Syracuse University

SURFACE

Syracuse University Honors Program Capstone Projects

Syracuse University Honors Program Capstone Projects

Spring 5-2017

Gravity 2

Yi Jin
Syracuse University

Follow this and additional works at: https://surface.syr.edu/honors_capstone

Part of the Physics Commons

Recommended Citation

Jin, Yi, "Gravity 2" (2017). *Syracuse University Honors Program Capstone Projects*. 1048.
https://surface.syr.edu/honors_capstone/1048

This Honors Capstone Project is brought to you for free and open access by the Syracuse University Honors Program Capstone Projects at SURFACE. It has been accepted for inclusion in Syracuse University Honors Program Capstone Projects by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

Gravity 2

A Capstone Project Submitted in Partial Fulfillment of the
Requirements of the Renée Crown University Honors Program at
Syracuse University

Yi Jin

Candidate for Bachelor of Science Degree
and Renée Crown University Honors
Spring 2017

Honors Capstone Project in Your Major

Capstone Project Advisor: ______________________
Walter Freeman, Professor

Capstone Project Reader: ______________________
Marjory Baruch, Professor

Honors Director: ______________________
Chris Johnson, Interim Director

© (Steven Jin May 2nd, 2017)

Abstract

The project that I chose to tackle for my capstone was coding my own physics based music visualizer. Doing so first required a strong foundational understanding of the OpenGL and GLUT visualization libraries. To build the visualizer, I took five main steps. First, I needed to open the music file and read the header. Second, I had to read in the file data window by window. Afterwards, I needed to apply a fast Fourier transform to each window. Lastly, I had to visualize the data and play the music. My end result was a Macintosh based music visualizer that allows you to play and interact with WAV music files.

Executive Summary

Gravity 2 is a first of its kind interactive music visualizer. Like a standard music visualizer, Gravity 2 uses a row of sixteen bars(with the option to add more) that oscillate up and down to represent music frequency as well as amplitude. The unique thing that separates it from other music visualizers is that each bar is made of a countless number of free moving balls. Forces are applied to these balls to have them bounce to the sound of music. As a user, you can interact with these balls by clicking to apply a force.

Gravity 2's main functionality is the ability to play and visualize WAV music files. By putting a music.wav file in the same directory as the program, the program will read, visualize and play the file. By clicking on anywhere in the window, you create an attractive force to all of the dots on the screen. Gravity 2 boasts a myriad of different options and features. By pressing on different keys on the keyboard you can toggle on and off different settings such as colors, force, and gravity.

Gravity 2 works by first reading in data from the beginning of the WAV music file. From this data, the program learns important information about the music such as the music quality and file size. The file then is read piece by piece. Each piece of data is subsequently analyzed, filtered and converted into usable information that can be displayed. Gravity 2 then takes this transformed information and represents it on the screen as a force.

Table of Contents

Abstract……………………………………………………………………… 3
Executive Summary…………………………..………………….………….. 4
Table of Contents…………………………………………………………. 5

Chapter 1:Humble Beginnings……..……………….……………………………… 6
Intro to OpenGL and GLUT….………..……………….….…………… 6
Gravity 1………………………………………….……………………. 7

Chapter 2: Gravity 2………………………………………………………..… 8
Wav Files……………………………………………………………….. 9
Windowing……………………….……………….…………..………….. 10
The Fast Fourier Transform………………………………………..11
Visualization and Music Playing……….……….…………………….. 12
Results………………………….……………….…….………………….. 12
Features…………………….……………………………………………….. 13

Chapter 3: Future Pursuits…..………………………………………………… 17

Works Cited.……………………………………………………………… 18

Chapter 1
Humble Beginnings

Introduction to OpenGL and GLUT Library

The first step in creating a C++ based music visualizer is a solid understanding of OpenGL and GLUT. OpenGL and GLUT are graphics libraries used to turn code into images. They put the "visualizer" in "music visualizer." The following is an overview of the integral concepts in OpenGL/GLUT programming and the main functions that I used to build my program.

