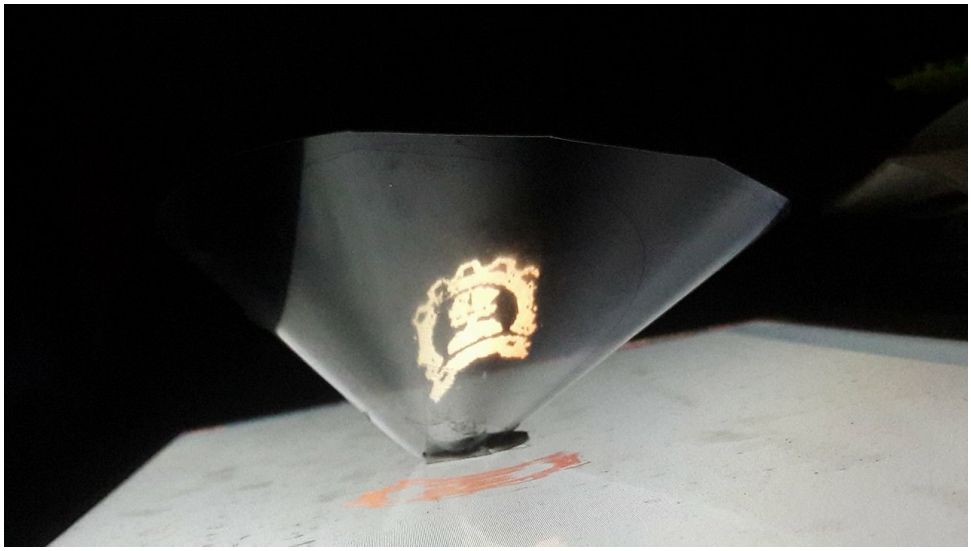
Pepper's Cone:
An Inexpensive Do-It-Yourself 3D Display
CS 775 Course Project

Kumar Ayush (140260016)

Sourav Bose (153059005)



Contents

1 Introduction

This project aims to reproduce the results of Xuan Luo's Pepper's Cone paper [1]. The authors advertise the project as an inexpensive DIY display. The setup is shown in Fig 1. It works on the principle of Pepper's Ghost [2] i.e. we simultaneously see a reflection of the image displayed on the screen with the background, which results in a perception of a floating 3D image. As we will discover, the "inexpensive DIY" claim holds valid only for using a pre-calibrated setup. The source code provided has a distortion map for providing the effect, which is primarily useful for a 51° cone on a 12.9" iPad Pro display. The primary challenge of our project was to attempt this calibration ourselves. Our procedure and limitations of the final result are described in the following sections.

The project has three modules. One, we need to invoke WebVR API on a browser to query the pose information for the display device. This allows us to rotate the 3D image about the axis of the display cone as the display device is rotated. Two, we need to use a WebGL rendering framework to render mesh objects. Three, we need to calibrate the distortion map that will be applied on the image displayed on the 2D screen such that the reflection from our plastic cone gives the expected image. The first two modules are relatively easy to implement given a vast number of example codes on the Internet, but the third module has several engineering challenges involved.

