# Illumination Reconstruction for Augmented Reality

by

**Daniel Cullen, B.Sc.**

**Dissertation**

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

**Master of Science in Computer Science**

**(Interactive Entertainment Technology)**

**University of Dublin, Trinity College**

September 2017

# Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Daniel Cullen

August 27, 2017

# Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Daniel Cullen

August 27, 2017

# Acknowledgments

I wish to give my thanks to my supervisor Dr. Aljosa Smolic. His guidance and advice throughout the project proved invaluable.

Daniel Cullen

*University of Dublin, Trinity College*

*September 2017*

# Illumination Reconstruction for Augmented Reality

Daniel Cullen, M.Sc.
University of Dublin, Trinity College, 2017

Supervisor: Aljosa Smolic

In this dissertation we propose a system to increase the realism of a virtual object placed a real environment. We aim to make the virtual object appear to be seamlessly integrated into its immediate surroundings, to achieve this we want to use information received by the camera to reflect details and changes of the background onto the object. These reflections should provide a real-time continuous way of updating the virtual objects appearance to make it truly feel like part of its environment.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

This dissertation aims to develop a system to add real time global lighting to virtual objects using information captured by the camera of the area surrounding the object. The aim behind creating this system is to add a sense of realism to the rendered object by having the objects appearance change depending on the brightness of its immediate environment is will make the object seem far more integrated into its environment.

Virtual objects with a more reflective material, such as a mirror or glass object, should show accurate real-time reflections of the nearby environment. More complex virtual objects that aren't completely reflective or are only partly reflective should experience changes in appearance that correspond to changes in the virtual objects environment. For example, imagine we have a virtual object with a partially reflective material rendered on a marker and beside the marker we have a light bulb that is switched off. If the light bulb was switched on then we should see that light reflected onto the virtual object from the direction of the light bulb.

## 1.1 Motivation

Augmented Reality has found uses in a variety of industries, including education [1], construction [3], and most prominently in the entertainment industry [5]. The recent global

phenomenon of Pokemon Go, an interactive augmented-reality based mobile application showcases the vast potential and large public interest in interactive AR and AR based entertainment with over 750 million downloads since its release in July 2016 [7].

Moving forward, augmented reality should look to avoid becoming a novelty as it may be prone to do, and become integral to how we communicate with each other, how we work and how we entertain ourselves. AR provides a digital overlay on top of a real world environment, this overlay should seek to be seamlessly integrated into its surroundings and should be adaptive and responsive to changes in its environment in real time. Performing these tasks to a high level should improve how well a virtual object is blended in with its surroundings and thus improve a users experience when using this technology.

## 1.2 Road-map

This dissertation starts with some background research in Chapter 2, then it describes the design of the system in Chapter 3. Chapter 4 describes the implementation and various challenges of coding the project. Chapter 5 provides an evaluation of the system and methods used, while Chapter 6 discusses future work and offers a conclusion and some discussion.

# Chapter 2

# Background Research

## 2.1 State of the Art

Most research in the area of illumination for augmented reality is based on using complex lenses/mirrors or building complete maps of the environment.

## 2.2 Use of a complex lens

T. Grosch et al. [4] describe a method whereby they use a HDR fish-eye camera to capture daylight from outside a room and then use this information to create lighting and shadows on a virtual object rendered inside a room. Using this approach means that the virtual objects brightness or appearance will depend on where the fish-eye camera is placed. For example, if the fish-eye camera is placed somewhere with lots of sunlight, then the virtual object will appear very bright, and the opposite if the camera is placed in a dark area. This set up is visualised in Figure 2.1.

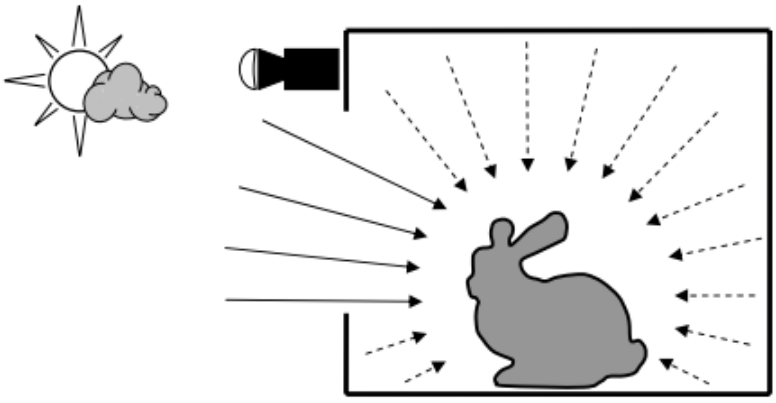


Figure 2.1: A graphic showing the set up described in the T. Grosch et al. paper [4]. Shown is a virtual object in a room with a fish-eye camera outside. Data recieved by the fish-eye camera is then projected onto the virtual object.

## 2.3 Use of extra apparatus

Another method of creating illumination for a virtual object is shown by S. Heymann et. al [2] . This method involves the use of a spherical mirror to capture a 360 degree image of the environment. This sphere image can then be unwrapped into a panorama image, then this panorama image is used to form an environmental cube map. This cube map can then be used to calculate light values which are then reflected onto the rendered virtual object. Figure 2.2 shows this method in action.

## 2.4 Use of multiple cameras

The use of multiple cameras is also a possible solution to the augmented reality illumination problem. As seen in K. Rohmer et al. [6] multiple HDR cameras can be used to provide a better representation than a single camera. The method described in this paper features multiple HDR cameras attached to a PC, this PC handles the intense computation involved with illumination reconstruction. The computed lighting is then projected onto the virtual object and displayed on a mobile device. This set-up can be seen in Figure 2.3.