Essentials:
The glutInit Quintuple: Comprised of glutInit, glutInitDisplayMode, glutInitWindowSize, glutInitWindowPosition, glutCreateWindow. These functions are a must for initializing a window inside main.
glutDisplayFunc/glutPostDisplay: These functions are called whenever the window is to be redrawn.
glutIdleFunc: This calls the inputted function continuously to perform background processing tasks/continuous animation.

Animation Related:
glBegin: Depending on the input mode, it is a "pen down" function used to draw various shapes on the window.
glClearColor: Clears the color of the window and sets to the inputted color
glColor3f: Specifies the color of the drawing.
glEnd: The pen up function.
glVertex2f/glVertex2f: Specifies a vertex ,whether in 2d/3d, while drawing.

Miscellaneous:
glutKeyboardFunc: Calls the inputted function when keyboard is pressed.
glutMouseFunc: Calls the inputted function when mouse is clicked.

How the Animation Works:
In order to simulate animation, the program repeatedly draws to the window to make it appear as if the objects on the window are moving. An important concept behind this is the idea of a time step. In my OpenGL code, the time step is the simulated fundamental change in time (dt) that dictates how much time passes per frame update. In order to simulate motion, I tell the circle how much to move per unit of time (dx/dt), which is the same as assigning the circle a velocity. By this same process, we can tell the circle how much velocity to change per time step to simulate acceleration(dv/dt). By being able to simulate acceleration, one can easily simulate force as Newton's second law tells us that force is derived from mass times acceleration.

Gravity 1

"Gravity 1," the predecessor to "Gravity 2", was my first substantial project after 2 months of learning OpenGL and GLUT. The idea for the project was to create a sandbox type application in which the user can interact with dots on the screen with a gravitational force via a mouse click.

In order to create the project, I first had to start out with only a single body(a dot on the screen). I used the glutMouseFunc method to access the mouse functionalities. From there, I was able to get the coordinates of the mouse, which I used along with the body coordinates to calculate a force separation vector. Calculating this vector allowed me to tell the body which direction to move and how fast. By making the force fall off by a factor of displacement squared, I was able to simulate gravitational pull on mouse click. By incorporating a "for" loop, I moved from applying the force on only a single body to a row of ten bodies. From there, although tedious, I jumped to applying the force to an array of 10,000 bodies. Upon simulating 10,000 bodies, I found that I had to lower the time step to keep things running smoothly as my computer was having trouble keeping up.

I was able to add a variety of features to "Gravity 1." By simply creating a global boolean and adding a negative sign to the force vector, I made a reverse gravitational force toggle. Alongside the toggle I also incorporated dampening, color, and even a black hole functionality to the project simply by playing around with the force vector formula. By combining all these various settings together, I was able to generate mesmerizing kaleidoscopic effects (See figure 1.1).

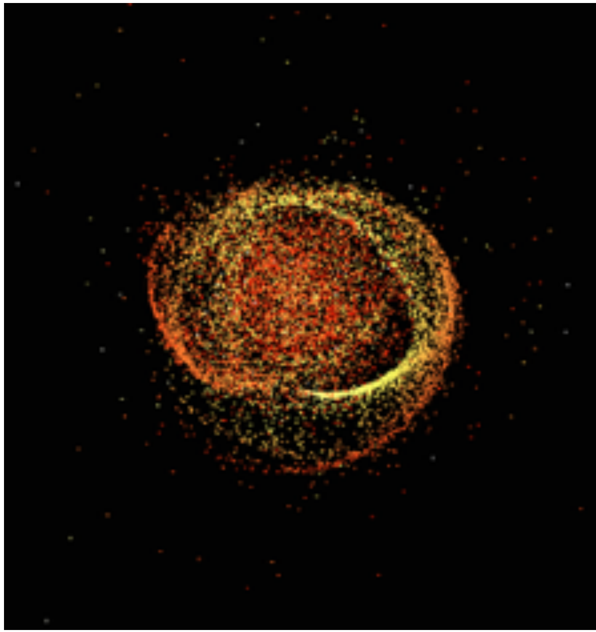


Figure 1.1: Screenshot from Gravity 1

Chapter 2
Gravity 2

Following my creation of Gravity 1, I was motivated to add on more functionality to it. It was when I decided to add color to the various points on the screen that I realized I could take my project one step further by adding the element of music. By having all the points on the screen move to the sound of music, I would have a new dimension to my project. My project would no longer remain a sandbox application, but would now become an interactive music visualizer.

