AR Shadowing Project

Project Team

Almog Brand

Dani Ginsberg

Lior Wandel

Project Leaders

Yaron Honen

Boaz Sternfeld

Boris Van-Sosin

Technion institute of technology

GIP lab

1. Hardware

1.1. Introduction

Augmented reality shadows are un-realistic compared to the shadows of “real world” objects created by the “real world” light module.

In Order to build a light model that we can deploy to the AR system, we first needed a way to measure the room’s light.

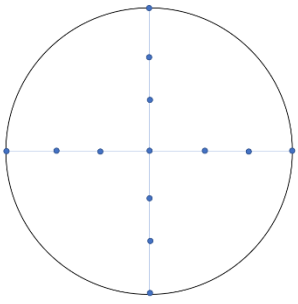
We came up with two main options –

- Wide lens camera that will take a picture of the room, then running an image processing algorithm to find areas that are suspected to be a light source.
- A dome shaped device with photoresistors on its arcs that will sample the light, then send the data to an algorithm that will build a light model.

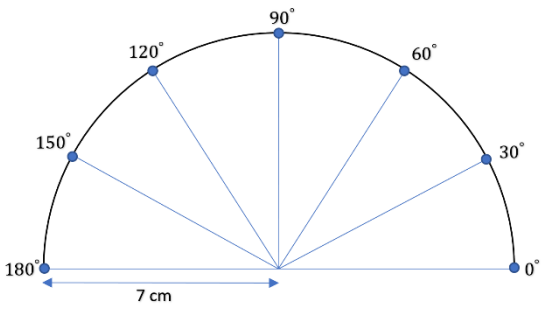
After we learned the basics of both fields – image processing and hardware design, we decided to go with the dome shaped device as we thought it will be more accurate in the positioning of the source of light.

In order to decide if the light is coming from the ceiling or the walls, we planed the device with two verticals, half-cycled arcs. Each arc has seven photoresistors on it, positioned in $0^{\circ}, 30^{\circ}, 60^{\circ}, 90^{\circ}, 120^{\circ}, 150^{\circ}, 180^{\circ}$ so that the photoresistor on 90° is shared for both arcs.

We decided to use HTC Vive’s tracker to get the coordinates and rotation of the device any time we take samples. We set the diameter of each arc to 14 cm, so the distance between the center of the dome to each photoresistor is 7 cm. This is enough space for the HTC tracker to be attached in the middle of the dome.



Top view



Side view

1.2. Hardware 1.0

1.2.1. Notes

We made some tests in order to decide how our device should be built, we divided the work to a few stages –

Stage 1: fitting a resistor that will support a full analog range. We built a simple cycle to test the best resistors that fit our purpose – getting the maximum range of analog data from the Arduino. The result was 1K-ohm resistors that got us the full range Arduino-UNO supports – 0-1023.

Stage 2: "noise avoidance" – A number of samples are needed so the result won't be affected by a glitch. Although we didn't run into a wrong sample in our tests, we coded the device so that it will sample each resistor 10 times and return their average.

Stage 3: "hardware gap" – A code fix that covers for a hardware difference between sensors. We tested 13 identical photoresistors, positioned in the same direction, under a wide light source. The result showed a small difference in the sample, between 1-3 units (in a scale of 1024)

In order to fix this "gap" we added a calibration mode that saves two values for each photoresistor –

- Dark value that we sampled in the darkness
- Bright value that we sampled with a strong close light source facing the device

Then, we show each photoresistor's private range, from dark value until bright value, on a 0-100 scale.

We repeated the test again, this time all photoresistors showed the same value.

Stage 4: We had to cover each photoresistor with a shrink in order to block light

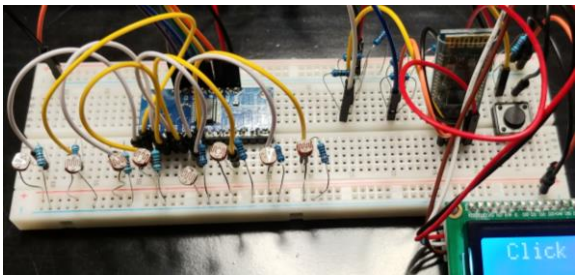
that hits the side of the photoresistor. Otherwise, a weak light from the side of the dome was interrupting the sample.

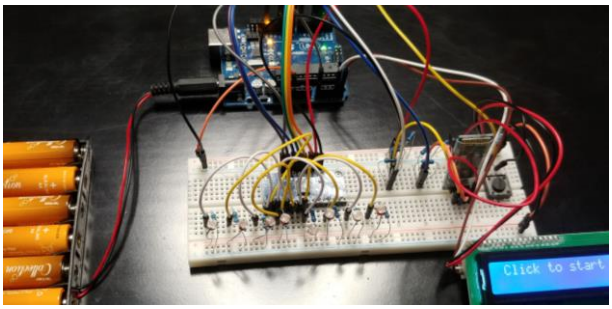
1.2.2. Parts list

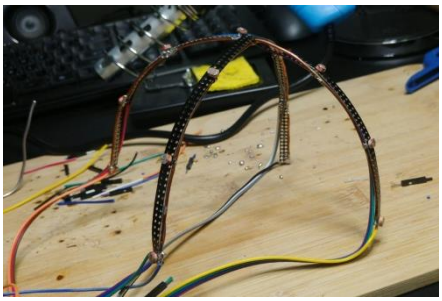
Parts for hardware 1.0

- Arduino UNO
- Battery pack
- Photoresistors X13
- 1k ohm resistor X13
- HTC-06 Bluetooth
- MUX
- Button
- 2X16 LCD screen

1.2.3. Scheme







1.3. Hardware 2.0

1.3.1. Notes

After we built the first prototype, we could measure lights and start developing our algorithm. Doing that, we experienced the lack of user interface of our device – there was no option to retake samples, we needed to run a full sample just to get the value of each photoresistor, there were to many cables and resistors etc.

We decided to focus on both algorithm developing along with making a new, better, version of the hardware.

The Arduino-UNO was replaced with the Olimex ESP32-Poe-ISO - A faster and smaller developing card, with a built-in WiFi, Bluetooth, ethernet and SD card slot. The ESP32 also has a wider analog range 0-4095, which gives us better accuracy.

The battery pack was replaced with a small Lipo battery.

The simple 2X16 LCD screen was replaced with a big, colorful, touch LCD screen –which allows us to build a user interface with a flexible menu – retake or accept samples.

