MAT-265 Open Projects in Computational Processes
Project Documentation

Experiments with Sonic Sculptures

Overarching theme:

Experimental abstract sound driven sculptures generated through the interaction of audio data and simple physics simulation.

Focus:

- Visualizing audio spectrum data from an audio source, audio clip or microphone into abstract 3D art worlds in Unity
- Development of a framework to use spectrum data to drive any phenomena and behavior in Unity
- Experimentation with generating abstract immersive sonic sculptures using audio data and physics simulation functionality

Methodology:

Unity game engine was used to prototype the experiments in this project. Audio in Unity is handled by a component called AudioSource which is attached to a GameObject to play back sounds in a 3D environment. An AudioListener is attached to the Camera to capture the run time mix based on player's movement and directionality.

In order to extract data from an AudioSource, we must create a reference to the 'AudioSource' component of a Unity GameObject and a pair of float arrays to store the spectrum data for each channel.

```
AudioSource _audioSource;
private float[] _samplesLeft = new float[512];
private float[] _samplesRight = new float[512];
_audioSource = GetComponent<AudioSource>();
```

We then run the GetSpectrumData() method for each channel on the audiosource component referenced in _audiosource. This method provides a block of the currently playing audio source's spectrum data. The number of samples must be a power of 2, with a minimum of 64 and maximum of 8192. There are six types of windows available to reduce leakage between

frequency bins/bands. The more complex the window type, the better the quality, but reduced speed. On testing the different window options, there was no noticeable difference at the level of sampling and computation that this experiment is working with so Unity's default window choice, FFTWindow.Blackman was used.

```
_audioSource.GetSpectrumData(_samplesLeft, 0, FFTWindow.Blackman);
_audioSource.GetSpectrumData(_samplesRight, 1, FFTWindow.Blackman);
```

SWITCH TO MANUAL

public void **GetSpectrumData**(float[] **samples**, int **channel**, FFTWindow**window**);

The spectrum arrays _samplesLeft and _samplesRight can be used directly to drive visualization, however since these are FFT values designed to give a precise reading, they will not create a lot of movement to drive easily perceptual changes. In order to condense the spectrum information for real time playback purposes, a set of eight and sixty four frequency bands were chosen to ultimately drive visualizations.

When the number of samples for the FFT function are set to 512, and with a sampling rate of 44100, there are 43Hz for every sample ((44100 / 2) / 512). We then categorize the samples into eight frequency bands. To find the range limits of each band, we run an iteration loop with a simple algorithm to set our ranges. Below is a table showing the number to be multiplied by 43 on each iteration to get the frequency count (number of frequencies in the range) in each iteration. The start of the range is the upper limit of the previous iterations frequency range plus one, and the upper limit of the current iteration is the start value plus frequency count.

0 - 2 = 86Hz = 0 - 86
1 - 4 = 172Hz = 87 - 258
2 - 8 = 344Hz = 259 - 602
3 - 16 = 688Hz = 603 - 1290
4 - 32 = 1376Hz = 1291 - 2666
5 - 64 = 2752Hz = 2667 - 5418
6 - 128 = 5504Hz = 5419 - 10922
7 - 256 = 11008Hz = 10923 - 21930

The numbers to be multiplied can be calculated using the Mathf.Pow(F,P) exponent function, which returns F raised to P. We can efficiently approximate frequency band ranges with shorter sizes at the low frequency and exponentially larger sizes for the very high frequencies that are generally not used in music.

8 Frequency Bands:

Two nested loops are used, one to calculate the sampleCount for each frequency band range and the other to populate the frequency band array with the average value of the FFT samples for that range.

```
private float[] _freqBand = new float[8];
void MakeFrequencyBands()
{
    int count = 0;

    for(int i = 0; i < 8; i++)
    {
        float average = 0;
        int sampleCount = (int)Mathf.Pow(2, i) * 2;

        if(i == 7)
        {
            sampleCount += 2;

        }
        for(int j = 0; j < sampleCount; j++)
        {
            if(channel == _channel.Stereo)
            {
                average += (_samplesLeft[count] + _samplesRight[count]) * (count + 1);
            }
            if(channel == _channel.Left)
            {
                average += _samplesLeft[count] * (count + 1);
            }
            if(channel == _channel.Right)
            {
                average += _samplesRight[count] * (count + 1);
            }
            count++;
        }
        average /= count;
        _freqBand[i] = average * 10;
    }
}
```