I found that five main steps were needed in order to reach my goal:

1. Open the WAV music file and read the header.

2. Read the music data in chunks called windows.

3. Apply the Fast Fourier Transform to each chunk.

4. Visualize the output of the Fast Fourier Transform.

5. Play the music.

WAV Files

For my project, I used a WAV file because of it's simple structure and easy to access data. WAV files (Wavefore Audio File Format) are a type of uncompressed audio file meaning that the audio quality isn't reduced or encoded to to save memory. The uncompressed nature of a WAV file is advantageous in that it removes the step of having to decode the audio data. The uncompressed nature of the WAV file is also a setback as it makes the file size larger and more cumbersome. For this reason, WAV files are no longer the preferred audio format and instead lighter and more compact formats like MP3's are used.

All WAV files consist of two main pieces. The first piece is the header, which contains the information about how the audio data is stored (See figure 2.1). The second piece is the actual audio data itself. The header always takes up the first forty four bytes of a WAV file. Every few bytes of the header represents a different value. For instance, the bytes in between forty and forty four tell you the total size of the audio data. The header itself is further subdivided in to three main categories: the "RIFF" chunk descriptor, the "fmt" sub-chunk, and the data sub-chunk. For the purposes of my project I only looked at four main sections of the header:

1. **NumChannels**: This tells you how many audio channels are encoded. For instance, a music with just one audio channel would only have a single stream of music playing. Most music, however, is recorded with two audio channels. One for the left and one for the right. This is why in some music you might notice that the sound coming out from your left earpiece is different from your right earpiece.

2. **SampleRate**: This tells you the number of data samples in a second for this audio file. Most music is recorded at a sample rate of 44100hz, which means there are 44100 samples per second.

3. **BitsPerSample**: This tells you how many bits are allocated per sample. Generally this number is eight or sixteen.

4. **Subchunk2Size**: As mentioned before, this refers to the memory size of the audio data. A typical four minute song is around forty megabytes.

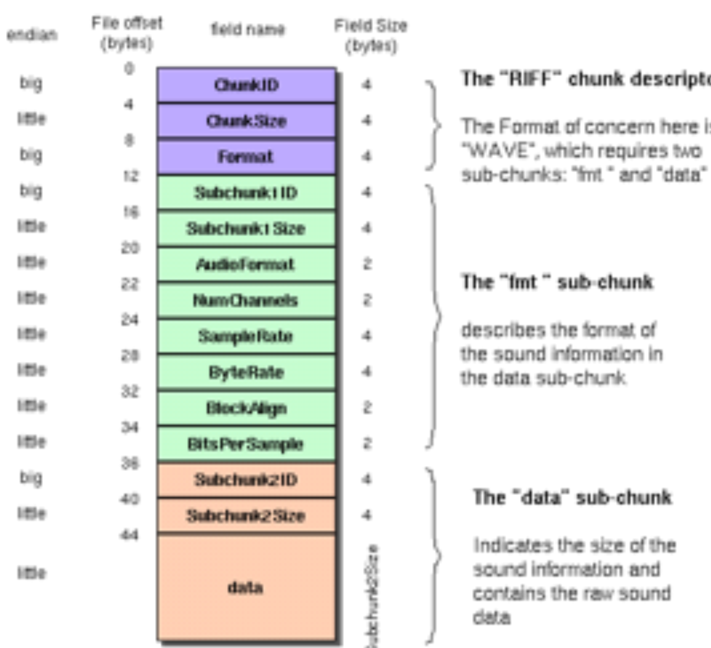


Figure 2.1: Anatomy of a WAV File Header

To read the header of the music: First, I located the music files. Then I used the C command fread to read in the header data piece by piece. I would start by reading in four bytes for the Chunkid. Subsequently, I would read another four bytes for the ChunkSize and another four bytes for the Format; so on and so forth until the header was all decoded.

