REAL-TIME INVERSE LIGHTING FOR AUGMENTED REALITY USING A DODECAHEDRAL MARKER

By

GLEN K. STRAUGHN

Bachelor of Science in Computer Science

University of Science and Arts of Oklahoma

Chickasha, Oklahoma

2014

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2018

REAL-TIME INVERSE LIGHTING FOR

AUGMENTED REALITY USING A

DODECAHEDRAL MARKER

Thesis Approved:

Dr. Blayne Mayfield

Thesis Adviser

Dr. Thomas Johnson

Dr. David Cline

Name: GLEN STRAUGHN

Date of Degree: MAY 2018

Title of Study: REAL-TIME INVERSE LIGHTING FOR AUGMENTED REALITY USING A DODECAHEDRAL MARKER

Major Field: COMPUTER SCIENCE

Abstract: Lighting is a major factor in the perceived realism of virtual objects, and thus lighting virtual objects so that they appear to be illuminated by real-world light sources—a process known as *inverse lighting*—is a crucial component to creating realistic augmented reality images. This work presents a new, real-time inverse lighting method that samples the light reflected off of a regular, twelve-sided (dodecahedral), 3D object to estimate the light direction of a scene's primary light source. Using the light sample results, each visible face of the dodecahedron is determined to either be in light or in shadow. One or more light vectors then are calculated for each face by either using the surface normal vector of the face as a light direction vector if the face is in light, or by reflecting the face's surface normal across the normal vector of every adjacent illuminated face in the case of shadowed faces. If the shadowed face is not adjacent to any illuminated faces, the normal vector is reversed instead. These light vectors then are averaged to produce a vector pointing to the primary light source in the environment. This method is designed with special consideration to ease of use for the user, requiring no configuration stages.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

LIST OF MATHEMATICAL EXPRESSIONS

Expression Page

CHAPTER I

INTRODUCTION

Augmented reality (AR) is an increasingly prominent subject in the field of computer science. With the proliferation of smart devices, AR has become more accessible to the average person than ever before. Moreover, the success of applications such as Niantic's *Pokemon Go* [1] has established firmly the concept of AR in the public consciousness.

As with many graphics-focused subjects, improving the perceived realism of virtual imagery, often referred to as "perceptual correctness", is a prominent subject in AR research. One of the most important factors in producing realistic renders is lighting; therefore imitating real-world lighting conditions is crucial for realistic AR. As manually describing the lighting conditions for an AR application would be tedious and prone to human error, it is best to automate the process of lighting AR scenes. The process of gathering real-world light information from images is known as *inverse lighting* and is a common topic of research in AR.

When proposing a solution to this problem, one of the most important factors to take into consideration is how much is known about the environment. Some AR implementations know nothing ahead of time, while others have detailed information about the scene in which they are to function. More information typically grants better results, but often comes at the cost of either portability or the need for special equipment.

Presently, the vast majority of inverse lighting methods require special equipment to which the average person would not have access, do not calculate lighting in real-time, or are difficult to set

up. For this reason, this research focuses primarily on creating an inverse lighting system that works in real-time and requires minimal effort and investment on the part the end user.

CHAPTER II

LITERATURE REVIEW

Section 2.1 Augmented Reality

Augmented reality refers to the process of superimposing virtual-imagery over video of a real-life scene. There are many ways of determining where and when to place virtual objects: GPS coordinates, gyroscopic sensors, and—perhaps the most common method—fiducial markers. Fiducial markers—or markers for short—are objects that an application can recognize easily using computer vision algorithms and are typically used to determine where to place a virtual object. Most often, these take the form of 2D images, appropriately called 2D markers. Theoretically, any image can be used as a 2D marker so long as it has a unique appearance from every viewing angle, though black and white images are most common, as the stark contrast between their features makes them easier to read. The process of identifying and cataloging the visual attributes of a 2D image for use as an augmented reality marker is known as marker training.

Alternatively, markers can take the form of a three-dimensional object. Often, a 3D marker is composed of several 2D markers that each form the face of a cuboid. That said, not all 3D markers are made from 2D markers. For example, human faces are well-suited for use as fiducial markers [2]. One of the common reasons for using a 3D marker over a 2D marker is to increase

the range of viable viewing angles, as 2D markers become difficult to read when viewed at shallow angles.

2.2 Inverse Rendering

As with any topic within computer graphics, one of the most prevalent subjects of research in augmented reality is how to create photo-realistic images. Unlike other graphical applications, whose rendered images exist in a vacuum, AR programs must render virtual objects over pictures or video of a real-world scene. This poses a unique challenge as virtual objects in AR appear next to real objects, thus making any visual incongruities between the two readily apparent. For this reason, it is useful to deduce information about the physical environment from the program's input. The process of making such inferences is known as *inverse rendering*.

Inverse rendering itself can be subdivided into multiple categories. This research will focus on the process of gathering information about real-world lighting conditions: a process fittingly called *inverse lighting* (IL).

Inverse lighting methods vary greatly in approach. Some methods choose to use little, if any, information about the deployment area. Others require meticulous recreation of the deployment area by modeling the scene's topography virtually, thus creating "phantom objects" that interact with the visual elements of the scene (shadows, occlusion, etc.) but are not themselves rendered. As shown in some of the referenced works, some techniques are capable of gathering such data about the surrounding environment at run-time using specialized equipment.

One of the most prevalent IL approaches is Image-Base Lighting (IBL). IBL uses a panoramic image of the surrounding environment, most commonly referred to as an environment map, from which light information can be extracted.

2.3 Previous Work

In 2004, Kanbara and Yokoya [3] presented an influential approach to inverse lighting. Their method used a standard black and white 2D marker with a small, shiny, black marble placed in the center of it. The program started by finding the marker, then used the position and rotation of the marker to find the location of the marble, which would be constant relative to the marker. After the marble was located, the computer would find any highlights on the surface of the marble. Then for each highlight it would determine the normal of the sphere at the highlight's location and calculate the angle of reflection, thereby locating the light source that created the highlight. This information then would be stored in a light map for use during rendering.

Two years later, Supan, Stuppacher, and Haller [4] used mirrored spheres, sometimes known as gazing spheres, for image-based lighting in AR. Using the reflection on the surface of the gazing sphere, the computer was able to make a 180º environment map.

Their article described three different camera configurations used in the study. The first involved having the AR marker and gazing sphere at a fixed distance from one another. The second used two cameras: one to look at the marker and another to look at the sphere. Finally, the third also had two cameras but—instead of a gazing sphere—it simply equipped the second camera with a fish-eye lens and gathers the environment map directly.

