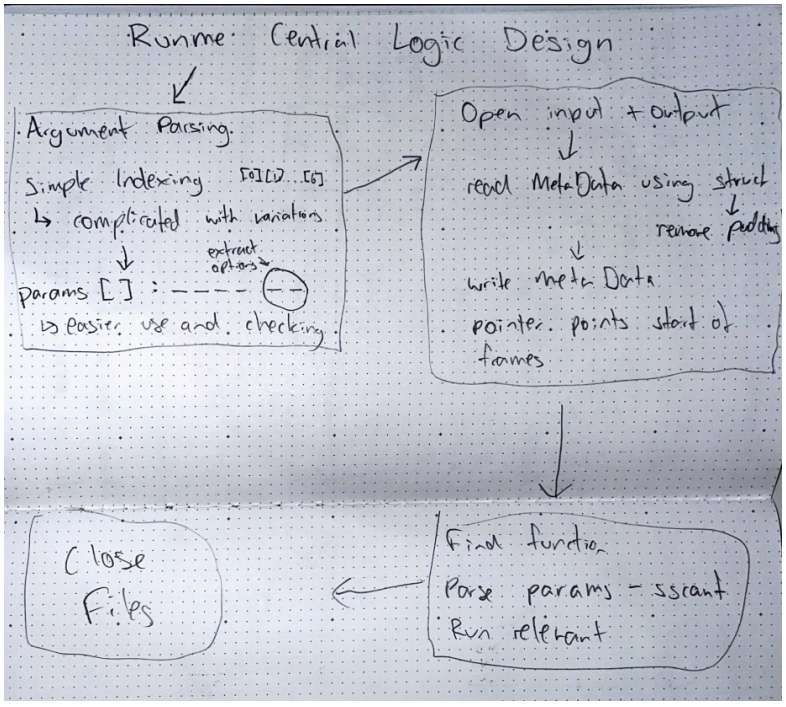
Video Filtering Coursework Report

Rose Laird
16/01/25

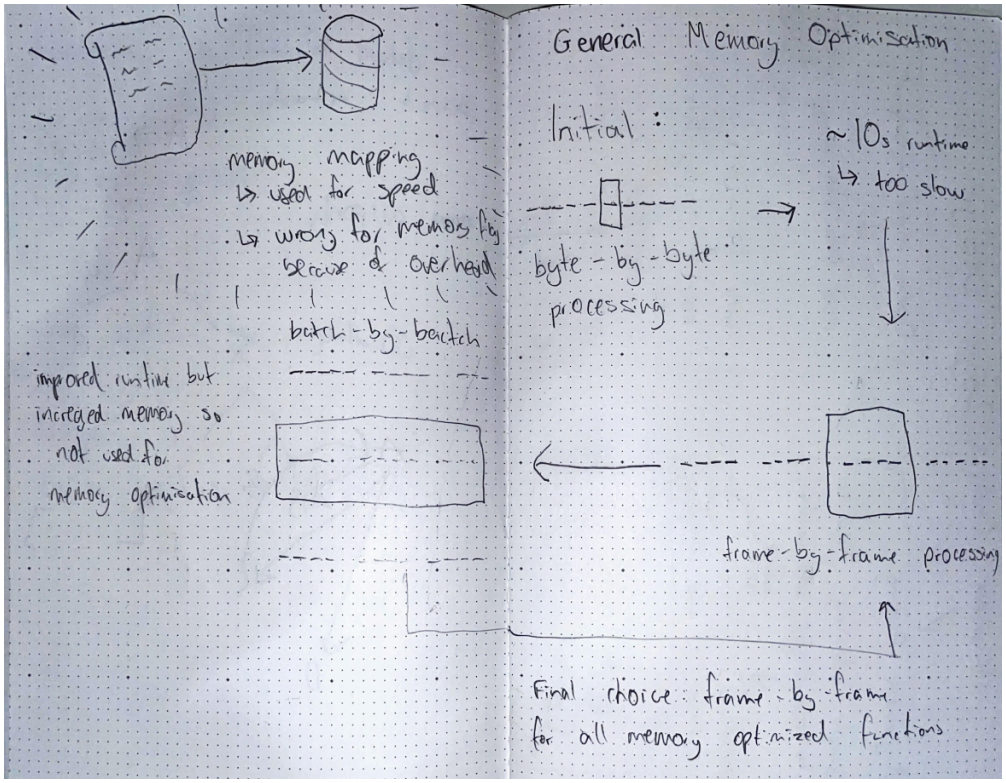
Overarching Design



My overarching design has remained similar to my first approach. I wanted to centralise as much of the functionality as possible to the *runme.c* to reduce redundant code; so the files are opened centrally, the meta data is extracted, parsed and written to the output file, so that all the file pointers are passed in pointing to the beginning of the frame section.
I chose to create a struct for the meta data for readability and maintainability, although I did run into an error where it automatically padded the data to 16 bytes, which I disabled.

Memory Optimisation

Throughout my design it has proved that reading frame by frame guarantees less than a KB of program memory, regardless of the number of frames, however if frames had significantly higher resolution it may break the 1kb boundary. In the graphs below my memory optimised functions are featured. It is interesting they are generally quite fast, likely because of the minimal overhead, important for the small file I tested.