Windowing

After reading the header and analyzing it's contents, the next step is to read in the audio data window by window. This process, known as windowing, is done because it is impractical to analyze all the music data at once and because it is necessary for producing each "frame" of the visualization. I found that the sweet spot for window size is generally either 2048 samples per window or 4096 samples per window. This translates to roughly .05s and .1s worth of data per window respectively. I made sure to sync my windowing with real time such that for every .1s that passed I was reading in .1s worth of data. I accomplished this by having the code consistently check the clock to see if the allotted time has passed.

One hurdle I had to overcome while windowing was taking into account the number of channels presents in an audio file. If a file has two audio channels then twice the number of samples had to be read in per window. Taking the number of channels into account was tricky. The file header might tell you that the sampling rate is 44100 samples per second, but this actually means that there are 44100 samples per second per channel. Thus a music file with two channels would actually have 88200 samples per second.

Another hurdle I had to overcome was the leakage associated with windowing. Because sound is naturally continuous, when you try to chop up the audio file into windows and process them, you get noise leakage from the adjacent windows. Consequently, a single note might end up looking like a spectrum of notes due to this leakage(see figure 2.2). To fix this windowing functions had to be implemented. Windowing functions are mathematical equations applied to windows to filter out the noise caused by windowing. We see from the image below the vastly different outcomes one receives when using a windowing function

For my project, I implemented and tested 3 different windowing functions: the Hann window, the Hamming Window and the Blackman-Nuttall window (see figure 2.2). Out of the three, I found that the Blackman-Nuttall window produced the best results due to the sharp peaks it would produce.

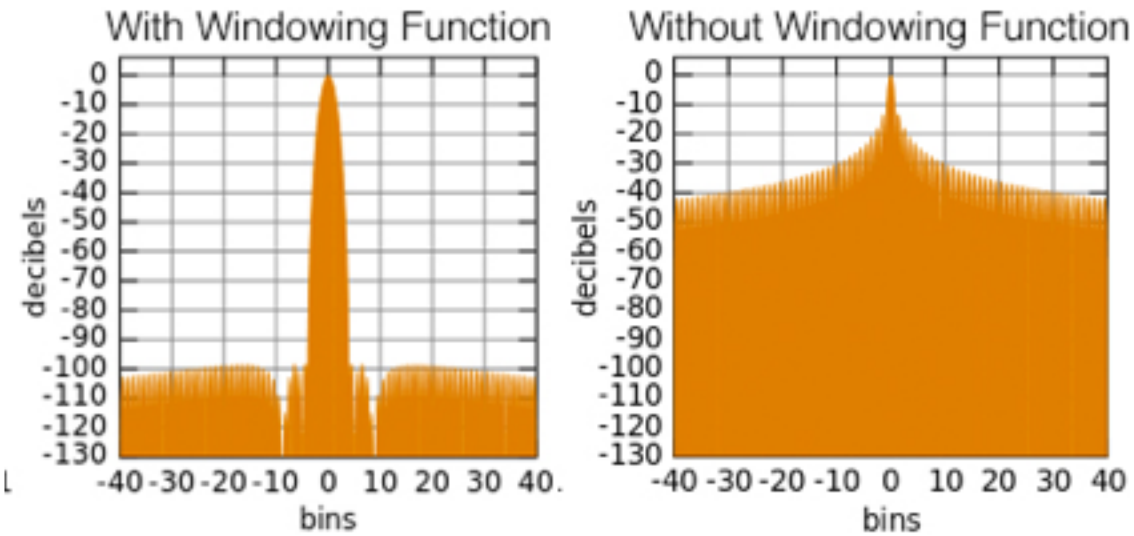


Figure 2.2: The Difference in Applying the Blackman-Nuttall Windowing Function

The Fast Fourier Transform

After reading in a window of data, the next step is to apply a Fast Fourier Transform on it. Normally, the Fourier transform is a slow algorithm, taking $O(n^2)$ exponential time to compute. The fast Fourier transform (FFT) is an optimized algorithm that is able to compute a Fourier transform in sub-exponential time O(nlogn). The Fourier transform is a crucial step because normally sound waves are represented as continuous oscillating wave (be it sine or cosine). By applying a Fourier transform, rather than getting continuous amalgamated mess, it allows you to obtain discrete note spikes(see figure 2.3). In terms of a music visualizer, each spike would equivalent to a bar.