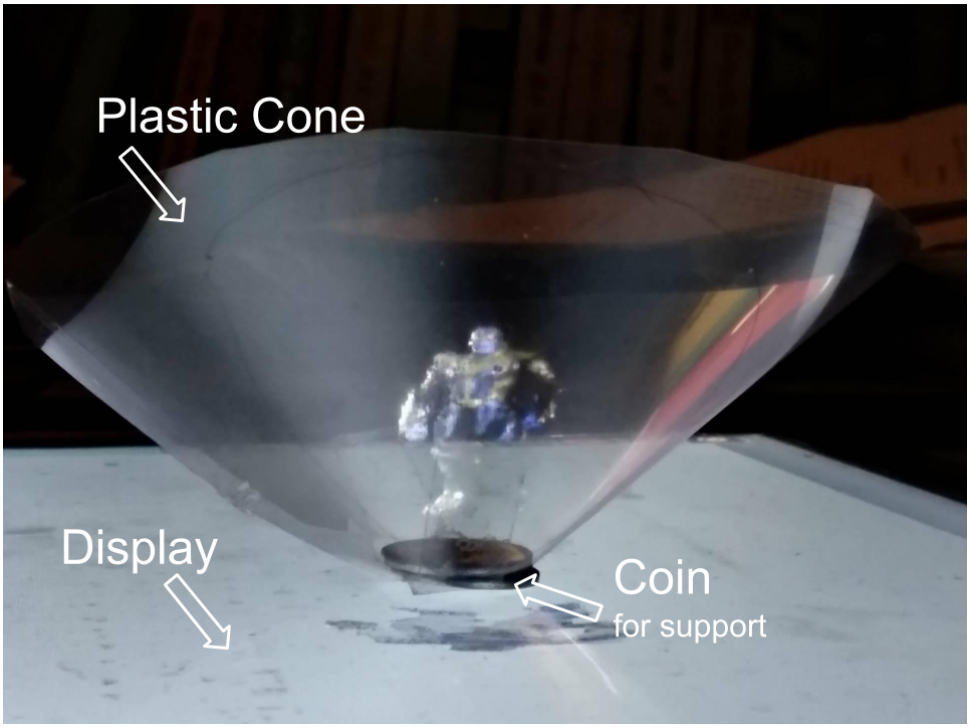


Figure 1: Setup demonstrating a 3D display of Thanos

2 Procedure

2.1 WebVR

We have used WebVR to query the device orientation. This is required to rotate the rendered mesh according to the orientation of the cellphone. We obtain the data as quaternions and do minimal processing to convert it to a rotation matrix which we apply on the mesh. The convertion of quaternion pose data to rotation angle is demonstrated through the code fragment shown as follows:

```
vrDisplay.getFrameData(frameData);
var newRot = new THREE.Quaternion();
newRot.fromArray(getPose(frameData));
var initRotInv = initRot.inverse();
var diffRot = newRot.multiply(initRot.inverse());
var diffAngle = new THREE.Euler();
diffAngle.setFromQuaternion(diffRot);
```

2.2 THREE.js Renderer

The rendering and ray casting of the input mesh is done using THREE.js. It provides a convenient way of rendering the mesh with appropriate methods for scaling/rotation/translation etc. The framework also allows us to specify objects and lights in a scene which is then rendered using their ray casting module. We integrate THREE.js with WebVR by reading rotation matrix and applying it to the mesh using standard methods. In the interest of time and unavailability of granular control on the rendering pipeline, we could not prototype a working demo of the realtime application of the distortion map on the final image. However, we have shown that this step is trivial once you have the access to the pixel data, in which case, you can directly set pixel values based on the distortion map as the distortion map does not change on rotation.

2.3 Calibration

The calibration is done using gray code structured lighting. A series of images with various sections painted in two contrasting colors representing 1 and 0. For each pixel, the temporal sequence of 1s and 0s defines a unique binary code. This code is used as the primary key to identify the mapped pixel on the reflected image. Example images are shown in Fig. 2. We have used OpenCV's structured light module which allows you to create a graycode pattern given a resolution. The module also has a decoding function but it is implemented only for the case of 3D reconstruction of a screen with imaging done for two cameras, hence we just extract the graycode pattern from an intermediate step in the module. All the images and the gray code GIF and video can be found along with the source code used to generate them in the repository. We have created the code for 640×480 resolution in the example discussed in the report.