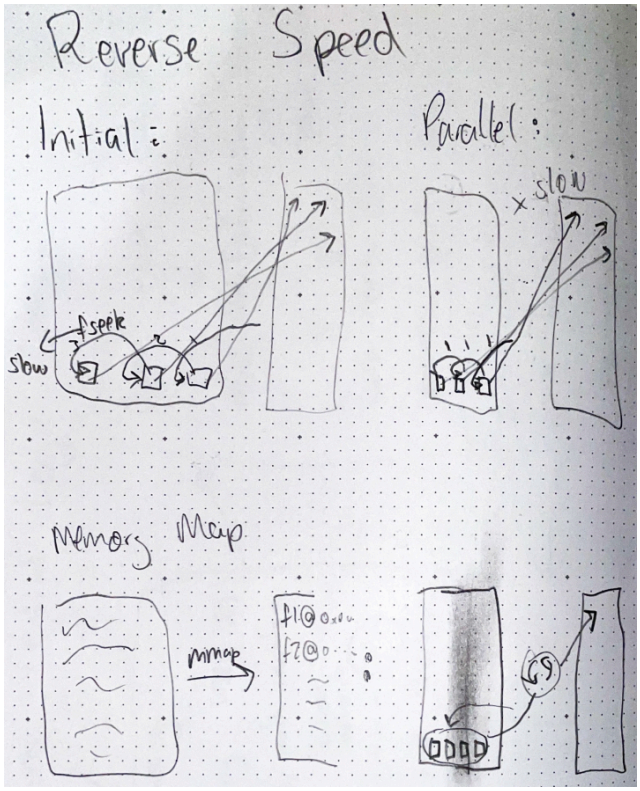


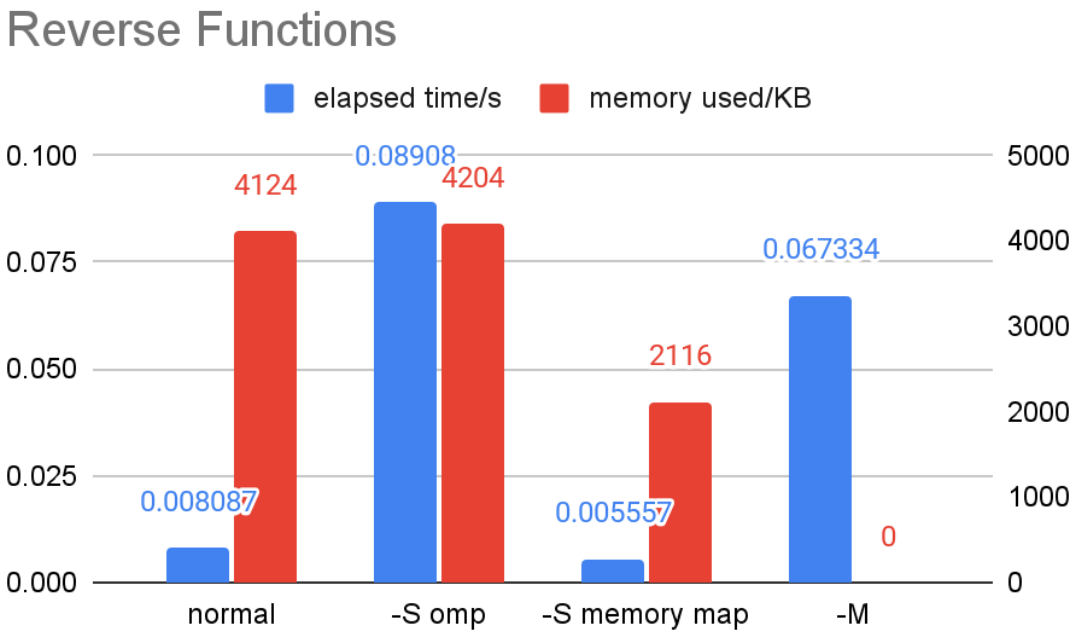
Speed Optimisation

reverse

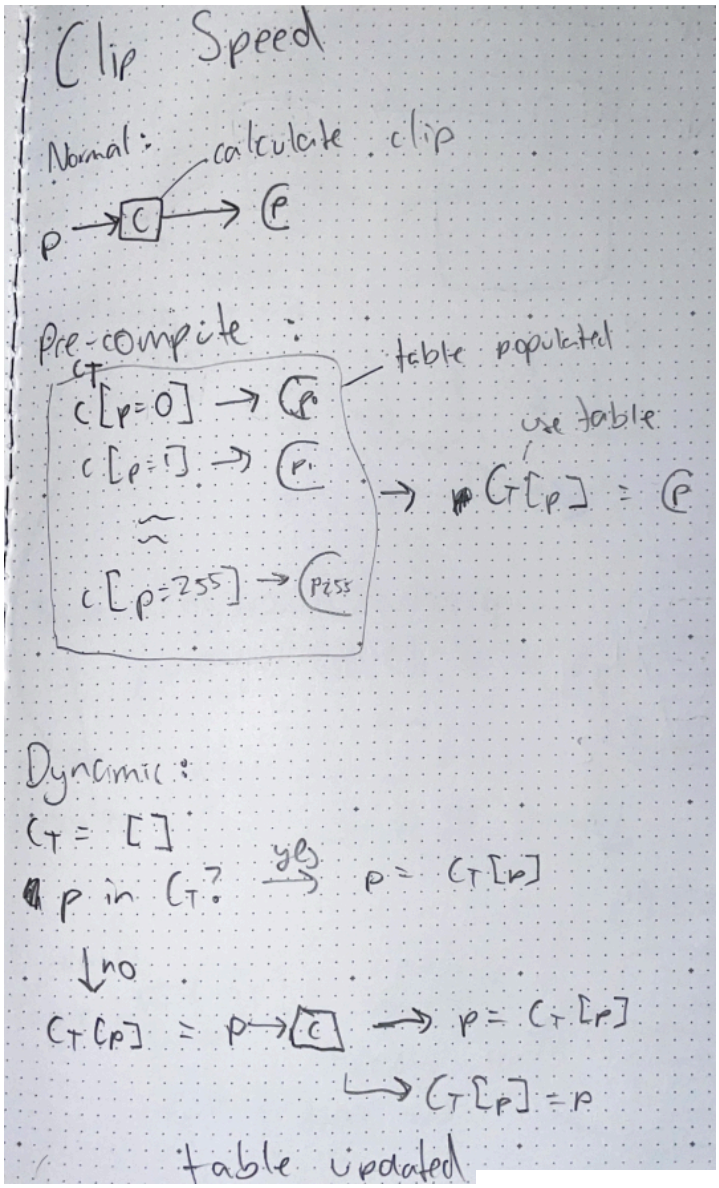
My initial design for reverse is to read the whole file into memory, and use pointers and fseek to read frames from the back of the file and then write them to the output file. I then used omp parallelisation but I found that it actually made the program run slower. This could be because of the overheads involved. I then looked into memory mapping as I read it reduces runtime for accessing memory non-sequentially - as done when reversing. I faced challenges doing so as it did not run on my Windows mounted WSL ubuntu remote window but I persevered and made it work by moving the folders from local to virtual. I also discovered it was more efficient to only memory-map the input file, as the output file was being written to sequentially. The result was a

superior runtime to normal and also a greatly reduced use of memory, which is particularly useful for larger files.