Figure 2.2: A demo of the set up described in the S. Heymann et al. paper [2]. Shown is a virtual object rendered on a marker, the spherical mirror can be seen in between the ears of the object.

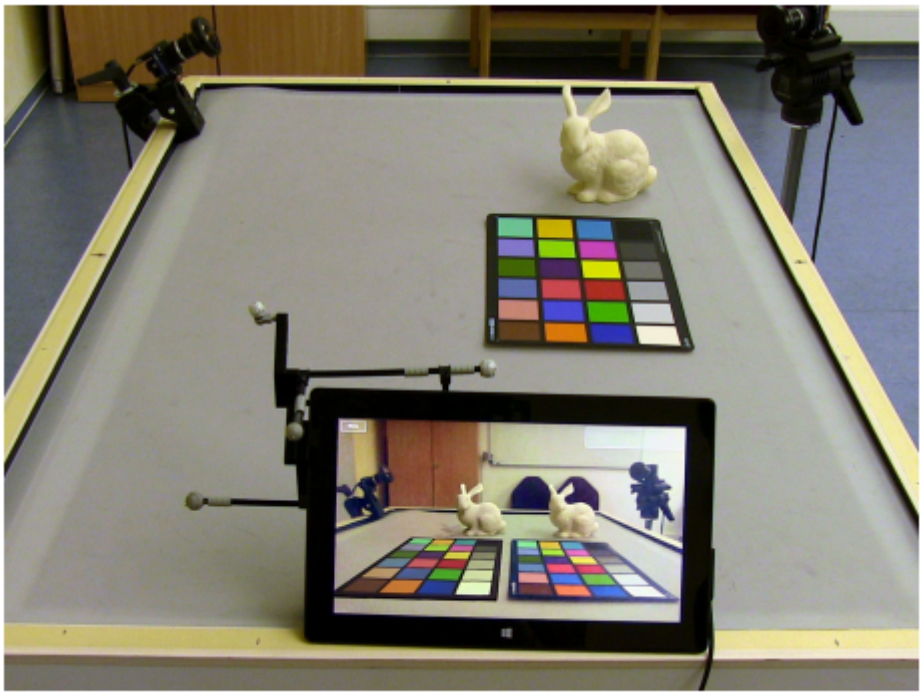


Figure 2.3: A demo of the set up described in the K. Rohmer et al. paper [6]. Shown is a tablet displaying some virtual objects. Multiple HDR cameras can also be seen on either side of the table.

## 2.5 Summary

Each of these methods for creating environmental illumination is valid and worthwhile. However, for this dissertation we propose a method that doesn't require extra pieces of equipment. The system only requires one camera and the camera only needs a regular lens, not a complex lens such as the one described in Section 2.1.

# Chapter 3

# Design

The overall goal of this project is to have a system which will gather information from a camera and use it to project reflections and lighting effects onto a virtual object. To achieve this, the live feed from the camera must be accessed and operated on every frame. Therefore the first step in the design is accessing the feed from the camera in the form of a 2D texture.

## 3.1 Extracting relevant data from the camera

When the camera background texture is found, the next step is how to use it in a way which we can provide an accurate representation of the real environment for the virtual object. One solution is to use this texture and map it directly onto an the object, but this often results in a mapping which will not correspond to the environment, that is to say for example that information from the area to the right hand side of the virtual object won't necessarily be mapped to its right hand side (see Figure 3.1). This random mapping is of no use in a real-world scenario, rather we wish for the system to provide a more intuitive mapping. One way to achieve this is to create a cube map which the virtual object is placed inside of. Then using a reflection probe the information taken from the cube map can be projected onto the object, this projection should provide an intuitive and accurate

mapping.

Figure 3.1: An example of random mapping where the pen should be reflected onto the left side of the cube map but is instead projected onto the right.

## 3.2 Cube map construction using the camera feed

There are a variety of different ways to construct the cube map, one such way is to use six two-dimensional textures as faces for the cube. This method of construction seems to be the most logical to use for this project, as we can divide the camera feed into smaller sections and we may also choose which section corresponds to which cube face. This gives us the ability to ensure that the area to the right of the virtual object is mapped to the right hand side of the object, thus eliminating the problem that came with the random mapping (see Figure 3.2). There are a variety of possible ways to divide the camera feed which may result in a valid/good result, these will be discussed in greater detail later.

## 3.3 Dynamic camera division

While a specific camera division may work very well for a virtual object at a certain position on the screen, the same division may not be suitable for an object rendered in a different location. For example, if we took the same division for each of the scenes in

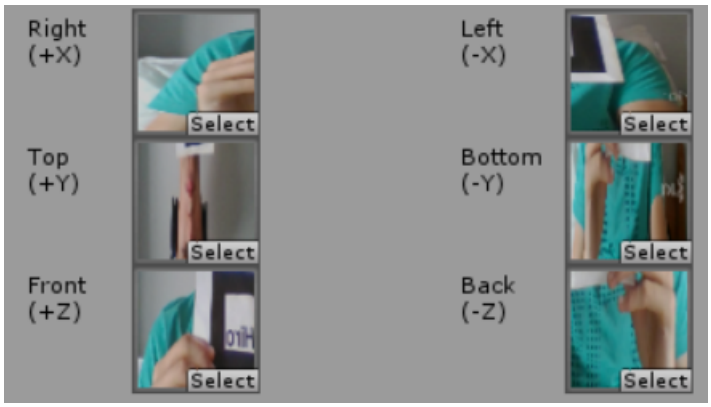



Figure 3.2: An example of a cube map when selected as shown by the Unity Inspector. A camera frame is split into six sections and each one is applied to a face.

