 1,000 colors will yield very high accuracy for most images but will have much less size reductions and have significant performance impacts. Even ignoring performance, choosing the correct number of colors that will yield the highest compression rate while also being visually identical in quality is another major task in itself, especially when considering that not all frames or images have the same color space. The truth is, while color quantization is an excellent way to reduce file size and still achieve *Visually Lossless Compression*, it depends too heavily on the provided number of colors and computing this color deterministically is another major problem of its own.

4.2.2 Connected Component Labelling (CCL)

A more logical way to approach the problem is to approach it as a partitioning problem, meaning we want to partition the image into a set of discrete adjacent parts that altogether make up the whole image. This is because that is exactly how the end result SVG will be created, as a set of SVG path elements that together form the whole image as if they are parts of a partition. This way of thinking lends itself extremely well to Connected Component Labelling (CCL) which is exactly that, dividing the image into a partition of connected components that together form the whole image. Connected Component Labelling does not in itself describe any particular implementation and thus can be implemented in many different ways depending on the context and desired outcome. However, the basic idea is that a single pass is made across the image and each pixel is checked to see if it fits within its neighbors' components, if so it is added to that component, if not it is the first in a new component. A second pass may be made to merge any components that are adjacent but this can be done in a single pass by checking to merge components after the initial component assignment. The concept itself is relatively simple with the intricate details left to the implementation which can heavily affect the results depending on how it is implemented and approached. Two CCL based approaches were used to attempt to tackle the vectorization problem, each differing only in the means by which a "component" is defined, the first using edges as components and the second using pixel colors as components.

4.2.3 Edge Based CCL Approach

The first approach using CCL is based on defining a component as an edge of the image. Edges can be easily detected using any edge detection algorithm, for this we used OpenCV's Canny edge detection but the actual edge detection is not the important factor here. For higher accuracy and confidence, a large sample size of edge detection passes with different parameters was averaged to create a single averaged edges image. This image is a monochrome image in which, a pixel with value of 0 is never an edge in any pass and a pixel with a value of 255 is an edge in every pass, thus we need to use some thresholding to determine what defines an edge.

Once that is determined however, the CCL algorithm can pass through the average edges image to find the connected edges that make up the image. The reasoning behind this is that in theory, this will return the edges of all the polygons that make up the image and thus by filling them with their average color, we can recreate the image using color regions. However, this is much easier said than done, and indeed there are many edge cases that need to be noted, specifically the effect of a single pixel. A single missing pixel could mean a polygon is incomplete, a single extra pixel between two unrelated polygons could mean the polygon is incorrect, basically the algorithm is very sensitive to any discrepancies within the averaged image. This can be tackled using further processing and analysis to undo some of these effects, but this would be very complicated and time consuming. In the end however, the edge based CCL approach did not yield the expected results and was very disappointing, although theoretically it can be perfected with time, that was something that was very limited and valuable and thus a new approach was explored.

4.2.4 Color Based CCL Approach

The second approach using CCL is based on defining a component as a region of similar colors based on the pixel values. This approach is the most logical way to approach the problem solely based on the desired output, i.e., SVG path elements of a single color each representing a color region. Since each pixel contains its color information, we can cluster pixels that are adjacent and have colors that are within a given delta threshold, that is to say, they are close enough in color to be part of the same color region. In essence we are performing a kind of color quantization since there will be reduced colors at the end of the process but unlike classic color quantization mentioned in the Color Quantization Approach section, the desired number of colors is no longer an input, solving the issue with that approach. As with other CCL approaches, the image is looped on and each pixel is queried to check whether it is eligible to be part of any of the components of its preceding neighbors, if so it joins that component, if not it is assigned its own new component where it is the first element. The reasoning behind this approach is that in theory each color region in the original image will naturally form a cluster which can then be reduced to a single color that is an average of all the pixels in that region. Thus, the image is transformed from a grid of unrelated pixels to a partition of adjacent color regions each being a single color that altogether form the entire image.

