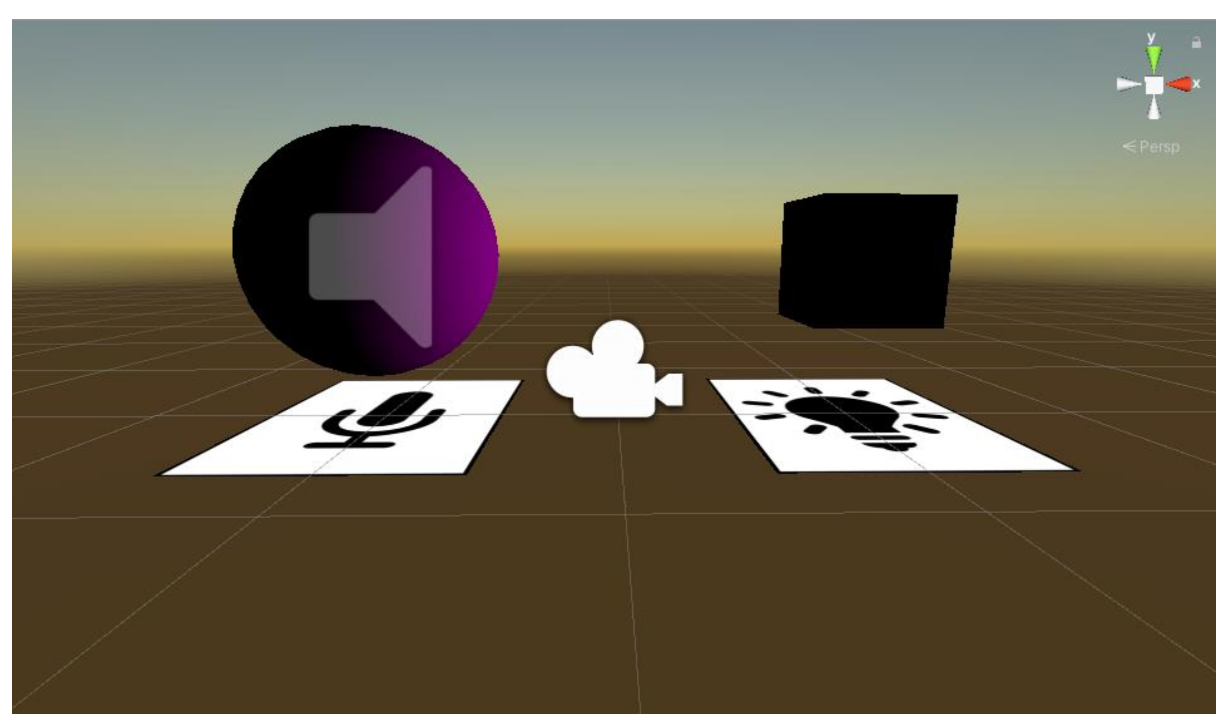
Ball of sound

Bas van Rossem
ARD
2020/06/02



Contents

Game setup....................2
Scene objects....................2
ARCamera....................2
ImageTarget....................2
Responder....................2
Lamp....................2
Sensing....................3
Microphone....................3
1. Read the microphone source....................3
2. Get the average loudness of the clip....................4
3. Get the most intense pitch....................4
4. Apply a sigmoid function to the values and scale....................5
5. Apply results to the material....................6
Light....................7
Projecting to reality....................8
1. Creating markers....................8
2. Creating marker database....................8
3. Loading database into Unity....................8
4. Adding objects to markers....................8
Shader....................9
Properties....................9
Texture....................9
Volume....................9
Pitch....................9
Light Point....................9
Position....................10
Color and light....................11
Vertex....................11
Fragment....................11

Game setup

The game consists of a single unity scene with several objects.

Scene objects

There are only a few objects needed to create the application. The structure of the objects can be seen in Figure 1.

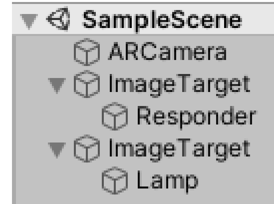


Figure 1 - Scene objects

ARCamera

This is the camera that is used by Vuforia. It takes the camera image and projects this behind all the objects in the scene.

This camera is used instead of the default camera.