After obtaining the environment map, the program blurred it to obtain an irradiance map and converted the maps from a spherical projection to a cubic projection. To extract lighting information, the environment maps were then down sampled; a light was placed at each texel (texture cell) and set to the color of the texel. Shadows were cast using standard shadow mapping techniques. To save on processing time, however, only one shadow map was updated on each frame.

Figure 1: Configuration two of three [4]. The main viewing camera is focused on the marker (the black and white texture next to the sphere) while a second camera observes the reflection on the mirror ball

At about the same time, Madsen, Jensen, and Andersen [5] also explored the potential of IBL for more realistic outdoor augmented reality. The method they proposed had a number of stipulations: it only worked outside, during the day, with the sun as the major light source, with no precipitation, with diffuse surfaces in the scene, and the program needed a rough 3D model of the environment. During calibration, the user manually captured an environment map of the scene and then identified the diffuse surfaces in the image. The intensity of the sun and indirect lighting then was determined by cross-referencing the amount of light hitting the diffuse surfaces and their corresponding 3D geometry. Unlike many image-based lighting systems, the program was capable of changing lighting conditions in real-time, though it required a pre-calculated "albedo map" of the environment—a special environment map which stores the diffuse colors of the surrounding environment—as well as a normal map to do so.

The team also place a significant amount of focus on the program's rendering pipeline. Shadows were handled via shadow volumes, which are cast by the virtual object and by phantom objects

imitating the scene's physical geometry. The scene's phantom objects were also used to occlude virtual geometry.

A year later, Madsen and Laursen [6] would further investigate the applications of IBL for augmented reality, with special consideration to aspects of lighting such as shadows. Their program started by obtaining a high dynamic range (HDR) environment map. It then used the median cut algorithm to locate light sources in the environment map, with shadows being handled by shadow mapping. Like Madsen's previous work, the program also made use of phantom objects that modeled real-world geometry. These objects occluded virtual objects, received shadows, and cast shadows on virtual objects.

The program also provided a workaround for the double shadow problem. The team chose to think of shadows as an absence of irradiance. This allowed them to divide areas of the environment map affected by real-world shadows by the irradiance term of the real-world shadows, which were calculated using the simulation's phantom objects. This effectively gave the diffuse albedo for those surfaces, which then allowed the program to apply the virtual shadows and real-world shadows via shadow mapping.

Finlayson, Fredembach, and Drew [7] proposed a means to identify which regions of an image were illuminated by which light source. Their program started with two images taken with a chromagenic camera, one normal and one with a filter applied. The lighting in the two images then was compared and contrasted to create a group of possible illumination relationships. These relationships then were compared to each other to find the best fit for each pixel.

In 2008, Nakano, Kitahara, and Ohta [8] researched what qualities make shadows believable in mixed-reality scenes, with an interest in finding ways to simplify light-source models without degrading perceived quality. The study included a variety of tests. The first test required volunteers to manipulate a virtual light until the shadow cast by a virtual cone matched that of a

real cone. The second involved changing the distance between two cones until the shadows were perceivably different.

The last and most involved test required volunteers to identify artificial images. The artificial images were rendered using image-based lighting; the light map was captured with a 180º fish-eye lens camera. As part of the test, the AR images were rendered with six different light map resolutions. The highest resolution was 2,048×2,048 pixels and the smallest was 8×8 pixels. Six images of a real object next to an artificial object were made, one for each resolution. The test consisted of two of the six images being selected randomly and shown in two second intervals to the subject. The subject then would decide which of the images was real by rating them on a five-point scale from negative two to positive two, with negative two representing certainty that the object on the left was real and positive two representing a high degree of confidence that the object on the right was real. Subjects made a total of fifteen comparisons. The team found that the images using the highest four light map resolutions scored almost identically. The images that used lower resolution maps than those, however, were far less convincing, with noticeable decreases between resolutions.

In 2010, Aittala [9] developed an inverse lighting pipeline for augmented reality based on the illumination found on diffuse surfaces. After his program detected the marker, the user would calibrate the AR scene's virtual lighting by using a plain, white ping-pong ball. The program used the illumination on the surface of the ping-pong ball to determine the direction and intensity of the real-world light sources. Alternatively, the user could simply rotate the marker in front of the camera to acquire lighting data.

A ping-pong ball was used for two reasons: it had a known geometry and the surface of the average ping-pong ball has a matte finish. Because the item was a sphere, determining the surface normal at any given point was a very simple and easy process, not to mention it would not be affected by the ball's orientation toward the camera. The matte finish was useful as highlights

on a shiny surface (specularity) are affected by the observer's viewing angle, while light coming off of non-shiny surface (diffuse lighting) is not affected by viewing angle, again simplifying inverse lighting calculations.

In the same year, Noh and Sunar [10] discussed how to generate soft shadows in AR. Their method utilized an image-based lighting scheme by using a reflective black sphere placed in the center of the marker as a light probe, much like Kanbara and Yokoya [3]. The program then would perform the median cut algorithm on the region of the screen occupied by the sphere to determine the light locations.

The technique the team tested was a soft shadow method by Heckbert and Herf. It involved overlapping multiple hard shadows of varying opacity to produce the appearance of a soft shadow. Ultimately, the team determined that, while it did produce higher quality shadows, the trade off in performance made the method impractical.

Pessoa *et al.* [11] proposed a number of rendering techniques to increase the realism of augmented reality. Their method started with the standard image-based lighting approach of capturing images of the surrounding environment for use in an environment map. Unlike many approaches, however, this map was not used globally for all virtual objects in the scene. On every frame, each virtual object would calculate four environment maps for its own use, making sure to include other virtual objects, including phantom objects, in the render. These individual environment maps were blurred in different amounts to match the intended material properties of the object.

Pessoa and company also proposed extensions to the standard bidirectional reflection function (BDRF). These included tangent rotation maps, which were 2D monochromatic textures that would store angles of rotation around the surface normal of an object. These were used to calculate anisotropic surface effects. The second addition was an adjustment to the calculation of the Fresnel component of an object's specular component.

Jensen *et al.* [12] implemented an AR method that used a reflection-sphere marker to gather light data and conducted a test to determine how believable their rendering method for shadows was. The marker they used was a standard 2D *ARToolkit* marker with a ping-pong ball covered in glossy, black paint placed in the center. The marker would track virtual objects, while the highlights on the surface of the sphere provided information about the surrounding lights. To interpret this data, they used the median-cut algorithm to zero in on the highlights and then used the sphere's normal at that point to project the light onto an environment map.