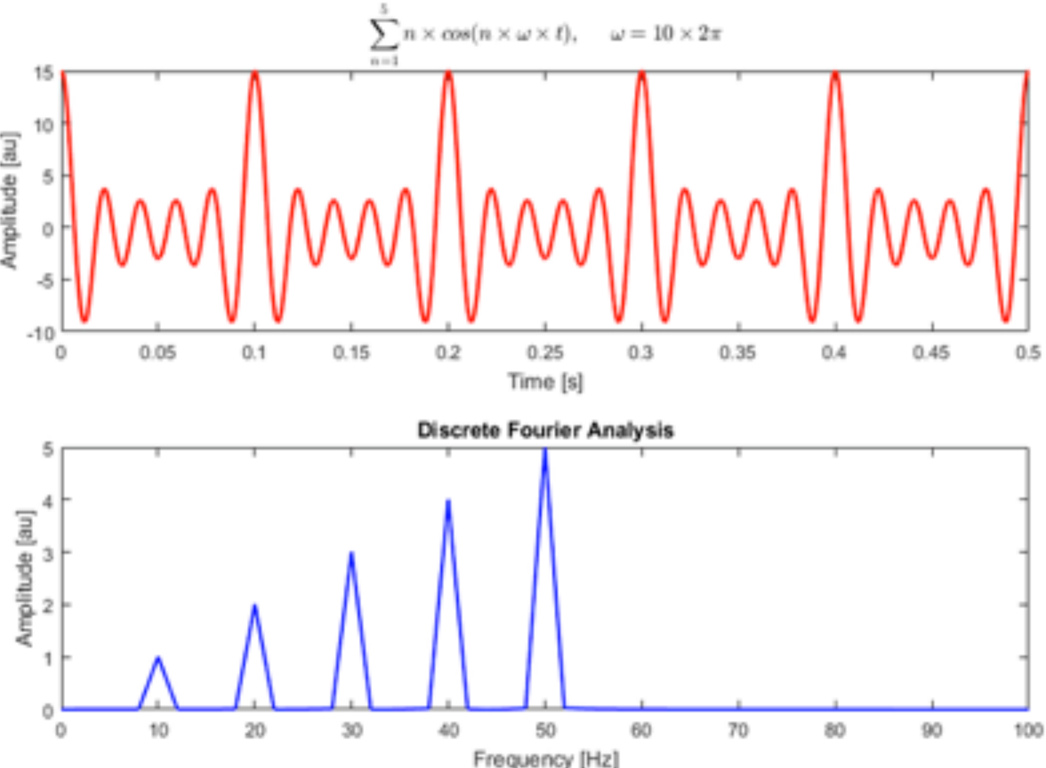


Figure 2.3: Example of Fourier Transform Discretization

To apply the fast Fourier transform, I used an external library called kissFFT. I took each window of data and ran kissFFT's fast Fourier transform function to get the desired output.

Visualization and Music Playing

Now that I had the output from the FFT, the next step is to visualize the data. To do so I started by splitting up my output into sixteen different bins, with each bin taking in a different frequency range. Each bin correlate to a music visualizer bar. It is important to note however that these bins are not of equal size because of the exponential nature of musical frequency. To further elaborate, C1 is about thirty hertz, and the next C an octave up is around sixty hertz. Our intuition tells us that the following C should be ninety hertz, but in actuality it is actually 120. The number of hertz between each note doubles with each octave. A bin size of thirty hertz would mean the notes A - E(60-90 hertz) on the low end of the spectrum. An octave up it would only cover the notes from A-C(120-150 hertz). Thus an equal bin size would result in bins covering an unequal range of notes. To resolve this issue, I doubled the size of each of subsequent bins.

Because I had sixteen bins, I divided my window lengthwise into sixteen columns. Based on the value in each bin, I then applied a force to all the points at the bottom of the corresponding column. I called this function my “kick” function, it is used to emulate how music visualizers kicks their bars up.