Figure 3.3 and Figure 3.4 the result would not be good. If we took a division which is optimized for the object in Figure 3.3 then the area to the left of the object would be assigned to the left face of our cube. If this same section was used as the left face of the cube for the object in Figure 3.4, the result would be unsatisfactory as this area is very far from the virtual object and would not accurately represent its immediate environment. It is therefore very important to have a system in place which changes how the camera feed is divided into sections to be used for the cube faces depending on the marker position in the real environment.


Figure 3.3: A scene with a virtual object rendered on a marker placed in the centre of the camera view.


Figure 3.4: A scene with a virtual object rendered on a marker placed on the right hand side of the camera view.

# Chapter 4

# Implementation

## 4.1 Using Unity

The Unity game engine was chosen as a platform to create the system on. This decision was made because of prior knowledge and experience using the game engine and coding in C#, and also because of Unity's augmented reality capabilities.

## 4.2 Using the Vuforia SDK

To handle the marker detection and tracking, the Vuforia software development kit was used. This software is well integrated with Unity and also supports development on mobile devices so was perfect for use in this project. To use Vuforia first a suitable image required to use as a marker. When a suspected good marker image is selected it is uploaded onto the Vuforia Development Portal which provides a rating for have effective the image will be when in use. Multiple images were checked for suitability, it was decided to use the image in Figure 4.1. Figure 4.2 shows the Developer Portals rating for the marker image.

After the image to be used was decided on, the Vuforia Image Manager generates a database for use in the Unity editor which was downloaded and imported into the Unity project. The database consists of several scripts and prefabs which are used for marker


Figure 4.1: The image used for the marker.

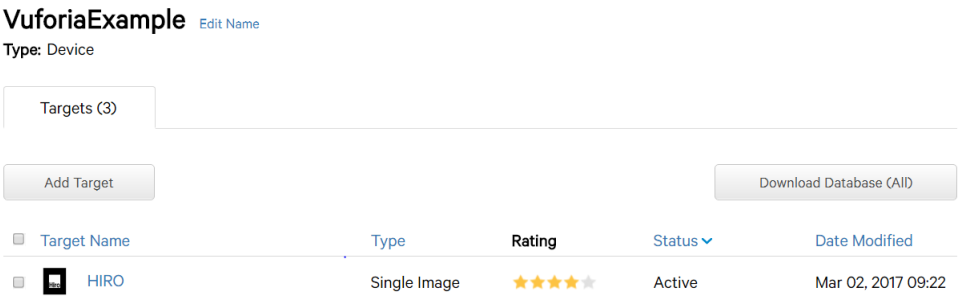



Figure 4.2: The rating the image used was given by the Vuforia Image Manager.

detection and tracking. Also included in the database are the selected images that are to be used for the targets. An ARCamera prefab and an Image Target object are added to the scene. There is a script attached to the ARCamera which allows a certain database of image markers to be loaded and activated.

The next step is to set the marker textures type to sprite to make it actually be able to be detected and tracked, then to add a virtual object we wish to be rendered on the marker. This is achieved by creating an object in Unity and setting it as a child of the marker object (see Figure 4.3).


Figure 4.3: An example of the Unity Hierarchy where a sphere object will be rendered on a marker called Image Target.

## 4.3 Background plane access

We need to access the live camera feed every frame and use it as a texture. It was first tried to directly project the camera feed onto an object while simultaneously displaying the camera in the Unity game screen. This results in an error where the camera will not be displayed or the texture on the object will be null, because the camera can't be split in this way.

To successfully access the camera feed, we are required to use the BackgroundPlane object in Unity. This BackgroundPlane is a child object of the ARCamera, and every frame the texture rendered to this plane is the live camera feed (see Figure 4.4). So by accessing this game object via C# script we may also have access to its material which we can then use as we wish. The following is code from a C# script attached to the BackgroundPlane object which stores the camera feed in a Texture2D object called im.

```
//Take in the camera feed as a 2D texture and name it "im"
        Renderer rend = GetComponent<Renderer>();
        Texture2D im = (Texture2D)rend.material.mainTexture;
        im.Apply();
```

If this code segment is run every frame then im will remain up to date with the live feed.

## 4.4 Division of the camera feed for an object in optimal position

As discussed in section 3.2 we now need to divide the camera into sections which will be then used as cube faces for our cube map. The first step is to find the camera image dimensions. Once the width and height of the screen is known, we can divide the camera using the pixel co-ordinates. For the division we require six sections, front, back, top, bottom, right and left. The division chosen was implemented as follows.