The second half of this study was focused on rendering perceptionally-correct shadows based on the information gathered from the first step. The team's method involved layering multiple shadows over each other and applying various visual effects to the shadows, such as blurring. Lastly, the authors recounted how they tested the realism of these shadows, namely through an online survey. Humorously, they found that the online participants where quite skeptical, with even the unaugmented control images only scoring 60% acceptance. After the survey, the team concluded that 64 overlapped shadows was the lower limit of what it took to create a convincing image.

In 2011, Jiang, Schofield, and Wyatt [13] proposed an algorithm to detect shadows in single images. Their method would start by performing a color segmentation algorithm. Afterward, the program made illumination maps for multiple light frequency bands and orientations. The program then used the color segmentation buffer and the illumination map to construct a shadow edge map. To save time, the algorithm would only check color segment edges to find shadow edges. Interestingly, the team used not one but three color spaces for this step: RGB, HSL, and LAB. Each of these were calculated at four different scales, resulting in a 36-dimensional matrix to be solved per pixel. Finally, once the shadow edge map was constructed and some additional refinement steps, the program labeled each pixel as either “shadow” or “not shadow.”

That same year, Chen, Wang, and Jin [14] created a means to estimate 3D geometry and lighting conditions from a single image with no prior knowledge of the scene geometry or lighting. The scene's 3D geometry would be estimated by running the image through one of two algorithms: one based on edge detection or one based on linear regression. Both of these algorithms were created via Markov training over sets of images with similar geometry. At the same time, the program decomposed the image into a reflective component and a shadow component. Using the virtual geometry, reflectance image, and shading image, the algorithm then constructed a sparse radiance map. The sparse radiance map would consist of *M* lights placed at regular intervals inside a hemisphere, each light having a different intensity to reproduce the lighting conditions in the original image. With all of this, the virtual object then would be rendered into the scene.

In the year 2012, Arief, McCallum, and Hardenburg [15] proposed a light direction estimation algorithm utilizing 3D cuboid fiducial markers. The team emphasized optimization of the algorithm to operate efficiently on mobile devices. The algorithm worked by using edge detection to identify the marker's shadow. Once the shadow was identified, the program located the corners of the shadow. Combining the pose data from the markers, the program then matched the corners of the shadow to the corresponding corners of the marker. With this information, the program then could determine the direction of the scene's major light.

The method they proposed had some limitations, however. First, it assumed the shadow was being cast on a flat surface. Second, the authors stated they were not able to perform shadow analysis on every frame and instead only checked the shadows on initialization. Third, the implementation the team used could only estimate one light source. Finally, the system did not work under certain lighting conditions, such as when the light source was very close to the marker.

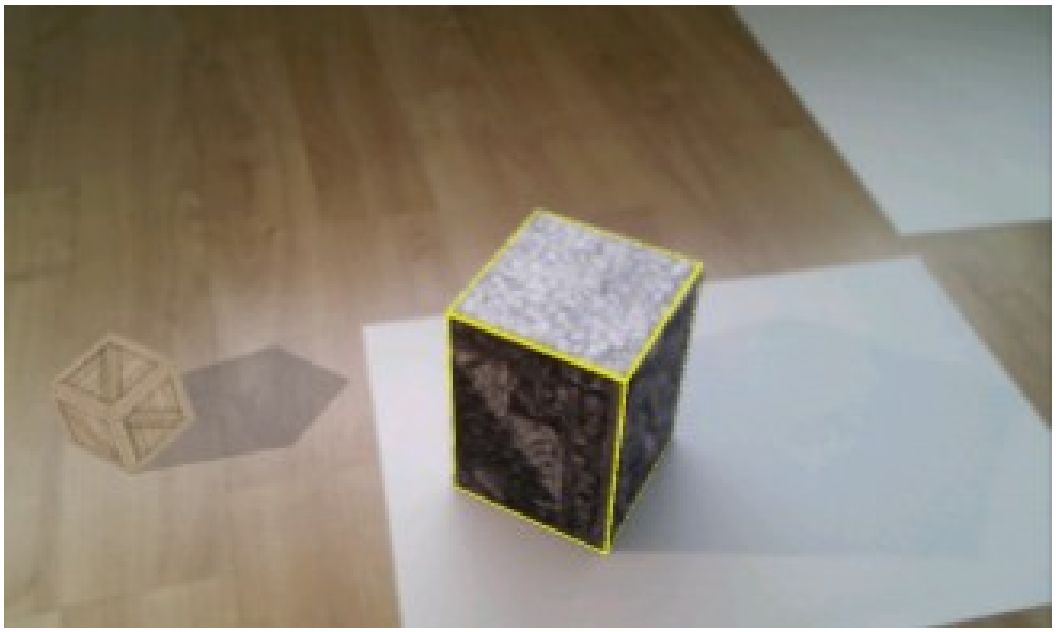

Figure 2: Image of Arief et al.'s program in action [15].

In 2014, Gruber *et al.* [16] discussed an augmented reality method that could calculate radiance transfer in real-time without the aid of a scene probe (e.g. a mirror ball). The algorithm used a camera with a depth sensor, in this case an XBox Kinect, to gather not only visual information, but data on the scene geometry. Once the program had a virtual representation of the real-world geometry, the color information from the image was compared with the real-world geometry to determine the lights' position in the scene. This was accomplished via simple Lambertian reflection calculations. The Depth information also was used to determine whether a virtual object should be occluded by a real-world object in the scene. One of the more noteworthy abilities of this system was the ability of users to cast shadows on virtual objects by moving their hands between the object and the light.

Kamboj and Liu [17] experimented with a variance cut algorithm for use in augmented reality, as opposed to the more commonly used median cut algorithm. As the name implies, the variance cut algorithm is much the same as the median cut algorithm, only it splits the image so that each subsection’s variance is minimized.

The authors' method began with obtaining an image from a camera with a fish-eye lens that had been pointed upward. This image served as a 180° environment map. The program then performed variance cutting on the image to determine the locations of the scene's lights. In the team's implementation, the light data then was exported to a script to be read into *3DS Max 2014*.

Michiels *et al.* [18] presented a novel augmented reality system based on omni-directional video. The research team used their own custom-designed, omni-directional camera, which they mounted to the top of a car. The custom camera was composed of six regular cameras, each of which were oriented in 60º intervals, making for a 50% overlap between cameras. Because the device captured footage in all directions, making an environment map to use for image-based lighting was trivial. The scene's primary light source was identified using the shadows in the surrounding environment. Visibility of virtual objects was handled using voxel cone-tracing.

To determine the car's position, and thus track without the aid of markers, the algorithm used feature tacking. Feature tracking was applied to the input from each camera individually. The results then were filtered via bundle adjustment to obtain the car's trajectory. This stage was completed offline.