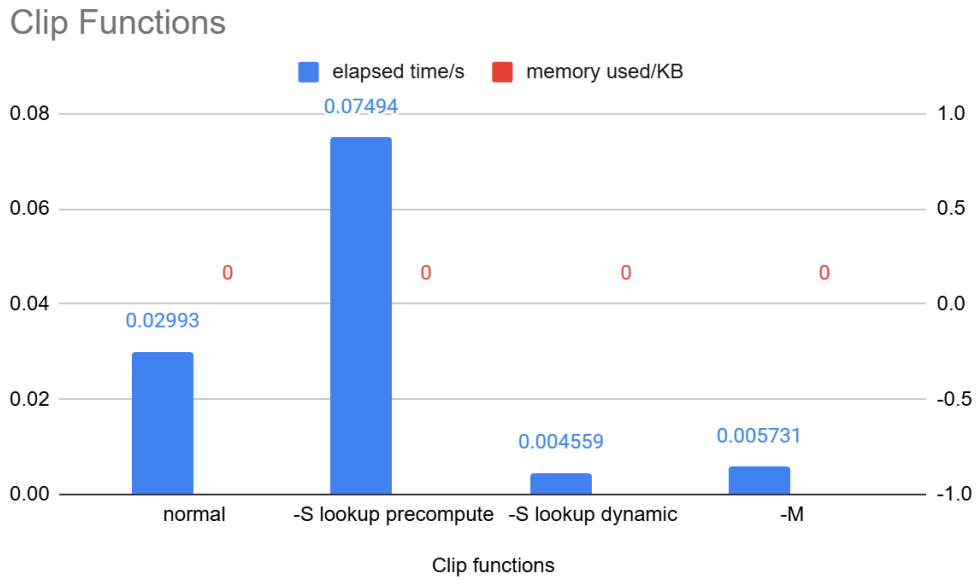




clip_channel

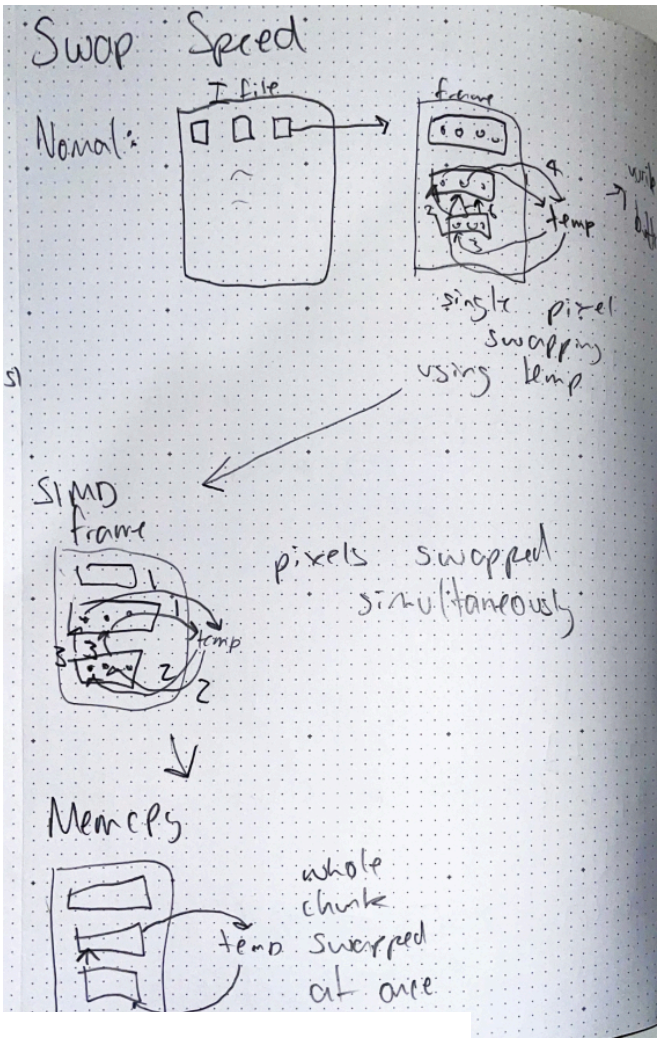


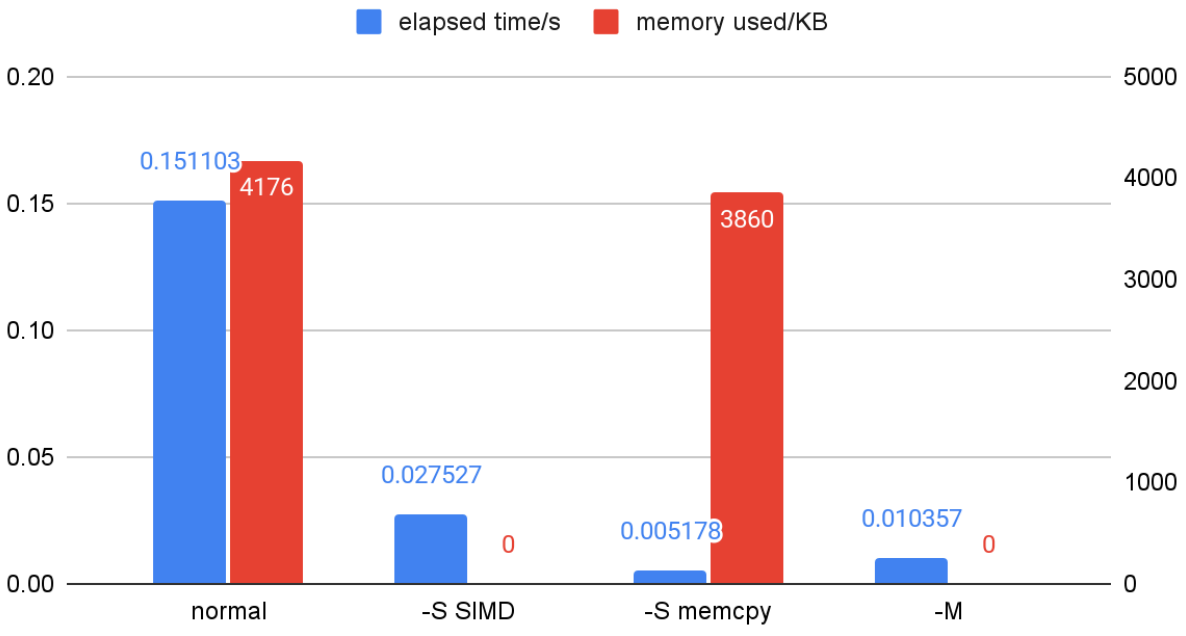
For *clip_channel*, my initial approach involved reading the file frame by frame, using a buffer to index channel sections, and clipping each pixel individually. However, I later optimized this by precomputing the clipping values once and storing them in a hash map, which eliminated redundant computations. To further enhance efficiency, I initialized the hash map as empty and dynamically added new pixel values during processing. This significantly improved runtime for *test.bin*, though it's worth noting this approach might be less effective for files with a broader range of color values.