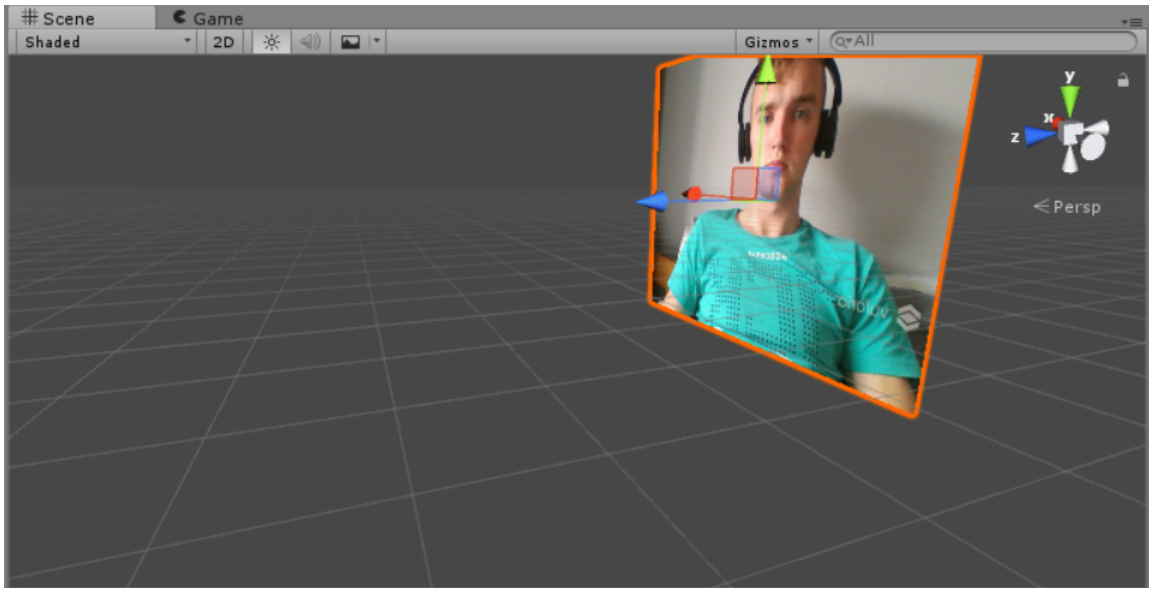



Figure 4.4: The BackgroundPlane object when seen in scene view.

We create new smaller textures by using the Texture2D functions GetPixels and SetPixels. We GetPixels to retrieve a Color array which contains the RGB values of the pixels we "Get". The arguments for the GetPixels function are:

```
Color[] Texture2D.GetPixels(int x, int y,
                            int blockWidth, int blockHeight);
```

The Color values of the pixels contained in the rectangle with its top left corner at the pixel with co-ordinate (x,y), of width blockWidth and height blockHeight. Using this array of Color values we can then use SetPixels on a newly created Texture2D of the same width and height to set this new texture as the rectangle we want.

```
void Texture2D.SetPixels(Color[] colors);
```

We then use the Texture2D.Apply() function to save the changes to this new texture. When creating the six different textures we use the naming convention

- xSection, ySection are the x,y co-ordinates of the top left corner of that section. So xTop and yTop are the co-ordinates for the top left corner of the Top section.
- sectionW, sectionH are the width and height of that section. So topW and topH

are the width and height of the Top section.

Now we go through how each section is defined for an optimally placed virtual object, which is an object rendered on a marker in the centre of the screen. For this we denote $w$ as the total screen width and $h$ as the total screen height. The definitions with regards positions and sizes of the sections are detailed in Table 4.1.

| Section | xSection | ySection | sectionW | sectionH |
|---|---|---|---|---|
| Top | 0 | 0 | $w$ | $h/4$ |
| Bottom | 0 | $3h/4$ | $w$ | $h/4$ |
| Front | $w/4$ | $h/2$ | $w/2$ | $h/4$ |
| Back | $w/4$ | $h/4$ | $w/2$ | $h/4$ |
| Left | 0 | $h/4$ | $w/4$ | $h/2$ |
| Right | $3w/4$ | $h/4$ | $w/4$ | $h/4$ |

Table 4.1: A table detailing the positions and dimensions of each of the six sub-sections of the camera feed

Using these co-ordinates and dimensions results in a division which can be seen clearly in Figure 4.5.

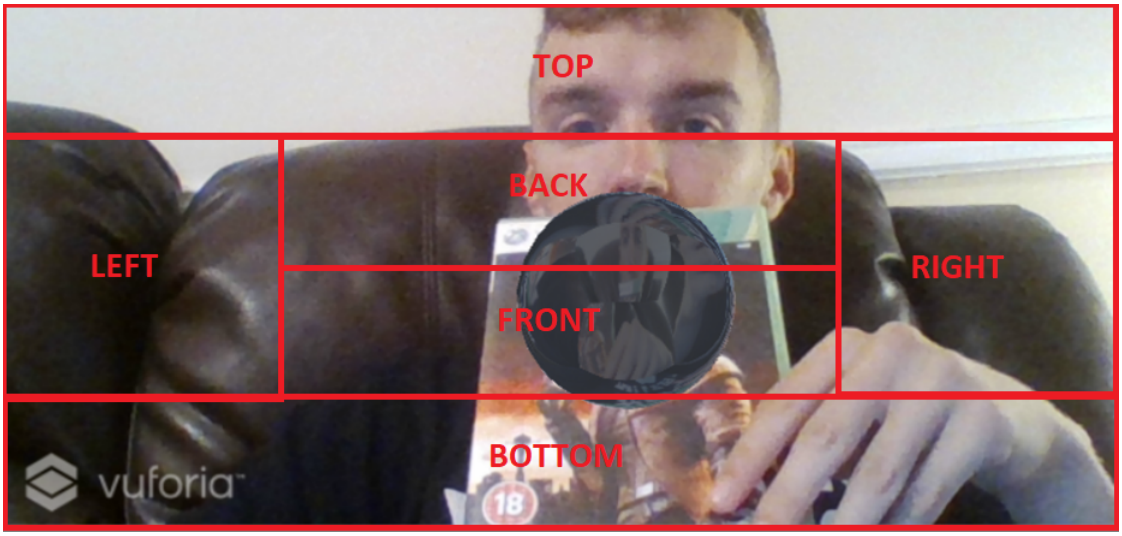



Figure 4.5: How the camera feed is divided using the parameters specified in Table 4.1.

## 4.5 Scaling and rotation fixes

Now we have six Texture2D objects which correspond to different sections of the camera feed. We can now define a cube map object in Unity and assign the textures to each cube map. While defining the cube map we choose the resolution of each face, which is the pixel length of each face. So if our cube map has resolution $R$, then each face has dimensions $R$ x $R$. Our six sections will likely not have the same dimensions, especially since only two out of the six sections will usually be square. Hence when we assign these sections to each face, Unity automatically compresses them so that they have size $R$ x $R$ instead of giving an error.

