Musically Composed Visuals-Draft

Fourier2 Team

Spring 2023

1 Team

Justin Choi, Ryan Gaffney, Ian Lips, Matt Dim

1. Our project is going along very well. We have the basics of audio processing that takes a wav file and performs the fourier transform. We also have managed to create an output of a visual based on our transformed audio. While we still need to work on more utilization of parallelization and connecting our programs coherently, I would give us a score of **4** on our progress.
2. I would say we require **Normal** feedback, as we have our plans and steps set for our implementation, yet if there are any glaring issues with the setup of our project it would be nice to know before committing further.
3. Our group has no issues in finding times to work and have meetings for our project. **Great**
4. At the moment I don't believe we have any questions

2 Overview

Our starting point for this project had two main topics, audio processing with fourier transform, and displaying an image from a julia set given an input. For the audio, our start consisted of a lot of research on how people process audio into a format to where we can use it as an input for our image generation. While we already knew we would perform fourier transform we still needed to understand what our initial audio would start as, and how the processing would result in a data type that can be transformed with fourier. After looking through several different ways of audio processing, we decided to go with the format of using wav files with FFT (fast fourier transform) processing. We will be able to parallelize this using OpenMP to perform the transformation across many threads. In order to deliver the information to our image generator, we decided to use a CSV format in order to feed our processed audio into the visualizer.

3 Fast Fourier Transform (FFT)

A lot of our time was used for researching and discovering libraries that we could utilize for our project. We knew that we would use OpenGL for our visualization, but for audio processing there are several different ways we could have gone about it. Thus, most of our time was not spent on implementation, but was used to understand how libraries worked and which libraries would be good candidates for our needs. We found multiple libraries in c and c++ that we have decided to utilize in our file conversion, signal processing, and graphics which will be included in our references.

The fourier transform output essentially keeps track of all the frequencies like a histogram of a song, and creates a magnitude to see how many times that frequency appears in a given song. We decided to use the wav file format for our implementation due to how they are easier to parse through for processing. MP3 files are compressed which results in smaller files as data is lost from compression. Going with this implementation was the most optimal way to create a format to easily input into our visualizer. The most difficult part of this initial draft was finding ways of integrating the two parts of our program. Building the visualizer, we had a difficult time understanding input, alongside a way of creating outputs of our audio processing.

Looking back at our Draft Deliverable, we have met a lot of the requirements set for ourselves. We have successfully gotten a song in wav format to be broken down and fed to an audio visualizer. However, there have been significant issues with trying to parallelize our fast-fourier transform. We've been trying to use OpenMP to try and get a