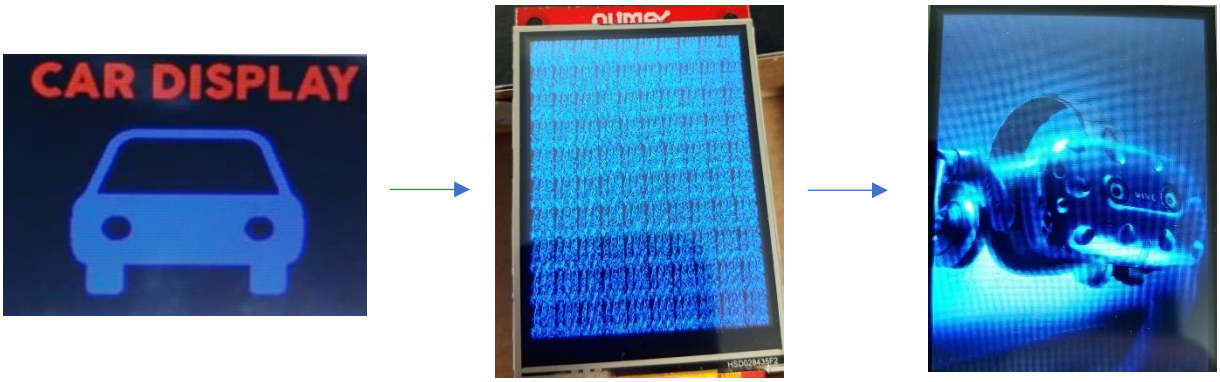
The photoresistors were replaced with better isolated photoresistors so we don't need to cover them with shrinks in order to block a light that hits their side.

For the calibration we added two features – the first is an option to skip calibration and use a default value of dark and bright values. The second is a unique way of bright sampling – a counter of 10 seconds that allows us to light every spot of the dome for the calibiration.

We added a demo mode – a graphical display of the device with its 13 photoresistors, each is represented with a small circle that contains a numeral value between 0-100 according to its sample reading. The value is also shown as a fill color of the cycle in greyscale, black for 0 value and white for 100 value.

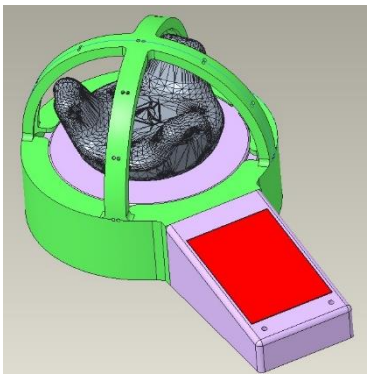
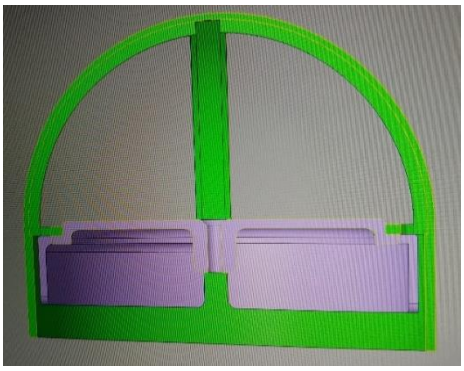
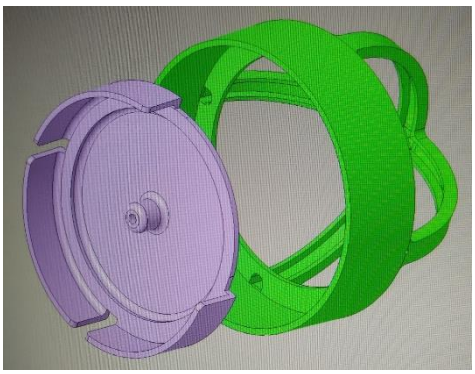
We ran into some Issues on the way:

- RGB color fixing – when we tried to upload a background picture to our screen, we found out there are only a few colors available in the screen's code package. We then found out that each color is represented in 16bit instead of full 24 bit (one byte for each red-green-blue color). By adding our own function that fixes the missing range for each color, we were able to display a full RGB image on the screen.



- Bitmap loading speed – displaying the background image pixel by pixel was a slow and visible process. We replaced this with loading the full image at once – a process that consumes a lot of memory from the developing card. We finally loaded the bitmap on an SD card, and loaded the picture in two stages – first half and second one. This solution was fast so the user can't notice the uploading process and it saves memory as the bitmap is stored on an external SD card.

In order to make the arcs steady, we designed a 3D sketch using Pro Engineer (aka Crio). We also positioned the LCD screen in between the arcs, so its light won't affect the samples, this, of course together with making the screen as dark as possible while taking a sample.