ImageTarget

The image target objects are the markers that are used in the scene. They have the marker images as image target. These objects are tracked by the Vuforia engine and the position of these markers in the scene control the world position of their child elements.

Responder

This object is a ball that will ultimately respond to the sound.

Besides its default attributes it has a script to get the microphone input (*MicrophoneInput.cs*), a material to render the new vertices, colors and light (*spikes*), and a script that updates the light position in the material (*LightPosition.cs*).

Lamp

The lamp object is to show where the light source is that is used in the responders *Lightpositon* script.

Sensing

Microphone

The main way of interacting with the program is to make use of the microphone. The general idea is to make the object of choice red or blue, depending on a low or high pitch, and to give the ball a “spikiness” depending on the volume.

The microphone input is handled by the *MicrophoneInput* script. The script starts by getting the audio source, restarting a timer that is needed in a later function, and muting the output so that does not output the microphone input. After the start function, the script has a few steps that it goes through every update:

1. Read the microphone source

This is done in the *MicrophoneIntoAudioSource* function.

```
void MicrophoneIntoAudioSource()
{
    if (Time.time - timeSinceRestart > 0.5f &&
        !Microphone.IsRecording(null))
    {
        src.clip = Microphone.Start(null,
                                    true,
                                    300,
                                    AudioSettings.outputSampleRate);

        while (!(Microphone.GetPosition(null) > 0))
        {
        }

        src.Play();
    }
}
```

The function does the following:

1. It checks if enough time has passed since the last time the function was called, to reduce lag from the input.
2. It starts the microphone to record.
3. It waits until it has some samples to play. If this is not done, there will sound problems as Unity is trying to play audio that is not really recorded. The number of samples that it waits for is at this moment 0, so that samples are played immediately after receiving them.

   Note: this step can also be used to introduce latency into the recording. If half of the sample rate is given as a value to wait for, the latency will be half a second. For example, with a microphone that records at 44100 Hz, half a second would be 22050 samples.

4. It will then play the audio so that it can be used in later functions. It is possible to play, even when the output volume of the mixer has been set to -80 Db, because the output is only output, the reduction in volume has no effect on the audio clip.

2. Get the average loudness of the clip

This is done in the *GetAverageLoudness* function.

```
public float GetAverageLoudness()
{
    float[] clipSampleData = new float[sampleDataLength];

    src.clip.GetData(clipSampleData, src.timeSamples);

    float clipLoudness = 0f;
    foreach (var sample in clipSampleData)
    {
        clipLoudness += Mathf.Abs(sample);
    }

    clipLoudness /= sampleDataLength;

    return clipLoudness;
}
```

The function does the following:

1. Create an array in which to store the clip data.
2. Get the data from the audio clip.
3. Using a foreach loop, add the absolute values of the samples. This must be the absolute value because the sample data range from -1.0 to 1.0. Without making the sample values absolute, the average result will be 0.
4. Average out the sample values
5. Return the average clip loudness.

3. Get the most intense pitch

This is done in the *GetMostIntensePitch* function.

```
public float GetMostIntensePitch()
{
    float[] clipSampleData = new float[sampleDataLength];

    src.GetSpectrumData(clipSampleData, 0, FFTWindow.Hamming);

    int highest_index = 0;
    for (int i = 0; i < sampleDataLength; ++i)
    {
        if (clipSampleData[i] > clipSampleData[highest_index])
        {
            highest_index = i;
        }
    }

    return highest_index;
}
```

The function does the following:

1. Create an array in which to store the sample data. The length of this array must be a power of 2. The length used is 2048, because of the high sample frequency of the microphone that is used.

2. Get the spectrum data from the source. In this case the Hamming method is used. This is because it gives values that are good enough for what it needs to do, and because the samples are calculated with a relatively simple calculation:

$$W[n] = 0.54 - \left(0.46 * \cos\left(\frac{n}{N}\right)\right)$$

Equation 1 - FFT Hamming (source: Unity)

3. With this data, the next step is to find which of the pitches in the audio clip has the highest intensity. This is done by using a simple for loop, comparing each data point with the established highest intensity.
4. The index of the item with the highest intensity is returned.

