# Development- Beamforming

## Introduction

Sprint 3 is the development phase of the internship. My goal of this sprint was to start on the important features of the project. This includes the topic of this document: Beamforming. With Beamforming, we can create an image of the location and intensity of the source. And using a color code and threshold, we can create a heatmap out of it.

**! Later on, after a discussion with Sorama, we came to the conclusion that Sorama's algorithm is a bit too complex for an entry-level intern to implement into an unity project. Instead, Sorama accepts a simple mock version of the beamforming !**

This documentation contains information and pictures of all the beamforming algorithms, as well as the ones worked on in Sprint 4.

## Old Implementation: Sorama's Beamforming

During this project, the idea first was of using the C# Beamforming Algorithm used by Sorama, instead of writing my own algorithm.

For the project, I would only need this method that would get the required components, turn the array into a buffer and create a Beamform Image.

```
// Read inputs
var holograms = ReadBinary<Complexf>(inputFile, micCount, holoCount);
var mics = ReadBinary<Point3f>(arrayFile,micCount);
var targets = ReadBinary<Point3f>(targetFile, targetCount);
var frequencies = ReadBinary<float>(frequencyFile, holoCount);

// Create buffers
var handler = new GeneralHandler();
var hologramBuffer = handler.CreateBufferFromArray(holograms);
var targetBuffer = handler.CreateBufferFromArray(targets);
var frequencyBuffer = handler.CreateBufferFromArray(frequencies);
var micsBuffer = handler.CreateBufferFromArray(mics);

// Create beamformer and beamform
var beamformer = new FrequencyDomainBeamformerSSE(handler);
var response = beamformer.BeamformMagnitude(343.0f, hologramBuffer, micsBuffer, targetBuffer, frequencyBuffer);
```

To put this in Unity, a "HeadManager" is created that will call others managers to fill in the variables. This HeadManager is called BeamformingManager. Every Manager will inherit from an interface, so make it easy to replace it with a different manager. BeamformingManager inherits from IBeamformingManager

```
public interface IBeamFormingManager
{
    2 references
    void SetMicrophones();
    2 references
    void GetArraySize();
    2 references
    void GetHologramSize();
    2 references
    void GetScale();
}
```

```
void Update()
{
    SetMicrophones();
    GetArraySize();
    GetHologramSize();
    GetScale();
}
2 references
public void SetMicrophones()
{
  _microphonesOnCamera = _microphoneController.GetMicrophones();
}

2 references
public void GetArraySize()
{
    _microphoneArrayHeight = _arraySizeManager.GetArrayHeight(_microphonesOnCamera);
    _microphoneArrayWidth = _arraySizeManager.GetArrayWidth(_microphonesOnCamera);
}

2 references
public void GetHologramSize()
{
    _hologram = _hologramManager.ReturnHologramSize();
}

2 references
public void GetScale()
{
    _scaleHeight = _hologramManager.GetScaleHeight(_microphoneArrayHeight);
    _scaleWidth = _hologramManager.GetScaleWidth(_microphoneArrayWidth);
}
```

## Microphones

The first thing for the beamforming was to create the microphones. A class was created called MicrophoneController which would inherit from IMicrophoneController.

```
public interface IMicrophoneController
{
    2 references
    void SetMicrophones();
    2 references
    List<Microphone> GetMicrophones();
}
```

SetMicrophones will find the microphones attached to the gameobject and add them to the _microphones List. The class also contains Methods that would set and reset the variables per microphone.

```
public void SetMicrophones()
{
    Microphone[] microphones = GetComponentsInChildren<Microphone>();
    if (microphones != null)
    {
        foreach (Microphone mic in microphones)
        {
            _microphonesOnCamera.Add(mic);
        }
    }
    else
    {
        Debug.LogError("No Microphones found");
    }
}

1 reference
void SetVariablesMicrophone()
{
    _logicManager.SetVariables(_microphonesOnCamera, soundSourceInRange);
}

1 reference
void ResetVariablesMicrophone()
{
    _logicManager.ResetVariables(_microphonesOnCamera);
}
```