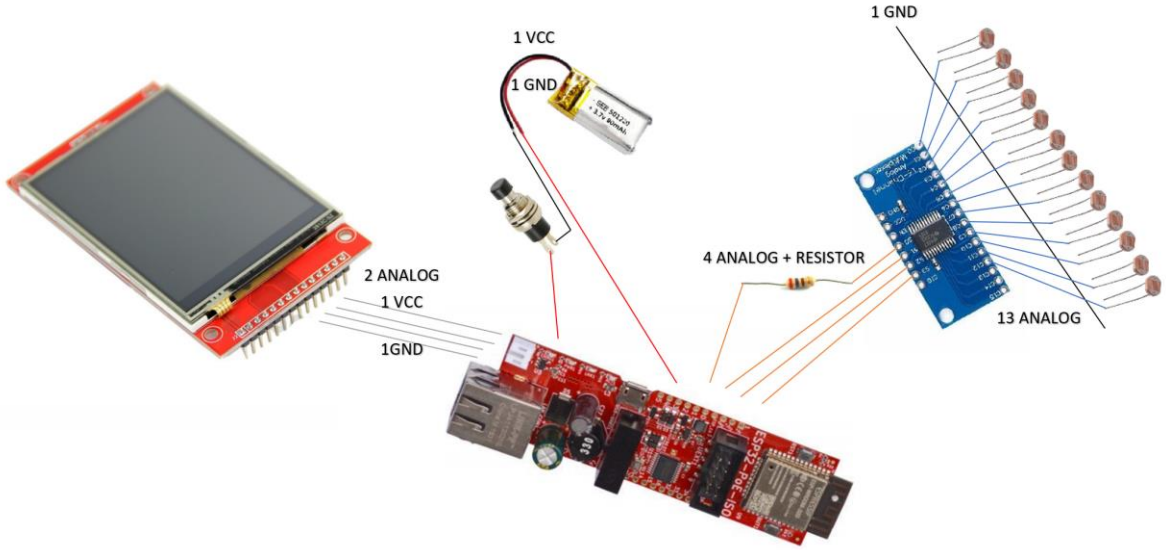




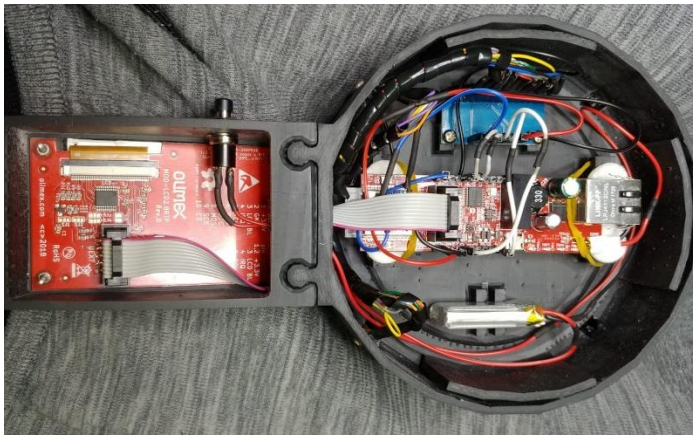
1.3.2. Parts list

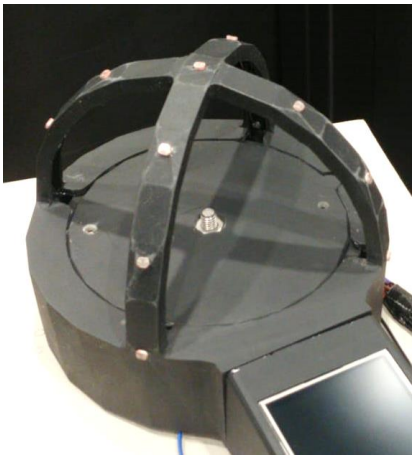
Parts for hardware 2.0

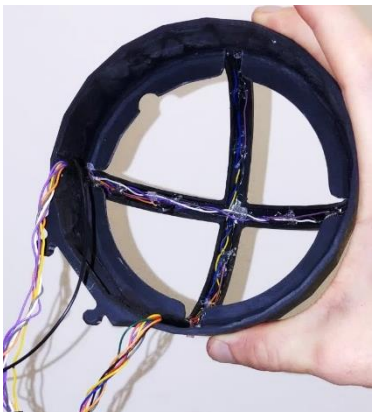
- Olimex ESP32-Poe-ISO
- Lipo Battery
- Electricity switch
- 1k ohm resistor
- Photoresistors X13
- MUX
- LCD touch screen

1.3.3. Scheme









1.4. Data Transfer flow

The data is transferred via Bluetooth from the device to a computer with a Bluetooth dongle. After each sample the device sends 14 values to the Bluetooth serial – first value is for communication protocol, if its 253 (a new sample), 254 (accept a sample), 255 (Done) and the rest 13 are for sample, each representing a number between 0 – 100 for each of the 13 photoresistors input.

The moment the data is received, we sample the HTC tracker and save the coordinates and rotation of the device the way it is located during the current sample. That way, if the user chooses to retake the sample, the tracker's data will be updated as well.

PosLog.txt

```
1  0.45 -2.11 9.06
```

When the user clicks the Done button the 255 signal is sent and the software closes two output files – first with the samples data, each line represents a sample with 13 values. Second is for tracker's data for each sample, each line shows the (x, y, rotation) of the tracker according to the sample in the same line in the other file.

Position.txt

```
1  0.45 -2.11 9.06
2  0.47 -2.13 332.17
```

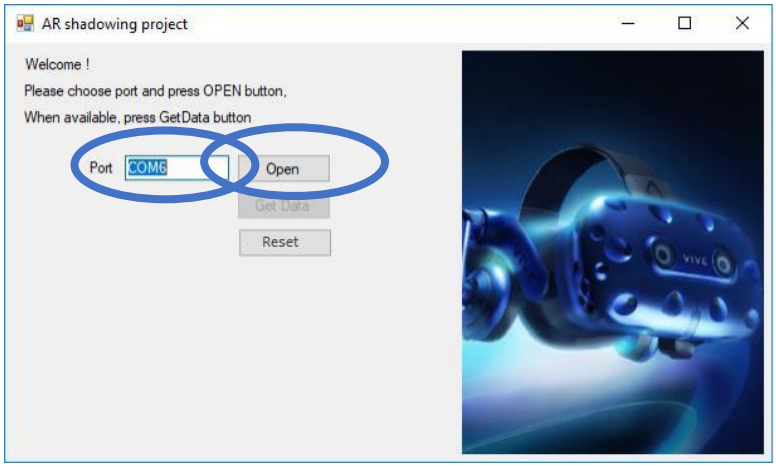
Input.txt

```
1  0 0 5 0 1 0 0 15 6 4 0 0 0
2  4 6 12 0 0 0 0 3 0 0 0 0 0
```

1.5. Software

We built a software that runs the entire process –

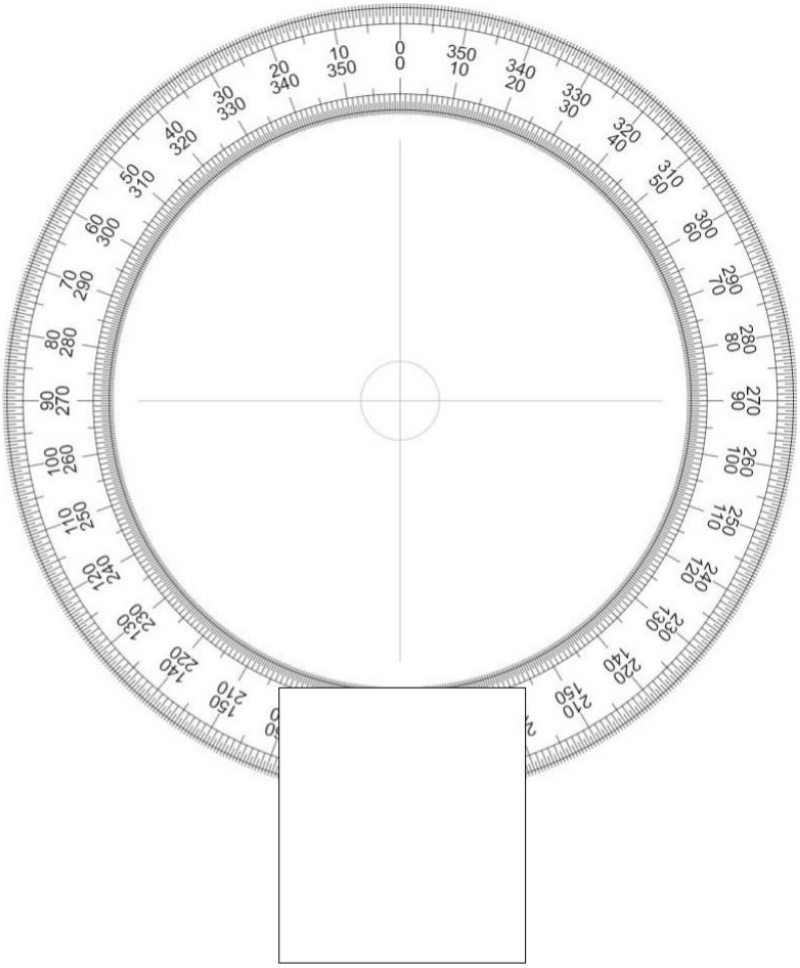
1. The user chooses the computer Bluetooth serial port and click "Open"



2. A failure will pop up an error message "cannot open port" that indicates the Bluetooth connection failed, the user needs to make sure the port number is correct.
3. A success will activate the "Get Data" button, clicking on it will set the software ready to get the data from the device.
4. When the user clicks "Done" on the device, the software will prepare the files and then run our algorithm. The output will be a Visual Studio Project with a light module that matches the real world. (Detailed in part 2. The Fusion Algorithm)
5. Clicking the "Reset" button will clear saved date and prepare the environment to a new light model building.