64 Frequency Bands:

Dividing up the samples for 64 frequency bands required a custom logic as the method used for the 8 frequency bands would not work here without modification. The table below shows the currently used distribution of FFT samples per band:

0-15 bands = 1 sample/band = 16
16 - 31 bands = 2 samples/band = 32
32 - 39 bands = 4 samples/band = 32
40 - 47 bands = 6 samples/band = 48
48 - 55 bands = 16 samples/band = 128
56 - 64 bands = 32 samples/band = 256
+= 512 samples total

```
void MakeFrequencyBands64()
    {
        int count = 0;
        int sampleCount = 1;
        int power = 0;

        for (int i = 0; i < 64; i++)
        {
            float average = 0;


            if (i == 16 || i == 32 || i == 40 || i == 48 || i == 56)
            {
                power++;
                sampleCount = (int)Mathf.Pow(2, power);
                if (power == 3)
                {
                    sampleCount -= 2;
                }

            }
            for (int j = 0; j < sampleCount; j++)
            {
                if (channel == _channel.Stereo)
                {
                    average += (_samplesLeft[count] + _samplesRight[count]) * (count + 1);
                }
                if (channel == _channel.Left)
```

```
            {
                average += _samplesLeft[count] * (count + 1);
            }
            if (channel == _channel.Right)
            {
                average += _samplesRight[count] * (count + 1);
            }
            count++;
        }
        average /= count;
        _freqBand64[i] = average * 10;
    }
}
```

Buffers were added as an option to slowly reduce the value of the bands to prevent abrupt scaling down, which helps with the aesthetics of the visualization.

```
void BandBuffer()
{
    for(int g = 0; g < 8; ++g)
    {
        if (_freqBand[g] > _bandBuffer[g])
        {
            _bandBuffer[g] = _freqBand[g];
            _bufferDecrease[g] = 0.005f;
        }
        if(_freqBand[g] < _bandBuffer[g])
        {
            _bandBuffer[g] -= _bufferDecrease[g];
            _bufferDecrease[g] *= 1.2f;
        }
    }

}
```

The arrays _audioBand[] and _audioBandBuffer[] are assigned and populated based on the frequency band range.

```
void CreateAudioBands()
{
    for (int i = 0; i < 8; i++)
    {
```

```
            if (_freqBand[i] > _freqBandHighest[i])
            {
                _freqBandHighest[i] = _freqBand[i];
            }
            _audioBand[i] = (_freqBand[i] / _freqBandHighest[i]);
            _audioBandBuffer[i] = (_bandBuffer[i] / _freqBandHighest[i]);
        }
    }
```

The normalized data from the _audioBand[] and _audioBandBuffer[] arrays are used to calculate the amplitude of the audiosource.

```
void GetAmplitude()
    {
        float _CurrentAmplitude = 0;
        float _CurrentAmplitudeBuffer = 0;

        for (int i = 0; i < 8; i++)
        {
            _CurrentAmplitude += _audioBand[i];
            _CurrentAmplitudeBuffer += _audioBandBuffer[i];
        }
        if (_CurrentAmplitude > _AmplitudeHighest)
        {
            _AmplitudeHighest = _CurrentAmplitude;
        }
        _Amplitude = _CurrentAmplitude / _AmplitudeHighest;
        _AmplitudeBuffer = _CurrentAmplitudeBuffer / _AmplitudeHighest;
    }
```

Visualization Experiments

Now that we have our audio data organized, we can start plugging in the frequency band values into parameters of objects in Unity. The main variables used are _audioBand[] and _audioBandBuffer[] for 8 bands; _audioBand64[] and audioBandBuffer64[] for 64 bands.