In order to make the code more readable and SOLID, A LogicManager class was created that would handle all the math related logic, such as calculating the Amplitude, distance and phase per microphone. LogicManager inherits from ILogicManager.

```
public interface ILogicManager
{
    2 references
    void SetVariables(List<Microphone> _microphones, SoundSourceSinewave _soundSource);
    2 references
    void ResetVariables(List<Microphone> _microphones);
    2 references
    float GetDistance(Vector3 _micPos, Vector3 _soundPos);
    2 references
    float GetAmplitude(float _amplitude, float _distance);
    2 references
    float GetPhase(float _frequency, float _distance);
}
```

```
2 references
public float GetAmplitude(float _amplitude, float _distance)
{
    //Amplitude decreases over distance.
    //Amplitude is inversely proportional to distance  A = 1/d

    //Since neither of them can be zero, we need to check that if the distance is bigger than 1
    if (_distance < 1)
    {
        return _amplitude;
    }
    float factor = (1 / _distance);
    float newAmplitude = _amplitude * factor;
    return newAmplitude;
}

2 references
public float GetDistance(Vector3 _micPos, Vector3 _soundPos)
{
    float distance =  Vector3.Distance(_micPos, _soundPos);
    //Make sure the distance is never negative
    return Mathf.Abs(distance);
}

2 references
public float GetPhase(float _frequency, float _distance)
{
    //Step 1. Get the wave length which is 2PI / frequency
    float waveLength = (2 * (Mathf.PI)) / _frequency;
    //Step 2. Calculate the remaining distance
    float remainingDistance = (_distance % waveLength);
    //Step 3. Get the phase
    float phase = (((2 * (Mathf.PI)) * remainingDistance) / waveLength);
    return phase;
}
```

```
2 references
public void ResetVariables(List<Microphone> _microphones)
{
    foreach (Microphone mic in _microphones)
    {
        mic.Frequency = 0;
        mic.Distance = 0;
        mic.Amplitude = 0;
        mic.Phase = 0;
    }
}

2 references
public void SetVariables(List<Microphone> _microphones, SoundSourceSinewave _soundSource)
{
    foreach (Microphone mic in _microphones)
    {
        mic.Frequency = _soundSource.Frequency;
        mic.Distance = GetDistance(mic.transform.position, _soundSource.Position);
        mic.Amplitude = GetAmplitude(_soundSource.Amplitude, mic.Distance);
        mic.Phase = GetPhase(mic.Frequency, mic.Distance);
    }
}
```

The way MicrophoneController works is that the object attached to it has a sphereCollider with a range that can dynamically change. When a soundsource enters this range, it will set the variables for every microphone and will always do that until that soundsource leaves the range.

```
1 reference
void ChangeRadius(float radius)
{
    if (_rangeReference != radius)
    {
        _rangeCamera.radius = radius;
        _rangeReference = radius;
    }
}

Unity Message | 0 references
private void OnTriggerEnter(Collider other)
{
    if (other.TryGetComponent<SoundSourceSinewave>(out SoundSourceSinewave soundSource))
    {
        soundSourceInRange = soundSource;
    }
}

Unity Message | 0 references
private void OnTriggerExit(Collider other)
{
    if(other.TryGetComponent<SoundSourceSinewave >(out SoundSourceSinewave soundSource))
    {
        soundSourceInRange = null;
        ResetVariablesMicrophone();

    }
}
```

```
void Update()
{
    ChangeRadius(_cameraRange);
    if(soundSourceInRange != null)
    {
        SetVariablesMicrophone();
    }
}
```

## Size

The next step was to calculate the width and height of the microphone array. This will be used later to calculate the Hologram and the positions of the target points.

