Development- Testing Environment

Introduction

For Sprint 1, the main goal was to find out what visualization methods are needed for the environment. The acoustic camera, Cam Iv64,'s main forms of visualization are heatmap, spectrum, and spectrogram. But for this project, only the heatmap and spectrum will be implemented. As an extra form of visualization, the idea came to visualize the wave that the sound represents. This environment is made to showcase the visualization and to see if it is fitting the project.

General

To start the project, we need to have a player moving around the scene. For the inputs of the player, I'm using the New Unity Input System that calls input via events.

The InputProcessor is an abstract class that other Input classes can inherit. This is to make the process more solid since the classes that handle the input will depend on this abstraction.

```
Unity Script | 2 references
public abstract class InputProcessor : MonoBehaviour
{
    public Vector2 moveInput;
    public Vector2 lookInput;

    2 references
    protected abstract void ProcessMoveInput();
    2 references
    protected abstract void ProcessLookInput();
}
```

The PlayerInput inherits Inputprocessor, so this one will be used for the player movement.

```
Unity Script (3 asset references) | 0 references
public class PlayerInput : InputProcessor
{
    private Vector2 playerInputMove;
    private Vector2 playerInputLook;

    Unity Message | 0 references
    private void Update()
    {
        ProcessMoveInput();
        ProcessLookInput();
    }
    2 references
    protected override void ProcessLookInput()
    {
        lookInput = playerInputLook;
    }

    2 references
    protected override void ProcessMoveInput()
    {
        moveInput = playerInputMove;
    }

    0 references
    public void OnMove(InputAction.CallbackContext ctx)
    {
        playerInputMove = ctx.ReadValue<Vector2>();
    }
    0 references
    public void OnLook(InputAction.CallbackContext ctx)
    {
        playerInputLook = ctx.ReadValue<Vector2>();
```

The PlayerInput is used in Playermovement, which handles both moving the player and looking around.

```
Unity Message | 0 references
void Update()
{
    ProcessInput();
    Moving();
    Looking();
}

1 reference
void ProcessInput()
{
    _moveInput = _inputProcessor.moveInput;
    _lookInput = _inputProcessor.lookInput;
}
1 reference
void Moving()
{
    Vector3 move = transform.right * _moveInput.x + transform.forward * _moveInput.y;
    _controller.Move(move * _moveSpeed * Time.deltaTime);
}

1 reference
void Looking()
{
    xRotation -= _lookInput.y;
    xRotation = Mathf.Clamp(xRotation, lookHeightMinLimit, lookHeightMaxLimit);

    _FPSCamera.localRotation = Quaternion.Euler(xRotation, 0f, 0f);
    transform.Rotate(Vector3.up, _lookInput.x);
}
```

HEATMAP

For the heatmap functionality, there are a few things that were created

- **Heatmap Shader**
- **Mock Soundsource**
- **Mock Microphone**
- **Heatmap Grid**
- **Removal of the trail**

For the heatmap shader, I've followed a simple tutorial that creates a shader that changes color upon impact from an object:

https://www.youtube.com/watch?v=Ah2rHGtOSbs

I've modified the code so that it changes color when colliding with a ray, shooting from the mock Soundsource. The shader contains certain point ranges on which the pixel's color will change.

```
float4 getHeatForPixel(float weight)
{

    if (weight <= pointRanges[0])
    {
        return colors[0];
    }
    if (weight >= pointRanges[4])
    {
        return colors[4];
    }
    for (int i = 1; 1 < 5; i++)
    {
        if (weight < pointRanges[i])
        {
            float dist_from_lower_point = weight - pointRanges[i - 1];
            float size_of_point_range = pointRanges[i] - pointRanges[i - 1];

            float ratio_over_lower_point = dist_from_lower_point / size_of_point_range;

            //now with ratio or percentage (0-1) into the point range, multiply color ranges to get color

            float4 color_range = colors[i] - colors[i - 1];

            float4 color_contribution = color_range * ratio_over_lower_point;

            float4 new_color = colors[i - 1] + color_contribution;
            return new_color;

        }
    }
```

```
fixed4 frag (v2f i) : SV_Target
{
    Initialize();
    fixed4 col = tex2D(_MainTex, i.uv);

    float2 uv = i.uv;
    uv = uv * 4.0 - float2(2.0, 2.0);  //our texture uv range is -2 to 2

    float totalWeight = 0;

    for (float i = 0.0; i < hit; i++)
    {
        float2 workPoint = float2(hits[i * 3], hits[i * 3 + 1]);
        float intensity = hits[i * 3 + 2];

        totalWeight += 0.2 * distanceSquare(uv, workPoint) * intensity * _Strength;
    }
    float4 heatmapColor = float4(getHeatForPixel(totalWeight));
    return (col * (1 - heatmapColor.a) + (heatmapColor * heatmapColor.a));
```