1.6. Testing

In Order to test the device with the algorithm we designed a test sheet to be placed under the device. This helped us testing the light from known and precise angles so we could match them to the algorithm's output.



2. The Fusion Algorithm

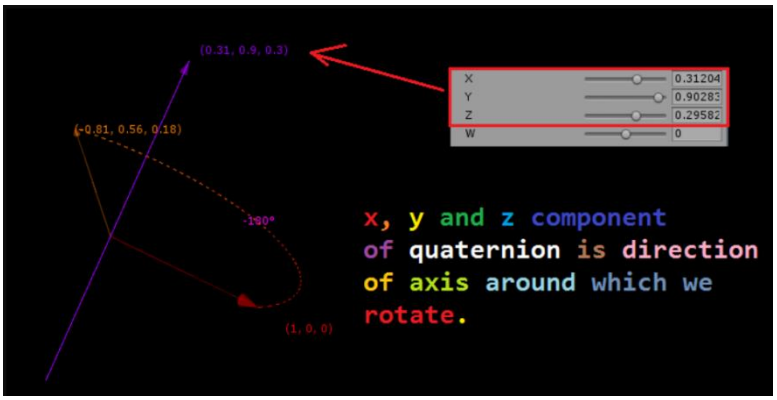
2.1. Introduction

The Fusion Algorithm is the linking part between the receptions made by the hardware and the light source formed by Unity.

2.1.1. HTC VIVE Tracker

In order to consider the hardware's position in space, we used the HTC VIVE Tracker and its self-location data in order to fit each measurement to the same initial axis system aligned to the first sample taken.

The tracker position is given as a quaternion [x, y, z, w] - a complex number with 'w' as the real part and x, y, z as imaginary parts.



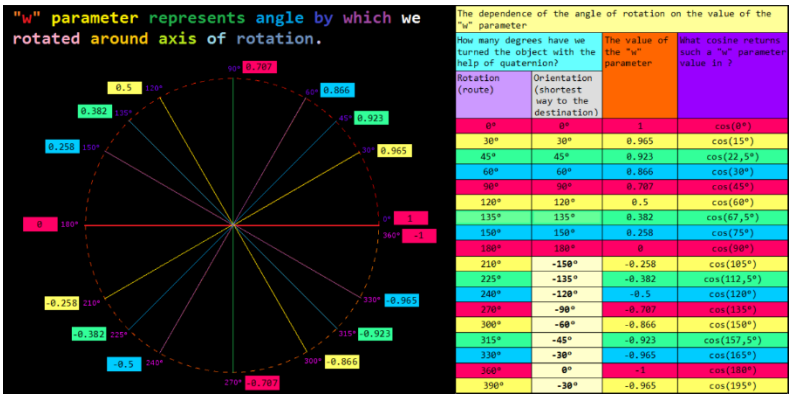


figure 1[1]

In order to extract the rotation details in degrees (the basic format is given in Euler angles) we used conversion function as part of the script running in the HTC Unity environment.

[1] figure 2-1- https://answers.unity.com/questions/147712/what-is-affected-by-the-w-in-quaternionxyzw.html

According to the way we designed the device, the Roll and Pitch rotations are fixed, therefore we are only interested in the Yaw rotation relative to Z axis.

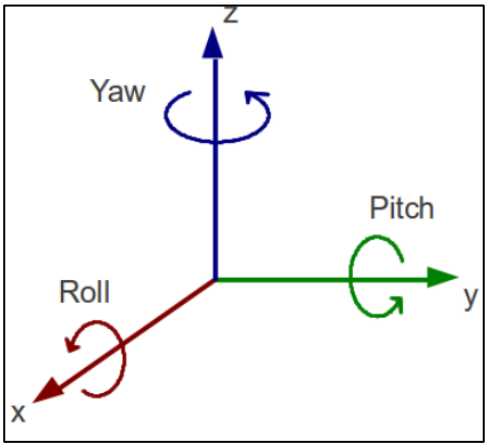


figure 2[2]

The tracker location is saved for each sample and in case the sample is approved (and not retaken) the position is recorded into the position log file.

2.2. Constraints

While working on the algorithm and the light model creation we had several constraints regarding the type of light source and the location of the device during the measurements:

1. **Focused light** - the measured light source needs to be narrow spotlight and not wide directional light.
2. **The device position -** the device should be found in the center of the light beam and not on the edge of the spotlight.
3. **Taking the measurements -** it's required that for each measurement one arc will be directed to the light source.

[2] Figure 2 – The 3D axis rotations.

2.3. The Algorithm Operation

2.3.1. Inputs and Outputs

The algorithm is capable of taking the samples of light from the device's samples of the environment and creating a light model of a single directional light source equivalent to it.

Input: The algorithm uses several input files:

1. input.txt – contains the measurements made by the hardware:

```
58 55 54 12 6 0 0 0 0 0 17 9 10
38 44 46 11 5 0 0 0 0 0 37 43 53
```

figure 3[3]

2. position.txt – contains the tracker's data (x , y , rotation) for each corresponding measurement:

```
0.36 -2.14 8.87
0.34 -2.15 329.4
```

figure 4[4]

3. fitting.txt – contains basic data for the light's distance -> light intensity and light intensity -> distance conversion functions:

fitting.txt - Notepad

File Edit Format View Help

0 0.05 0.1 0.15 0.2 0.25 0.3 0.35 0.4 0.45 0.5 0.6 0.7 0.8 0.9 1 1.1 1.26 1.42 1.58 1.74 1.9 2.06 2.22 2.38 2.54
100 97 84 80 75.25 61 58 52 52.25 44 38.75 31.5 25 23.25 19.25 15.5 15 12 10 9 7 5 5 4 4 3

figure 5 [5]

Output: The algorithm's output is four directional light sources, one for each quarter, with its intensity and rotation angles (Yaw and Pitch):

```
47.66 208.36 82.84
41.09 311.15 61.15
73.18 36.22 10.08
0.00 180.00 0.00
```

Figure 6 [6]

[3] Figure 3 – input example
[4] Figure 4 – position example
[5] Figure 5 – the fitting file
[6] Figure 6 – the output file

2.3.2. The Algorithm functionality

After we are done taking our measurements, the samples are sent to the fusion algorithm.