The Class ArraySizeManager is used to calculate the width and height. The class inherits from IArraySizeManager

```
public interface IArraySizeManager
{
    2 references
    void GetMinandMaxX(List<Microphone> _microphones);
    2 references
    void GetMinandMaxY(List<Microphone> _microphones);
    2 references
    float GetArrayWidth(List<Microphone> _microphones);
    2 references
    float GetArrayHeight(List<Microphone> _microphones);

}
```

ArraySizeManager gets the Minimum and Maximum of each axis and uses that to calculate the Width and Height of the microphone array.

```
2 references
public float GetArrayHeight(List<Microphone> _microphones)
{
    GetMinandMaxY(_microphones);
    return (_arrayYMax - _arrayYMin);
}

2 references
public float GetArrayWidth(List<Microphone> _microphones)
{
    GetMinandMaxX(_microphones);
    return (_arrayXMax - _arrayXMin);
}
```

GetMinandMax checks every microphone for its position on the acoustic camera. If its X/Y is higher than the max, it will set as that max. And if its X/Y is lower than the minimum, it will set as that min.

```
public void GetMinandMaxX(List<Microphone> _microphones)
{
    //These will be used to determine the width (Max - Min)
    float highestMaxX = 0;
    float lowestMinX = 0;

    foreach (Microphone mic in _microphones)
    {
        Vector3 micLocalPosition = mic.Position;

        float currentX = micLocalPosition.x;

        if (currentX > highestMaxX)
        {
            highestMaxX = currentX;
        }
        else if (currentX < lowestMinX)
        {
            lowestMinX = currentX;
        }
    }
    _arrayXMax = highestMaxX;
    _arrayXMin = lowestMinX;
}

2 references
public void GetMinandMaxY(List<Microphone> _microphones)
{
    //These will be used to determine the Height (Max - Min)
    float highestMaxY = 0;
    float lowestMinY = 0;

    foreach (Microphone mic in _microphones)
    {
        Vector3 micLocalPosition = mic.Position;

        //Even though it's Y, Unity sees the z-axis as up and down
        float currentY = micLocalPosition.z;

        if (currentY > highestMaxY)
        {
            highestMaxY = currentY;
        }
        else if (currentY < lowestMinY)
        {
            lowestMinY = currentY;
        }
    }

    _arrayYMax = highestMaxY;
    _arrayYMin = lowestMinY;
}
```

## HOLOGRAM

The next step is to create a hologram, an invisible square with a Width and a Height where the target points will be placed.

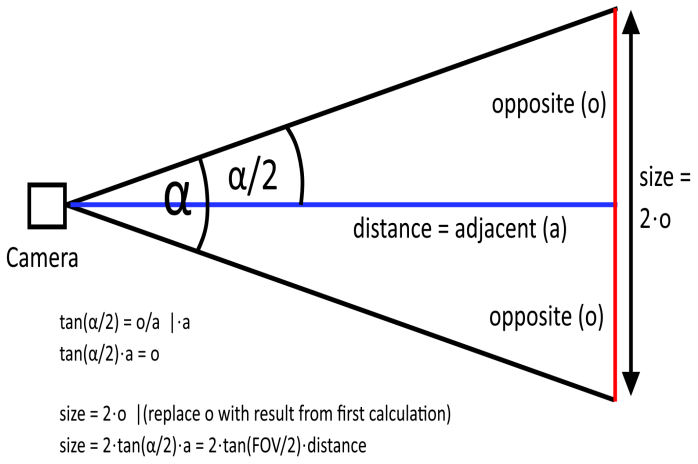
The class HologramManager is used to Calculate and Return the Hologram, but is also used to calculate the ScaleMultiplier used for the target points. HologramManager inherits from IHologramManager

**Note:** Since the algorithm uses Complex for the hologram, it will also return a Complex in ReturnHologramSize. A complex consists of a Real and an imaginary Part ( a + bi)

```
public interface IHologramManager
{
    3 references
    float CalculateHologramSize();
    2 references
    float GetScaleWidth(float _width);
    2 references
    float GetScaleHeight(float _height);
    2 references
    Complex ReturnHologramSize();
}
```