Rohmer [19] detailed a mobile augmented reality system that offloaded some operations to a desktop computer. The system was primarily intended for use indoors. It used image-based lighting, though instead of utilizing an environment map captured ahead of time or performing an initial scan of the surrounding environment on startup, it captured the environment map in real-time using multiple HDR cameras fitted with fish-eye lenses.

To prevent latency issues, most of the work was performed on the mobile device. The PC interpreted the environment data captured by the cameras and would relay the lighting data to one or more mobile devices. The PC also handled radiance transfer calculations. Using models of the scene geometry, which were constructed ahead of time, the PC projected the radiance from the surrounding environment onto the geometry and would register the results into a “radiance

atlas," which tracked both direct and indirect radiance. Direct radiance was simulated by inserting colored point lights into the scene and indirect radiance was computed via a pre-calculated radiance transfer function.

Borg *et al.* [20] documented their study on what factors are necessary to achieve perceptionally-correct AR. Some of the factors and techniques tested included shadows, highlights, silhouetting, and the addition of visual noise to the rendered image to simulate the visual noise produced by web cameras. For their experiment, the team used marker-based AR to track the position of the virtual object. Rendering and tracking were handled using the *Unity* game engine and the third-party *Vuforia* plugin. The virtual objects to be projected, a mostly diffuse-surfaced object and an object with shiny highlights, were virtually reconstructed from photographs via an *Autodesk* function. The lighting for the scene was captured via a camera fitted with a 180º fish-eye lens and stored in an environment map. The light positions then were calculated offline and stored in a text file.

The test had volunteers view images of the AR scene for four seconds; the volunteers then would determine whether or not the image they were shown was real. There were two types of test images: ones where only virtual objects were present and ones where a virtual object and a real object where in the same image. The test originally intended to let users rotate the camera around the scene but rotating the view point made the virtual objects jittery, ruining the illusion. Instead, the objects where viewable from one of three predetermined angles, with the transition between the points not being shown to the volunteers.

Ultimately, the tests revealed that a lack of visual noise does reduce the perceptual correctness of an AR simulation, though it usually took test subjects multiple viewings to notice. The team also noted that shadows in particular are effective in convincing observers of an AR image's authenticity. Specifically, the test results indicated that shadows with rendering artifacts—especially around their edges—acted as dead giveaways of an object's virtual nature.

Interestingly, scenes that used a high amount of ambient light, and by extension had less discernible shadows, were less likely to be identified as real by test subjects, even when the object on screen was real.

Knor and Kurz [2] described a real-time inverse lighting system that used a human face as both a fiducial marker and a light-probe. As the authors explain, faces are well suited for this task for a multitude of reasons. Firstly, faces have a relatively consistent geometry between individuals. Secondly, faces have enough surface detail to perform inverse illumination from almost every angle. Lastly, using a human face to extract lighting information from video does not require any sort of specialized cameras.

The process started with an offline stage in which pictures of faces under different lighting conditions were fed to a learning algorithm to produce a model for the radiance transfer function of the average face. Using this function, the program could estimate light direction color and intensity in the input image without a calibration stage. It could also track the user's face, allowing projected virtual objects to follow the user's head movement.

In 2015, Alhajhamad, Sunar, and Kolivand [21] proposed an algorithm to gather light information from images of indoor scenes. Captured images first were converted from RGB to HSV color space and then converted to black and white using Otsu's threshold algorithm. The black and white image then was filtered using a Gaussian blur to provide smoother approximations. Once the image was filtered, the program performed a contour detection algorithm to locate areas of shadow. Light direction was determined by comparing a detected light's center of mass to the center of mass of the shadows in the image to obtain a vector. Light intensity was determined by comparing the angle between shadows. To determine which shadows were cast by which lights, the program assumed the light closest to each shadow was the caster.

The algorithm proved to run quickly on the test hardware, maintaining above 40 frames per second. Moreover, the method provided accurate estimations even immediately after the lighting

in the scene changed. The authors did note, however, that room size could affect the accuracy of their method.

Einabodi and Grau [22] implemented a new technique for augmented reality. Their method used four cameras: one with a fish-eye lens to make an environment map and three to act as “witness cameras” to help calibrate the simulation. After the initial calibration, which included determining the witness cameras' orientation relative to the main camera, the lights' positions would be triangulated and given a location in 3D space. The primary direction of the lights were also found via least-square optimization, more specifically the Levenburg-Marquardt method.

Kán, Unterguggenberger, and Kaufmann [23] detailed an image-based lighting scheme intended for mobile devices. The program would start by having the user perform a full 360º scan of the surrounding area in both yaw and pitch with his or her phone's camera. Using the visual input in conjunction with the phone's gyroscopic sensor, the program then would generate an environment map and map it to a spherical projection. Next the BDRF was used to calculate multiple reflectance maps to produce visually consistent reflections. Lastly, the augmented image could be enhanced with ambient occlusion. Unlike the other steps, however, ambient occlusion had to be calculated offline ahead of time for the sake of performance. According to the authors, the setup process in its entirety took an average of 52 seconds.

Buteau and Saito [24] presented a method to extract the positions of point lights from images of indoor scenes. It used a Kinect to capture the environment geometry of both the image and the real-world. The program then split the image's diffuse and specular components. It used the RANSAC (random sample consensus) algorithm to project the diffuse component onto a plane and then produced an initial estimation of light locations and direction. The specular component was used to create a second set of estimates. These two estimates then were converted into a point cloud and the DBSCAN (Density Based Spatial Clustering of Application with Noise) algorithm would be applied to the point cloud to create the final set of lights. While the system

worked admirably, the research team noted that the last step could be finicky when used on large rooms and, as a result, did not always produce quality results.