The Mock Soundsource will shoot a ray in the direction it is looking. When colliding with the microphone, it will send the coordinates of the texture it collided with to both the gridheatmap and the shader heatmap.

```
void CreateLineTowardsHeatmap()
{
    Vector3 sourceDirection = transform.TransformDirection(transform.forward);
    RaycastHit hit;

        bool hitMicrophone = Physics.Raycast(transform.position, sourceDirection, out hit, Mathf.Infinity, LayerMask.GetMask("Heatmap"));
        if (hitMicrophone)
        {
        //ReplaceDistance(hit.distance);
        //Grid
        microphoneHeatmap.SetValues(hit.textureCoord.x, hit.textureCoord.y);
        //Shader
        addHitPoint(hit.textureCoord.x * 4 - 2, hit.textureCoord.y * 4 - 2);
        }
        Debug.DrawRay(transform.position, sourceDirection * hit.distance, Color.green);

}
```

Since shaders can't use multidimensional arrays, Points[] are used to store three different values to the shaders: XPosition, YPosition, and Intensity.

```
public void addHitPoint(float xp, float yp)
{
    Points[HitCount * 3] = xp;
    Points[HitCount * 3 + 1] = yp;
    Points[HitCount * 3 + 2] = Random.Range(1f, 3f);


    HitCount++;
    HitCount %= 32;


    _Material.SetFloatArray("hits", Points);
    _Material.SetInt("hit", HitCount);
}
```

During development, another heatmap tutorial was found that also uses a shader.

```
uniform int _Points_Length = 0;
uniform float3 _Points [20];        // (x, y, z) =
position
uniform float2 _Properties [20];    // x = radius, y =
intensity

sampler2D _HeatTex;

half4 frag(vertOutput output) : COLOR {
    // Loops over all the points
    half h = 0;
    for (int i = 0; i < _Points_Length; i ++)
    {
        // Calculates the contribution of each point
        half di = distance(output.worldPos,
_Points[i].xyz);

        half ri = _Properties[i].x;
        half hi = 1 - saturate(di / ri);

        h += hi * _Properties[i].y;
    }

    // Converts (0-1) according to the heat texture
    h = saturate(h);
    half4 color = tex2D(_HeatTex, fixed2(h, 0.5));
    return color;
}
```

```
    public Vector3[] positions;
    public float[] radiuses;
    public float[] intensities;

    public Material material;

    void Start ()
    {
        material.SetInt("_Points_Length", positions.Length);
        for (int i = 0; i < positions.Length; i ++)
        {
            material.SetVector("_Points" + i.ToString(),
positions[i]);

            Vector2 properties = new Vector2(radiuses[i],
intensities[i]);
            material.SetVector("_Properties" + i.ToString(),
properties);
        }
    }
```

Comparing both shaders, I choose the first one since, while it wasn't as fluid as the second one, there was more control over this one, and was easy to understand.

For the Grid system of the heatmap, I followed along with this tutorial:
https://www.youtube.com/watch?v=mZzZXfySeFQ

There's an initializer class that initializes the grid and sets the values

```
[SerializeField] private GridMeshGenerator heatmapVisual;
private HeatmapGrid grid;
Unity Message | 0 references
void Start()
{
    heatmapVisual = GetComponentInChildren<GridMeshGenerator>();
    grid = new HeatmapGrid(50, 50, 0.5f, new Vector3(transform.position.x,transform.position.y));
    heatmapVisual.SetGrid(grid);
}

0 references
public HeatmapGrid GetGrid()
{
    return this.grid;
}

1 reference
public void SetValues(float x, float y)
{
    x = x * grid.GetWidth();
    y = y * grid.GetHeight();
    int gridx = Mathf.FloorToInt(x);
    int gridy = Mathf.FloorToInt(y);

    grid.AddValue(new Vector3(gridx,gridy,0),1,3,6);
}
```