For these set of experiments, a template visualization was created using 64 Cylinder 3D objects representing each of the 64 frequency bands. Tests were done with 8 bands but the more interesting visualizations emerged out of using 64 bands of frequency ranges. The scales of the cylinders were driven by the value of the frequency bands using the transform.localScale function and the following vector calculation:

```
transform.localScale = new Vector3((_audiopeer._AmplitudeBuffer * _scaleMultiplier) +
_startScale, (_audiopeer._AmplitudeBuffer * _scaleMultiplier) + _startScale,
(_audiopeer._AmplitudeBuffer * _scaleMultiplier) + _startScale);
```

_startScale and _scaleMultipliers are user set variables that are visible in the Unity Editor menu under each cylinder object, or any object that contains a script with this functionality in it.

The visualizations explored how audio data reacts with physics simulation data. Each set of cylinders were made to attract to different objects using an attraction force script. The cylinders were also given rigid body physics dynamics and colliders so they would interact with each other as they cycled through their attraction logic.

```
Rigidbody _rigidbody;
__public Transform _attractedTo;
__public float _strengthOfAttraction, _maxMagnitude;
 Vector3 direction = _attractedTo.position - transform.position;
_____ _rigidbody.AddForce(_strengthOfAttraction * direction);
```

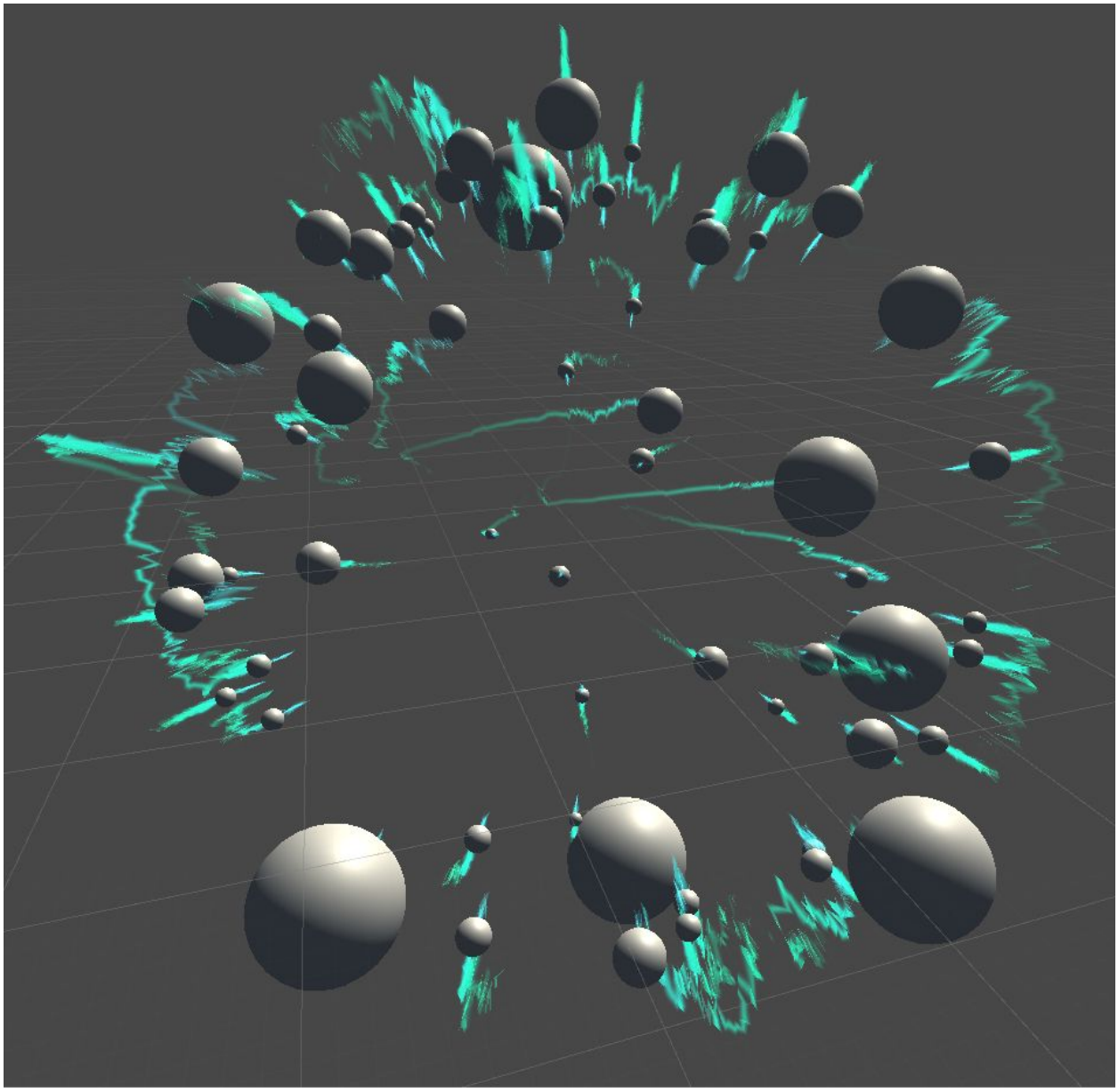
With this basic setup, the geometry was cycled between cylinders, spheres and cubes being a combination of attractor object or attracted particle objects or both. The scale of the attractor object was also driven by either a frequency band or the amplitude of the audiosource. Trail renderer components were also added to each particle object to draw a path of their movement over time. The trails are set to decay after five seconds.

