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.

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).

In order to get the frames of the video you can download 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.

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. Doing this for every pixel should create a picture that would smoothly blend over the length of the video.

After finishing that implementation, I noticed that the resulting pictures had noticeably rough transitions between pixels,

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.

I used PIL to access the images and build a numpy array that was easier and faster to operate on.

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\*. After implementing, 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. However, 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. In the end, while it took more significantly more time to code than Java it ended up being substantially faster. After using O flags, cutting down on how many malloc calls I had and using additional space to store some values that I had previously been recalculating, I was able to get the code to run about 4x faster than the best Java implementation.

Tribology internship