Animation Vectorization and Compression

Bassam Helal

September 2020

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, 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

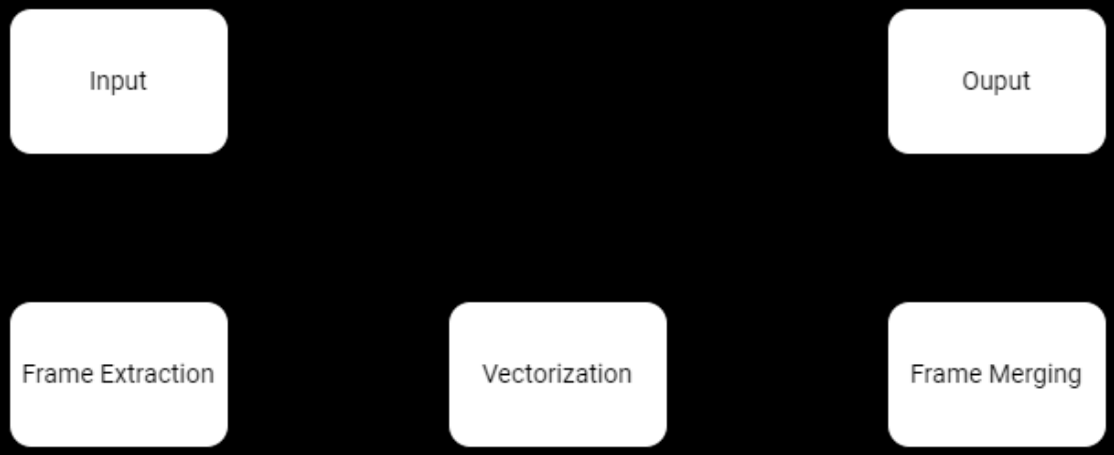
3.1 Current Research

3.2 Current Applications

3.3 Limitations

4 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, 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. 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 lower file sizes. Finally, we explain the output that the process will produce and what an ideal output will be like, particularly in comparison to the input file.

4.1.1 Input

First, it is important to understand the input that the process will be expecting and detail what makes an ideal input file. The minimum requirement is that the input file be a video in any popular format, specifically, one that Ffmpeg can use (so basically all formats), however, this project specifically 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 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 if 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 signal interference 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 is like this, but a surprising number of cartoon videos can come very close to this description, with very few complications such as shadows and light gradients. 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 and thus can be described as being a truly 2D cartoon. On the other extreme end cartoons with heavy use of photographs and 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*, *Looney Tunes* cartoons as well as 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 a video of an animated cartoon and the ideal input is a video of an animated cartoon that has highly vectorizable qualities.

4.1.2 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.

4.1.3 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

5.1 Methodology

5.2 Schedule (Gantt Chart)

5.3 Risks

5.4 Testing

6 Evaluation

6.1 Accuracy

6.2 Compression

7 Results and Findings

7.1 Results

7.2 Findings

8 Challenges

8.1 Toolchain

8.2 Performance

8.3 Minimizing Human Interaction

9 Reflection

10 Future Work

11 Conclusion