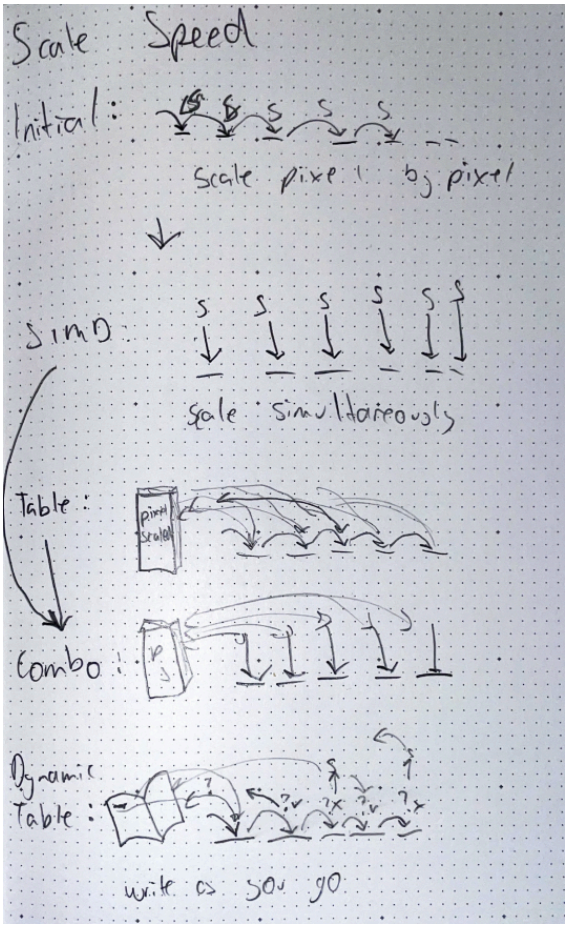
swap_channel

My initial implementation used a basic loop to iterate over each pixel, swapping channel values directly. While functional, it lacked efficiency. I next explored SIMD instructions to process multiple pixels simultaneously. This approach significantly improved performance by reducing the number of iterations. However, alignment issues and setup overhead limited its effectiveness for smaller frames. Finally, I tested *memcpy* for swapping channel sections. By copying and swapping blocks of memory at once, this method proved to be the fastest for larger test files, leveraging optimized memory operations. However, its benefit diminishes for smaller resolutions or fewer channels.

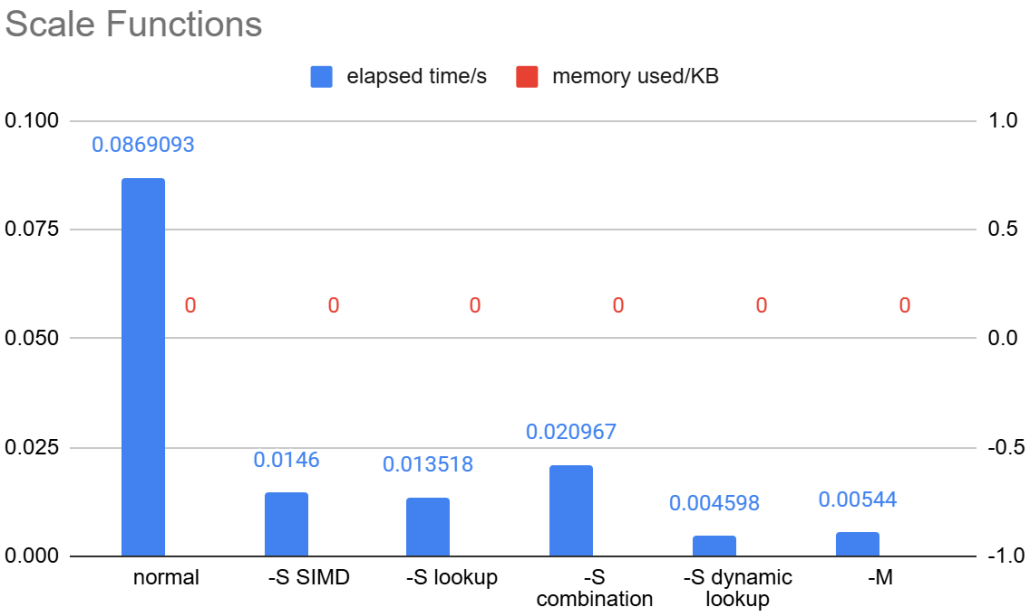




scale_channel



My first implementation for scale_channel used a basic loop, scaling each pixel one at a time. While simple and functional, it was slower due to the repeated multiplication for every pixel. Moving to SIMD improved efficiency by processing multiple pixels in parallel, taking advantage of modern processor capabilities. Introducing a lookup table for precomputed scaled values further reduced runtime by eliminating repeated calculations, though it required additional memory. Combining SIMD with the lookup table proved less effective due to the overhead of managing both. Ultimately, an on-the-fly lookup approach balanced speed and simplicity, avoiding memory overhead while remaining efficient.