The frequency data for 64 bands was then used to drive the color properties of the materials and shaders associated to the particle objects, such as surface shaders and trail shaders. A color variable was driven by the frequency data, which in turn modified the material and shader attributes in real time. A user defined color gradient is set before run time and divided by the number of rows to determine the color for each frequency band.

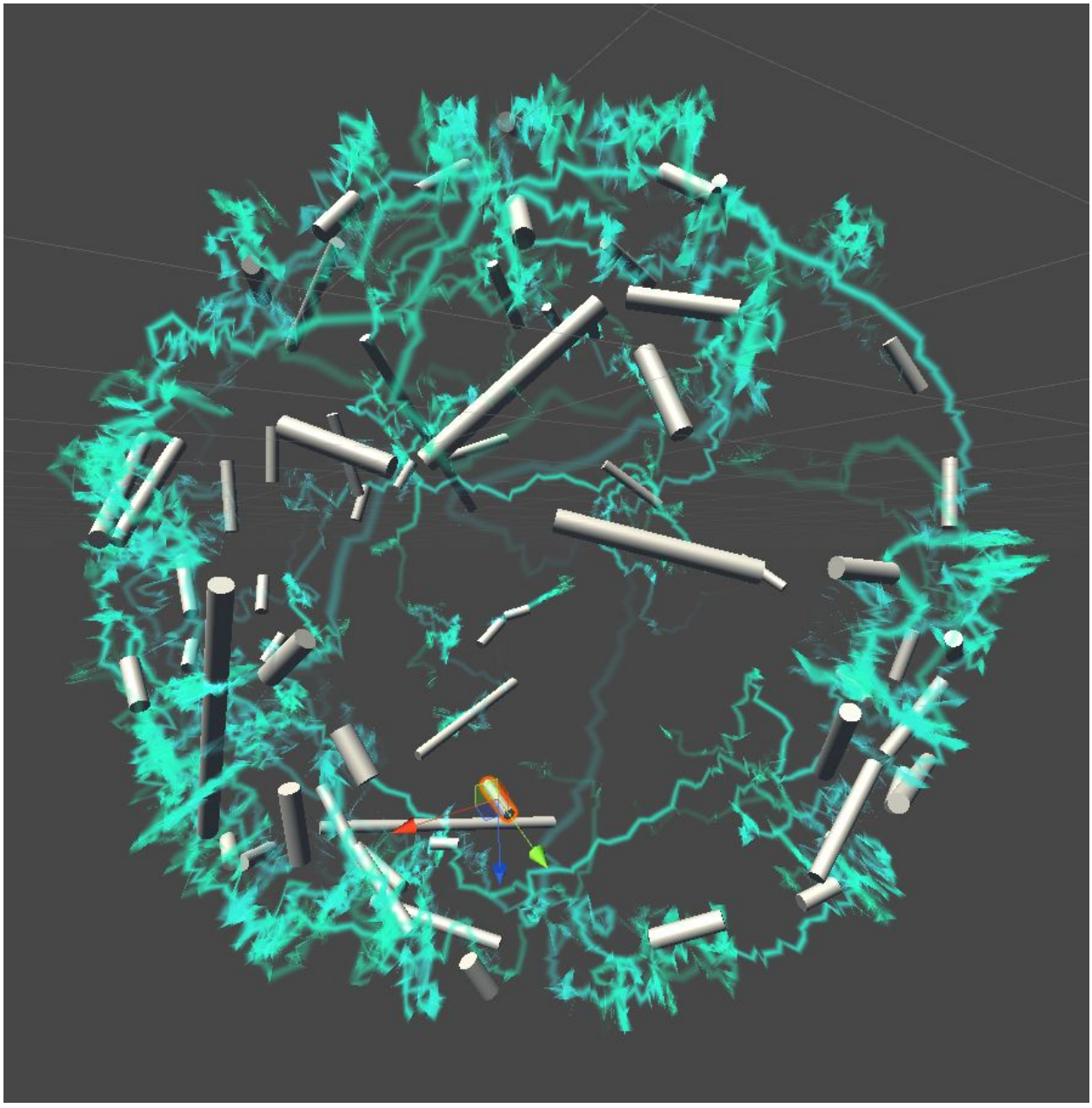
```
_color[i] = _colorGradient.Evaluate((1f / 64f) * i); // set colours to the gradient, dividing the
gradient by the number of rows. You can change the gradient in the inspector
Color DiffuseColor = new Color (
                                (_color [i].r * _audioPeer._audioBandBuffer64 [i]) *
_diffuseColorMultiplier,
                                (_color [i].g * _audioPeer._audioBandBuffer64 [i]) *
_diffuseColorMultiplier,
                                (_color [i].b * _audioPeer._audioBandBuffer64 [i]) *
_diffuseColorMultiplier, 1); // the diffuse color is always updating, to the bandbuffers, and
multiplied by the amount specified on the slider
                        _materialLine [i].SetColor ("_Color", DiffuseColor); //apply the color above
_____ _materialTrails[i].SetColor("_Color", DiffuseColor);
```