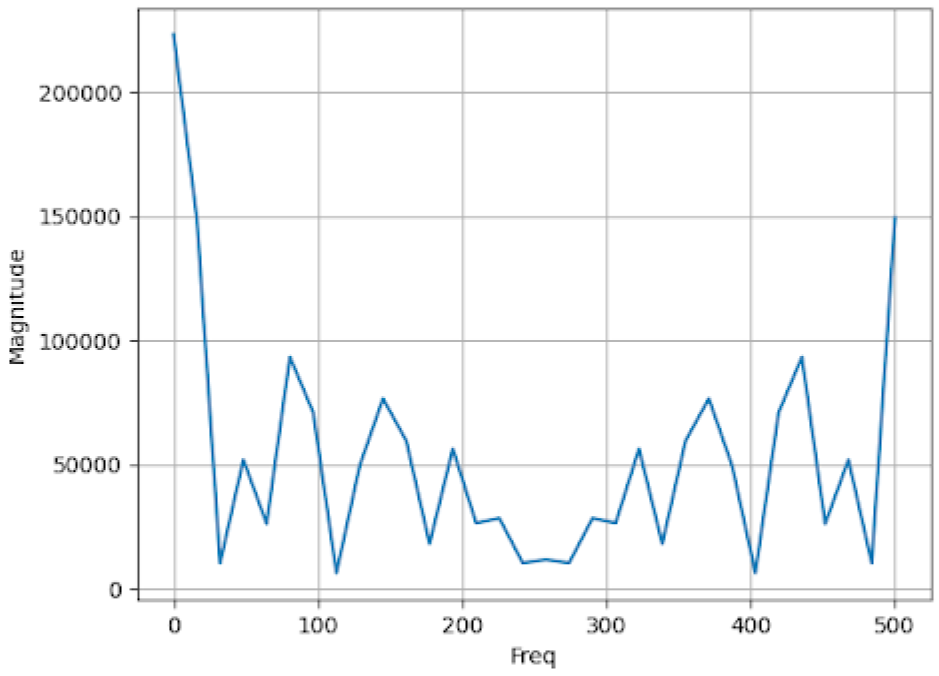


Figure 1: Output from the FFT Algorithm

noticeable speedup in computation. For now we have only noticed the program actually slowing down in real practice, so that part of the deliverable I think was a little too optimistic to have done at this time in the project. The code we've ended up with does output an image from song data, but I believe we would like to make a modification in getting the parallelization done with OpenMP and Cuda done in the next part of the project instead of done by the draft period.

Work that needs to be done before the due date of the project starts with getting computation parallelization working. We will still attempt to get our fast-fourier transform library to speedup with OpenMP. If this does not result in any speedup, we may have to pivot into using another library. After doing so we want to get the cuda implementation done with the OpenGL visualizer, so that it can be run on the gpu and meet the project specification. Once parallelization is working across the whole project, we may want to normalize our data to better work with the visualizer and make the animations more seamless.

4 Visualization

So far what I have done is create our serial fractal visualizer in C utilizing OpenGL's framework. Currently our images are being generated via a modified julia set algorithm to incorporate mobility with our "camera". In other words you can zoom in/out and move the camera around. In addition there are also keybinds mainly for debugging, and finding values that work well for image generation. They do however also allow for exploration of the fractal that is being generated. So in a sense our visualization tool is multifunctional. I have also modified our animation function to utilize values from the fast fourier transform. These values are currently being used as a modifier for the values responsible for the generation of our fractal/visual. I tried having those values be the actual input to our fractal generation algorithm, and also tried scaling them. This However illustrated jarring results and will require further tuning. I plan on solving this by normalizing our values and messing with how the values are scaled to get cleaner results. So for now the extracted values from the FFT algorithm are being used as modifiers for our algorithm. A demonstration of the visuals can be seen at `https://www.youtube.com/watch?v=gugts8mUX-U`. In addition some more fractals generated from my algorithm can be seen below. I have also spent some time thinking about how to visualize the colors in our algorithm and started messing with that. The colors correspond to the current iteration of our julia set which gives some really pretty, and some really trippy results. This is important because I hope to maintain that functionality once cuda is being utilized.

Most of the obstacles that were encountered pertained to OpenGl. I did have to learn a little bit about how it worked but that wasn't too much of an issue as I have had some prior experience with graphics. Another issue as I hinted to above is how the values outputted from the FFT algorithm are interpreted. It was initially so bad I ended up giving myself a migraine. So I will only show our algorithm exactly correlating to the output of the FFT when I have got that working and it's not painful to look at. I also had some issues running the visual side of the program on my desktop at home which is running windows as opposed to my laptop running linux. This will require some more research but for now i'll write it up as I configured something wrong on my Desktop.

In terms of the original proposal I feel like this is almost exactly what I was thinking. Some modifications are going to have to be made to how values are interpreted from the FFT output but that shouldn't be too much of an issue.

In order to wrap up this portion of our project, it mostly comes down to implementing cuda, and tuning variables to get aesthetically pleasing and not jarring images. So In terms of finishing by the deliverable due date I think we'll be ok on the visual side of things as I do feel a little ahead of schedule as of writing this.