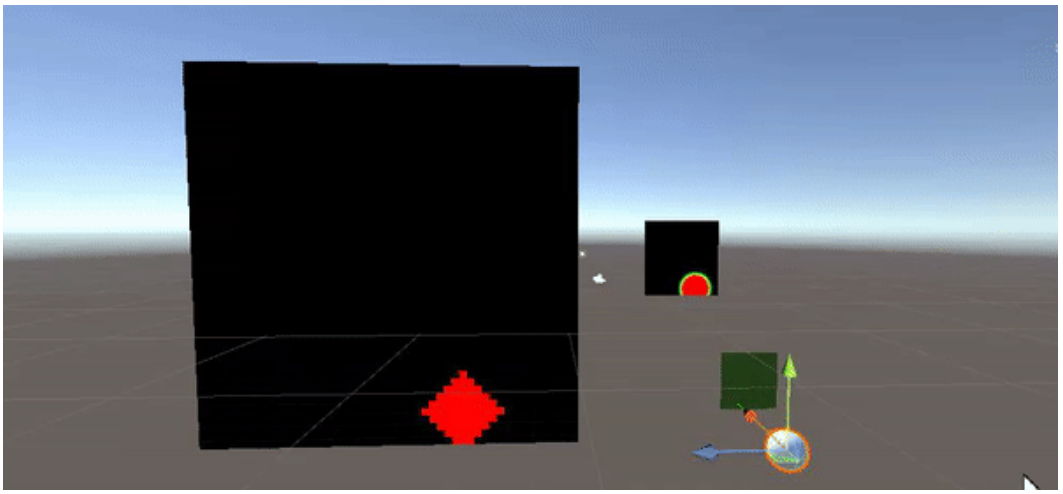
Then there is also a GridMeshGenerator that creates the mesh for the grid and updates it every time a new value gets added.

```
2 references
void UpdateHeatMapVisual()
{
    MeshUtils.CreatEmptyMeshArrays(grid.GetWidth() * grid.GetHeight(), out Vector3[] vertices, out Vector2[] uvs, out int[] triangles);

    for (int x = 0; x < grid.GetWidth(); x++)
    {
        for (int y = 0; y < grid.GetHeight(); y++)
        {
            int index = x * grid.GetHeight() + y;
            Vector3 quadSize = new Vector3(1, 1) * grid.GetCellSize();

            int gridValue = grid.GetValue(x, y);
            float gridValueNormalized = (float)gridValue / HeatmapGrid.HEAT_MAP_MAX_VALUE;
            Vector2 gridValueUV = new Vector2(gridValueNormalized, 0f);
            MeshUtils.AddToMeshArrays(vertices, uvs, triangles, index, grid.getPosition(x, y) + quadSize * .5f , 0f, quadSize, gridValueUV, gridValueUV);
        }
    }
    mesh.Clear();
    mesh.vertices = vertices;
    mesh.uv = uvs;
    mesh.triangles = triangles;
    mesh.RecalculateBounds();
}
```

Lastly, there is a grid class that adds the value to the grid and changes the color based on its value

```
public void AddValue(Vector3 worldPosition, int value, int fullValueRange, int totalRange)
{
    int lowerValueAmount = Mathf.RoundToInt((float)value / (totalRange - fullValueRange));

    GetXY(worldPosition, out int originX, out int originY);
    for (int x = 0; x < totalRange; x++)
    {
        for (int y = 0; y < totalRange - x; y++)
        {
            int radius = x + y;
            int addValueAmount = value;
            if (radius >= fullValueRange)
            {
                addValueAmount -= lowerValueAmount * (radius - fullValueRange);
            }

            AddValue(originX + x, originY + y, addValueAmount);

            if (x != 0)
            {
                AddValue(originX - x, originY + y, addValueAmount);
            }
            if (y != 0)
            {
                AddValue(originX + x, originY - y, addValueAmount);
                if (x != 0)
                {
                    AddValue(originX - x, originY - y, addValueAmount);
                }
            }
        }
    }
```

Having these scripts allows this to happen:



During testing, the conclusion was that we should use the shader instead of the grid, as the shader looks more fluidly than the grid system.

SINEWAVE

For the sinewave visualization, the following scripts were made in order to test it out.