This algorithm has 4 main stages:

1. Stage 1 – The sample fitting
   For each sample taken, the first action performed is to calculate the different measurements and divide them into four quarters.
   The sample fitting is performed in the "*getCoordinates*" method, which receive a sample (array of 13 measurements) and tracker rotation arguments.
   The 13 measurements represent the receptions of the 13 photoresistors as follows:

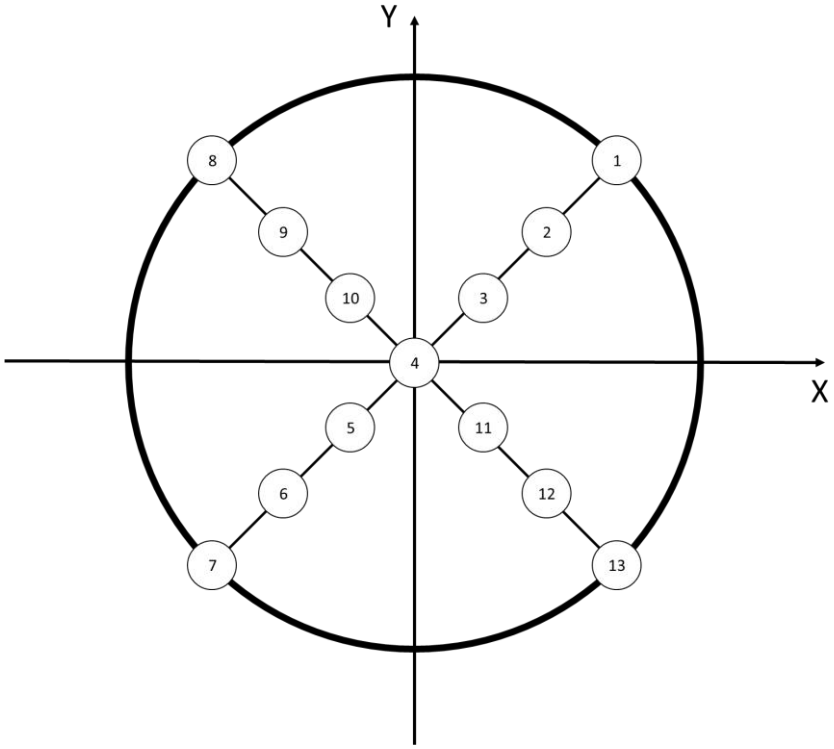


figure 7[7]

According to photoresistor position and the tracker rotation, each measurement is then inserted into a matrix that holds the (x, y, z) position data of each measurement.
For example, the measurement of photoresistor number 2 is calculated as follows:

```
[rot_x(cos_theta(90), sin_theta(90) * cos_alpha(30),rot),
rot_y(cos_theta(90), sin_theta(90) * cos_alpha(30),rot),
sin_theta(90) * sin_alpha(30)]
```

[7] Figure 7– scheme of the photoresistors positions on the dome.

The rotation value is calculated by combining the hardware rotation (45° as we can see on figure 3) and the tracker rotation given from the tracker.
The data is then passed to rot_x and rot_y methods that calculate the actual rotation in space according to these formulas:

```
def rot_x(x_point,y_point,theta):
    return x_point*cos_theta(theta) - y_point*sin_theta(theta)

def rot_y(x_point,y_point, theta):
    return x_point*sin_theta(theta) + y_point*cos_theta(theta)
```

figure 8[8]

The result of this part is a matrix containing the fitted sample according to the light intensity measured on each photoresistor and the device's rotation in space. For example, this is the intermediate result for the input shown in figure 3:

```
The matrix is:
 [[ 0.5896     0.8077     0.       ]
 [ 0.5105936  0.6994682  0.5      ]
 [ 0.2948     0.40385    0.866    ]
 [ 0.         0.         1.       ]
 [-0.2948    -0.40385    0.866    ]
 [-0.5105936 -0.6994682  0.5      ]
 [-0.5896    -0.8077     0.       ]
 [ 0.8077    -0.5896     0.       ]
 [ 0.6994682 -0.5105936  0.5      ]
 [ 0.40385   -0.2948     0.866    ]
 [-0.40385    0.2948     0.866    ]
 [-0.6994682  0.5105936  0.5      ]
 [-0.8077     0.5896     0.       ]]
```

```
The matrix is:
 [[ 0.9686     0.2487     0.       ]
 [ 0.8388076  0.2153742  0.5      ]
 [ 0.4843     0.12435    0.866    ]
 [ 0.         0.         1.       ]
 [-0.4843    -0.12435    0.866    ]
 [-0.8388076 -0.2153742  0.5      ]
 [-0.9686    -0.2487     0.       ]
 [ 0.2487    -0.9686     0.       ]
 [ 0.2153742 -0.8388076  0.5      ]
 [ 0.12435   -0.4843     0.866    ]
 [-0.12435    0.4843     0.866    ]
 [-0.2153742  0.8388076  0.5      ]
 [-0.2487     0.9686     0.       ]]
```

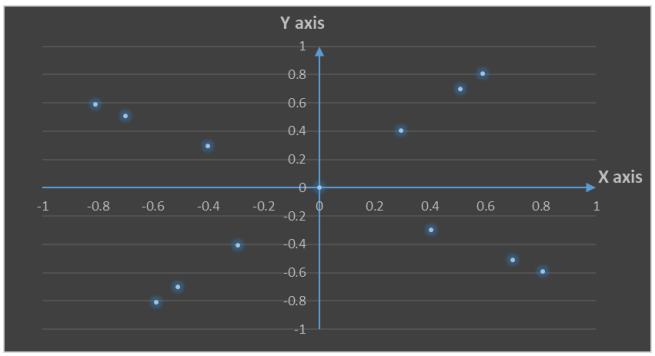


Figure 9[9]

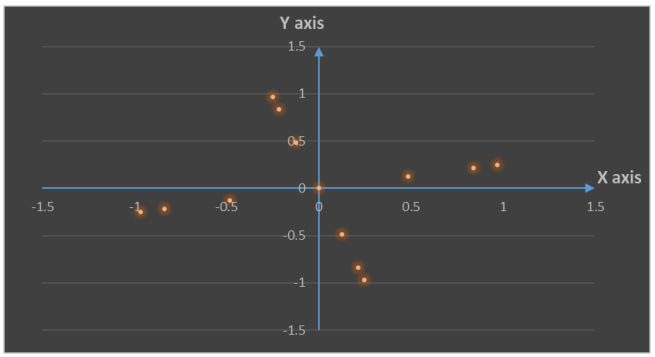


Figure 10[10]

[8] Figure 8 – the calculation of rotated x position and y position
[9] Figure 9 – the matrix for the first input sample
[10] Figure 10 – the matrix for the second input sample

2. Stage 2 – Light intensity do distance conversion

For getting only the lighted part of the matrix, we need to multiply the rotated matrix with equivalent distance of the measured light intensity.