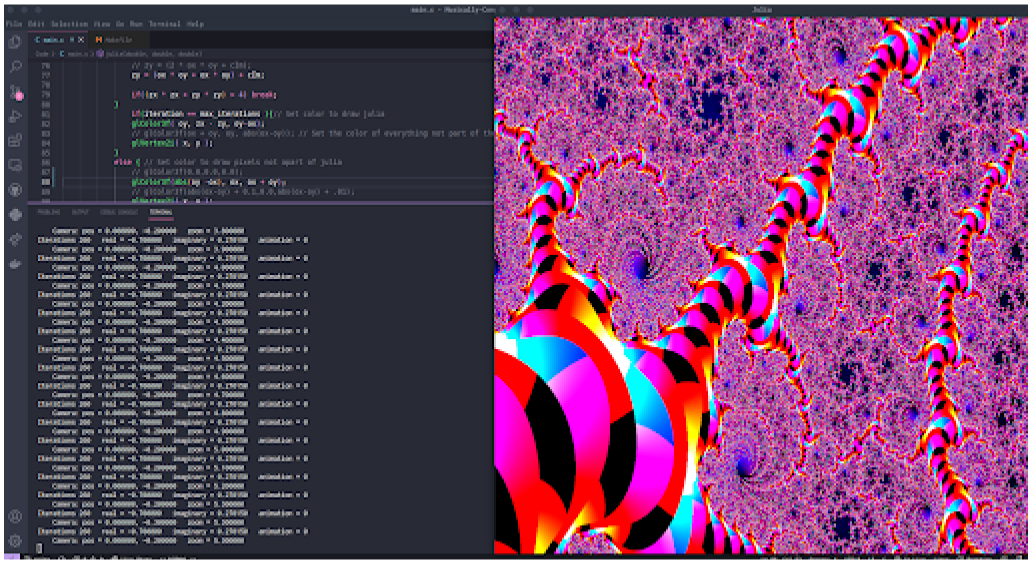


Figure 2: Julia Set image where the non julia set parts correspond to values in an iteration. This could have some cool results in the final version.

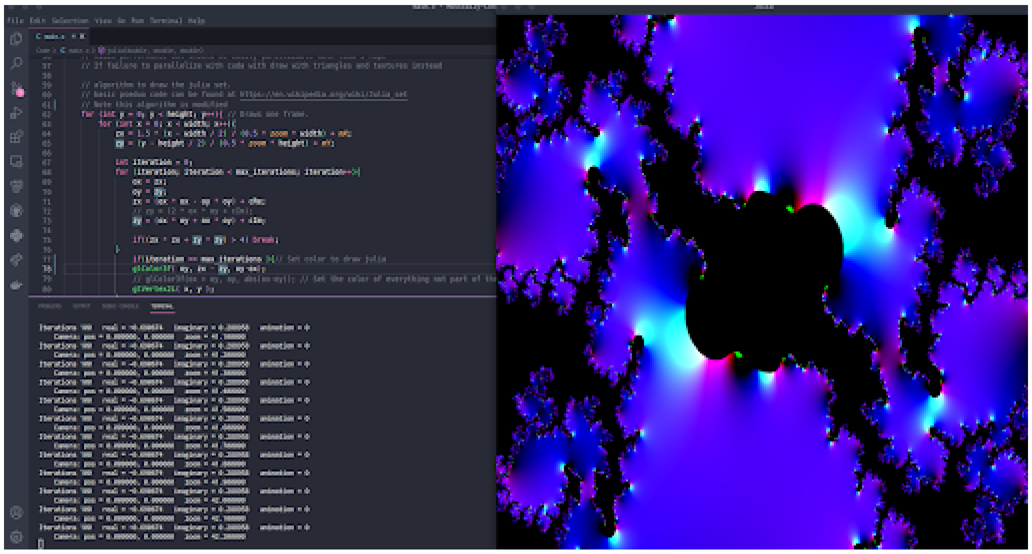
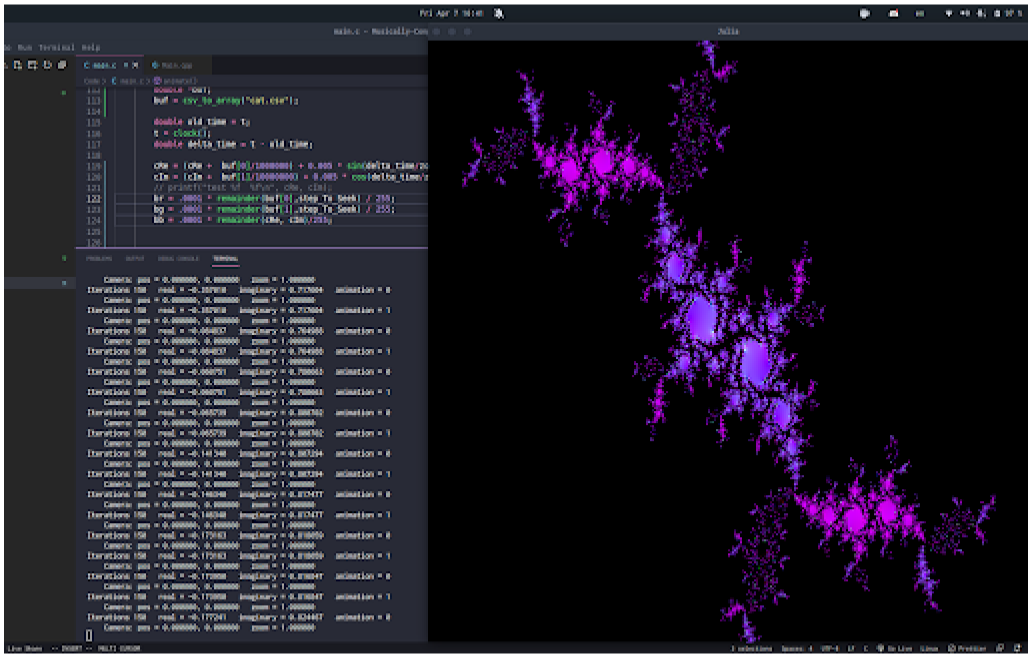