While being fairly complex and computationally expensive, this approach yields shockingly positive results both in terms of size reduction and image accuracy with any data loss incurred being minor. However, upon further inspection, a huge issue became obvious; the total number of components that was created was surprisingly high, an image with at most perhaps a few thousand color regions results with tens of thousands of color regions. The distinct colors were still reduced so color quantization was successful but an efficient partitioning was not and upon further investigation and research, it was clear that the culprit was anti-aliasing. Anti-aliasing is an effect popular in raster graphics that is used to make edges less sharp by

adding blurs between them, and in the case of two different color regions, pixels with colors *in between* those two colors. What ends up happening is that our algorithm cannot determine which region these *in between* pixels are because their colors are not close enough to either region and so they are assigned their own region. This is shown when we analyze the number of components with less than 5 total pixels and the result is a whopping 97% of all regions have less than 5 pixels in total and upon investigating, it is clear that these are the pixels at the edges of color changes and thus the ones responsible for anti-aliasing.

4.3 Limitations and Drawbacks

5 Software Engineering

This section will detail the software engineering aspects of the project's development, specifically the methodology used to develop, the risks of the project and the software testing techniques used in the project's development.

5.1 Methodology

The software methodology used for the development of this project is a very lean agile methodology aptly named, Lean-Agile. This methodology inherits heavily from modern agile methodologies following all the core principles and values but adds principles adapted from lean manufacturing, particularly emphasizing minimal waste and minimal prediction. While a more commonly known methodology such as Scrum could have been used, it seemed more appropriate to use a methodology that has extreme unpredictability built into it nature especially during the growth phase of the COVID-19 pandemic, which caused many unpredictable events. The pandemic's potentially adverse effects on the development of the project meant that it was critical to use a methodology that is extremely efficient and has little fixed planning with a very easy way of adapting to changes. This proved to be extremely beneficial, more so than expected because of the many major roadblocks that were faced during development which are further explained in detail in the Results section. The constant occurrence of very negative results, especially after a complex and time-consuming amount of work put in meant that plans had to change and adapt rapidly in accordance to this, which is common and expected in any research based project such as this one. As a result, while the Lean-Agile methodology is an unorthodox methodology choice, it was absolutely the correct one to use and has been the saving grace of the time management of this project since very few other methodologies could have allowed for the rapid adaptation and movement of this project.

5.2 Schedule

While the Lean-Agile methodology is not particularly well suited to long term predictions, it is still important to have a schedule in place in order to keep track of time and to have a general idea of how many tasks are left and how much time is available to complete them, even if the predictions of time may be incorrect. This is why a Gantt Chart was used to plot a schedule containing the general phases of the project's development against the time allocated to work on the project.

Gantt Chart

- Frame Managing (video to frames and frames to video)
    - Video to frames (ffmpeg)
    - Read frame image data
    - Write frame image data (SVG and PNG)
    - Combine frames to a playable video
- Vectorization
    - Core programming (ImageTracerJS)
    - Edge detection
    - Connected Component Labelling
    - Path/Polygon parsing
- Writing
    - Latex setup
    - User evaluations
    - Resource compiling
    - Core writing

5.3 Risks

Naturally, a project of this scale that is very experimental in nature has many multi-faceted risks associated with it that can severely impact the development of the project if not correctly mitigated. As such, it is important to detect these risks as early as possible and have detailed and usable mitigation strategies to counter such risks and avoid their possibly devastating effects.

Risks Chart

- Bad time management
- Data Loss
- Hardware performance issues
- Tools too difficult
- Tools not cooperative
- Personal issues (stress, burnout)
- Covid related crap
- Poor results
- Catastrophic fatal disaster ;)

5.4 Testing

Testing is an important part of software engineering, as it allows developers to ensure that the program behaves as expected even after changes to the code, this means that refactors or new implementations can easily be done since any breaking changes will fail the tests that previously passed. This makes developing new changes or code more reliable since there is a pillar of trust knowing that if the tests pass, then the software is behaving correctly, assuming of course the tests are well written and defined. In order to ensure that the application is behaving in the expected way, the project needed some kind of testing facilitation. A variety of classic testing methodologies and philosophies were used in this project namely, unit testing, integration testing and acceptance testing.

