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.

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.

I would build the image by taking pixels from different frames of a video depending on some 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 the Python library youtube-dl to download YouTube videos. Originally, I used VLC Media player to get all the frames of the video as images but later I used FFmpeg to make the whole process more streamlined. I used PIL to access the images and build a numpy array that was easier and faster to operate on.

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 by writing my own priority queue implementation that supported a change priority function and by condensing the representation of a grid point. Originally, I have 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 adds up. I was able to reduce all of 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. Finally, I decided to code the algorithm in C. It would be a good opportunity to learn 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 extra values I was able to get the code to run about 4x faster than the best Java implementation.

Tribology internship