However, this compression causes significant problems for our implementation. Figure 4.6 shows the difference between a texture which doesn't go through this automatic compression and one that does.

To resolve this issue, we scale each of the textures every frame using our own code instead of Unitys automatic scaling. Once the textures are properly scaled before being assigned to the cube map there are no issues.


Figure 4.6: The texture on the left has gone through the automatic compression whereas the texture on the right has not.

Some of the textures used for the cube map faces must also be rotated so that they are applied to the cube map correctly. A comparison of the difference between rotated or flipped textures can be seen in Figures 4.7 and 4.8.

Rotating and flipping some of these textures increases the accuracy of the system, at some computational expense. The textures which required rotation were the front, top and left sided face textures.


Figure 4.7: Here the texture assigned to the front of the cube map has not been flipped. It can be seen at the bottom of the sphere that the green cup is reflected onto the right hand side instead of the left hand side.


Figure 4.8: In this image the front texture has been flipped and can be seen to be correctly reflected on the right hand side.

## 4.6 Dynamic Division

The current division of the camera feed is performed without taking into account the marker position on the screen. This division is devised for an object rendered on a marker in the centre of the screen. When using this division, the reflection results for an object

rendered on a marker in a different location would not be very accurate. For example if a marker was placed on the right hand side of the screen, then the environment directly in front of the object is mapped to the right hand side of the object instead of the front of the object. In fact the environment directly to the left of the object will be set as the front face of the cube map, which is incorrect. This problem is illustrated in Figure 4.9.

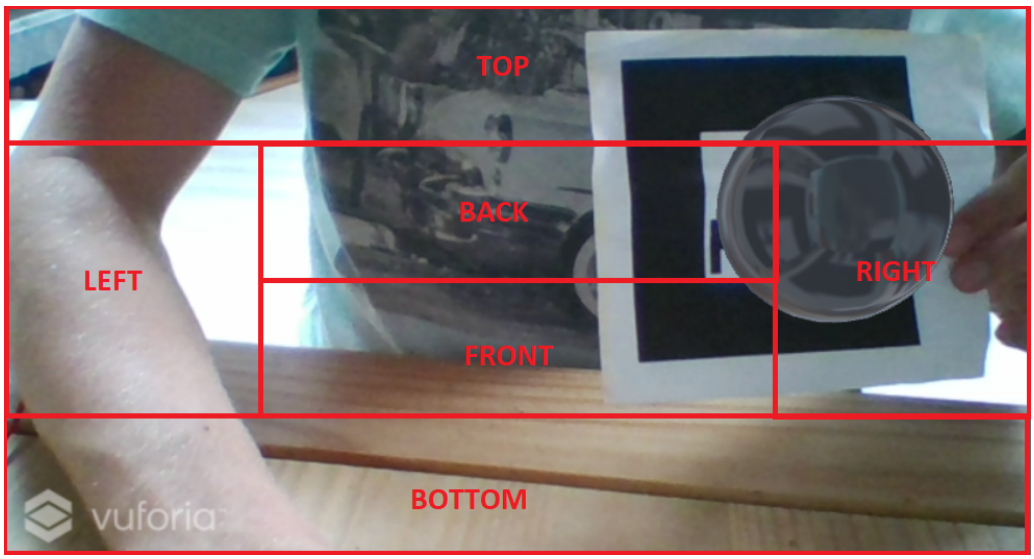



Figure 4.9: In this image the object is rendered on a marker on the right hand side of the screen. The problem is that the area in the middle of the screen is still marked as the front of the object, when ideally this area would be the left hand side and the right hand side would be the front face of the cube map.

To combat this issue, an algorithm whereby the marker position in terms of screen co-ordinates would be taken into account when dividing the screen into six parts was devised. This algorithm would read in the centre of the markers position in screen co-ordinates, then move the different sections accordingly. So if the marker is on the right hand side of the screen, all sections would be shifted over to the right. If a section goes "out of bounds" i.e. some of the section moves off the screen, then the algorithm moves the section back to the nearest valid location and this then becomes the appropriate face section. This method was chosen so we could maintain the current section dimension sizes which gave a very good result for an object rendered on a marker in the centre of the screen. Figure 4.10 shows a scenario where the centre of the marker is placed at a

location which is half way between the top and bottom of the screen, and three quarters of the way across the screen. In this scenario, the right, top and bottom sides would all be in invalid locations. The algorithm would then move these sections back to the closest valid locations, which in this case would be to move them a quarter of the screens width to the left, or in other words, move them so that their right-sided border is the same as the right-sided edge of the screen.

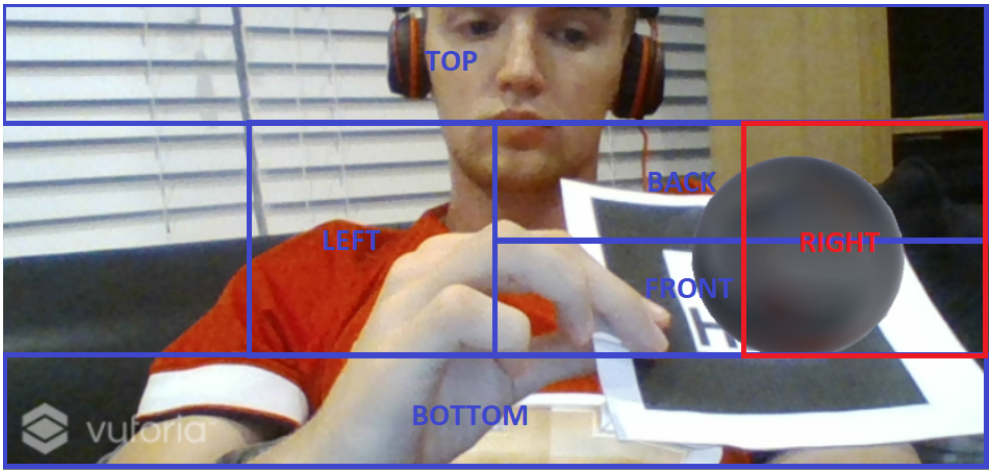



Figure 4.10: A scenario where the marker is rendered on the right-hand side of the screen. The division is adjusted by the dynamic division algorithm.