Therefore, we need to build a non-linear conversion function that will convert every light intensity to accurate distance.

In order to find the exact function, we used an online tool [11] with our fitting samples (figure 5) and got the following function:

$$f(x) = \frac{A-d}{1+\left(\frac{x}{c}\right)^b} + d$$

While using this function and our fitting samples with the curve_fit optimize library in python we received this function graph:

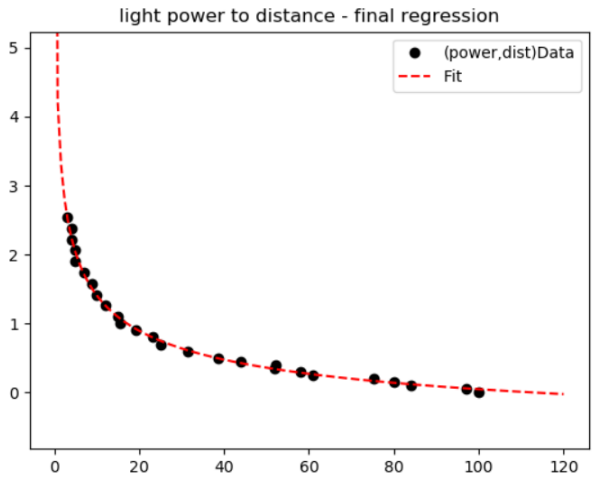


Figure 11[12]

[11] https://mycurvefit.com/

[12] Figure 11 – the curve_fit function graph of distance as function of the light power

Using this function, we converted our measurements into a distance and multiplied it with the previous matrix:

```
The final matrix is:

[[25.13011165 34.42603659  0.        ]]

[[23.07626723 31.61245089 22.5974897 ]]

[[13.5885758  18.61515039 39.91759375]]

[[  0.          0.        140.41657355]]

[[-58.88883592 -80.67251149 172.99094948]]

[[-46.29447824  33.79376547  99.27205188]]

[[-114.50949771   83.58895611   81.85468454]]

[[-125.16386085   91.36636419    0.        ]]
```

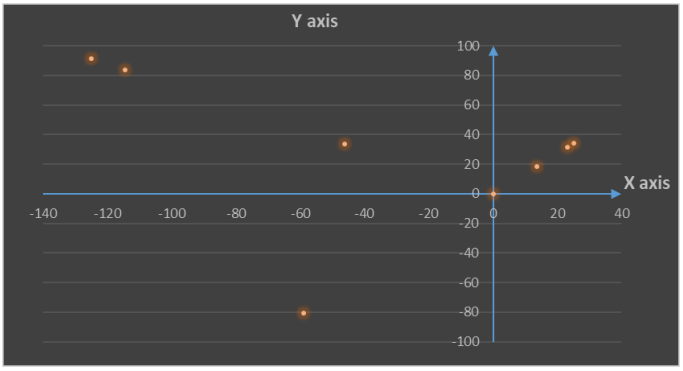


Figure 12[13]

```
The final matrix is:

[[62.36125898 16.01202262  0.        ]]

[[47.39392258 12.16897434 28.25077084]]

[[26.24123939  6.73776196 46.92321559]]

[[  0.          0.        147.26679648]]

[[-105.20746004  -27.01331335  188.12649266]]

[[-8.18924717 31.89426944 57.03166908]]

[[-12.42993821  48.41028607  28.85660912]]

[[-11.69282945  45.53950385   0.        ]]
```

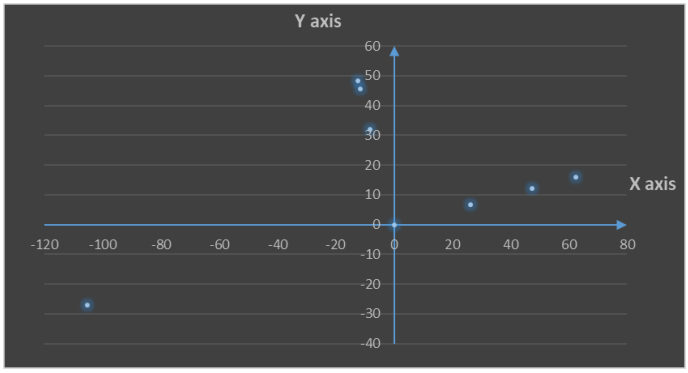


Figure 13[14]

3. <u>Stage 3 – Unifying Coordinate Systems</u>

The next part of the algorithm is determining the base to our coordinate system.
Therefore, we decided to choose the first (x, y) data from the tracker as our center of the coordinate system and any additional sample will be calculated relatively to it.

In order to perform this unifying process, we take every individual measurement (it's x, y coordinates) and transform it to the unified coordinate system while taking in consideration the current tracker location.

[13] Figure 12 – the calculated matrix for the first input sample
[14] Figure 13 – the calculated matrix for the second input sample

The next step is to determine in which quarter each measurement is found and classify the measurements into one of the four quarters.
Then, we calculate in each quarter it's summed location and setting one location for each quarter:

```
The points in quarter  0 is:  [array([15.44873867, 21.16340947, 50.73291425]), array([45.33214032, 11.63958631, 25.05799548])]

The points in quarter  1 is:  [array([-71.4919592 ,  52.18727144,  80.38582749]), array([-10.77067161,  41.94801979,  28.62942607])]

The points in quarter  2 is:  [array([-58.88883592, -80.67251149, 313.40752303]), array([-78.90559503, -20.25998501, 251.54496685])]

The points in quarter  3 is:  [array([0., 0., 0.]), array([0., 0., 0.])]
```