CalculateHologramSize uses the formula= size = 2 x (Tan(FOV/2)x Adjacent) to calculate both the width and height of the hologram.

```
public Complex ReturnHologramSize()
{
    _widthHologram = CalculateHologramSize();
    _heightHologram = CalculateHologramSize();
    return new Complex(_widthHologram, _heightHologram);
}

3 references
public float CalculateHologramSize()
{
    float opposite = (Mathf.Tan(FOV / 2) * Adjacent);
    return (2 * opposite);
}

2 references
public float GetScaleWidth(float _arrayWidth)
{
    return (_widthHologram / _arrayWidth);
}

2 references
public float GetScaleHeight(float _arrayHeight)
{
    return (_heightHologram / _arrayHeight);
}
```




FOV and Adjacent are Properties, as seen by the Capital First Letter, used by the logic of the CalculateHologramSize. Both Properties are using a Unity Component as a reference.

```
[SerializeField]
Camera _acousticCamera;
[SerializeField]
SphereCollider _range;
1 reference
public float FOV
{
    get { return _acousticCamera.fieldOfView; }
    private set { }
}
1 reference
public float Adjacent
{
    get { return _range.radius; }
    private set { }
}
```

## TargetPoints

The next step is to create the target points that will be used for the beamforming. The TargetPointManager is used to get and create the Target points. TargetPointManager inherits from ITargetPointManager.

```
public interface ITargetPointManager
{
    void CreateTargetPoints(List<Microphone> _mics, float scaleX, float scaleY, float range);
    void UpdateTargetPoints(float range);
    List<TargetPoint> GetTargetPoints(List<Microphone> _mics, float scaleX, float scaleY, float range);
}
```

TargetPointManager will create the list when the list is empty, otherwise it will just return the list.

```
public void CreateTargetPoints(List<Microphone> _mics, float scaleX, float scaleY, float range)
{
    foreach (Microphone mic in _mics)
    {
        TargetPoint tp = Instantiate(_targetPointPrefab, Vector3.zero, Quaternion.identity);
        tp.transform.parent = this.transform;
        tp.SetPosition(mic.Position, scaleX, scaleY, range);
        _targetPoints.Add(tp);
    }
}

public List<TargetPoint> GetTargetPoints(List<Microphone> _mics, float scaleX, float scaleY, float range)
{
    //If the list is empty, it will create the list, otherwhise it will retrieve it
    int count = _targetPoints.Count;
    if (count == 0)
    {
        CreateTargetPoints(_mics, scaleX, scaleY, range);
    }
    return _targetPoints;
}
```

TargetPoint has a method that would set its Position based on the range and the scale that was previously calculated.

```
[System.Serializable]
public class TargetPoint : MonoBehaviour
{
    [Header("Position")]
    [SerializeField]
    Vector3 _position;

    public void SetPosition(Vector3 referencePosition, float scaleX, float scaleY, float range)
    {
        //Set the coordinates of the target point
        float TpX = (referencePosition.x * scaleX);
        float TpZ = (referencePosition.z * scaleY);
        float TpY = range;

        _position = new Vector3(TpX, -TpY, TpZ);
        transform.localPosition = _position;
    }

}
```


## New Implementation: Simple Beamforming

After a discussion with Sorama about the beamforming, it was decided that the previous implementation would be too complex for now and that I should use the mock version made in Sprint 1. However, while the mock version did work, it only worked when you stand in front of the sound source, which I wasn't happy with to be used in the final version. I decided to make my own version of beamforming for this project.