4. Apply a sigmoid function to the values and scale

The loudness and the index of the highest pitch have been calculated, but these values are way too high or too low. To solve this, the *sigmoid* function is applied and some scaling is done, this will put the values to a value between 0 and 1.

```
float new_vol = (sigmoid(GetAverageLoudness() * 25) - 0.5f) * 2;
float new_pit = (sigmoid(GetMostIntensePitch() / 100) - 0.5f) * 2;
```

This is what each of the lines of code do:

1. Scale the initial values of the loudness and pitch. The average loudness is a number between 0.0 and 1.0, so when that number is put through the sigmoid function, the result will most certainly be close to 0. Multiply the initial value with 25, this results in a number that can be used in the shader functions.

   The intensity pitch is a number between 0 and 2048 (the index of the array of samples), so this will almost most likely be a very high number. Divide the number by 100 to get a useful result from the sigmoid function.

2. Put the value of the loudness and pitch into a sigmoid function. A sigmoid function brings very low negative numbers to 0 and very high positive numbers to 1. The calculation is as follows:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Equation 2 - Logistic sigmoid function

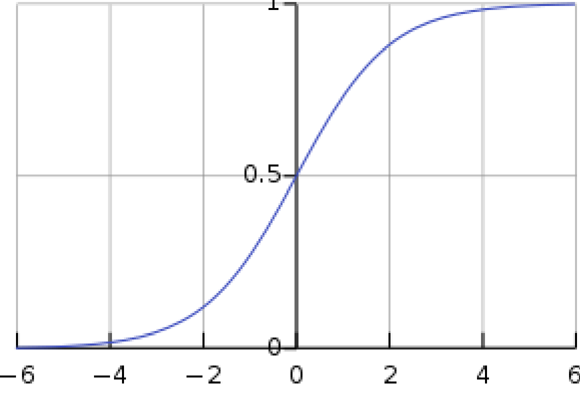


Figure 2 - The logistic curve (source: Wikipedia)

The sigmoid function is represented in the code as follows in code as follows:

```
private float sigmoid(float x)
{
    return 1 / (1 + (float)Math.Exp(-x));
}
```

3. Because the numbers have at least the value of 0, Figure 2 shows that the values that are returned will always be at least 0.5. The goal is to have a number that is between 0.0 and 1.0. To do this, 0.5 is subtracted from the sigmoid result.
4. Now the numbers are between 0.0 and 0.5. Multiply this result by 2 and the result will be between 0.0 and 0.1.

5. Apply results to the material

Finally, the calculated results are set in the material, so that the connected shader can use them.

```
spiky.SetFloat("_AudioVolume", new_vol);
spiky.SetFloat("_AudioPitch", new_pit);
```

Light

To calculate the light for the shader a position for a light source is needed. This is controlled by the *LightPosition* script.

This script has only one function, the update function.

```
void Update()
{
    Vector4 change = new Vector4(
        lamp.transform.position.x,
        lamp.transform.position.y,
        lamp.transform.position.z,
        0);

    spiky.SetVector("_LightPoint", change);
}
```

This function checks the position of the lamp object and sets this position as the *_LightPoint* variable of the material.

Projecting to reality

To make the final connection with the real world, marker-based tracking is used. It is implemented using the Vuforia engine of unity. Because there is almost no code, this part is kept brief.

1. Creating markers

To get the best result with using Vuforia, you need non-organic, non-symmetrical markers. Markers have been created for both objects in the game: a microphone for the ball object and a light for the light object.

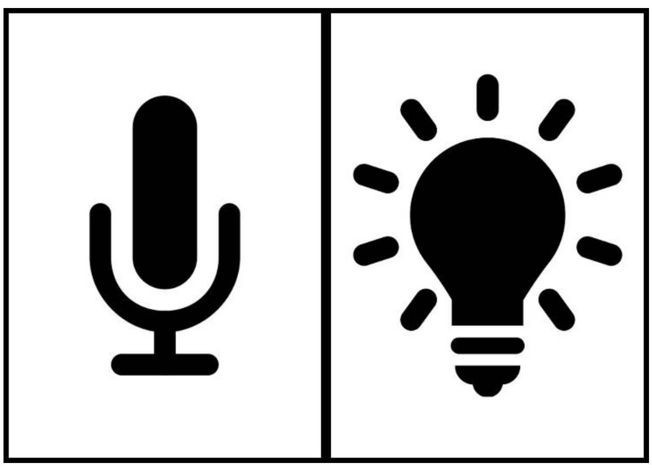


Figure 3 - Markers used in game

2. Creating marker database

To use the images, two methods could be used: load them as images into the game or make use of a database. The decision was made to use a database that can be made on the Vuforia website[1]. This is done because if it is ever decided to change the markers, the only change would be to update the database.