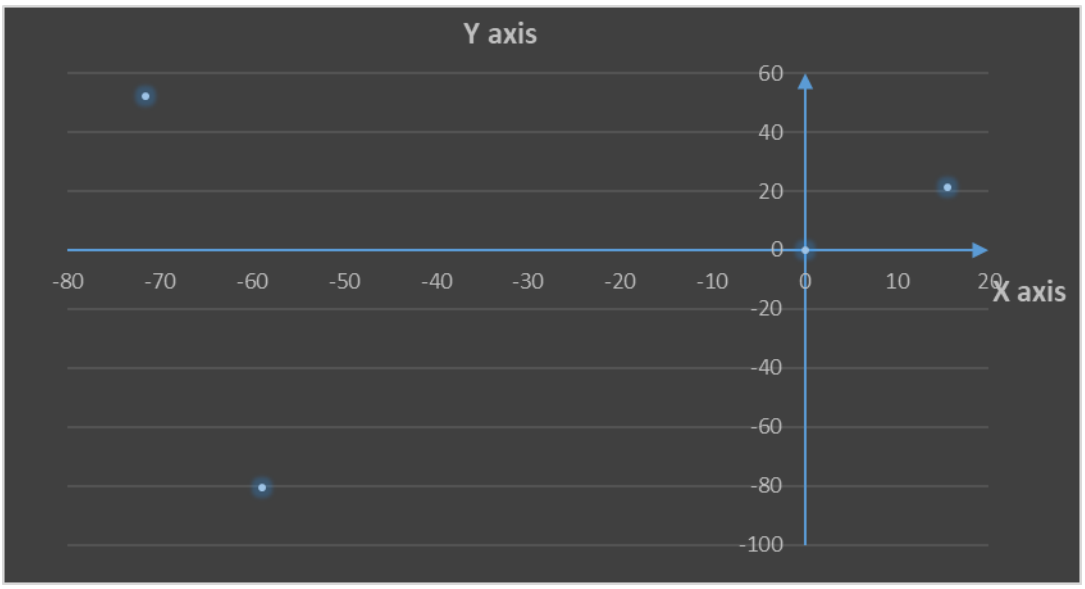


Figure 14[15]

[15] Figure 14 – the final 4 light sources, one for each quarter

4. Stage 4 – Create final light source

At this point we have four light source positions, and we want to determine its intensity and direction.
Therefore, we first need to calculate the opposite conversion function the same way we did in part 2 of the algorithm.
We used the same online tool[16] and received the following function:

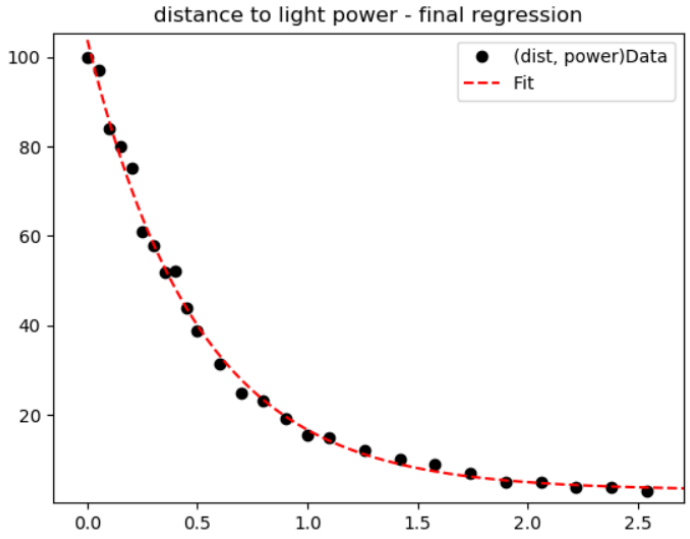


Figure 15[17]

The conversion result in our case is:

```
calculating Light Source....
The light power of quarter  0 is:  82.84128186340446

The light power of quarter  1 is:  61.145096518437796

The light power of quarter  2 is:  10.075645654943632

The light power of quarter  3 is:  0
```

Figure 16[18]

[16] https://mycurvefit.com/
[17] Figure 15 – the curve_fit function graph of light power as function of the distance
[18] Figure 16 – result of light source power after the conversion

The final action will be to calculate the rotation angle and the elevation angle of the light source, and we will do that using the (x, y, z) positions of the light sources.
In addition, for compatibility with the Unity coordinate system (in the measured room environment) we needed to rotate our result by 180 degrees.

The final result for each quarter in will be:

```
result of quarter  0  is:
Position =  [15.19521975  8.20074894 18.94772743]
elevate angle =  47.6573
rotate angle =  208.3554
light_power =  82.84128186340446


result of quarter  1  is:
Position =  [-20.5656577   23.53382281  27.25381339]
elevate angle =  41.0891
rotate angle =  311.1495
light_power =  61.145096518437796


result of quarter  2  is:
Position =  [-34.44860774 -25.23312413 141.23812247]
elevate angle =  73.178
rotate angle =  36.22230000000002
light_power =  10.075645654943632


result of quarter  3  is:
Position =  [0. 0. 0.]
elevate angle =  0
rotate angle =  180
light_power =  0
```

Figure 17[19]

[19] Figure 19 – the final output that will be passed on to Unity

3. Unity, Hololens & HTC Vive Tracker

3.1. Unity & Hololens

After all the data is ready from the device and the algorithm, we then need to show the finished result within unity, and the Hololens.

During this process we encountered a few difficulties we shall elaborate about here:

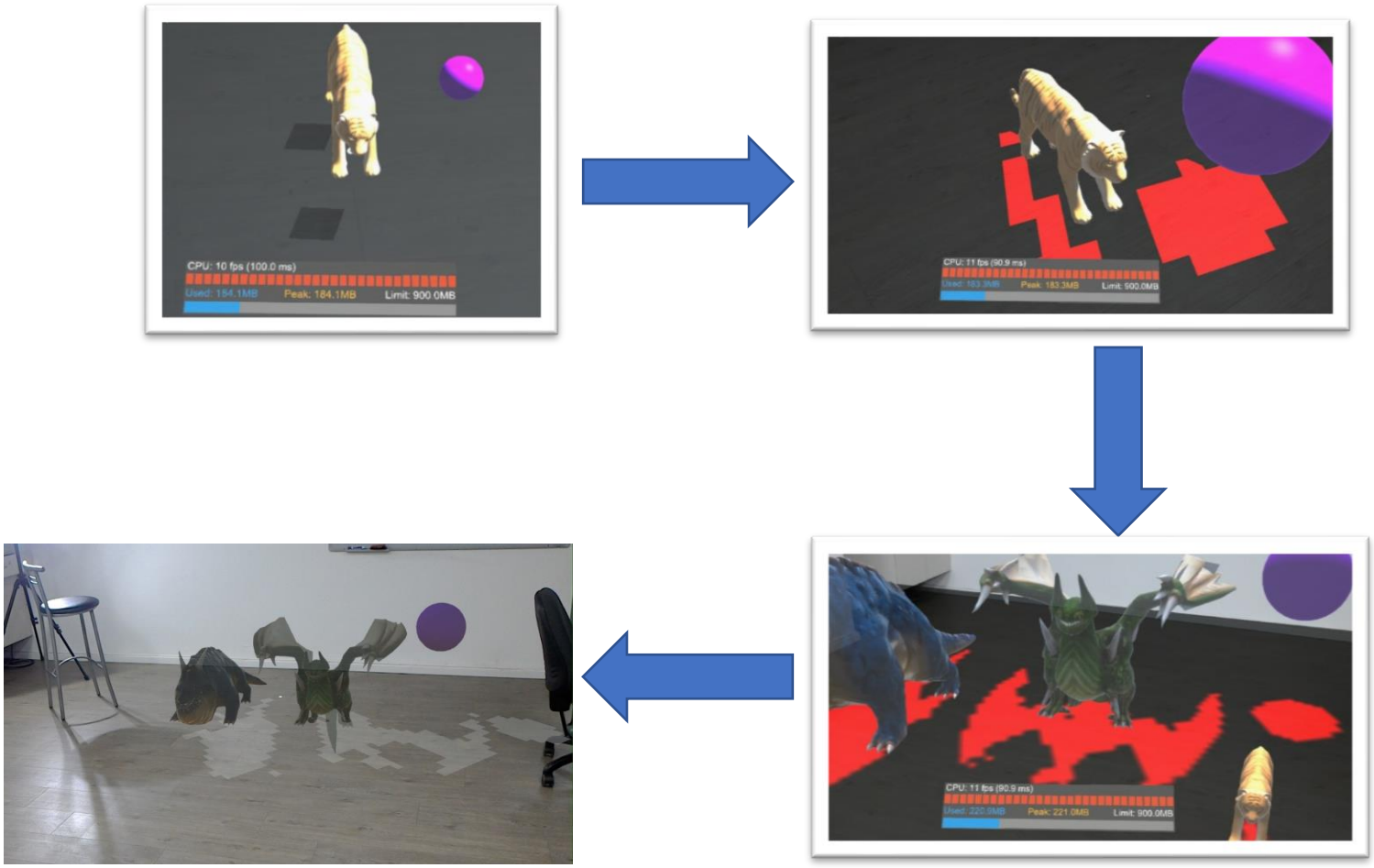
- Hololens limitations.
- Unity shaders.