The dynamic division is handled via C# script. Below is an example of the code used to check if the right-hand section is in a valid position. In the script the values for oX and oY are the top left corner of the back section, these co-ordinates are used as a reference point. Again, w and h are the width and the height of the screen in pixels respectively. First the width and height of the section are set. Then xRight and yRight are found using oX and oY (xRight,yRight are the x,y co-ordinates of the top left corner of the right section). Then it is checked whether or not xRight plus rWidth is less than the screen width. If it isn't then xRight is moved to be rightW away from the right hand side of the screen (i.e. w - rightW). The process is then repeated in the vertical direction using yRight, rHeight and h.

```
rightW = w / 4;
```

```
rightH = h / 2;
//Check right locations
xRight = oX + w / 2;
if (xRight > w - rightW) xRight = w - rightW;
yRight = oY;
if (yRight > h - rightH) yRight = h - rightH;
```

This algorithm is used for each of the six sections, resulting in the locations of each section moving depending on marker position and all sections remaining in valid locations.

After this algorithm was successfully implemented the system implementation was finished for use on a PC with a webcam in the Unity Editor. The next step of the implementation was to try and create a usable Android build that would work on Android phones and tablets.

## 4.7 Android build

A build for Android was created using the settings and code that was used for the PC build in the hope that the system correctly functioning would translate across platforms. However, there were significant problems when the system was tested on an Android device. After the some debugging, and consulting the logs provided by the Android Device Manager program that comes with Android Studio, it was found that the problems seemed to be stemming from the background texture. It was found that the smaller textures that were to be used for the cube map faces were in fact, null textures. After some investigation, it was discovered that where the background texture was automatically set as readable on the PC version, this was not the case on Android. Hence the use of GetPixels was returning no information, which meant the system did not work at all.

After some research and further debugging a solution to this problem was found where a copy of the background texture is created every frame which can be set as readable. We still need to actually fill this copy texture with the background even though the original

texture is unreadable. This can be achieved using the function Graphics.Blit(), which essentially copies the texture without it having to be readable. Once this is done we can use GetPixels on the copied texture which allows the system to function as on PC.

The changed code can be seen in use here.

```
//Create a copy of the camera image in order to set it as readable
RenderTexture tmp = RenderTexture.GetTemporary(w, h, 0,
    RenderTextureFormat.Default, RenderTextureReadWrite.Linear);
Graphics.Blit(im2, tmp);
RenderTexture previous = RenderTexture.active;
RenderTexture.active = tmp;
Texture2D im = new Texture2D(w, h);
im.ReadPixels(new Rect(0, 0, w, h), 0, 0);
im.Apply();
```

The Android build also required some flipping and rotation, but once this had been performed the system ran the same as it did when using on a PC with a webcam.

# Chapter 5

# Results and Evaluation

## 5.1 Examples of the system

The following are some examples of the system under different conditions. Figures 5.1 shows a complex reflective model with no major light source. Figure 5.2 shows the same virtual object but this time there is a torch light which can be seen reflected on the object from the torch direction.


Figure 5.1: A complex model with no major light source nearby.

Figures 5.3 and 5.4 show the same model with a much less reflective material. It can be seen that the brightness or darkness of the virtual objects immediate environment is not portrayed on the object. This is because the reflections from the cube map are not shown due to the reflective properties of the material.


Figure 5.2: A complex model near a torch light.


Figure 5.3: A complex model in a dark environment.

## 5.2 Evaluation of performance on mobile devices

The system was implemented on a Samsung Galaxy A5 mobile phone. There were significant problems with frame rates on the mobile device. Most likely this was due to the computational intensity of the system, especially when the cube map resolution is high (above 128x128). Figure 5.5 shows an example of the system in use on the mobile device. Note that in this instance the rotation and flipping functions for the cube map faces has not been used to investigate if that is where the computational bottleneck lies. It is clear that the system is functional, but the significant frame rate problems mean that the system is not real-time. Because of this, it is not usable as we would wish since changes in the environment would take a long time to be reflected onto the object, which would not make the object seem more realistic or more integrated into its surroundings.


Figure 5.4: A complex model in a bright environment. This model uses the same non-reflective material as the model in Figure 5.3 so the two models appear the same despite the different environments.

## 5.3 Impact of cube map face resolution

An investigation was carried out to examine the effect of cube map face resolution on the overall quality of reflections and on the frame rate of the system. This experiment was carried out on a Dell Inspiron 15 5000 series laptop. The aim of carrying out this investigation was to see if there was a cut-off point between quality of result and speed at which environmental changes are picked up and shown on the object.

To do this, the system was run for an object rendered in a scene and the average frame rate over one hundred frames of video was recorded. This was carried out for resolutions of 2 by 2 all the way up to 512 x 512 for the same scene. The results of this can be seen in Figure 5.6.

Looking at the results of this experiment, there seems to be a significant cut-off in performance (in terms of frame-rate) for cube map face lengths greater than 64 (i.e. resolutions greater than 64 x 64). So it is ideal at this moment in time to use resolutions less than this cut-off if possible.