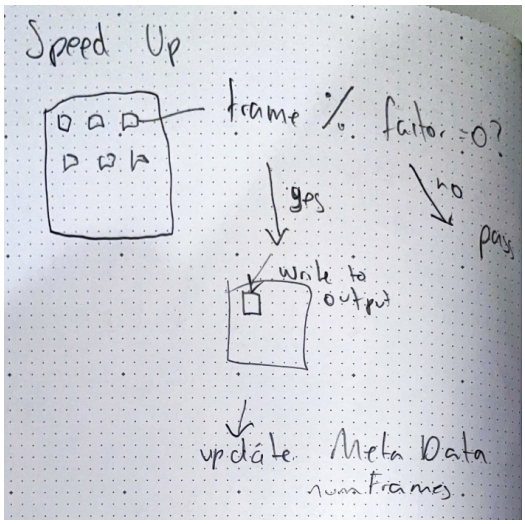


Extra Credit

speed_up

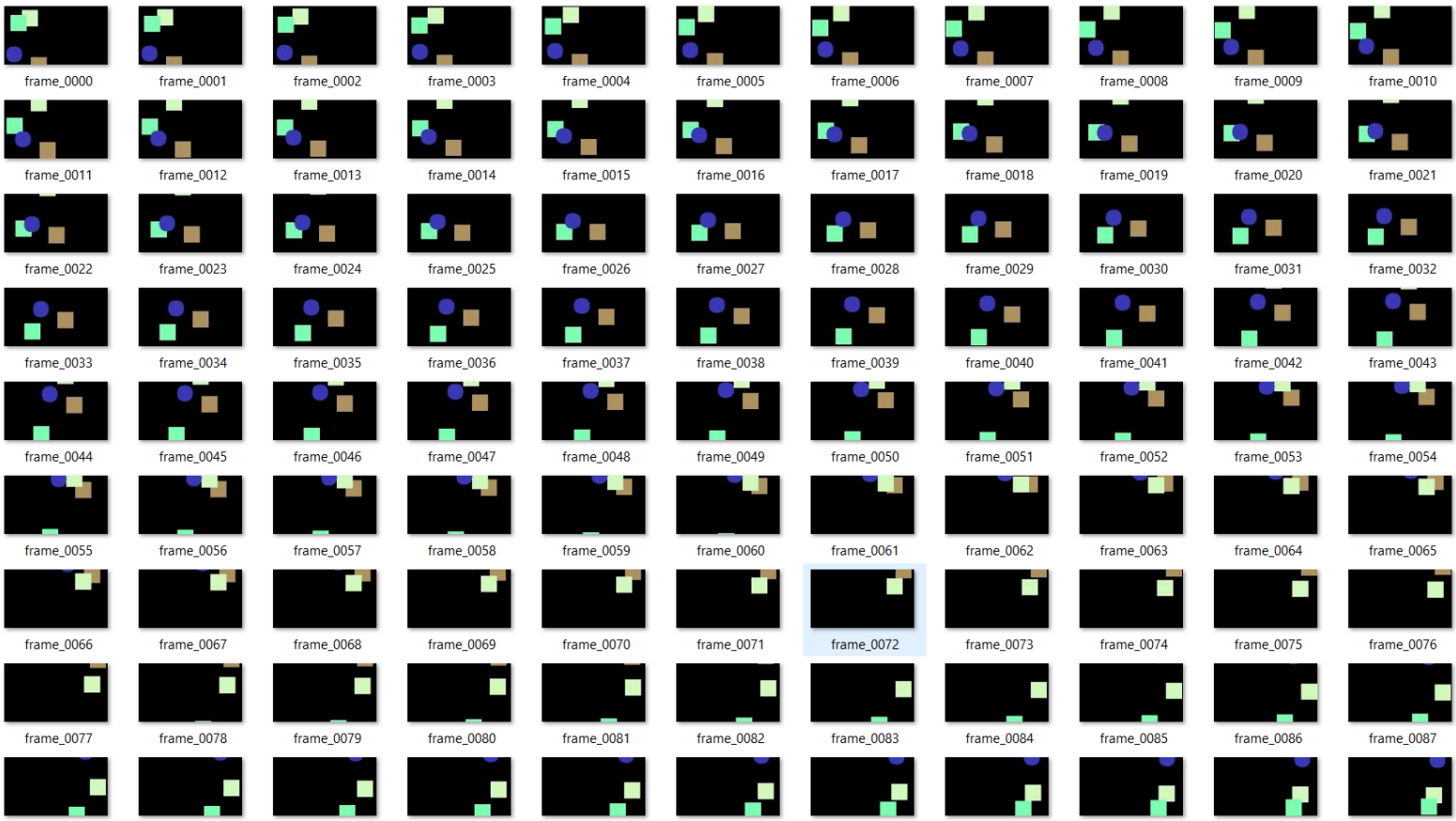
The fist function I wanted to design was a speed up/stop motion function that simply made the video quicker by removing a proportion of frames.