Results:

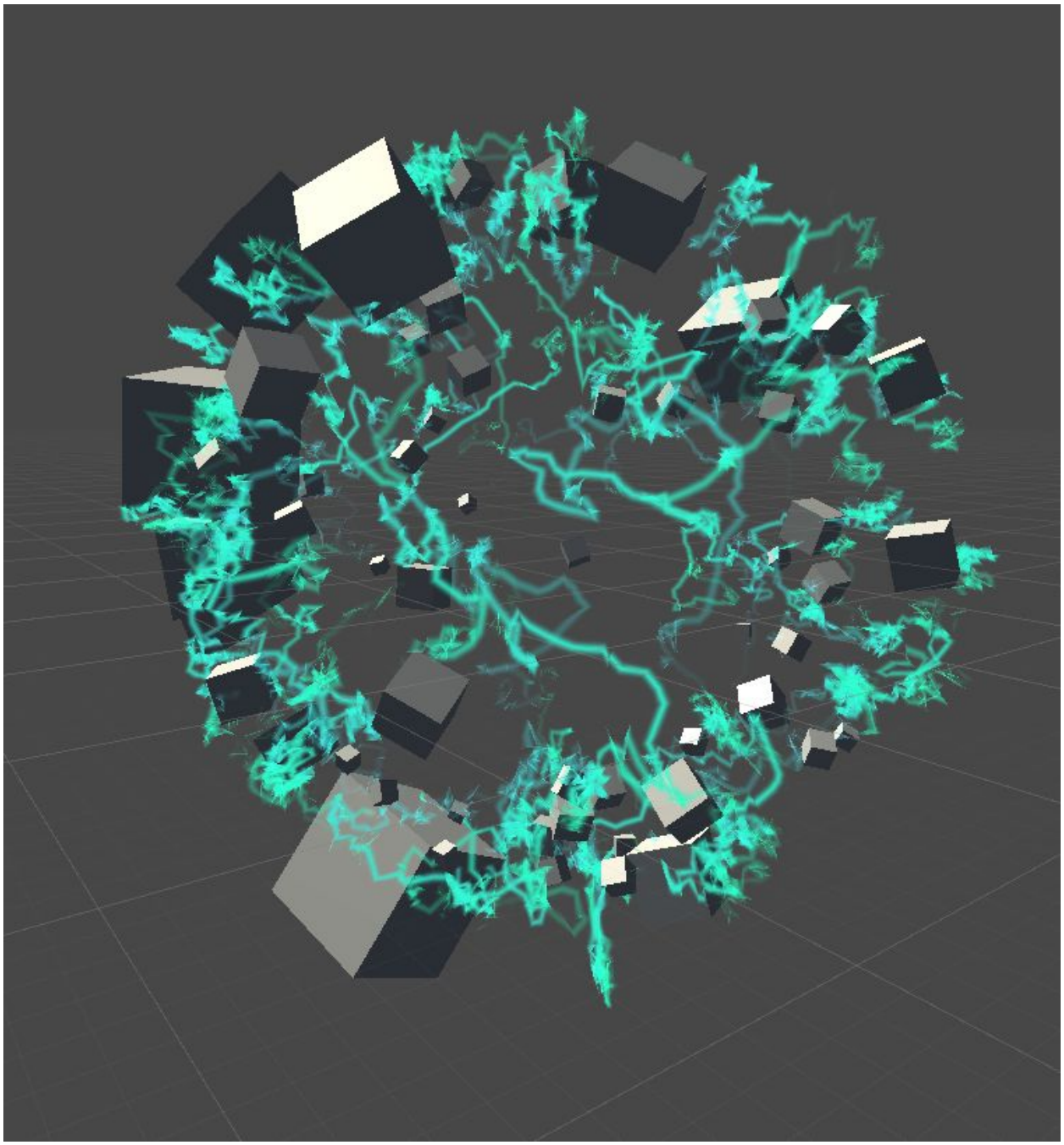
The sphere attractors produced the more immersive and compelling visualizations as the curved surfaces allowed the particles to spread out around the entire surface area. With cylinders and cubes, particles seemed to bunch up either around the equator of the cylinder or at the center of the four lateral sides. Below are the images showing the visualizations produced with sphere attractors:



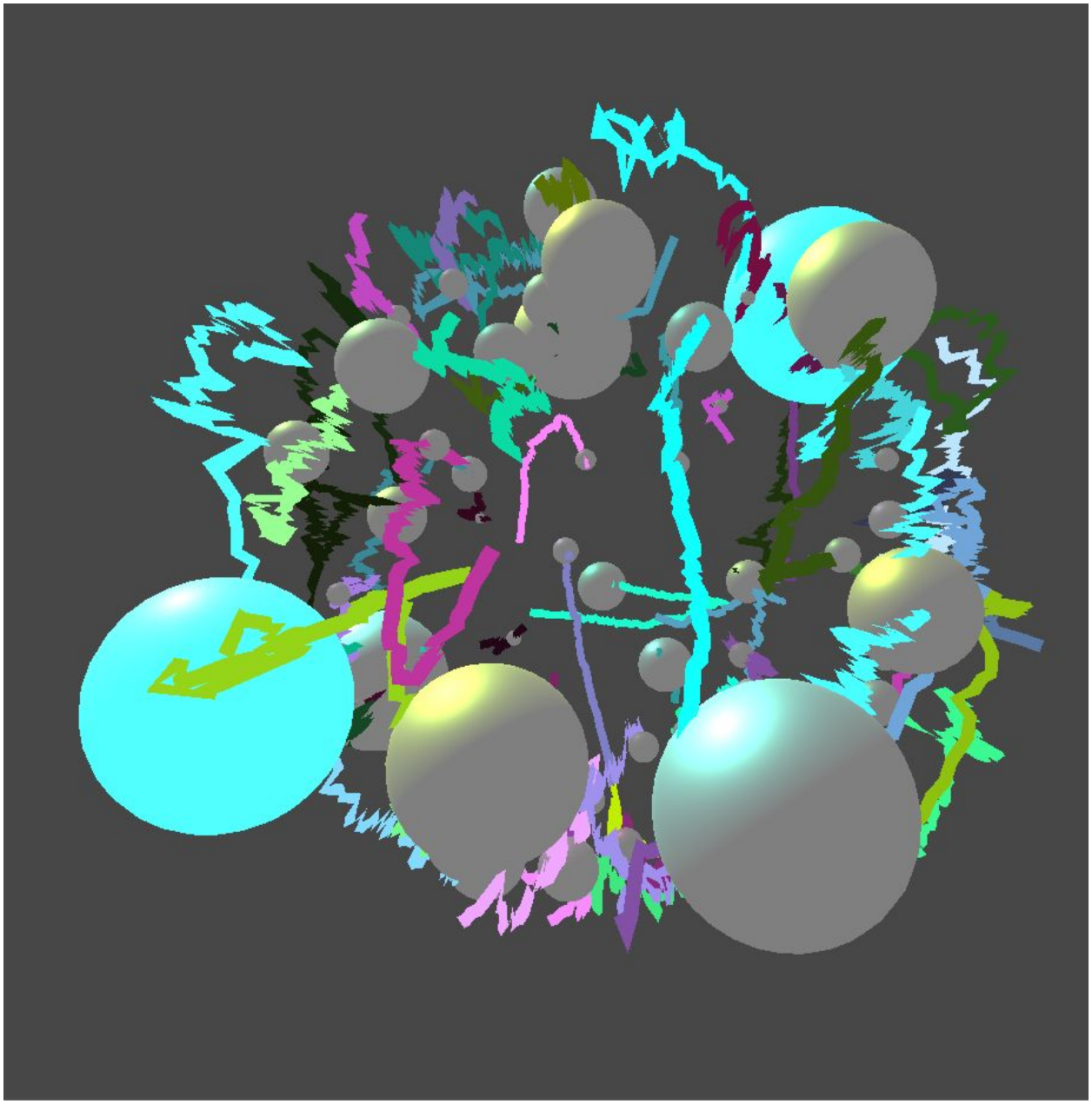
Sphere Attractor Sphere Particles



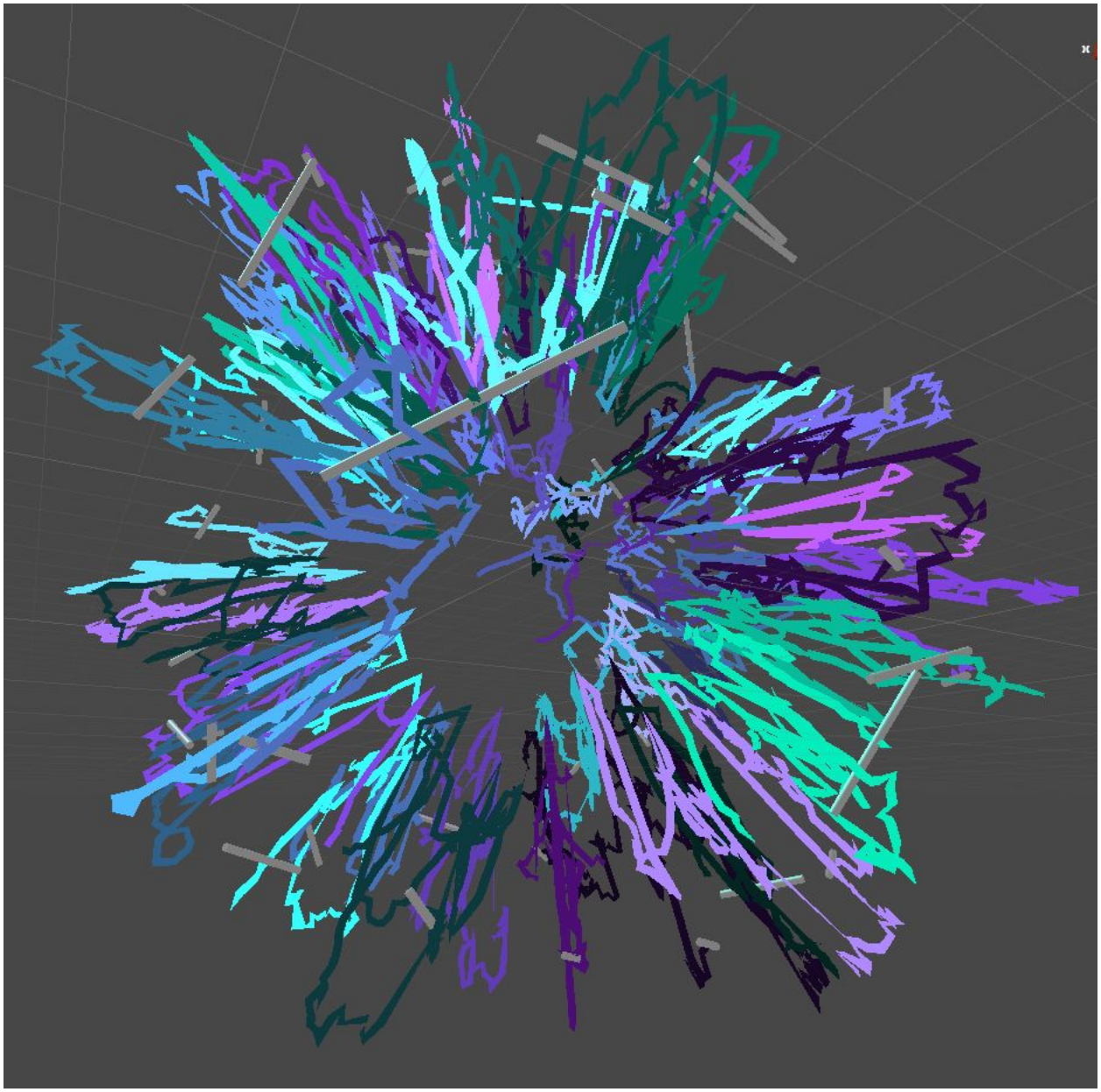
Sphere Attractor Cylinder Particles



Sphere Attractor Cube Particles



Music driven colors for Spheres on Sphere



Music driven colors for Cylinders on Sphere

Conclusions:

The interaction between audio frequency data and physics simulation was interesting to observe. When the particle objects were cylinders, the visualization produced the most noise, variance and asymmetry, while the spheres produced more uniform and symmetric visualizations. The cubes were on the noisy side, but their movements were not as erratic as the cylinders.

This project has been an exercise in understanding and exploring Unity's audio analysis capabilities as well as developing a framework to use audio spectrum data to drive any phenomena in an interactive experience. On that front, the code developed to drive the attractors, particle objects and colors using audio data will be reused for any future functionality that requires audio spectrum analysis.