3.1.1. Hololens limitations

As part of our project, we were assigned the hololens in order to show the light model after all measurements were done.
While dealing with the hololens a few issues came up:

- The hololens file system:
  Our original plan was to “pull” the data from within the scene through a unity C# script. But after trying to do so we concluded that after deploying the scene and starting to run it on the hololens there isn’t access from the hololens to the pc’s file system while connected via USB, or from the pc’s file system to the hololens’s.
  Because the light is created after the scene starts running, and obviously the location is unknow prior to taking the samples with the device, meaning the C# script runs in the scene, not prior to the scene. Because the scene starts on the hololens.
  So, we needed to find a way to plant the data within the C# script before we deploy the project to the hololens.
  To solve this issue, we put a “~” character in the C# script which is later substituted with the output from the algorithm. This means that instead of us being able to run the measurements while the scene is running and then when the output is ready the light is created, we need to take the measurements, deploy the scene to the hololens and then run it.
- The hololens color:
  The unity projects its AR scene onto reality by projecting the digital scene related objects onto the lenses of the glasses, that fact means that anything black does not show up on the hololens, as there isn’t a way to project black. Shadows by default, are shown in unity to be black, to change them to a different color we need to use a custom shader.
- The hololens quality:
  The whole scene and scene calculations are done on the hololens. This means that its calculations aren’t as strong as ones run on a pc, this leads to poorer performance when calculating real time shadows. We can’t use baked shadow maps, as the light source location is not known prior to the scene running, and the objects in the scene are not static, both are a must for using baked shadow maps.

The poor performance on the real time shadow calculations lead to the shadows not looking as good on the hololens as they do in the unity editor. A lot of tweaking and setting changes were needed in order to get the shadows looking better on the hololens, closer to how they look in the unity editor.



Pictures above show the progress made with the shadow quality until our final result.

3.1.2. The unity shaders

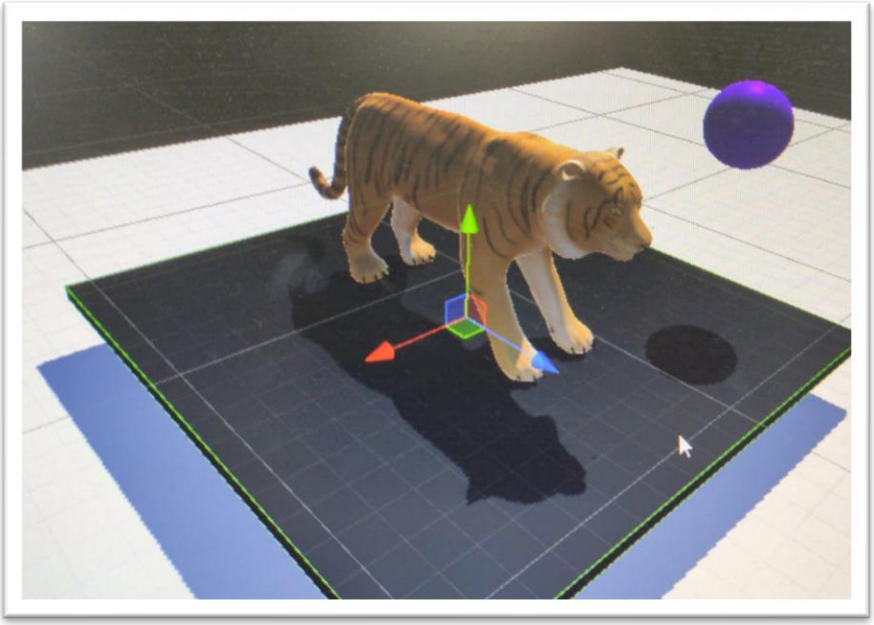
Part of the requirements were to show the result of the light module creation in the augmented reality, part of it is showing she shadows on the real floor, rather than a digital floor.

The built-in unity shaders don't show shadows on non-augmented objects.

Because of this fact, together with the fact mentioned before about the color black in the hololens, we needed to create custom shaders to use in our projects.

Custom shaders, to try and get the quality of the shadows to look better on the hololens, to get the shadows in a different color, initially red, then on to a grey color to better match real life shadows, and most importantly to get shadows to cast on the real-life objects.

An example of how well shadows look in unity.



3.2. HTC Vive Tracker

As mentioned previously we used the HTC Vive's tracker, in order to retrieve the location and rotation of the device while taking the samples of the room's light module.
In order to work better with the Hololens, we had to use Microsoft's AR Toolkit.
It allowed us to set up and work better in building the scene and setting it up for the hololens.
To work with the HTC Vive, we needed to work with Steam VR, and also, we couldn't get the tracker to work independently without the headset.
So, we decided to work with the HTC vive's headset connected.
When we tried to insert the Steam VR plugin into the hololens's project, there were problems that came from different settings each plugin wanted to set, Steam VR and AR Toolkit. As the needed data from the tracker can and is moved via the filesystem, we decided we can run two separate projects on the pc, one for the HTC Vive, and another for the hololens.
So, that is what we do, run the HTC Vive's project which is pretty much empty, solely for the purpose of retrieving the trackers location into an output file located on the pc's filesystem, and run the hololens's project so after the output from the algorithm is done, we can deploy and run the scene.