Since both the sound source and microphone should contain X, Y, Z, Frequency, Amplitude, and Phase, An abstract class called SinewaveComponent is created and will be inherited by both classes.

```
public abstract class SinewaveComponent : MonoBehaviour
{
    public float X;
    public float Y;
    public float Z;
    protected PlayerMovement player;

    [SerializeField]
    protected float _frequency;
    [SerializeField]
    protected float _amplitude;
    [SerializeField]
    [Range(0, (2 * Mathf.PI))]
    protected float _phase;

    6 references
    public float Amplitude { get { return _amplitude; } set {_amplitude = value; } }
    6 references
    public float Frequency { get { return _frequency; } set { _frequency = value; } }
    4 references
    public float Phase { get { return _phase; } set { _phase = value; } }
    2 references
    protected virtual void Initialize()
    {
        player = FindObjectOfType<PlayerMovement>();
        this.X = this.transform.position.x;
        this.Y = this.transform.position.y;
        this.Z = this.transform.position.z;
    }
```

Both the sound source and microphone class have a method that will always look at the player. This is for the clear readability of the UI attached to the game objects.

```
Unity Script (3 asset references) | 0 references
public class SoundSource : SinewaveComponent
{
    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()
    {
        base.Initialize();
    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        LookAtPlayer();
    }

    1 reference
    void LookAtPlayer()
    {
        transform.LookAt(player.transform);
    }
```

The Microphone also contains methods that handle setting the Frequency, Amplitude, and Phase.

```
1 reference
void SetAmplitude()
{
    float amplitude = soundsource.Amplitude;
    if (_realisticAmplitude)
    {
        float distance = manager.CalculateDistanceBetweenSourceAndMicrophone(this.transform, soundsource.transform);
        float newAmplitude = manager.CalculateAmplitude(distance, amplitude);
        Amplitude = newAmplitude;
    }
    else
    {
        Amplitude = amplitude;
    }

}

1 reference
void SetFrequency()
{
    Frequency = soundsource.Frequency;
}

1 reference
void SetPhase()
{
    float distance = manager.CalculateDistanceBetweenSourceAndMicrophone(this.transform, soundsource.transform);
    float newPhase = manager.CalculatePhase(Frequency, distance);
    Phase = newPhase;
}
1 reference
void SetAttributes()
{
    SetFrequency();
    SetAmplitude();
    SetPhase();
}
```

The wave contains a class called SinewaveGenerator that draws the wave depending on the sound source's frequency, amplitude, and phase. It also contains a bool for realistic amplitude or not.

```
void DrawSineWave()
{
    float originalAmplitude = _amplitude;
    float twoPi = 2 * Mathf.PI;

    _lineRenderer.positionCount = _bufferPoints;
    for (int currentPoint = 0; currentPoint < _bufferPoints; currentPoint++)
    {
        float progress = (float)currentPoint / (_bufferPoints - 1);
        Vector3 lerpedDistance = Vector3.Lerp(beginPosition.position, endPosition.position, progress);
        float currentPosition = Vector3.Distance(beginPosition.position, lerpedDistance);

        //when enabled the amplitude will decrease over distance
        if (_realisticAmplitude)
        {
            originalAmplitude = _sinewaveManager.CalculateAmplitudeForPosition(_amplitude, currentPoint, beginPosition, _lineRenderer);
        }

        float y = (originalAmplitude * Mathf.Sin(twoPi * ((_frequency * currentPosition))+ _phase) + lerpedDistance.y);
        _lineRenderer.SetPosition(currentPoint, new Vector3(lerpedDistance.x, y, lerpedDistance.z));
        _lineRenderer.SetPosition(currentPoint, new Vector3(lerpedDistance.x, y, lerpedDistance.z));
    }
}
```