The last task that needed to be completed for the music visualizer was to play the music. To accomplish this task, I used a C library called SDL2 and SDL Mixer. Playing the music was simply a matter of finding the file and then plugging it into the SDL Mixer play function.

The main difficulty here was in linking the frameworks associated with the two libraries and getting all the files to play nice with each other.

Results

The final product is a program that is able to successful play and visualize WAV music files. The program audio and visual perfectly align, and the visualization appears to accurate depict the music. The program also boast a plethora of features that can be mixed and matched to create for mesmerizing effects.

The program however does suffer from multiple flaws. One of which being optimization. The program has a tendency to run noticeably slower with the addition of more points. Another flaw that the program has is it’s limitation to Macintosh OS. Through testing I found that the program only works successfully on Apple computers. Additionally, the program often has complications searching for the music file unless certain criteria are met.

Features and Controls

To run the program: double click on the Gravity 2 Unix Executable file and the program should start. To swap out music file: locate the music.wav file and replace it with a music.wav file of your own. The name must be “music.wav” or else the program will not be able to locate it.

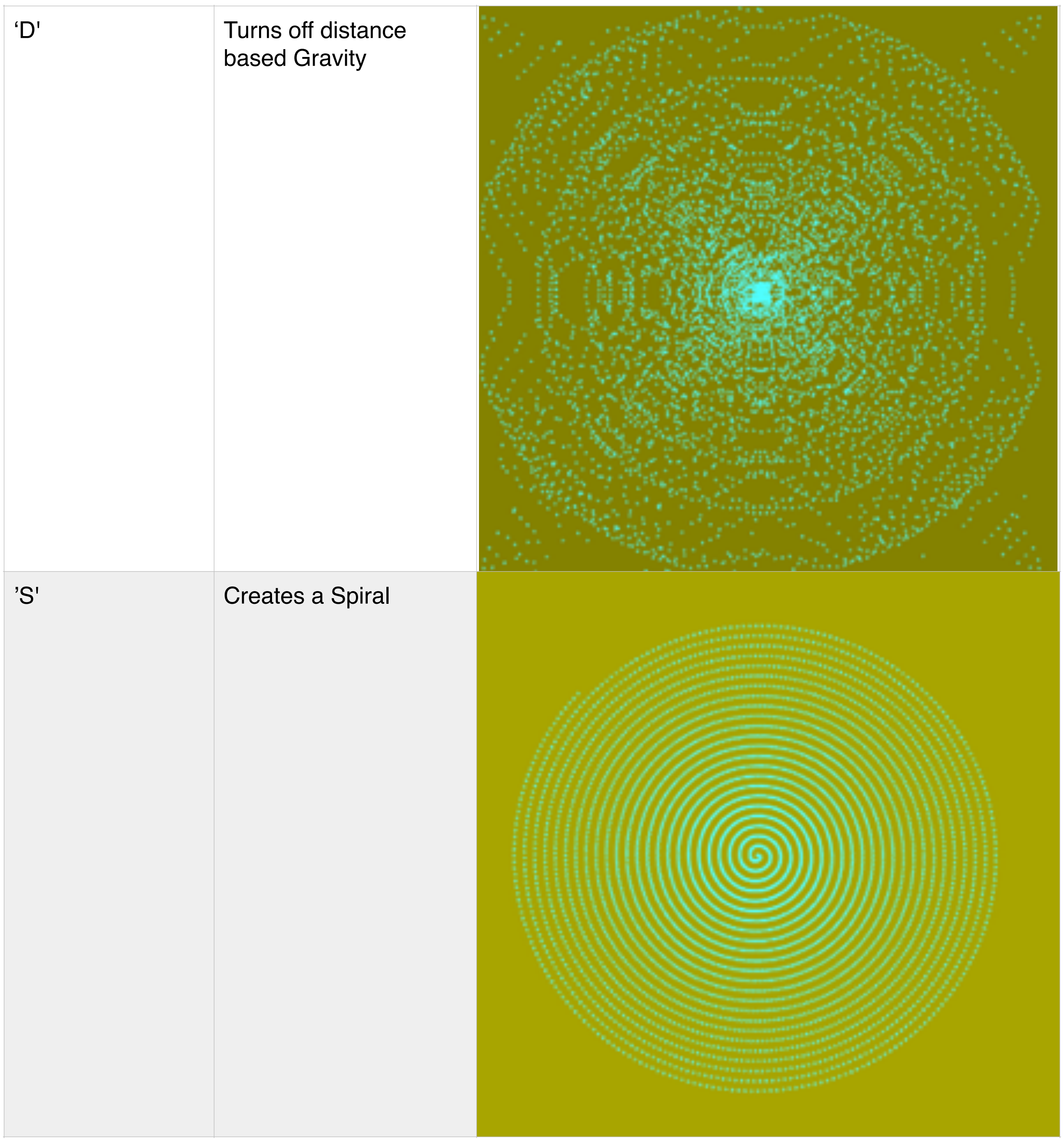
The file is available for download at: Jinnovate.com/Gravity2.zip

