AME 394

Computation Photography

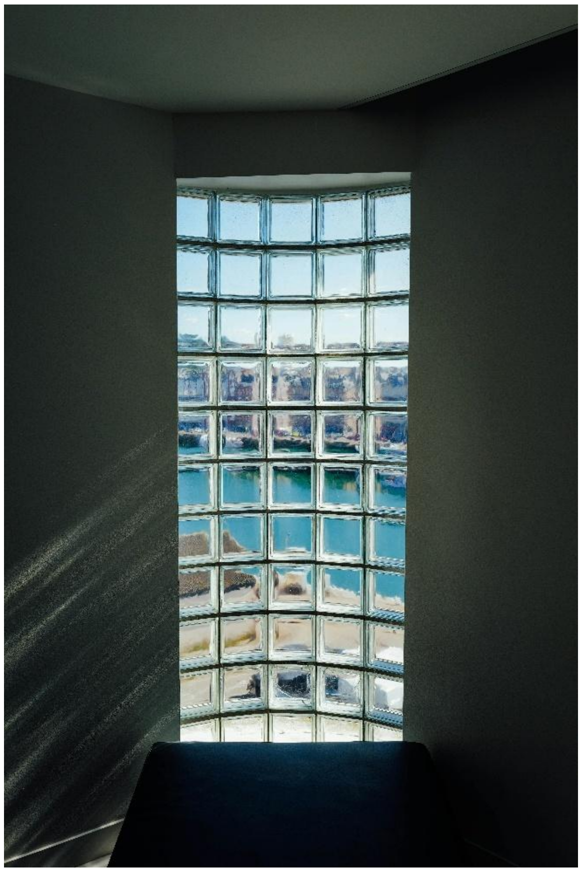
Final Project – Python Custom Cubism Filter

Ethan Rudy

Prof. Tinapple

Introduction

I loved the uniformity that some of the examples of cubism brought to the image, the perfectly aligned corners. It reminded me of a *Glass Block* window.



Scenic View Through Glass Blocks at Seaside, Source

I loved the idea of creating something to do it for me, not out of laziness, but because I am at heart a no life programmer. I love a challenge!

Process Documentation

The process wasn’t *too* awful, as I opted for a simpler language Python. Given that it’s an interpreted language, the filter takes around 50 milliseconds to run. This could be done closer to real time (~12ms, estimated from game development framerate goals) if I was to preload the image and use a faster, compiled language like C++. I would also have to rely on third party libraries like stb_image instead of using Pillow, Python’s image manipulation library.

Once I decided on my tools, I had to figure out the design of my algorithm. My program flow is as such: Load Image, apply cubism filter, Output image. Super simple! I built it this way because I would like to add even more filters! Like pixel sorters, all kinds of fun stuff.

Okay, the algorithm. First things first, we’ll need to recognize that unless we want to lose pixel data to overlap, our cubes should be cubes, not rectangles. So, we need a square cube dimension that is a factor of both our width and height, to evenly fit inside our image. In my case my image is (2400, 3000) pixels, so I chose a (200, 200) cube. But of course, when I wrote the code, I added cube width and height as modifiable parameters for other images. Bad code if it only works once. Given our cube dimensions and our imaged dimensions, we can find the total number of cubes, in my case 180

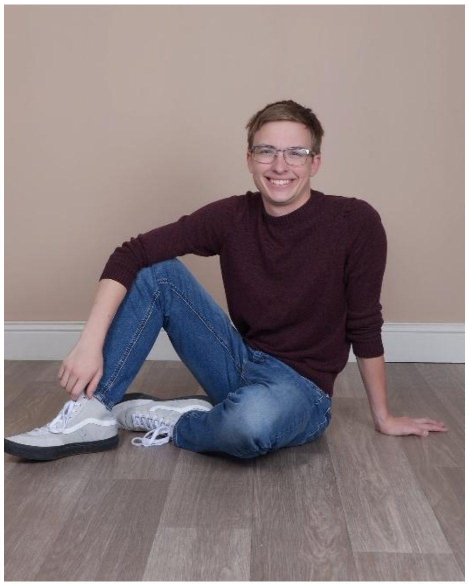
180 rotations is all that separates us from our desired output. But what *type* of rotation matters! Customization is key. I added another parameter to the filter, rotation direction. Direction is as follows: CW, CCW, 180, and Random. You may be wondering “What does random mean?”, Every block is randomly rotated one of the other three options!

Now that we know where we’re rotating and in what direction, it’s as easy as iterating over the pixels (with the step value being the cube dimension, 0, 200, 400) and creating a region of pixels in memory to rotate. Ex: (0, 0) to (200, 200) Luckily, I was lucky enough to read the Pillow documentation and found a transpose function! Rotating

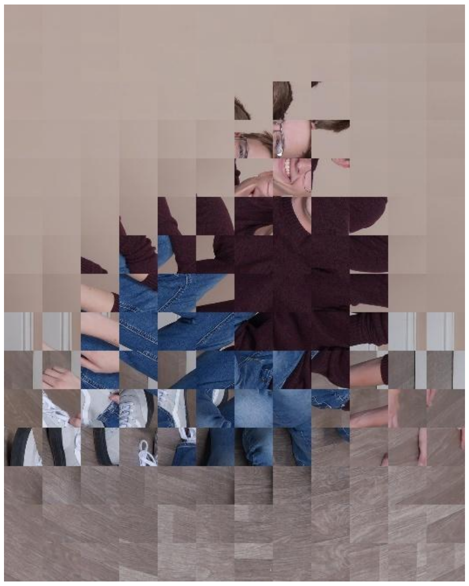
2D arrays is NOT my specialty so having a function that does the heavy lifting was a big relief. Final step! We have a rotated region, so we just paste the rotated pixels onto the original!