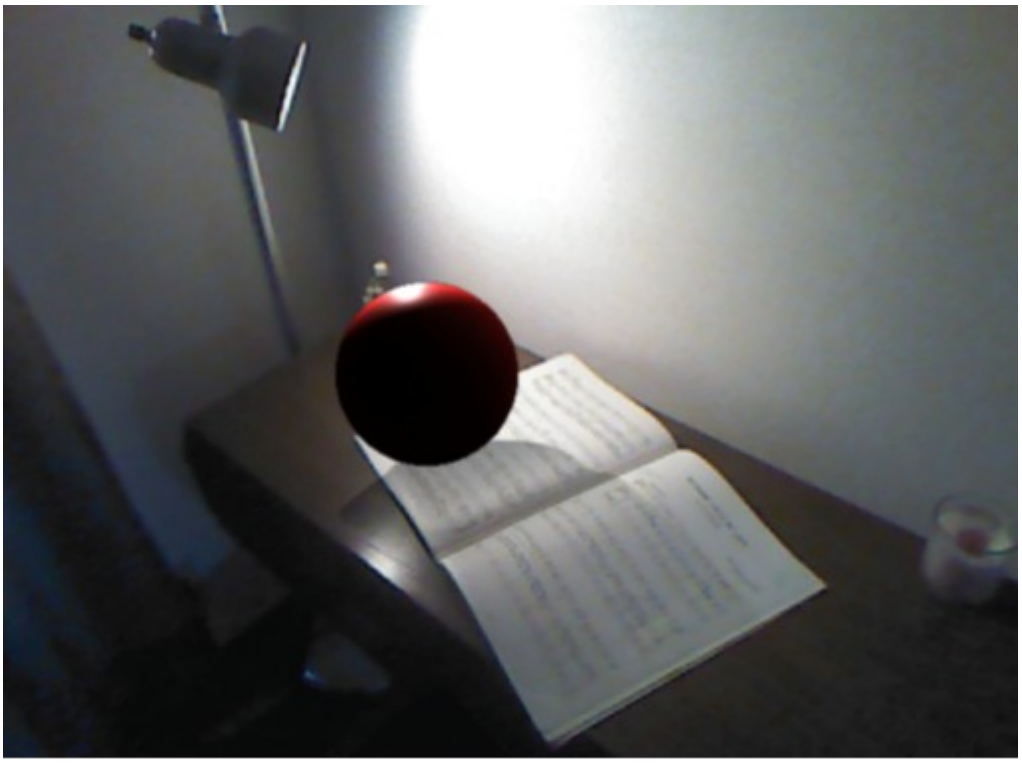

Figure 3: Image of Buteau and Saito's method detecting radiance from a wall [24].

In 2016, Soulier, Selzer, and Larrea [25] created a solution to make a low-cost light sensor for augmented reality. The team's method involved using eight light-dependent resistors (LDR) attached around the circumfrence of a hemisphere in 45º intervals with an additional one situated on top of the hemisphere. These LDRs were wired into an Arduino UNO Micro-controller, which in turn was connected to a PC application that forwarded the light data to the mobile platform running the AR program. The AR program itself ran on the mobile distribution of the Unity game engine and the Vuforia plugin for Unity. The team concluded that the results were encouraging, but cited the need for a PC to act as a middleman between the phone and Arduino board as cumbersome.

In 2017, Kasper *et al.* [26] described a new method they developed to estimate light sources. They focused specifically on physically-based rendering methods such as path tracing. The algorithm started by generating a 3D virtual geometry of the scene in question. This was accomplished via a depth camera and a triangulation algorithm. Next, the surface albedos of the objects in the scene were assigned to the mesh vertices corresponding to the same location in screen space. Any holes in the mesh were ignored. An environment map was used to establish the lighting conditions around the scene. To sample lights, ray tracing was performed on a per-pixel basis. Lastly, the weights of the lights sampled-per pixel were adjusted with sequential Monte Carlo filtering.

CHAPTER III

PROPOSED SOLUTION AND THEORY

Section 3.1: Review of Problem to be Solved and Overview of Proposed Method

As previously discussed in section 2.1, augmented reality (AR) is the process in which one or more virtual objects are placed into an image or a video feed of a real-world environment. One common goal in the development of AR programs is to make the insertion of virtual object as seamless as possible. To achieve this, virtual objects need to be affected by the same environmental factors as the surrounding physical environment so as to minimize noticeable visual incongruities. For this reason, being able to gather information reliably about the lighting conditions of a real-world environment, a process known as inverse lighting (IL), is a critical first step in creating immersive AR.

This research focuses on developing a dynamic, real-time solution to the inverse lighting problem. The proposed method is composed of three primary stages: marker tracking, luminance sampling, and light direction estimation. It assumes there is one dominant light source in the scene, and produces a direction vector pointing in the direction that the dominant light is shining in.

In the marker tracking stage, an image is obtained from the camera and processed to find a 3D dodecahedral marker (DM). The position and rotation of the DM—also known as its pose—are

then extracted. This pose information then is used by the luminance sampling and light direction estimation stages.

In the luminance sampling stage, the program takes color samples from each face of the DM. The luminance value of each color sample then is calculated and the values are averaged to get an average luminance value for each face. The light estimation phase compares the luminance value of each face to a threshold value to determine whether a face is illuminated or in shadow. The light estimator then generates light vectors for each visible face, and then averages the results.

These processes will be elaborated on in more detail in the following sections.

Section 3.2: Dodecahedral Marker and Tracking

As with many augmented reality systems, the method developed by this research utilizes marker-based tracking, more specifically 3D marker tracking. The 3D marker used in this research takes the form of a regular dodecahedron composed of twelve 2D markers, one on each face. Each 2D marker is unique and can be used to differentiate faces from one another. Lastly, the faces of the DM are made of a matte material, as specularity would complicate both tracking and light estimation.

Section 3.2.1: Reason for Using a Dodecahedral Marker

Typically, 3D markers are used to increase an AR system's versatility. Unlike a 2D marker, a 3D marker can be viewed from any angle, thus greatly increasing the robustness of the simulation. As useful as this property is, it is merely a secondary factor in the context of this research. In this research, the DM is not only a marker, but a light probe as well. In general, a flat 2D surface only reveals how much light is coming from the direction it is facing. By increasing the number of observable surfaces, however, the amount of lighting information available increases. Moreover, by arranging the surfaces in a configuration in which their spacial relationships to one another are known, the information gathered from them can be compared to make further deductions. As such, a 3D marker makes for a feasible light probe.

The choice of a dodecahedron was motivated by a similar line of reasoning. While it is standard to make 3D markers cubes or cuboid objects, cuboids have the unfortunate property of being able to be viewed in such a way that only one face is visible at a time, thus eliminating the aforementioned advantage of 3D markers. For this reason, a regular polyhedron with a greater number of faces was chosen. The dodecahedron was chosen as being, more or less, a reasonable middle ground between a cube and an icosohedron. As an added bonus, fitting readable AR markers on the pentagonal faces of a regular dodecahedron is less challenging than the triangular surfaces of a regular octohedron or icosohedron.

Section 3.2.2: Tracking

To track the DM, the program obtains an image from a web-camera and uses marker identification on it to find the 2D markers. Once a face marker is identified, its pose relative to the camera is determined using AR software. This pose information is only for one face of the DM, however, and not the DM itself. Because the position of each face remains constant relative to the DM, the offset each face has from the center of the DM can be represented as a four-by-four transformation matrix, which in turn means the pose of the DM relative to the camera can be found by applying the inverse transformation of the face's offset to the pose of the face. The formula for this is given as:

$$P_{DM} = F_C F_{DM}^{-1} \tag{3.1}$$

where

- P_{DM} is the transformation matrix of the dodecahedral marker relative to the camera,
- F_C is the transformation matrix of one of the faces of the DM relative to the camera, and
- F_{DM}^{-1} is the inverse transformation of the face relative to the center of the DM.