5.4.1 Unit Testing

5.4.2 Integration Testing

5.4.3 Acceptance Testing

6 Evaluation

6.1 Accuracy

6.2 Compression

7 Results

8 Challenges

As shown by the previous sections in this document, this project has been far from trivial, in fact it was much more difficult and challenging than initially anticipated despite already expecting a high level of difficulty. While the project has been overall a very mentally stimulating, rewarding and fulfilling project it is important to note some of the major challenges that were faced during the development. This serves the purpose to inform future readers and those interested in this project about some of the challenges faced and how to either overcome them early, or more likely, to expect and prepare for them. While some of these issues can be mitigated earlier with further planning and research, the most difficult and major challenges are those inherent to the vectorization process and conversion of raster graphics.

8.1 Toolchain

8.1.1 JavaScript

8.1.2 Concurrency

Arguably the most preventable and frustrating challenges faced were those that were a direct result of the toolchain used, especially when there exist other tools that do not have these problems. The most frustrating of these is the lack of maturity in the JavaScript ecosystem. This can be seen with the very minimal Node.js standard library which contains only the most necessary built in functionality and instead delegates anything else to libraries, frameworks and other open source contributors. While this does have benefits it also has major disadvantages especially coming from the vast ecosystems of the JVM and Android which both come with large fully featured but modular standard libraries that provide standard ways to perform common operations. This weakness is shown extensively whilst using any JavaScript collections such as Array, Set or Map, which provide only the most basic functionality leaving common operations like copying and adding an element to an index up to the user or to some external dependency. However, perhaps the best example for an argument of the JavaScript ecosystem's immaturity is the fact that JavaScript supports object oriented concepts, including classes and methods, but does not allow programmers to declare when two objects are equal or not. The Set collection contains only distinct elements by first checking if an element already exists before inserting it and ignores it if it already exists. JavaScript does not allow the programmer to determine how this equality is checked which is done using `===` by default, instead, if the programmer wants a Set of custom objects they have to implement this themselves or use an external library, further adding to dependency hell.

8.2 Performance

8.2.1 Hardware Utilization

8.2.2 GPU Programming

8.3 Minimizing Human Input

9 Reflection

10 Future Work

Despite being given many months to work on this project, it would be dishonest to claim that it is close to being completed, as there are still many future endeavours and tasks that can be completed given more time. These tasks were left out as they were either lower priority or out of the scope of the project's initial aims and objectives and were thus cut for time saving.

10.1 Optimization

As discussed in depth in the preceding sections, one of the main issues with the project currently is the performance of the processing. This was never a high priority to begin with as it was clear early on that in order to aim for high accuracy and high compression, more instructions needed to be executed and thus more time was needed to execute them. However, given more time this is an angle that is important to investigate and tackle as the more complex the process becomes, the longer it takes to view any results, leading to significantly lower productivity. One of the main ways this can be achieved is by using the machine's hardware more efficiently especially by using more of the resources and making use of the many processing cores most modern machines possess today. As discussed earlier,

10.2 Future Research

Towards the end of the development cycle many new ideas began to make themselves apparent especially after the multiple consecutive failures to show major positive results. These ideas were noted and left aside in order to focus on the main goal of the project as they strayed slightly from the scope of the project and would have required more than the given time in order to truly explore correctly. However, given more time, these ideas should be further explored and tested as they can show further positive results and shed some light and insight onto the vectorization problem. The most important idea is the potential mixing of raster and vector graphics in a new graphics format in order to leverage the best of both formats and attempt to solve the issues of both. A potential algorithm would identify easily vectorizable areas of an image that would alter the data of the original image only less than a given maximum data loss threshold. These areas of the image would be vectorized and represented in vector graphics. Then, for the areas where vectorization fails to yield any positive results, usually because of noise, rasterization artifacts such as anti-aliasing or other non-vectorizable qualities such as gradients, the algorithm would keep those areas as raster pixel-based. As is made obvious by the simplistic high-level explanation, this idea is still not well thought out or researched, but given more time this is something that can be explored and researched in depth.

11 Conclusion