Final Outcome(s)

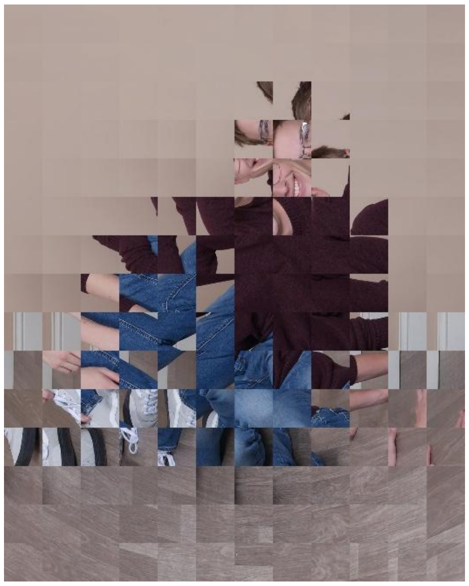
Original Photo



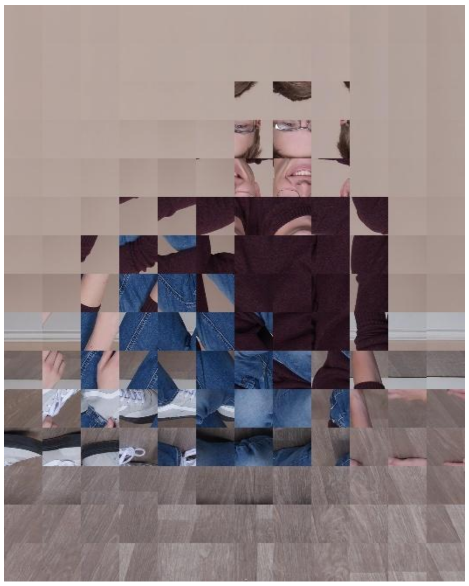
Clockwise Rotation 200x200



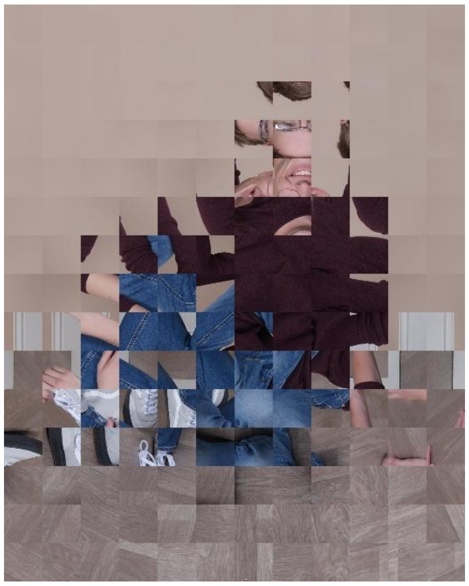
Counterclockwise Rotation 200x200



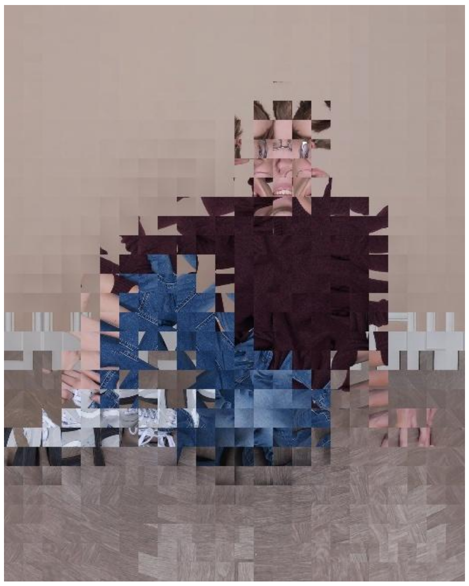
180 Degree Rotation 200x200



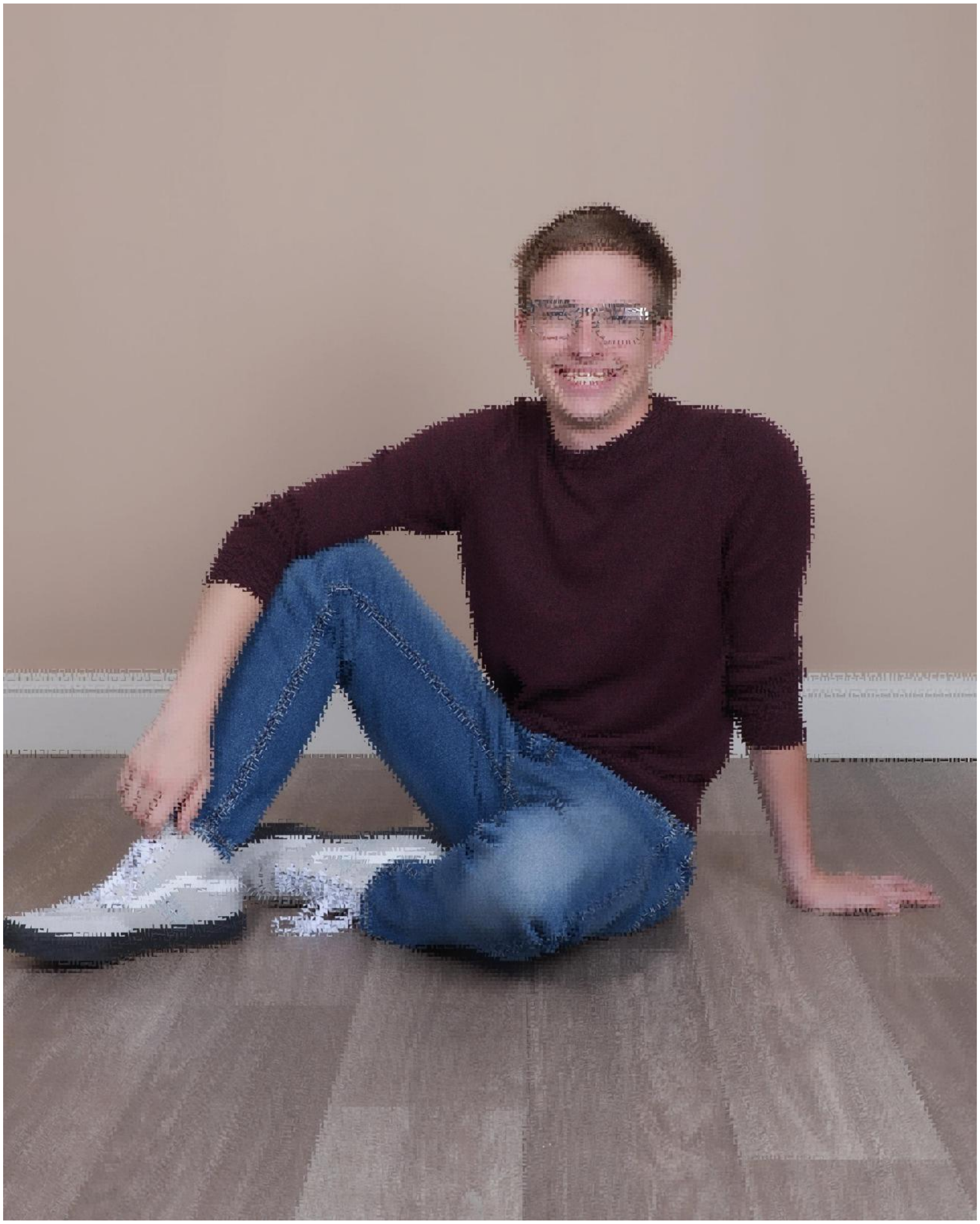
Random Rotation 200x200



Clockwise Rotation 100x100



Clockwise Rotation 10x10



72,000 Cubes, 450ms, Produces an interesting blur

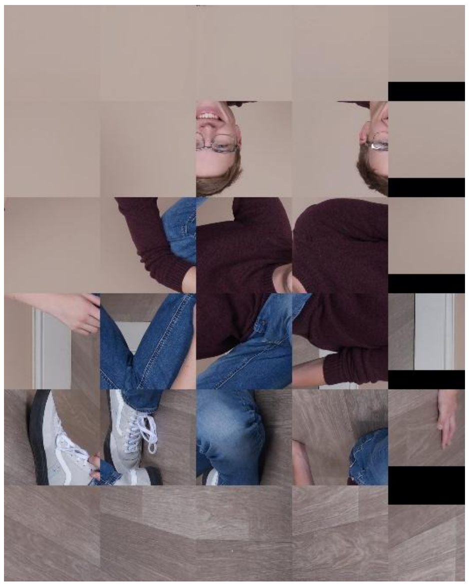
Express the Making

The creation of the filter, found here, wasn't *too* difficult to make. As a junior in CS, if I can't figure it out, I have bigger problems! I loved the challenge throughout the entire process. Writing the filter took about 1.5 hours after a couple hours of ideating and a nice shower full of algorithmic thinking.

Major breakthroughs included the transpose function as well as the crop function, so I didn't have to manage triple channel 2D arrays of pixels manually, instead leaving them in a pixel array structure.

My only struggle with the filter was my own incompetence. I accidentally offset the regions by multiplying by cube dimensions, neglecting to think that I was already scaling by pixels! So I was rotating like 740,000 pixels, which just does nothing if you were wondering. It took a bout 40 minutes of combing the documentation, looking to see if it was deprecated, only to figure out that it was completely my fault.

Random Rotation 500x500



This is some artifacting caused by grid dimensions that are not common factors of width and height (2400 % 500 != 0), so we introduce nonexistent pixels!