Obtaining the pose of the DM is important for two reasons. First, when placing objects into a scene, the DM provides a better anchor point than any individual face, as a face may or may not

be visible at any given time. Secondly, for instances when a face theoretically should be visible but is not detected by the AR system, the DM pose can be used to locate where on the screen the face should appear. This second factor is especially useful in the luminance sampling phase.

Section 3.3: Luminance Sampling

After the DM has been located, the program can begin gathering information about the surrounding lighting conditions from its surface. Before such information can be gathered, however, the reflective properties of the surface in question must be taken into consideration. The two basic types of light reflection are diffuse and specular. Diffuse, (sometimes referred to as Lambertian) surfaces are surfaces that are rough and thus scatter light that hits them in every direction. Specular lighting occurs on smooth and glossy surfaces where light is reflected in a singular direction dependent on the heading of the incoming ray and the normal vector of the surface.

To simplify lighting calculations, the majority of inverse lighting techniques assume the surface being sampled is purely one or the other. For this research, the surface of the DM is assumed to have a Lambertian surface as that is the surface most conducive for marker-based AR.

Section 3.3.1: Deciding From Where to Sample

When and where to gather color samples starts with determining the visibility of each face of the DM. As to be expected, if the marker on one of the DM's faces is detected, then that face is visible and the transform obtained from it can be used without any further processing. Sometimes, however, a marker that should be clearly visible is not detected by a system's AR software. Finding the pose of any particular face relative to the camera is as simple as reversing the operations used to obtain the DM's pose. As mentioned previously in section 3.2, the formula for obtaining the pose of the DM is given as

$$P_{DM} = F_C F_{DM}^{-1} \tag{3.2}$$

So to obtain a face's pose relative to the camera, P_{DM} is multiplied by the offset matrix of the face, F_{DM}.

$$P_{DM} F_{DM} = F_C F_{DM}^{-1} F_{DM} \tag{3.3}$$

Multiplying a matrix by its inverse produces an identity matrix

$$P_{DM} F_{DM} = F_C I \tag{3.4}$$

Which simplifies to

$$F_C = P_{DM} F_{DM} \tag{3.5}$$

Once of the pose of the undetected face is obtained, it then is multiplied by a perspective matrix to account for the effects of perspective in the actual image. The program then checks if the angle between the face's normal vector and the program's view vector is within acceptable bounds. This is accomplished by taking the dot product of the face's normal and the view vector. Because $A \bullet B = \|A\|\|B\|\cos(\theta)$ and both the normal and view vector have a magnitude of one, the dot product of the two gives the cosine of the angle between them. With this information, it is possible to tell if the marker is facing toward the camera and by how much. It is not enough for a side to be facing the camera, however, as meaningful information can be very hard to glean at very shallow angles. For that reason, the undetected face is only sampled if the cosine is above an a predetermined threshold, as specified in the program's configuration file. For ease of reference, the process of locating an undetected face and sampling from it will be henceforth referred to as *projected sampling* and the process of sampling from a detected face as *direct sampling*.

After all of the visible faces are accounted for, the program can proceed with using the pose information to sample from the faces of the DM. Samples are taken from predetermined points on the faces, the coordinates of which are described relative to the original image the 2D marker was trained from. For simplicity's sake, it was decided that the X and Y coordinates of a sample

point would be bounded on the interval [-1.0, 1.0], with -1.0 mapping to the left and bottom edges and 1.0 being the right and top edges of the image. As to be expected, the point (0, 0) corresponds to the center of the marker.

To sample from a point, the point's 2D coordinates are converted to a 4D vector of the form $[x\ y\ 0\ 1]^T$, assuming a Z-up coordinate system. The *W* element of 1 indicates the vector is a point in space and not simply a direction. This enables the point to be multiplied by the perspective matrix and the 4×4 pose matrix of the point's corresponding face, thus making the point's position relative to the coordinate system of the DM face. The formula for this operation is given as

$$S_C = PF_C S_F \tag{3.6}$$

where

- S_C is the sample point relative to the camera,
- P is the perspective matrix,
- F_C is the pose of the face relative to the camera, and
- S_F is the four element sample point relative to the face (i.e. the source image).

Once the position of the point is determined in the AR simulation's arbitrary coordinate system, the 3D coordinates of the simulation must be converted to 2D screen coordinates so that color values can be extracted from the current camera image. The process of obtaining the screen coordinates begins with obtaining the image plane coordinates by using the perspective divide for both the point's x and y values:

$$S.x_{Image\ Plane} = \frac{S.x_C}{S.z_C} \quad \text{and} \quad S.y_{Image\ Plane} = \frac{S.y_C}{S.z_C} \tag{3.7 \& 3.8}$$

Next, the image plane coordinates are converted into texture coordinates, which range from (0, 0) in the lower left corner to (1, 1) in the upper right corner. This is accomplish via the following operations:

$$S.x_{Texture} = \frac{S.x_{Image\,Plane} + 1}{2} \quad \text{and} \quad S.y_{Texture} = \frac{S.y_{Image\,Plane} + 1}{2} \qquad (3.9 \text{ \& } 3.10)$$

Finally, the pixel coordinates are calculated using the following formula:

$$S.x_{Pixel} = S.X_{Texture} * w \quad \text{and} \quad S.y_{Pixel} = (1 - S.y_{Texture}) * h \qquad (3.11 \text{ \& } 3.12)$$

where

- w is the camera frame's width in pixels, and
- h is the camera frame's height in pixels.

Note that the y coordinates need to be inverted due to the fact that pixels in bitmap images are typically indexed from top left to the bottom right instead of the bottom left to the top right.

Section 3.3.2: Sampling From the Captured Camera Frame

Once a pixel coordinate is obtained, the program obtains the RGB color value from that location. During sampling, the value of each channel is converted from a one-byte range (0-255) to a four byte float value in the range [0, 1]. The luminance value of this sample then is calculated using the perceptional luminance function provided by the *GLM* library, which is given as

$$L = 0.33r + 0.59g + 0.11b \qquad (3.13)$$

where

- L is the luminance,
- r is the red channel,
- g is the green channel, and
- b is the blue channel.

