Animation Vectorization and Compression

Bassam Helal

September 2020

Abstract

Table of Contents

1 Introduction **1**

2 Background **2**
- 2.1 Raster Graphics
- 2.2 Vector Graphics
- 2.3 Image Compression
- 2.4 Video Graphics
- 2.5 Toolchain

3 Related Work **4**
- 3.1 Current Research
- 3.2 Current Applications
- 3.3 Limitations

4 Implementation **5**
- 4.1 Pipeline
  - 4.1.1 Input
  - 4.1.2 Frame Extraction
  - 4.1.3 Vectorization
  - 4.1.4 Frame Merging
  - 4.1.5 Output
- 4.2 Vectorization Methods
  - 4.2.1 Color Quantization Approach
  - 4.2.2 Connected Component Labelling (CCL)
  - 4.2.3 Edge Based CCL Approach
  - 4.2.4 Pixel Based CCL Approach
- 4.3 Limitations and Drawbacks

5 Software Engineering **8**
- 5.1 Methodology
- 5.2 Schedule (Gantt Chart)
- 5.3 Risks
- 5.4 Testing

6 Evaluation **9**
- 6.1 Accuracy
- 6.2 Compression

7 Results and Findings **10**
- 7.1 Results
- 7.2 Findings

8 Challenges **11**

9 Reflection **12**

10 Future Work **13**

11 Conclusion **14**

1 Introduction

A paragraph or so for each of:

- Project Description
- Motivation
- Existing Literature
- Limitations of the literature that we will attempt to tackle
- Aims and Objectives
- Discoveries and results we found after our attempts
- Section Signposting

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, ie, the current common standard for graphics data representation. We will then introduce its counterpart vector graphics, a different way to store graphics data that is growing in popularity and has its own advantages and disadvantages. 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 this all ties into video graphics and describe some of the few differences between image graphics and video graphics.

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

Here we throw a lot of the current stuff from both research and industry and show their flaws and limitations (which we may or may not have tackled)

3.1 Current Research

- Brief summary of what we will go over (signposting)
- New Machine Learning methods for Vectorization
- New Machine Learning methods for Compression
- Novel approaches and ideas within Vectorization such as mesh gradients etc (no ML)
- Brief summary of everything we just mentioned

3.2 Current Applications

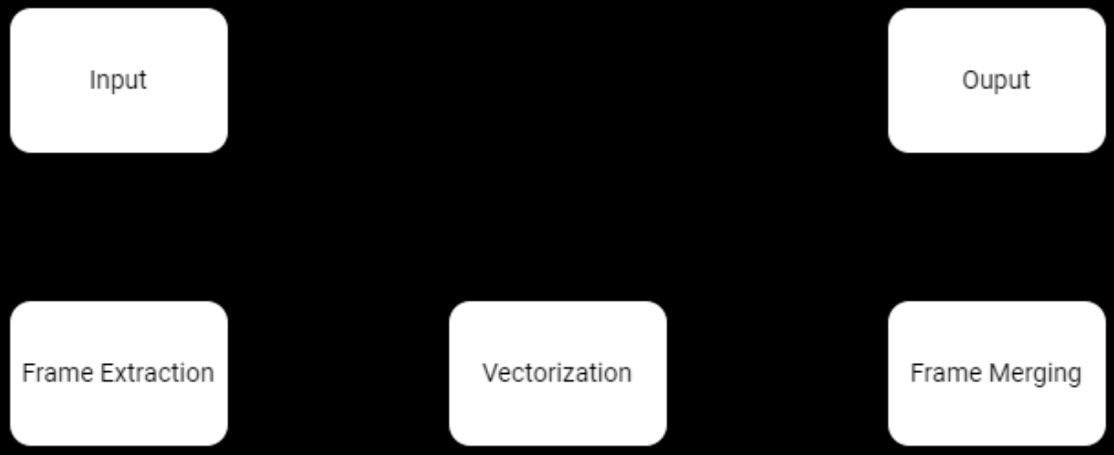
- Brief summary of what we will go over (signposting)
- Adobe Illustrator, Logos, Topography, Medical stuff like X rays etc Potrace
- Animations and Games using Vector as the development method
- Current Image & Video compression algorithms such as PNG, H265 and H266 (no ML)
- Brief summary of everything we just mentioned

3.3 Limitations

- Brief summary of what we will go over (signposting)
- Why ML is absolute hot trash
- Show off the flaws with existing technologies while showing how we tackled (and hopefully beat) them
- Discuss inherent limitations of this process, such as issues with Vector Graphics and input data being varying etc
- Brief summary of everything we just mentioned

4 Implementation

4.1 Pipeline



This pipeline image describes the the project's implementation from a high level view. Each phase of the pipeline is further detailed.