We create a map file using this graycode video. A few lines from the map look like this:

```
01011001011010010110010110100101100101,71,238
01011001011010010101010110100101100101,71,239
01011001010110010101010110100101100101,71,240
```

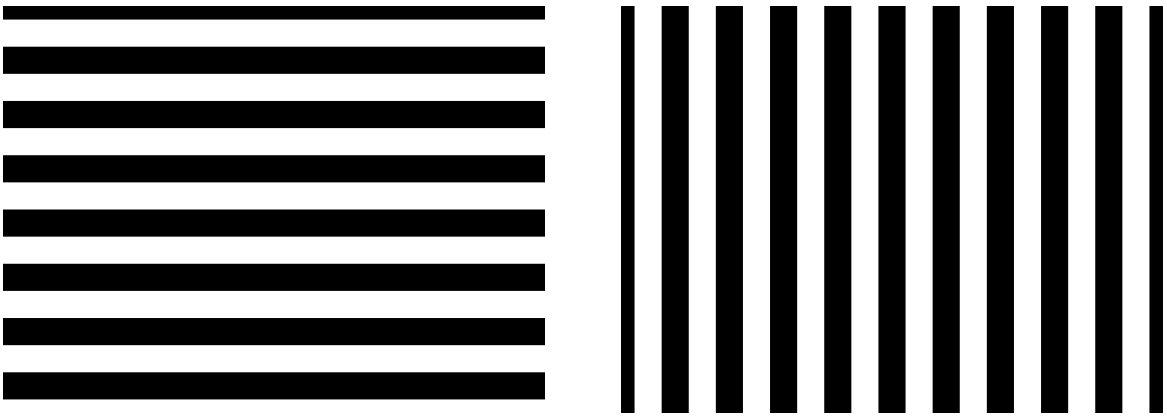


Figure 2: Example frames from the graycode video

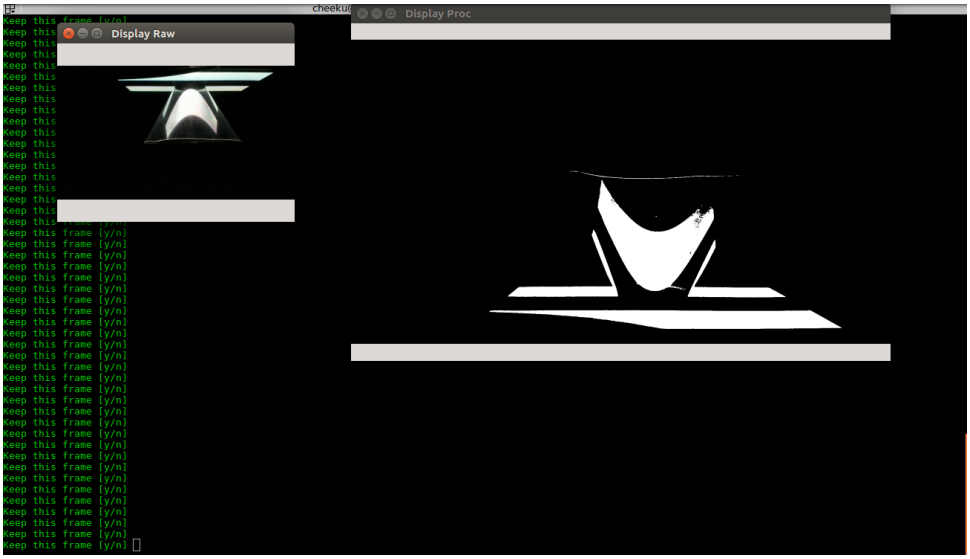


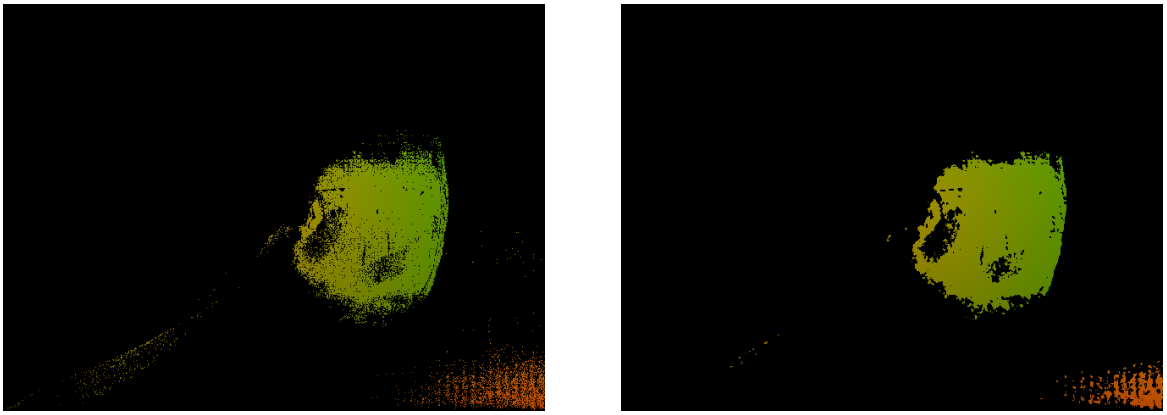
Figure 3: Selecting keyframes for calibration

```
4 01011001010110010110010110100101100101,71,241
```

For this resolution, there are 38 frames. Each line represents the binary code for a pixel followed by the pixel coordinates.

Next, we record a video of the graycode pattern reflected off the cone. We manually go through all frames to choose the best 38 frames to be used for calibration. Now we have a similar map file as above described with a binary code followed by pixel coordinates. This procedure is shown in Fig 3

We call the reference map the *direct map* and the recorded map the *inverse map*. The next task is to find entries with matching binary codes. To do this efficiently, we convert the binary codes to integer representation, sort them and then do a linear search along both the maps. Now, we have a mapping of pixels on the display (s, t) to pixels on the recording (u, v). The pixel coordinates (u, v) need to be scaled to 640x480 or whichever resolution we use for creating the graycode pattern. An interesting image is $im[s, t] = [u', v', 0]$, where $u' = \frac{u}{h}$ and $v' = \frac{v}{w}$, where w and h are width and height; (640×480) in our case. For our example case, this is shown in Fig 4 We can apply a Median Blur to find some missing values. The authors of the original paper [1] have



(a) Original *(b) Median Blurred with a kernel of size* 3×3

Figure 4: UV Plot for the Distortion Map

(a) Original *(b) Distorted with blurred map*

Figure 5: Applying the distortion map

used some Gaussian kernel to interpolate the values but we could not understand it well. We tried a Gaussian Blur within our limited understanding but got worse results than the Median Blur.

The points in the corner of Fig 4 represent the pixels matched from the direct capture of the display screen. While finally applying the distortion map, we remove them.

We finally apply the distortion map. This is done with the following line of code.

$$\text{distorted\_image}[s,t] = \text{image}[u,v]$$

. The results are shown in Fig. 5

Other than Fig 1, a couple more of our images captured using this map are shown in the next section.

