Hello, my name is JT Klenke, I’m an undergraduate at Cornell University studying computer science interested in a summer internship in an entrepreneurial environment. I have experience coding in Java, Python, OCaml and C/C++. I’m a fast learner and a hard worker eager to take on new challenges. By summer my completed coursework will include Object Oriented Programming and Data Structures, Functional Programming, and Algorithms. Here is a link to my personal portfolio website: jtklenke.github.io

About Me

I’m an undergraduate studying computer science in the College of Engineering at Cornell University. I enjoy math and physics and spend my free time flying drones, taking pictures, and throwing pottery. I’m part of the computer science team of Cornell’s Aerial Robotics club which competes in the annual International Aerial Robotics Competition. Additionally, I work with Misha Padidar as a research assistant working on optimizing the coil shape for stellarator fusion reactors.

My Projects

Evolutionary algorithm assisted antenna design

I wrote an evolutionary algorithm in Python to optimize the antenna shape for communications antennas, the final design is to be used on F-5 Navy training jets. The plate antenna operates from 108-400MHz, the evolutionary algorithm designed antenna is about twice as good as the old design and exceeds the specification by 25%. Here is the paper I wrote on the process: Evolutionary Algorithm Designed Broadband Plate Antenna for F-5 Vertical Stabilizer.

Timelapse photo creation

Inspired by this (https://www.ted.com/talks/stephen\_wilkes\_the\_passing\_of\_time\_caught\_in\_a\_single\_photo?language=en) Ted Talk by Stephen Wilkes, I wanted to create a version of these photos procedurally. Stephen Wilkes creates them manually, painstakingly picking out the most interesting subjects and times of day, manually blending it together to create his images. I wanted to simplify the process, using frames from a timelapse video and procedurally blending over time to create a similar effect.

The first method that I wanted to try was to take columns of pixels from different frames of the timelapse. This would be easy to implement and give me a rough idea of how a more advanced method would look. I would specify a width that the final image would be as well as the number of columns of pixels I would take from each frame. From that, the final image could be constructed. The result is bands of pixels from separate frames arranged next to each other.

It's surprisingly tricky to find 24h time-lapses of interesting things with still cameras on YouTube (most are of the night skies, which stays relatively similar throughout the video, have moving cameras, have watermarks, or are of boring subjects.) I found two that work well enough for experimentation but suffer from one of the above issues: [30 Days Timelapse at Sea | 4K | Through Thunderstorms, Torrential Rain & Busy Traffic](https://www.youtube.com/watch?v=AHrCI9eSJGQ&) and [24 Hour Time Lapse](https://www.youtube.com/watch?v=x1Os8NcSrbI). The first has a watermark and text plus I found out everything happens too fast for this method to work effectively, the second is just a boring subject and the total number of frames is a bit too short to do everything I wanted (925 frames) but was great for testing because there's a good span in time but not much changes frame to frame. The last video I found was [Timelapse Los Angeles / Santa Monica Beach California](https://www.youtube.com/watch?v=Ew7N65F6oYc). This was a good video with a still camera and interesting subject over a long enough time that there was good variation but unfortunately, the video is super short. There are only 616 frames and only 470 of those are useful (the rest are black or darkened from a fade in and fade out).

To get the frames of the I downloaded the YouTube videos using one of many online tools, they're kind of sketchy if your careful to only download the video they seem to be fine, then I used VLC Media Player to get frames. Later I would streamline this process using the Python library youtube-dl to download YouTube videos and FFmpeg to get all the frames as .jpg images.

I then used Image from PIL to get pixel data from each frame. My final code simply creates a black image of the appropriate width and height and then filles it in with pixels in some nested loops, the final image is then showed. The result of the first attempt is a little underwhelming but shows some promise and some flaws to be remedied. Here it is on the 30 Days at Sea and the 24 Hour videos.

(image)

This has one pixel (column) per frame and a width of 1200 pixels.

(image)

This one is 925 pixels wide and 2 pixels per frame.

There are a few obvious issues to this method, first, the width has to be manually capped by the number of pixels per frame and the length of the video second, it just doesn't look that great, if more than 1 or 2 pixels are used in each band, they become extremely obvious. Below is an example with 4 pixels per frame and you can clearly see the bands start to appear.

(image)

While I think this is a good proof of concept that can make interesting images, it's not a great method.

For my second iteration I would generalize the process by taking pixels from different frames based on a more arbitrary function. For example, the distance of a pixel from the center would decide what frame that pixel would be taken from. This solved the issue of manually setting the width and number of pixels per column, the final image would have the same dimensions as a frame from the video and the pixel distribution would be decided by the function. I ended up having a scaling coefficient to the function determined by the maximum value of the function and the size of the “useful frames” of the video. There’s still some manual parameter tuning. but I suppose this whole project is more art than science anyway. Doing this for every pixel should create a picture that would smoothly blend over the length of the video.