4.1.1 Input

An animated cartoon video, particularly one with vectorizable qualities, which we will further detail

- Low or no use of gradients so it has discrete colors
- Low or no noise
- No 3D stuff like shadows and reflections

Show examples of good cartoons and have images, so Fairly OddParents, Teen Titans Go as well as some less optimal cartoons like old Tom and Jerry, Hannah Barbera and Ed, Edd n Eddy

4.1.2 Frame Extraction

4.1.3 Vectorization

Conversion of each Frame to an SVG which will have a single SVG Path element for each color

4.1.4 Frame Merging

SVG Frame Joining which will have some smart compression built-in

4.1.5 Output

An SVG Video that is of course highly scalable and has a lower size than the original if upscaled and with very minimal data loss

4.2 Vectorization Methods

Here we will explain our story of the last few months, the many approaches and their failures. This section should implicitly show the reader that the problem is very hard, harder than initially thought to be. If we show the difficulty and the struggle it can explain any poor results and can further show overall contribution that we tried something difficult

4.2.1 Color Quantization Approach

From the general process of the library ImageTracer.js

- Explain Color Quantization
- Advantages (very accurate, fast, good for logos and web based stuff)
- Disadvantages (Too dependent on quantization number of colors, large size and poor speed with high number of colors)
- Show pictures of original vs some different results from this approach (to show why it fails)

4.2.2 Connected Component Labelling (CCL)

Using Connected Component Labelling (CCL) to create discrete but adjacent components

- Explain CCL
- Reasoning behind approach
- Possible disadvantages
- Signpost for the following approaches

4.2.3 Edge Based CCL Approach

Using OpenCV Canny Edge detection

- Explain OpenCV and Canny Edge Detection
- Reasoning behind approach (edges should form polygons which will be color regions) and the expected result

- Advantages (fast, easy, accurate for edge detection)
- Disadvantages (not all edges form polygons and 1 pixel incorrect can have huge effects)

4.2.4 Pixel Based CCL Approach

Using the color of each pixel to form regions with close enough or similar colors

- Explain pixel data (RGBA) and what it means to be close to a neighboring pixel
- Reasoning behind approach (an image can be a partition of several color regions)
- Advantages (surprisingly accurate, little data loss, huge size savings)
- Disadvantages (slow, complex, Anti-Aliasing and noise affects result so too many 1 pixel regions)

4.3 Limitations and Drawbacks

Here we explain the overall limitations of all the processes we used and really any future process because of the nature of Vectorization of Raster data

- Brief summary of what we will go over (signposting)
- Speed or performance (never realtime and requires HPC on GPUs for good speeds) and optimizations required
- Lossy-ness, the transformation of Raster to Vector is by nature lossy because we are undoing Rasterization artifacts
- Size, this can be an issue for low resolution images or for very complex images, rasterization is often better
- Brief summary of everything we just mentioned

5 Software Engineering

Just the SE stuff and how we applied it during development, briefly mention the changes Covid did to the development

5.1 Methodology

Extreme Lean Agile because life is unknown beyond 2 days in these strange times we live in :/
Here we can show how the poor results affected our schedule and how things shifted because of unexpected results, we can definitely show that this is common in real life and is a good representation and preparation of the real world which has unexpected delays and poor results all the time and how we overcame it all (less sleep and more coffee :D)

5.2 Schedule (Gantt Chart)

Followed it well but started falling apart towards the end, very accurate to the real world :D
Mention that we started dissertation somewhat early to avoid the risk of running out of time

5.3 Risks

Time Time Time!!!

5.4 Testing

Short and quick, show some basic testing stuff
Unit Testing for accuracy and of course Manual Acceptance testing (to ensure images actually open, are not corrupt etc)

6 Evaluation

Two main angles, accuracy to the original and compression rate from the original As much as possible we need to emphasize that we have done something great, no lying, just selective and sometimes exaggerated benchmarks in order to further our claim that we have accomplished something meaningful

6.1 Accuracy

- Define accuracy (aka lossy-ness or how much data is lost from original)
- Quantitative methods (compare each pixel to itself), and compare with other compression methods like JPEG
- Qualitative methods (human eye side by side) and compare with other methods like JPEG

6.2 Compression

- Define compression rate (aka how smaller is the result compared to original or some other)
- Quantitative methods (compare against original, AND against others including upscaled and uncompressed)
- Qualitative methods (file transfer speeds of smaller sizes and how smaller files are better for users)

7 Results and Findings

Here we show both the results of our development but also our own personal findings that someone who wants to research this kind of thing should be aware of that I discovered

7.1 Results

The good, the bad and the ugly (with detailed explanations :D)

7.2 Findings

- This is a very difficult problem, more difficult than initially thought to be
- GPU acceleration is very hard but improves speed shockingly well
- Node sucks for concurrency and doesn't have true multi-threading
- Differing input will have different results because of the limitations of vectorization

8 Challenges

- Toolchain used was not very mature or helpful sometimes
- GPU programming is insanely difficult but yields huge benefits (leading to very difficult decisions)
- Optimizations to increase speed are also difficult and especially so on a non-multi-threaded environment like Node
- Reducing the number of human inputted arguments to create a fully autonomous deterministic system that is not ML based and produces very accurate results is very difficult

9 Reflection

- How would you do it differently if to start from scratch (Use JVM or Native and optimize for GPU early)
- Project Management (Fairly good given the circumstances)
- So much new knowledge about many different things
- Very difficult but very mentally stimulating and rewarding project that I really enjoyed

10 Future Work

- Use better tools that allow for native multi-threading and easier and direct access to GPU
- Better optimizations and better compression
- Explore having a mixed raster and vector solution to solve the areas where vector fails
- Buy better hardware for faster development :D

11 Conclusion

- Summarize everything we said
- Conclude with final thoughts about the good, the bad and how this can further improve the world etc