3 Results

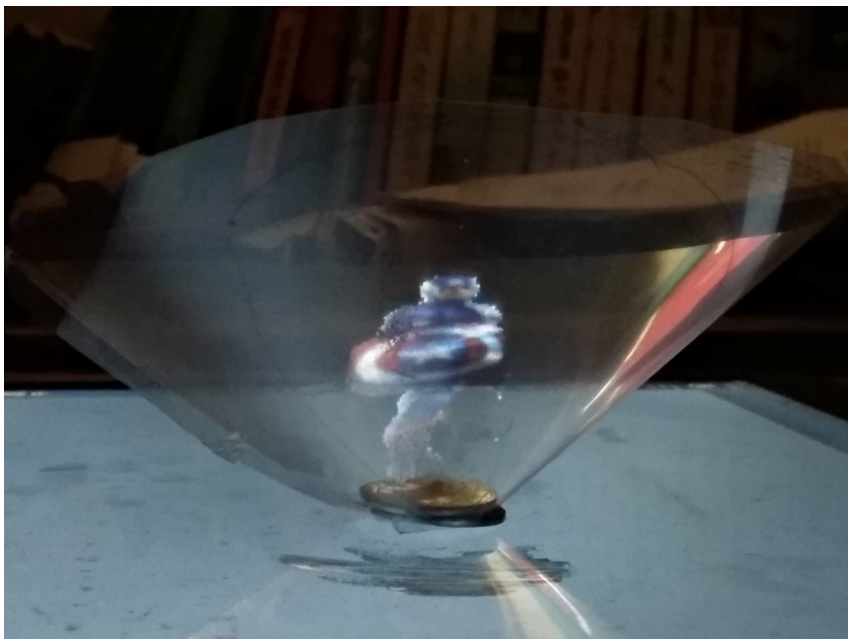


Figure 6: Captain America

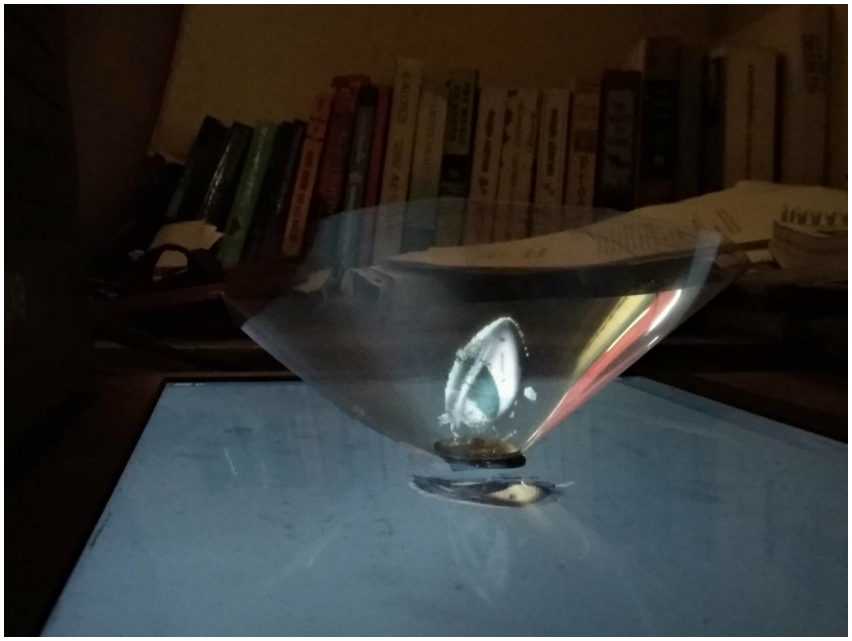


Figure 7: Spaceship

4 Challenges

The following are some practical challenges that we faced:

- The material of the cone must be carefully selected. Too thin will have thin film interference effects and too thick will have double reflection effects. The authors of the original paper do mention this, but we were unable to find a perfect material. We are currently using plastic sheets used for lamination, which has some thin film interference but it is not noticeable.
- The calibration must be done in a complete dark room with no objects around it. We were only able to successfully create a distortion map for the central front part of the cone because there were always objects or a wall near the display device.
- The high frequency frames of the graycode get aliased unless the video is captured in 4K HD. If we are using a small cone for a smartphone display, this is again a problem because the view plane angular size of the reflection of a pixel is less than the resolution of our camera. A large cone or reflector and a high definition camera, which is steady are very important for calibration.
- The non-distorted image has constraints on the position of the object of interest, which means that it is non-trivial to let the user specify what object to display.

5 Discussion

We tried to reproduce Xuan Luo's Pepper's Cone [1]. It is inexpensive and the effect looks brilliant. The DIY claim, however, does not extend if we want to do our own calibration. We have highlighted the challenges and we hope that our work will be helpful to someone trying to build their own Pepper's Cone 3D display.

References

[1] Xuan Luo, Jason Lawrence, and Steven M Seitz. Pepper's cone: An inexpensive do-it-yourself 3d display. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 623–633. ACM, 2017.

[2] Wikipedia. Pepper's ghost
`https://en.wikipedia.org/wiki/Pepper's_ghost`
last accessed on [03-05-2018].