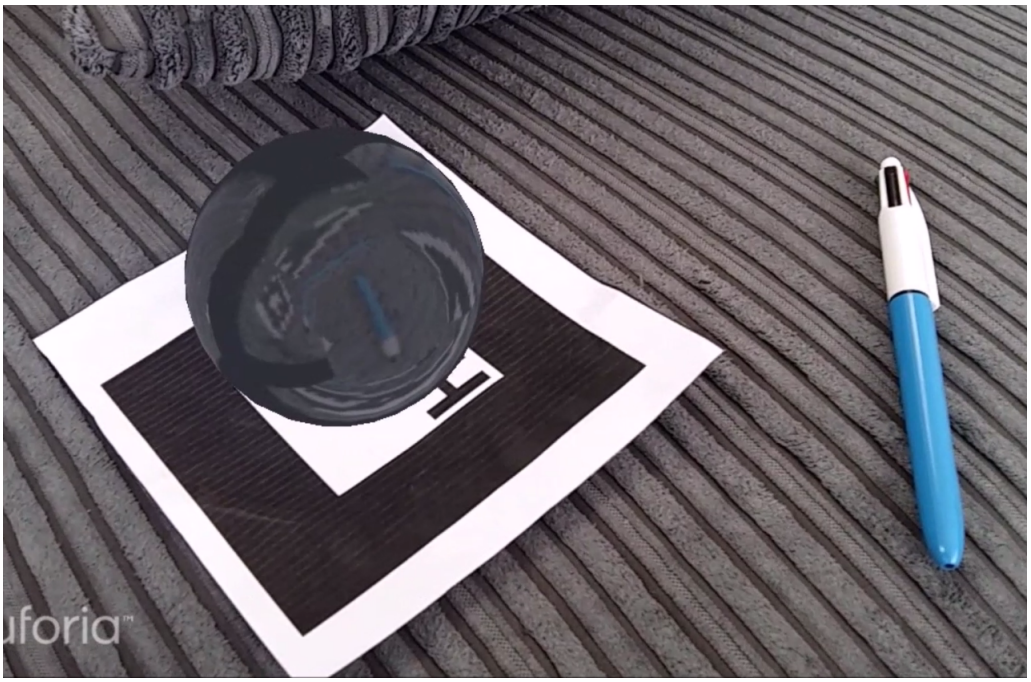



Figure 5.5: Example of the system in use on a Samsung Galaxy A5 mobile phone.

## 5.4 Comparison of different resolutions on quality of result

The quality of reflections for different cube map face resolutions was also investigated.

Figure 5.7 shows a sphere with a very reflective material attached to it. This material is used to demonstrate the exact quality of the reflections for the varying resolutions. It can be seen that the system provides very detailed reflection quality, even for resolutions as low as 16 x 16. This coupled with the previous experiment where it was found that large resolutions (greater than 64 x 64) can slow down the system show that using lower resolutions is definitely acceptable.

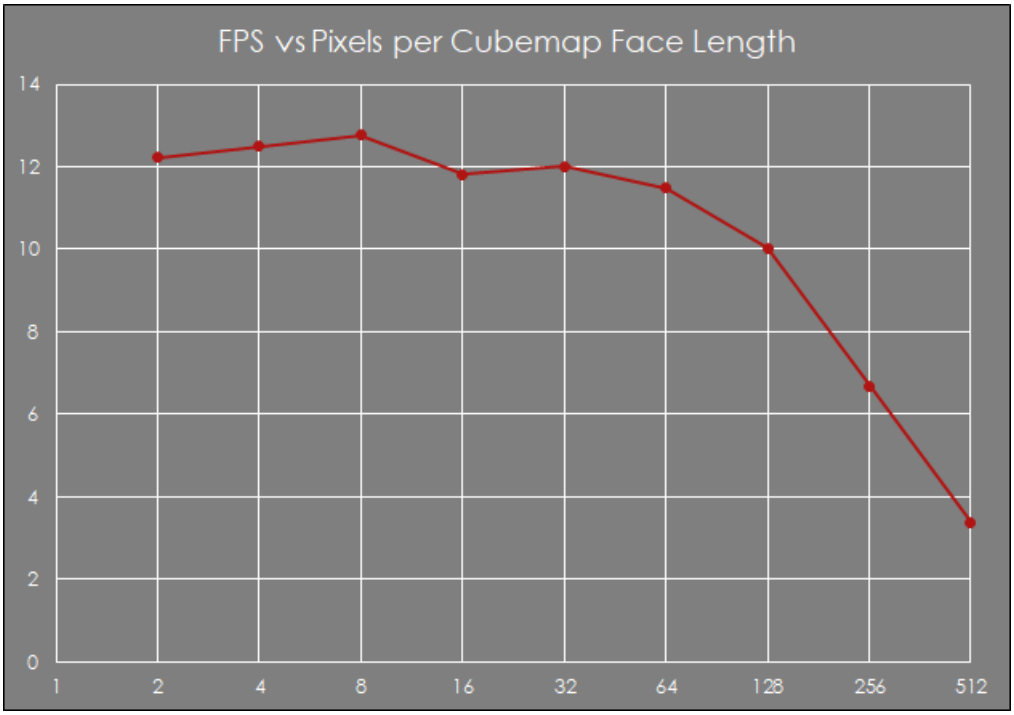



Figure 5.6: A graph displaying results of FPS vs Cube map Face Length (in pixels).