Unfortunately this luminance value cannot be used in its current state. Even when fully illuminated, different colors have different luminance values. For example, using the above formula, one can determine a fully illuminated, purely red sample would have a luminance value of 0.33 while a fully illuminated, purely green sample would have a luminance of 0.59. Ergo, to determine how illuminated a face is, the luminance values must be normalized. This is done by dividing the obtained luminance value by the maximum luminance value at that point. The maximum value is calculated by taking a color sample from a reference image of the marker at the sample point and calculating the luminance. The reference image is assumed to be fully illuminated.

After every viable point is sampled and the resulting values are normalized, each face's sample values are averaged and stored for use in the light estimation phase.

Section 3.3.3: Reasons for Using Predetermined Sample Points

Predetermined sample points were chosen over random sampling for a number of reasons. To use random sampling, the program would need to sample not only the camera frame, but the reference image as well to obtain the maximum luminance value. This would double the number of samples required per point. This also would mean that the reference image for each face would need to be read into the program, which in turn means an import function would have to be written for image files. Another potential pitfall is if the program randomly selected a black region from which to sample. Because the maximum luminance value of a black region is 0, the program would have to randomly select another region to prevent a division by 0. Likewise, if the sample region were near the border of two regions of differing colors, the program may end up sampling from the wrong region. This could result in samples that were improperly normalized, thus throwing off the accuracy of the final result. While the effects on performance would probably be negligible, it was decided the potential decrease in accuracy and increase in workload made random sampling was more trouble than it was worth.

Section 3.4: Light Estimation

Once the pose information of the DM and the faces' average luminance values are gathered, the program finally has everything it needs to begin the light estimation phase. In this phase, the aforementioned information is used to estimate a primary light vector, which is to say a vector that points in the direction that the environment's dominant light source is shining. The light estimator also finds the primary light source's intensity as well as the intensity of the environment's ambient light, which is to say light that has reflected off surfaces in the environment and illuminates all sides of the objects in the environment.

Section 3.4.1: Preliminary Steps

The core idea of the light estimation phase is to generate a list of light vectors, one for each visible face of the DM, and find the weighted average of them. Before these vectors can be derived, however, the program needs to determine which faces are illuminated (i.e. receiving direct light) and which are shadowed (i.e., receiving only ambient light). This is important because illuminated faces and shadowed faces use different methods of deriving preliminary light vectors. This distinction is made by comparing the face's average luminance to a threshold value. Attempts were made at setting the threshold value dynamically based on the overall illumination of the DM, but this proved difficult in practice due to a variety of reasons (see section 5.1 for more details). For these reasons, the threshold is set ahead of time in the program's configuration files.

Next, the program needs to know the normal vector of each visible face. Fortunately, it already has this information. One useful (and woefully under-reported) property of 4×4 rotation matrices is that (assuming column-major order) their first three columns, when converted to vectors, are the transform's local X, Y, and Z axes respectively [27]. So, with that in mind, the normal vector of a face is simply the local "up" vector of its pose. For this research, that means the pose's third column, as *ARToolkit* uses a Z-up coordinate system for its markers. Lastly, it should be noted

that this information is retrieved after the perspective matrix is applied to the pose so that the normal is orthogonal to the face as it is perceived by the user.

Section 3.4.2: Calculating Light Vectors

Both the illuminated face and shadowed face light vector generation algorithms operate under one basic assumption: the more a side faces a light source, the more direct light will hit it, and thus the more illuminated it will be.

Generating light vectors for illuminated faces is a fairly straight-forward process. If a face is determined to be illuminated, then the preliminary light vector is simply the face's normal vector multiplied by its luminance value. Multiplying the normal vector by the average luminance value for that side weights the resulting light vector so that more illuminated face's have a greater contribution than less illuminated faces.

Generating a preliminary light vector for shadowed sides is not quite as simple. If a face's average luminance is less than the threshold, it is assumed that the face is not receiving much, if any, direct light, which means it is not facing a light source. For this reason, the normal cannot be used as the face's preliminary light vector, as that would orient the final primary light vector toward the shadowed face. Instead, the program must orient the light away from shadowed faces.

To create the preliminary light vector for a shadowed face, the program checks the average luminance value of all of the sides adjacent to the shadowed face. For each illuminated face, the program reflects the normal of the shadowed face across the normal of the illuminated face. The reflected normal vectors then are averaged and the result is normalized, producing the unweighted preliminary vector for that face. If none of the faces adjacent to the shadowed face are illuminated, it is assumed that the light is on the opposite side of the DM and the preliminary light vector defaults to the negation of the shadowed face's normal vector.

Next, the program must determine the weight for the preliminary light vector. Going back to the assumption stated at the beginning of this section, the darker the face, the further it is rotated away from a physical light source. Therefore, how dark the face is relative to the rest of the DM can be used to weight the vector. Using the difference between the luminance value and the illumination threshold is not sufficient, however, as that would not take into consideration factors such as ambient light.

Even when one side of an object is facing away from a light source, the surface is illuminated by a small amount of light that has reflected off of other surfaces and diffused throughout the environment. This diffused light is known as ambient light. Because ambient light hits surfaces even when they are not facing a light source, the ambient light should be treated as a constant factor when examining the lighting on the DM. Therefore, the scene's ambient light intensity is regarded as the lowest possible light value a face can have. This means that if a face's average luminance is equal to the scene's ambient intensity, the face has the highest possible relative darkness. With that in mind, relative darkness is given as:

$$d = 1 - \frac{l_f - a}{t - a} \tag{3.14}$$

Where

- d is the relative darkness,
- l_f is the average luminance of the face,
- a is the ambient intensity,
- t is the illumination threshold, and
- $1 > t \geq l_f \geq a$.

The above formula produces a relative darkness value within the range [0, 1]. The domain of this function is [a, t]. Finally, d is inversely proportional to l_f, meaning greater average luminance values correlate to smaller relative darkness values.

Once the relative darkness is determined, the preliminary light vector for the shadowed face is weighted by it.

When all of the preliminary light vectors have been calculated and weighted, the program averages them together and normalizes the resulting vector to get the primary light vector.

Section 3.4.3: Primary Light Vector and Ambient Intensity

Direction is not the only significant attribute of light. There are other factors to consider for an AR simulation's lighting, namely the intensities of the primary light and the environment's ambient light. The former is easy to estimate with the information already available. To find the intensity of the primary light, the program selects the greatest average luminance value from the currently visible faces.

In section 3.4.2, the nature of the ambient light term was determined to be the lowest light value possible for a visible face. Similarly, to find the primary light intensity, the ambient light intensity is in principle the lowest luminance value. Unfortunately, there are a few complications. Unlike illuminated faces, the markers of which are detected easily, unilluminated faces can be difficult for the system to spot, and thus hard to gather data from. The lowest luminance value obtained may very well be above the illumination threshold. This is problematic when simply using the naive approach of using the lowest luminance value, as rendered images with high ambient light tend to look washed out and flat, thus breaking immersion. To work around this, the maximum value the ambient can be is 0.5. If the lowest average luminance value is above this threshold, the program assumes no unilluminated faces are visible and the ambient intensity defaults to 0.3.