The examples showed that this system can almost be used as a plug and play asset on any Unity object that has an AudioSource component. In this way, there still is a lot more experimentation left to do in order to find every permutation and combination of visualization that is possible using just the variables and audio values derived from this project. The prospect of quickly prototyping functionality in order to validate a concept is where Unity really shines, and the ability to develop experimental functionality as a modular asset allows for creative media artists to eventually start plugging individually developed functionality together in order to create new forms of code based or simulation based art and abstract immersive environments.

The next stages for this project involve further development of this framework using Unity's particle system, high definition render pipeline and GPU processing to enable simulation of thousands and possibly millions of particles, while using the maximum number of samples available through the FFT function - 8192. With flocking behavior and advanced physics simulation, there are a lot of applications for this functionality in building 3D objects that deform over time based on the interaction between them and the particle objects driven by frequency bands. This requires development of an efficient way of working with real time deformation, which can be computationally intensive.

References & Inspiration:

Soundself:
https://www.soundself.com/

Cymatics:
https://www.youtube.com/watch?v=Q3oItpVa9fs
https://www.youtube.com/watch?v=ftoFKlTcYEc

Golan Levin:
http://www.flong.com/projects/messa/
https://youtu.be/STRMcmj-gHc

The Well:
https://www.mat.ucsb.edu/g.legrady/academic/courses/06w256/projects/final/will-anne-marie/index.html

George Legrady MAT

https://www.mat.ucsb.edu/g.legrady/academic/courses/overview.html
https://www.mat.ucsb.edu/g.legrady/academic/courses/06w256/06w256.html

Adrien M & Claire B
L'ombre de la vapeur: https://vimeo.com/278181935
Mirages et miracles: https://vimeo.com/248983439