It works like this:

1. Sound source is in range of the acoustic camera
2. Microphones get the amplitude, frequency, phase and distance calculated
3. Microphone with the shortest distance will be used as a location.
4. Get the size of both the microphone size and the camera's feedback plane.
5. Calculate the scale and use it on the position.
6. Set the scaled position in the heatmap and the other information in the FFT converter for a spectrum

## Beamforming Manager

A class called CurrentBeamformingManager was created, which inherits from IBeamformingManager as well. It contains methods required to get the microphones, get the closest microphone, make a heatmap and set the variables in the spectrum.

```
public class CurrentBeamformingManager : MonoBehaviour,IBeamFormingManager
{
    IMicrophoneController _microphoneController;
    IArraySizeManager _arraySizeManager;
    IScaleManager _scaleManager;
    IHeatmapManager _heatmapManager;
    [Header("Spectrum")]
    [SerializeField]
    Example01_time_to_frequency _fftConverter;

    List<Microphone> _microphones;
    [Space(5)]
    [Header("Size Of Microphone Array")]
    [SerializeField]
    float _microphoneArrayWidth;
    [SerializeField]
    float _microphoneArrayHeight;
    [Space(5)]
    [Header("Scale")]
    [SerializeField]
    float _scaleWidth;
    [SerializeField]
    float _scaleHeight;
    [Space(5)]
    [Header("Components used for Spectrum and Heatmap")]
    [SerializeField]
    SoundSource _soundSource;
    [SerializeField]
    Microphone _closestMicrohpone;
```

In the update method, it checks for the closest microphone. If, for some reason, there isn't one, it will not create a heatmap and spectrum. It also checks for a sound source and will reset the variables if one cannot be found.

```
void Update()
{
    _soundSource = _microphoneController.GetSoundSource();
    if(_soundSource != null)
    {
        SetUp();
    }
    else
    {
        ResetVariables();
    }
}

private void SetUp()
{
    GetMicrophones();
    GetClosestMicrophone();
    if (_closestMicrohpone)
    {
        //Heatmap Functionality
        MakeAHeatMap();
        //Spectrum Functionality
        SetVariablesToFFT(_closestMicrohpone.Frequency, _closestMicrohpone.Amplitude, _closestMicrohpone.Phase);
    }
}
```

The GetMicrophones will set up the microphones in the microphoneController and return those microphones to be used by the beamformingManager.

GetClosestMicrophone compares all the distances of each microphone and gets the shortest one.

```
public void GetMicrophones()
{
    _microphones = _microphoneController.GetMicrophones();
}
void GetClosestMicrophone()
{
    float closestDistanceRef = float.MaxValue;
    foreach (Microphone m in _microphones)
    {
        if (m.Distance <= closestDistanceRef)
        {
            closestDistanceRef = m.Distance;
            _closestMicrohpone = m;
        }
    }
}
```

The GetArraySize gets the height and width of the microphones array. The GetScale gets the scale of the size of the microphone and the scale of the plane used as the camera's feedback. The MakeAHeatmap uses the scale and then uses that to add the microphone position to the heatmap.

```
    #region Scale calculation
    public void GetArraySize()
    {
        _microphoneArrayHeight = _arraySizeManager.GetArrayHeight(_microphones);
        _microphoneArrayWidth = _arraySizeManager.GetArrayWidth(_microphones);
    }

    public void GetScale()
    {
        _scaleHeight = _scaleManager.GetScaleHeight(_microphoneArrayHeight);
        _scaleWidth = _scaleManager.GetScaleWidth(_microphoneArrayWidth);
    }
#endregion
    #region Heatmap Functionality
    void MakeAHeatMap()
    {
        GetArraySize();
        GetScale();
        AddMicrophonePositionToHeatmap();
    }
```

SetVariablesToFFT simply grabs the variables and put them into the fft converter that creates a spectrum for us.

```
void AddMicrophonePositionToHeatmap()
{
    _heatmapManager.AddPointToHeatmap(_closestMicrohpone, _scaleWidth, _scaleHeight);

}
#endregion
#region Spectrum Functionality
void SetVariablesToFFT(float freq, float amp, float phase)
{
    _fftConverter.freq[0] = freq;
    _fftConverter.Amp[0] = amp;
    _fftConverter.phase[0] = phase;
}
#endregion

void ResetVariables()
{
    _closestMicrohpone = null;
    _fftConverter.ResetValues();
}
```