3. Loading database into Unity

The database was downloaded and installed it into Unity. After that, the images could be added as markers into the scene.

4. Adding objects to markers

Adding the objects to the markers is as easy as can be. Simply make the objects a child of the marker and it is done.

[1] https://developer.vuforia.com/target-manager

Shader

The *spikes* shader is applied to the *spikes* material. This shader handles both the vertex and the fragment calculations needed to create the "angry ball" effect.

Properties

To create the effect, the shader needs several properties:

- A texture
- The volume
- The pitch
- A light point

```
Properties
{
    _NoiseTex("Noise", 2D) = "white" {}
    _AudioVolume("Volume", float) = 0
    _AudioPitch("Pitch", float) = 0

    _LightPoint("Light Point Position", Vector) = (0,0,0,0)
}
```

Texture

The texture is a Perlin noise texture, this noise texture is used to get random numbers that do not differ too much from one another.

Volume

The volume is also needed in the calculation of the spikiness of the ball.

Pitch

The pitch is used to determine the color of the vertex.

Light Point

The light point is a point in the world that is used to calculate the diffused light for each fragment.

Position

The calculation of the new position of the vertex is done in the *vert* function.

```
void vert(
    in float4 vertexPos : POSITION,
    ...
    out float4 pos : SV_POSITION)
{
    // Position
    float4 pixel_location = float4(
        vertexPos.x + (_Time[1] / 5),
        vertexPos.y + (_Time[1] / 5),
        0,
        0);
    float random_number = tex2Dlod(_NoiseTex, pixel_location).r *
                          _AudioVolume;

    pos = UnityObjectToClipPos(vertexPos  + (random_number * vertexPos));

    ...
    return;
}
```

The function does the following:

1. Get a pixel location to retrieve from the Perlin noise texture. This is done by taking the x and y positions of the vertex and adding time to that value. This will create the effect of movement in the end.
2. The tex2Dlod function gets the pixel value of a specified location on the specified texture. In this case the r pixel value at the pixel location is used. Only the r value is needed, because the Perlin noise texture is in black and white. This means that the r, the g, and the b values are the same in a pixel.
3. To get the final random value, the pixel value is multiplied with the audio volume.
4. To calculate the new position, the random number (r in Equation 3) is multiplied with the current position and the current position is added to this multiplication.

   By multiplying a relative offset is calculated, and by adding that relative offset to the current position the desired new position is calculated.

$$pos_{new} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} + \left(r * \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \right) = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} + \begin{bmatrix} rx \\ ry \\ rz \\ rw \end{bmatrix} = \begin{bmatrix} x + rx \\ y + ry \\ z + rz \\ w + rw \end{bmatrix}$$

Equation 3 - Calculation of new vertex position

5. The last step is to send this new position to the camera, so the camera can render the vertex using this new position.

Color and light

To calculate what color the pixel should be that is eventually rendered on the screen, several things are needed: the color of the pixel on the object, and what the lighting will do to the pixel color.

Vertex

Besides the position of the vertex, some other information is needed in the fragment shader. This information is outputted in the vertex shader. The vertex shader also does some incredibly simple color calculations to change the color of the vector.

```
void vert(
    in float4 vertexPos : POSITION,
    in float4 normal : NORMAL,
    ...
    out float3 worldNormal : TEXCOORD1,
    out float3 worldPosition : TEXCOORD2,
    out float4 col : TEXCOORD0)
{
    worldNormal = UnityObjectToWorldNormal(normal);

    worldPosition = mul(unity_ObjectToWorld, vertexPos);

    float blue = 0.0 + _AudioPitch;
    float red = 1.0 - _AudioPitch;

    col = float4(red, 0, blue, 1);

    return;
}
```

Several things happen:

1. The normal of the vertex is transformed into a world normal. This is done so that the orientation of the object does not matter when calculating the lighting.
2. Next, the position of the vertex in the world space needed, so that it is possible to calculate the difference in position between the fragment and the light point when calculating the lighting.
3. Some color calculations, so that when pitch is higher, the color is bluer, and when the pitch is lower, it is redder.