Figure 3: Another example of the julia set showcasing the functionality of the camera. The camera gives us more ways to generate an image. Might also be an interesting test case when it comes to using cuda.



```
    double *buf;
    buf = csv_to_array("cat.csv");

    double old_time = t;
    t = clock();
    double delta_time = t - old_time;

    cRe = (cRe +  buf[0]/10000000) + 0.005 * sin(delta_time/
    cIm = (cIm +  buf[1]/10000000) + 0.005 * cos(delta_time/
    // printf("test %f  %f\n", cRe, cIm);
    br = .0001 * remainder(buf[0],step_To_Seek) / 255;
    bg = .0001 * remainder(buf[1],step_To_Seek) / 255;
    bb = .0001 * remainder(cRe, cIm)/255);
```

```
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.107818   imaginary = 0.712604   animation = 0
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.107818   imaginary = 0.712604   animation = 1
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.064837   imaginary = 0.704986   animation = 0
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.064837   imaginary = 0.704986   animation = 1
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.060791   imaginary = 0.700063   animation = 0
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.060791   imaginary = 0.700063   animation = 1
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.065739   imaginary = 0.800702   animation = 0
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.065739   imaginary = 0.800702   animation = 1
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.141340   imaginary = 0.807294   animation = 0
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.141340   imaginary = 0.807294   animation = 1
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.140240   imaginary = 0.817477   animation = 0
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.140240   imaginary = 0.817477   animation = 1
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.173163   imaginary = 0.818059   animation = 0
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.173163   imaginary = 0.818059   animation = 1
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.173050   imaginary = 0.836047   animation = 0
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.173050   imaginary = 0.836047   animation = 1
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
Iterations 150   real = -0.177241   imaginary = 0.824467   animation = 0
        Camera pos = 0.000000, 0.000000   zoom = 1.000000
```

Figure 4: Another example of the image generation attempting to showcase the animation, and the creative images we can get from the output of our FFT algorithm.

References

[1] `https://en.wikipedia.org/wiki/Julia_set`

[2] `https://github.com/Himanshu4746/Morphy`

[3] `https://github.com/KyleSmith19091/Fourier-from-WAV-file/blob/master/Main.cpporphy`