The following are a list of notable controls and toggles that the program has.

Control	Description	Screenshot
R	Resets the Screen	
0 - 9	Change Colors of Screen and Dots	

Numbers ’n'	Changes the Number and size of Particles	
m’ + ‘d’	Bars are now radial around the mouse rather than at the bottom of window	
‘l'	toggles borders	
‘g'	toggles gravity	
‘v'	toggles air resistance	

Mouse Click	Applies a gravitational force on all points	
‘r’	Toggles Repulsive Force	

‘D'	Turns off distance based Gravity	
’S'	Creates a Spiral	

Chapter 3
Future Pursuits

The great thing about this project is that there is never a point at which the project is fully complete. There are always new features that can be added to the program and optimizations that can be made. Three improvements I hope to add to my program are:

1. **MP3 compatibility**: As stated earlier, the WAV file format is extreme large and bulky. Consequently, no one really uses WAV files to store their music. Having the program also work with MP3 would make the program more accessible and inviting to use. I hope to implement this in the future by incorporating a WAV to MP3 library
2. **Note Detection**: Another feature I would like to add is the ability for the program to differentiate notes. For instance, it would be able to make all 'A' notes red or 'B' notes blue. To achieve this I would have to increase my window size to be able to successfully pinpoint what notes are being played.
3. **User Interface**: I would also like to incorporate a general user interface so that controls become more intuitive to use. I would likely implement this by importing an external prebuilt GUI library.

Works Cited

A discrete Fourier analysis of a sum of cosine waves at 10, 20, 30, 40, and 50 Hz. Digital image. N.p., n.d. Web. <https://en.wikipedia.org/wiki/Fast_Fourier_transform#/media/File:FFT_of_Cosine_Summation_Function.png>.

Blackman–Nuttall window; B = 1.9761. Digital image. N.p., n.d. Web. <https://en.wikipedia.org/wiki/Window_function#/media/File:Window_function_and_frequency_response_-_Blackman-Nuttall.svg>.

A discrete Fourier analysis of a sum of cosine waves at 10, 20, 30, 40, and 50 Hz. Digital image. N.p., n.d. Web. <https://en.wikipedia.org/wiki/Fast_Fourier_transform#/media/File:FFT_of_Cosine_Summation_Function.png>.

"Fast Fourier Transform." Wikipedia. Wikimedia Foundation, 03 May 2017. Web. 08 May 2017. <https://en.wikipedia.org/wiki/Fast_Fourier_transform>.

"Fourier Transform." Wikipedia. Wikimedia Foundation, 04 May 2017. Web. 08 May 2017. <https://en.wikipedia.org/wiki/Fourier_transform>.

Microsoft WAVE soundfile format. Digital image. N.p., n.d. Web. <http://soundfile.sapp.org/doc/WaveFormat/wav-sound-format.gif>.

Rectangular window; B = 1.0000. Digital image. N.p., n.d. Web. <https://en.wikipedia.org/wiki/Window_function#/media/File:Window_function_and_frequency_response_-_Rectangular.svg>.

"Window Function." Wikipedia. Wikimedia Foundation, 05 May 2017. Web. 08 May 2017. <https://en.wikipedia.org/wiki/Window_function>.