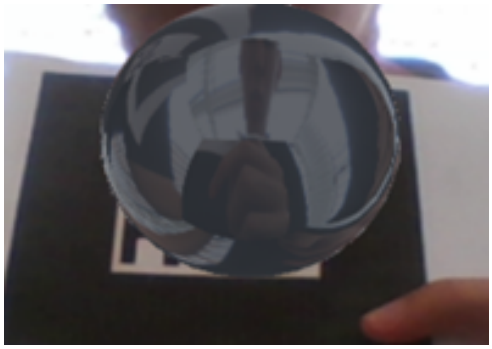
(a) 2 x 2

(b) 4 x 4

(c) 8 x 8

(d) 16 x 16

(e) 32 x 32



(f) 64 x 64

(g) 128 x 128

(h) 256 x 256

Figure 5.7: Examples of different cube map face lengths.

# Chapter 6

# Conclusion

The aim when beginning this project was to develop a system whereby a virtual object placed into a real environment is more seamlessly integrated into its immediate environment. This was to be achieved using data taken from the environment which would be reflected onto the virtual object, thus making the object truly seem immersed in its surroundings. I feel the goal of this project has been achieved. Throughout this dissertation most images which contain a virtual object contain one with a perfect reflective material, this was used to highlight the exact effectiveness of the system and the quality of the reflections that it produces.

However when the system is in use in the real world, it is intended for objects without this perfect reflective material. It is intended to affect the objects appearance depending on how reflective or non-reflective its material is. For example, an object with a completely non-reflective material would have very little or no effect from the system, but a system with a material some some reflective materials would have a greater reaction to reflections provided by the system. An example of the system in use for a more complex object can be seen in Figures 6.1 and 6.2. In Figure 6.1 the virtual object is rendered in a scene with no prevailing light source in any direction. In Figure 6.2 we see the same object placed in the same scene, except this time there is a light source coming from the mobile phone screen. The system assigns the area below the object to the bottom face of the cube map,

so we see the blueish-greenish light reflected onto the object. It seems to come from below the object, which shows the system working correctly and intuitively.

Figure 6.1: A complex object rendered on a marker.

There is definitely scope to further the project from the point I have reached here. Investigation into where the bottlenecks are in terms of the Android build is one obvious way to conduct some further study. It is highly desirable to have the system perform to the level it does on PC, due to the enormous potential of augmented reality for these mobile devices such as phones and tablets.

It also may be worthwhile comparing and contrasting different camera background divisions to the one I have chosen to use here. While the division used for the majority of the project provides good results, there may be possible divisions which give even better results, or divisions in which the sections are smaller than the ones used here. This would result in less per-pixels calculations and thus would make the system less computationally expensive. Having less per-pixel computations could also be the key to having the system perform in real-time on Android devices.


Figure 6.2: A complex object subject to light from below.

Another way to further the project would be to work on a rotation function for the cube map. In the current version the cube map faces always map to the same side of the object. For example, the front face will always map to the front of the object. This may not be ideal, as if the object is rotated such that the front of the object doesn't face the camera, then reflections from the front of the object will be mapped to whatever side the "front" of the object is instead of exactly to the front of the object. This issue could potentially be resolved using the rotation values of the marker, which can be accessed through the Vuforia prefab for the marker object.

In conclusion, the system showcased here does produce good results and has the potential to be used in a real world scenario. As augmented reality becomes a more integral part of everyday life, features that provide better integration for virtual objects into the real world environment will become more and more important. Using environmental reflections as we have here is just one of a multitude of ways to try increase the realism of augmented reality moving forward.

# Bibliography

[KS03] Hannes Kaufmann and Dieter Schmalstieg. "Mathematics and geometry education with collaborative augmented reality". In: *Computers and Graphics* 27.3 (2003), pp. 339–345. ISSN: 0097-8493. DOI: http://dx.doi.org/10.1016/S0097-8493(03)00028-1. URL: http://www.sciencedirect.com/science/article/pii/S0097849303000281.

[Hey+05] Sebastian Heymann, Aljoscha Smolic, Karsten Mfffdfffdller, et al. "Illumination reconstruction from real time video for interactive augmented reality". In: (Jan. 2005).

[RBW05] H. Regenbrecht, G. Baratoff, and W. Wilke. "Augmented reality projects in the automotive and aerospace industries". In: *IEEE Computer Graphics and Applications* 25.6 (Nov. 2005), pp. 48–56. ISSN: 0272-1716. DOI: 10.1109/MCG.2005.124.

[GEM07] Thorsten Grosch, Tobias Eble, and Stefan Mueller. "Consistent Interactive Augmentation of Live Camera Images with Correct Near-field Illumination". In: *Proceedings of the 2007 ACM Symposium on Virtual Reality Software and Technology*. VRST '07. Newport Beach, California: ACM, 2007, pp. 125–132. ISBN: 978-1-59593-863-3. DOI: 10.1145/1315184.1315207. URL: http://doi.acm.org/10.1145/1315184.1315207.

[PC15] Klen Čopič Pucihar and Paul Coulton. "Exploring the Evolution of Mobile Augmented Reality for Future Entertainment Systems". In: *Comput. Enter-*

*tain.* 11.2 (Jan. 2015), 1:1–1:16. ISSN: 1544-3574. DOI: 10.1145/2582179.2633427. URL: http://doi.acm.org/10.1145/2582179.2633427.

[Roh+15] K. Rohmer, W. Buschel, R. Dachselt, et al. "Interactive Near-Field Illumination for Photorealistic Augmented Reality with Varying Materials on Mobile Devices". In: *IEEE Transactions on Visualization and Computer Graphics* 21.12 (Dec. 2015), pp. 1349–1362. ISSN: 1077-2626. DOI: 10.1109/TVCG.2015.2450717.

[Sar] Samit Sarkar. "Pokemon Go hits 650 million downloads". In: (). URL: https://www.polygon.com/2017/2/27/14753570/pokemon-go-downloads-650-million.