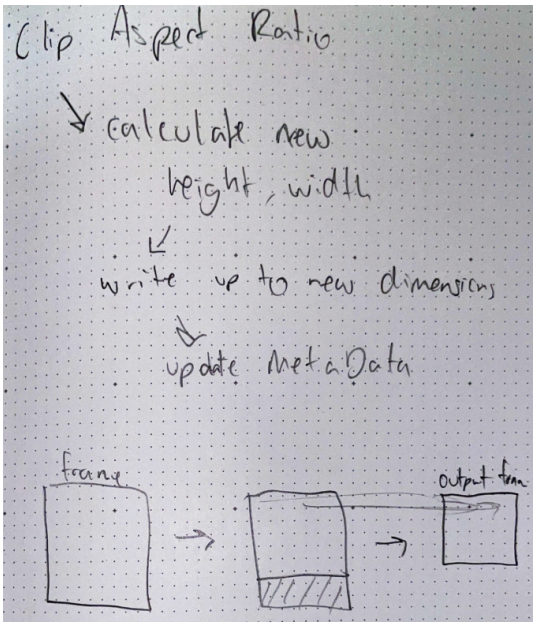
Usage: ./runme data/test.bin rose_s.bin fast_forward 4
Result: 0.002336 seconds 0 KB



crop_aspect

One function I thought might be useful to film-makers is crop to aspect ratio. I chose to crop instead of add black bars because it would look less unsightly for non black backgrounds, but the consequence is some image is lost.
Usage: ./runme data/test.bin rose_s.bin crop_aspect 16:9
Result: 0.007712 seconds 0 KB


frame_0000
frame_0001
frame_0002
frame_0003
frame_0004
frame_0005
frame_0006
frame_0007
frame_0008
frame_0009
frame_0010
frame_0011
frame_0012
frame_0013
frame_0014
frame_0015
frame_0016
frame_0017
frame_0018
frame_0019
frame_0020
frame_0021
frame_0022
frame_0023
frame_0024
frame_0025
frame_0026
frame_0027
frame_0028
frame_0029
frame_0030
frame_0031
frame_0032
frame_0033
frame_0034
frame_0035
frame_0036
frame_0037
frame_0038
frame_0039
frame_0040
frame_0041
frame_0042
frame_0043
frame_0044
frame_0045
frame_0046
frame_0047
frame_0048
frame_0049
frame_0050
frame_0051
frame_0052
frame_0053
frame_0054
frame_0055
frame_0056
frame_0057
frame_0058
frame_0059
frame_0060
frame_0061
frame_0062
frame_0063
frame_0064
frame_0065
frame_0066
frame_0067
frame_0068
frame_0069
frame_0070
frame_0071
frame_0072
frame_0073
frame_0074
frame_0075
frame_0076
frame_0077
frame_0078
frame_0079
frame_0080
frame_0081
frame_0082
frame_0083
frame_0084
frame_0085
frame_0086
frame_0087


Clip Aspect Ratio
calculate new
height, width
write up to new dimensions
update MetaData
frame
output frame