## MicrophoneController

A class called CurrentMicrophoneController inherits from IMicrophoneController and uses two sub-managers for the logic handling, such as calculating the variables and for handling the range with the sphereCollider. InitalizeMicrophones grabs the existing microphones on the object and added them to the list.

```
public class CurrentMicrophoneController : MonoBehaviour, IMicrophoneController
{
    ILogicManager _logicManager;
    IRangeManager _rangeManager;
    [Header("The microphones on the camera")]
    [SerializeField]
    List<Microphone> _microphonesOnCamera;
    // Start is called before the first frame update
    SoundSource _soundSourceLocatedInRange;
    void Start()
    {
        _logicManager = GetComponent<ILogicManager>();
        _rangeManager = GetComponent<IRangeManager>();

        InitializeMicrophones();
    }

    public void InitializeMicrophones()
    {
        Microphone[] _micsFound = GetComponentsInChildren<Microphone>();
        if (_micsFound != null)
        {
            foreach (Microphone m in _micsFound)
            {
                _microphonesOnCamera.Add(m);
            }
        }
        else
        {
            return;
        }
```

IRangeManager contains methods to get the detected source, get the range and change the range.

```
public interface IRangeManager
{
    float GetRange();
    void ChangeRange(float range);
    SoundSource GetDetectedSoundSource();
}
```

RangeManager inherits from IRangeManager. RangeManager uses a sphereCollider as its range and uses OnTriggers (methods provided by Unity) to detect if an object gets in range or not.

```
SphereCollider _range;
public float Radius
{
    get { return _range.radius; }
    set { _range.radius = value; }
}
[SerializeField]
float referenceRange;
SoundSource _soundSourceLocatedInRange;

// Start is called before the first frame update
void Start()
{
    _range = GetComponent<SphereCollider>();
}
void Update()
{
    if(Radius != referenceRange)
    {
        ChangeRange(referenceRange);
    }
}
public void ChangeRange(float range)
{
    Radius = range;
}

public float GetRange()
{
    return referenceRange;
}
```

**For the array size, it uses the same one as seen here.**

## ScaleManager

The interface IScaleManager contains methods to get the scaleWidth and scaleHeight.

```
public interface IScaleManager
{
    float GetScaleWidth(float _width);
    float GetScaleHeight(float _height);
}
```

The class ScaleManager inherits from IScaleManager and it gets the scale of the height and width by using it to divide the height and width of the screen.

```
public class ScaleManager : MonoBehaviour, IScaleManager
{
    [SerializeField]
    GameObject screenObject;
    [SerializeField]
    float _widthScreen;
    [SerializeField]
    float _heightScreen;
    // Start is called before the first frame update
    void Start()
    {
        _widthScreen = screenObject.transform.localScale.x;
        _heightScreen = screenObject.transform.localScale.z;
    }

    public float GetScaleHeight(float _height)
    {
        return Mathf.Abs(_heightScreen / _height);
    }

    public float GetScaleWidth(float _width)
    {
        return Mathf.Abs(_widthScreen / _width);
    }

}
```

## HeatmapManager

The interface IHeatmapManager has one method that uses the microphone and scales to add a point to the heatmap.

```
public interface IHeatmapManager
{
    void AddPointToHeatmap(Microphone closest, float scaleX, float scaleY);
}
```