After implementing this more general form, the time to build an image skyrocketed from just a few seconds to an hour at which point I ended the program. What was taking so long? It turns out that while my first implementation only opened each image once (400-1,000 images), the new one opened a new image for every single pixel, about 2 million. This was obviously a complete waste of time. Fixing it required splitting the process into two steps, first figure out what pixels would use what frames, second opening each image once and copying the pixels from that frame over. This sped it up immensely and the code was back to running under a minute. Eventually I decided it would be easiest to get all the pixel data from every frame, store that in a numpy array and then simply grab the data when I needed it. I folded this array building process into the separate code that included the video download and frame extraction. After running this code with the center distance function on the Santa Monica Beach video I got this image.

(image)

Much more interesting. However, after finishing that implementation, I noticed that the resulting pictures had noticeably rough transitions between pixels.

(image)

To fix this, I decided to use a weighted average of a pixel’s value over some time, this should soften the transition essentially by “blurring” the photo over the time axis. I was hoping this would mimic the effect of long exposure photography. This new method took significantly more computer power, while the original version sampled just 1 value per pixel the new version would sample 64 pixels and then perform a weighted average, which took additional time.

To speed up the new version I looked to parallelize the computation using the multiprocessing library. I further optimized the code by precomputing the Gaussian distribution (which is what I used as the weighted average values), pre-retrieving the image pixel values and storing them in a np array, and doing all three color channels at once using a matrix multiplication rather than doing the dot-product on a per channel basis, after that I was able to get the time to build an image down to around 10 minutes on my laptop, still slower than I’d like but not excruciatingly slow.

(image)

There are still some noticeable artifacts that I suspect come from YouTube's video compression being compounded by averaging over many frames, especially around the ferris wheel. Additionally, with this new technique some of the color detail and contrast is lost in the ferris wheel which is something I might revisit. I'd like to take some of my own timelapse video and use this tool and see what results I can get.

A\* pathfinding

As part of my work on the motion planning team of Cornell’s UAV club I implemented and benchmarked a few different implementations of A\* and related algorithms in Java and C.

The objective was to find the shortest path between two points on an NxM grid where some grid points are marked as obstacles. I first implemented Dijkstra’s shortest path algorithm which would be extremely slow for the given use case but was simple to implement and helped me understand the core algorithm behind A\*. Unsurprisingly, Dijkstra’s algorithm was slow, especially on large grids.

With a better understanding of the algorithm, I moved on to implementing A\*. Deciding that now was as good a time as any, I also created some test cases using JUnit after all if it’s not tested, its broken, and wouldn’t you know, it was broken. After some significant debugging I realized my mistake, as an optimization I had only created an instance of the point class when I “discovered” a point, what I had done wrong was to instantiate a new point even if it had already been discovered. Honestly, with such a major oversight I was shocked how few test cases it failed. To fix this I used Java’s HashMap to store all the discovered points and only created one if it didn’t exist.

After testing with JUnit and benchmarking A\* I was surprised at how slow it was. Looking for ways to speed it up I found an algorithm that takes advantage of the regularity of the grid called jump point search (JPS) which is said to be significantly faster. The original idea for JPS comes from this (<http://users.cecs.anu.edu.au/~dharabor/data/papers/harabor-grastien-aaai11.pdf>) paper by Daniel Harabor and Alban Grastien but I found this (<https://zerowidth.com/2013/a-visual-explanation-of-jump-point-search.html>) website by Nathan Witmer to give a great visual intuition behind the algorithm. Unfortunately, for almost every obstacle placement I found JPS to be much slower than traditional A\* and for very large grids it would run into the recursion limit.

Still looking for a faster implementation I tried to optimize by original A\* implementation. I did so first by writing my own priority queue implementation that supported an ‘increase priority’ method and second by condensing the representation of a grid point. Originally, I had a grid point class that stored the g-cost, f-cost, parent point and x and y coordinates of the point. While this was simple and readable, it came with all the overhead of a Java class, which for many points added up. I was able to reduce all the information down so that it was able to fit into a single long (a 64-bit number) then I built a class to abstract that structure away from the core A\* algorithm. This helped quite a bit but was still slower than I would have liked.

I decided to code the algorithm in C. It would be a good opportunity to refresh some basic C and should yield a faster implementation. I decided that I would start by implementing a linked list which I used to store the final path. I also started by implementing the priority queue as a linked list, just iterating over the whole list, and then returning and splicing out the “minimum” element. I knew this would be very slow (O(n) as opposed to O(lg n) from a binary heap) but it would let me debug the A\* algorithm, give me some extra practice in C and give me an idea as to the speed of C compared to Java. Additionally, instead of using a hash map to store the discovered points I just used an array with all of the potential points, while this would be very memory inefficient it would be very fast and easily implemented.

After getting that implementation working, I noticed that it was actually significantly slower than Java, originally, I just attributed that to the slow priority queue but even after I modified the Java priority queue to loop over every element it was still slower. I looked for ways to speed up the C code and realized that there were a few obvious things that were slowing it down. First, I cut down some unnecessary malloc calls I had, second, I used additional space to store some values that I had previously been recalculating, most importantly, I used O flags, O2 sped up the code by almost 4x.

Finally, with some guarantee that it would be faster than Java, I implemented the priority queue as a binary heap. In the end, while it took more significantly more time to code than Java it ended up being substantially faster. I was able to get the code to run anywhere from 4-100x faster than the best Java implementation.

Tribology internship