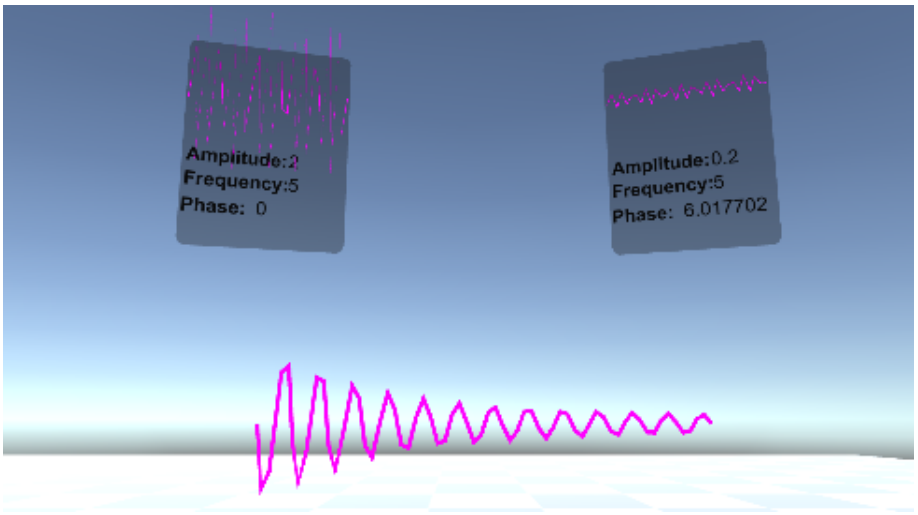
Lastly, there is the SinewaveManager that does the calculation of the attributes such as amplitude and phase.

```
1 reference
public float CalculateAmplitude(float distance, float amplitude)
{
    if (distance < 1)
    {
        return amplitude;
    }
    float factor = (1 / distance);
    float newAmplitude = amplitude * factor;
    return newAmplitude;
}

1 reference
public float CalculatePhase(float frequency, float distance)
{
    float waveLength = (2 * (Mathf.PI)) / frequency;
    float remainingDistance = (distance % waveLength);
    float phase = (((2*(Mathf.PI)) * remainingDistance)/waveLength);
    return phase;

}
```

To visualize the attributes of both the sound source and microphone, a class was created that would give the values to the UI.

```
1 reference
void GiveValuesToUI()
{
    amplitudeValueText.text = component.Amplitude.ToString();
    frequencyValueText.text = component.Frequency.ToString();
    PhaseValueText.text = component.Phase.ToString();
}
```

Having these scripts will result in the following scenario:



During Testing, the conclusion came that this form of visualization won't be used but the scripts will. Examples are the sound source and microphone scripts, but also methods that calculate the amplitude and phase.

SPECTRUM

For the spectrum functionality, I tested it out first with actual audio, since unity already had a method that would allow you to retrieve FFT samples with different algorithms.

```
void Update()
{
    spectrum = new float[512];

    _audioSource.GetSpectrumData(spectrum, 0, FFTWindow.Blackman);

    for (int i = 1; i < spectrum.Length - 1; i++)
    {
        Debug.DrawLine(new Vector3(i - 1, spectrum[i] + 10, 0), new Vector3(i, spectrum[i + 1] + 10, 0), Color.red);
        Debug.DrawLine(new Vector3(i - 1, Mathf.Log(spectrum[i - 1]) + 10, 2), new Vector3(i, Mathf.Log(spectrum[i]) + 10, 2), Color.cyan);
        Debug.DrawLine(new Vector3(Mathf.Log(i - 1), spectrum[i - 1], 1), new Vector3(Mathf.Log(i), spectrum[i], 1), Color.green);
       Debug.DrawLine(new Vector3(Mathf.Log(i - 1), Mathf.Log(spectrum[i - 1]), 3), new Vector3(Mathf.Log(i), Mathf.Log(spectrum[i]), 3), Color.blue);
    }
}
```

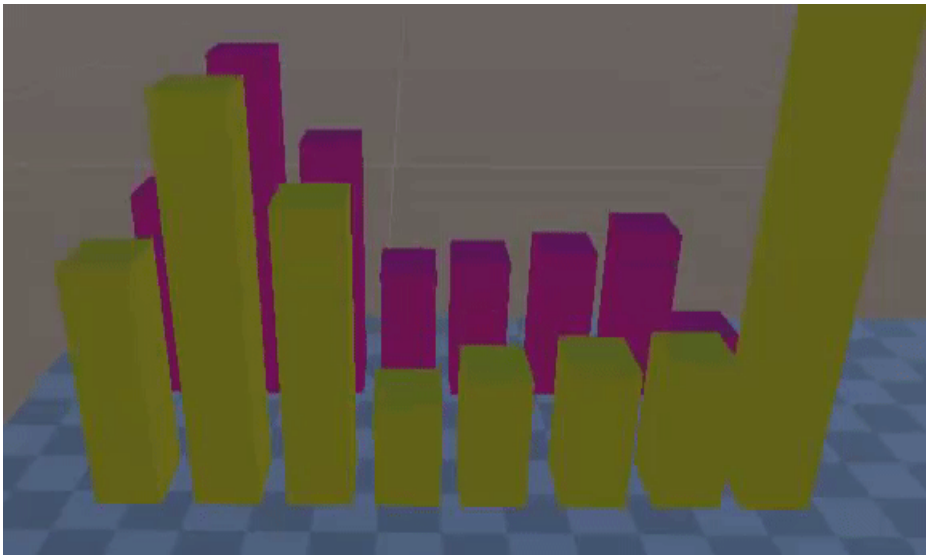
For one of the visualizations, I follow this tutorial: https://www.youtube.com/watch?v=Ri1uNPNlaVs and to have blocks function as the spectrum data.

```
void Update()
{
    if(_useBuffer)
    {
        transform.localScale = new Vector3(transform.localScale.x, (AudioPeer.bandBuffer[band] * scaleMultiplier) + scaleMultiplier, transform.localScale.z);
    }
    if (!_useBuffer)
    {
        transform.localScale = new Vector3(transform.localScale.x, (AudioPeer.freqBand[band] * scaleMultiplier) + scaleMultiplier, transform.localScale.z);
    }
}
```