Fragment

The last step in the process is the calculation of the pixel color in combination with the light.

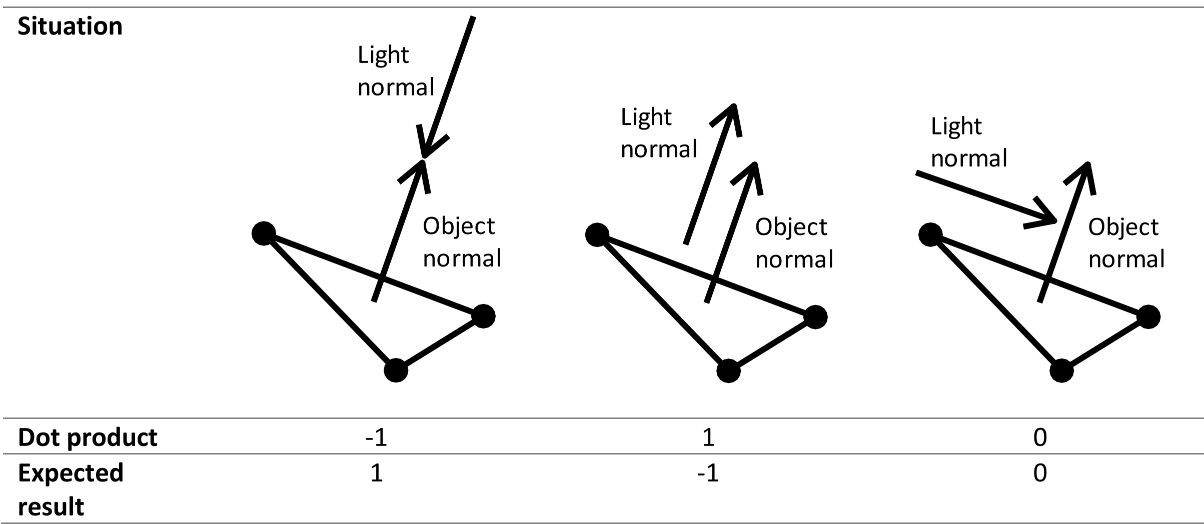
```
float4 frag(
    in float4 pos : SV_POSITION,
    in float3 worldNormal : TEXCOORD1,
    in float3 worldPosition : TEXCOORD2,
    in float4 col : TEXCOORD0) : COLOR
{
    fixed3 lightDifference = worldPosition - _LightPoint.xyz;
    fixed3 lightDirection = normalize(lightDifference);
    fixed intensity = -1 * dot(lightDirection, worldNormal);
    col = fixed4(intensity, intensity, intensity, 1) * col;
    return col;
}
```

The function works as follows:

1. The difference in location of the light point and the pixel is calculated. The swizzle operator (*.xyz*) is used to get the x, y, and z values. This turns the light point values into a 3d vector. This is used to determine the relative position of the light to the pixel.
2. The difference is normalized to get the real direction of the pixel in a vector. The vector now has a length of 1. This will make it possible to calculate the difference in angle.
3. The dot product is applied to the two normalized vectors. When the dot is applied product on two normalized vectors, the result will be a value between -1 and 1. When the two vectors are pointing towards each other, the dot will be -1. If they point in the same direction, the dot product will return -1. And if they are perpendicular, the dot product returns 0. See Table 1 for a visual representation.

   In other words: the dot product of two normalized vectors will be equal to the cosine of the two vectors. And thus, the angle between the two.

   When a light shines directly onto out pixel, it means that the normal vectors point towards each other. This will result in a dot product of -1. However, because the color is multiplied by this intensity, it needs to be as bright as can be in this situation, so the value is flipped. This way, when the light shines upon the object, the color will not be removed.



Situation			
Dot product	-1	1	0
Expected result	1	-1	0

Table 1 - Normal directions and dot product

4. The color is multiplied with the intensity, with a 1 for the alpha. And the final color is calculated.
5. The final color is returned so it can be rendered on the screen.