The class IHeatmapShaderManager inherits from IHeatmapManager. It uses the same code as explained in this document.

```
public class HeatmapShaderManager : MonoBehaviour, IHeatmapManager
{
    [Tooltip("This one must be filled with the meshrender the heatmap material has applied too")]
    [SerializeField]
    MeshRenderer meshRenderer;
    Material _material;

    float[] _points;
    int _hitCount;
    // Start is called before the first frame update
    void Start()
    {
        _material = meshRenderer.material;
        _points = new float[32 * 3];
    }
    public void AddPointToHeatmap(Microphone closest, float scaleX, float scaleY)
    {
        Vector3 pos = closest.Position;
        Vector3 scaledPos = ScalePosition(pos, scaleX, scaleY);

        ApplyPointToShader(scaledPos);
    }
```

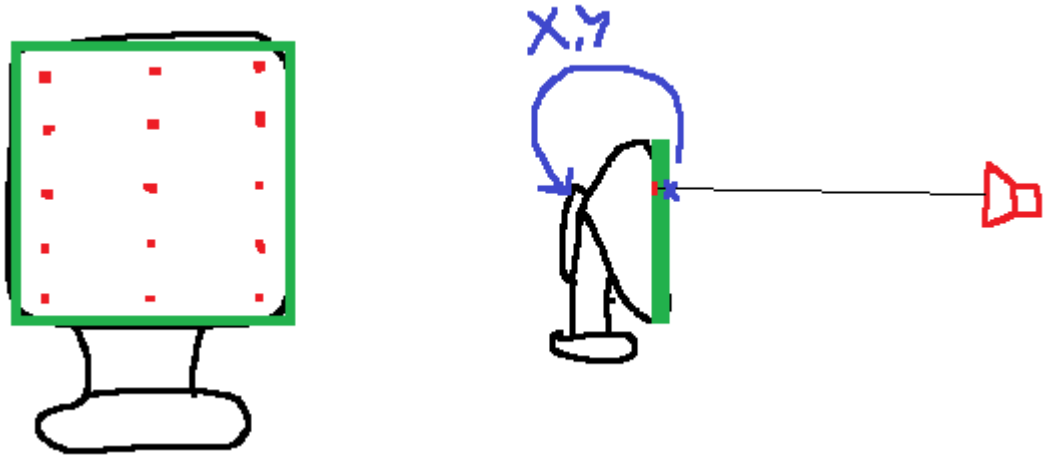
The only difference is that there is a new method called ScalePosition which scales the microphone position based on the scales.

```
public Vector3 ScalePosition(Vector3 micPosition, float scaleX, float scaleY)
{
    return new Vector3(micPosition.x * scaleX, micPosition.y, micPosition.z * scaleY);
}

void ApplyPointToShader(Vector3 position)
{
    _points[_hitCount * 3] = position.x * 4 - 2;
    _points[_hitCount * 3 + 1] = position.z * 4 - 2;
    _points[_hitCount * 3 + 2] = Random.Range(0.01f, 0.03f);


    _hitCount++;
    _hitCount %= 32;


    _material.SetFloatArray("hits", _points);
    _material.SetInt("hit", _hitCount);
}
```

## Improvement of Beamforming

During testing, it was discovered this algorithm wasn't really working, mostly due to the scales and sizes being really tiny, around the 0.01. So to change it, I decided to combine both versions of the beamforming, both this one and the mock version from Sprint 1. Here's how it works.

At the microphone array, a hidden plane stays a tiny bit in front of the microphones. When the closest microphone and sound source are found, a linecast is drawn from the sound source to that microphone. If the linecast collides with the hidden plane, it will send the texture coordinates of that hit to the beamforming manager.




The class HeatmapRaycast is created and is similar to HeatmapShaderManager. The only exception is that it does not contain the scales.

```
public void AddPointToHeatmap(Microphone closest, SoundSource soundSource)
{
    RaycastHit hit;
    if (Physics.Linecast(soundSource.transform.position, closest.transform.position, out hit))
    {
        ApplyPointToShader(new Vector3(hit.textureCoord.x, 0, hit.textureCoord.y));
    }

}
```

***After testing, some changes were done to the beamforming, those changes can be found [here](here).***