The sound source contains the Audiopeer Class that gets the spectrum data and groups them by frequency bands. It also adds a band buffer to it.

```
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++)
        {
            average += samples[count] * (count + 1);
            count++;
        }

        average /= count;

        freqBand[i] = average * 10;
    }
}
```

```
void BandBuffer()
{
    for (int i = 0; i < 8; i++)
    {
        if(freqBand[i] > bandBuffer[i])
        {
            bandBuffer[i] = freqBand[i];
            bufferDecrease[i] = 0.005f;
        }
        if (freqBand[i] < bandBuffer[i])
        {
            bandBuffer[i] -= bufferDecrease[i];
            bufferDecrease[i] *= 1.2f;
        }
    }
}
```

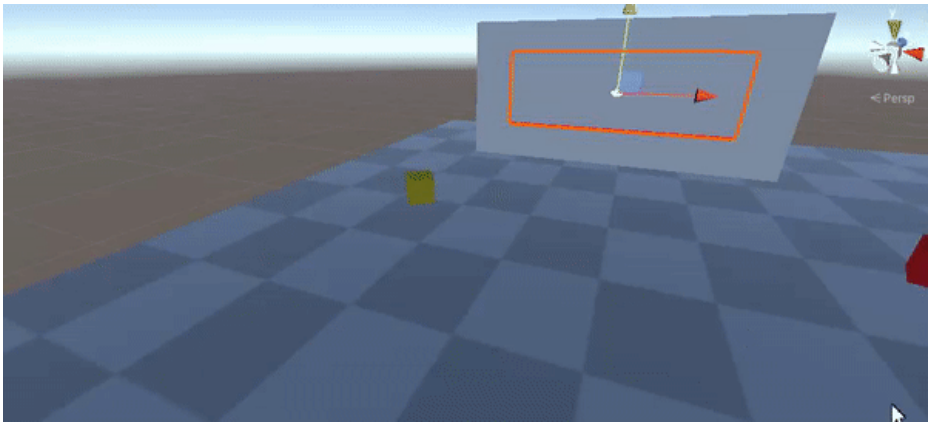


The yellow one does not use a band buffer and the pink one does.

After that, I began looking for making a spectrum using script-written frequency, and I found an asset called FFT Fast Fourier Transform: https://assetstore.unity.com/packages/tools/audio/fft-fast-fourrier-transform-152492 which does exactly what needs to be done to create a spectrum.

```
//perform complex opterations and set up the arrays
Complex[] inputSignal_Time = new Complex[windowSize];
Complex[] outputSignal_Freq = new Complex[windowSize];


inputSignal_Time = FastFourierTransform.doubleToComplex(Y_inputValues);


//result is the iutput values once DFT has been applied
outputSignal_Freq = FastFourierTransform.FFT(inputSignal_Time,false);



Y_output = new double[windowSize];
//get module of complex number
for (int ii = 0; ii < windowSize; ii++)
{
    //Debug.Log(ii);
    Y_output[ii] = (double)Complex.Abs(outputSignal_Freq[ii]);
}
```

Having these scripts will result in the following scenario:



During Testing, the conclusion came that for visualization we should use both the visualization on the acoustic camera, but also on a separate plane for more details.