CHAPTER IV

RESEARCH METHODOLOGY

Section 4.1: Implementation

The first step in testing is implementing the proposed method. This involved writing a program that performs light estimation using the technique detailed in chapter 3, deciding how to represent the information it will be using, and what software was used to create the program.

Section 4.1.1: The Program

The test program was written in C++, using the C++ 11 specifications. The program was compiled and debugged using Microsoft Visual Studio 2013.

AR tracking was handled by *ARToolkit* [28], an open-source AR library commonly used in AR research. *ARToolkit* was chosen both for its low cost (free) and for its high portability (an uncommon trait for AR API's), which allowed greater flexibility when determining both the development and testing device. This is important, as being able to run the program natively on the development device, in this case a *Windows* PC, not only eliminated the need for additional hardware, but also decreased the layers of abstraction when attempting to debug the program, thereby expediting development.

The render code was written using modern, shader-based *OpenGL* [29], with the shaders themselves being written in GLSL (OpenGL Shader Language). Additionally, the program makes use of the *OpenGL Extension Wrangler* library [30], a utility library that manages the

loading of *OpenGL* extensions and helps avoid system-specific compatibility issues. *GLUT*, the *OpenGL Utility Toolkit*, [31] was used to create and manage the application window and handle keyboard input. The program utilizes the OpenGL Mathematics library (GLM) [32] for its numerous vector and matrix calculations.

The program uses YAML files to track settings and scene data. This greatly reduces the time it takes to change variables in the program, as all of the important settings are grouped in human-readable files. Parsing the configuration file on the C++ side of things is handled by the *YAML-CPP* [33] library. Specifics on the configuration files are given in section 4.1.2.

Section 4.1.2: Threshold or NFT Markers

One of the features of *ARToolkit* is the ability to select between two types of 2D marker tracking: threshold-based markers and natural feature tracking (NFT) markers. Originally, the program was intended to use NFT markers due to their more robust tracking capabilities once detected. Unfortunately, NFT markers proved to be ineffective, as detecting them in the first place proved to be a very error-prone process that often would require fine tuning depending on the program's resolution (for more details see section 5.1). Ultimately, threshold-based markers were decided upon for their ease of use and being far less resource-intensive to detect and track.

As the name implies, threshold-based marker detection uses a thresholding algorithm to convert the camera image to a monochromatic image. This can prove to be problematic for a program that has to work under a variety of lighting conditions, as the pattern of the marker can be lost if the face of the marker is too dark or too light. To work around this issue, the test program scans the camera frame multiple times with the threshold being adjusted each time to improve the chance of the scene's markers being detected. How many passes to perform, the starting threshold value, and the amount to increment the threshold after each pass are all obtained from the program's configuration file.

Section 4.1.3: Data Files

As stated in section 4.1.1, YAML files are used to store program settings and locate data files. The settings file includes parameters for the tracking phase (e.g. error tolerance) and light estimation (e.g. the value for the shadow threshold). It also lists the other files used in the program, such as 3D models and shader programs. A second YAML file contains metadata for all of the fiducial markers in the scene. The metadata includes what type of marker the AR pattern is (threshold-based or NFT), a unique name to identify the marker, and the inverse offset matrix of the marker.

In addition to the YAML files, the program also uses text files to define sample points. There is one sample point description file which specifies the offset matrix for each face of the DM and denotes the location of another text file that lists sample point coordinates and maximum luminance values. These files are simple text files instead of YAML because this feature was added before it was decided to incorporate YAML.

3D meshes are stored using the Open Game Exchange (OpenGEX) file format [33]. OpenGEX is a text-based file format intended for use in video game development. It was chosen for its emphasis on avoiding the bloat common of other file formats and its openly-available specification, both of which make parsing easier. Moreover, it plays nicely with OpenGL by listing all relevant vertex information in a set order, an issue encountered when trying to use Wavefront OBJ files. Custom parsing functions were written for importing the mesh information into the program.

Finally, AR markers are defined in their own text-based file format. This format is proprietary to *ARToolkit* and is generated using tools that come with the package.

Section 4.2: Tools

The hardware used in testing includes a PC, a webcam, a regular dodecahedral object to serve as the DM, a light source, and a non reflective surface on which to place the DM. The test platform

is a desktop computer that runs *Windows 10*. It has a 3.7 GHz, 64-bit Intel i3 processor; an Nvidia GeForce GT 430 graphics card; and eight GB of RAM. The webcam is a 720p Logitech USB camera. The regular dodecahedron is a large, foam, 12-sided die, approximately 10 cm in diameter. AR markers were printed on card-stock and attached to the face of the die via double-sided tape. The light source is a desk-lamp. The non-reflective surface is a 28×22 inch (approx. 71×56 cm) piece of black poster board.

Section 4.3: Experiment Design

When evaluating the effectiveness of the proposed IL method, this research was primarily concerned with two quantitative metrics: the accuracy of the estimation and how efficiently that estimation is calculated. Calculating the accuracy was done by finding the angle between the estimated light vector and the real-world light vector from the light to the DM, with an angle of zero degrees being the best possible result and 180º being the worst. For each experiment, the average relative error and percentage error was tabulated.

As is typical of real-time graphical applications, the efficiency the method was measured in frames per second (FPS). FPS refers to the number of render cycles a program performs in one second. The target FPS was 30. FPS was tested by running the program at different resolutions and measuring the FPS over a period of ten seconds and recording the average. The resolutions used in testing were: 640×480, 800×600, 960×720, and 1280×960. Frame rates were gathered both for when the DM was and was not within view of the camera.

Tests were conducted to evaluate the method's accuracy depending on three different variables. These conditions were the orientation of the DM relative to the camera, the light source's position relative to the DM, and whether or not the surface of the DM was strictly black and white or gray-scale. To test the first two, the lamp was placed two feet (approx. 61 CM) from the DM and rotated around it in 45° intervals. For each positioning of the lamp, the DM was orientated in one of two ways: viewing the DM so that one side is directly facing the camera, with six faces visible

(henceforth known as “viewed straight” or “straight”), and viewing the DM so that an edge is facing the camera, with only four faces visible (henceforth known as “viewed turned” or “turned”). (See Figure 4.) Ten sample vectors were gathered and then averaged for each light position and DM orientation. The camera was not moved during testing.

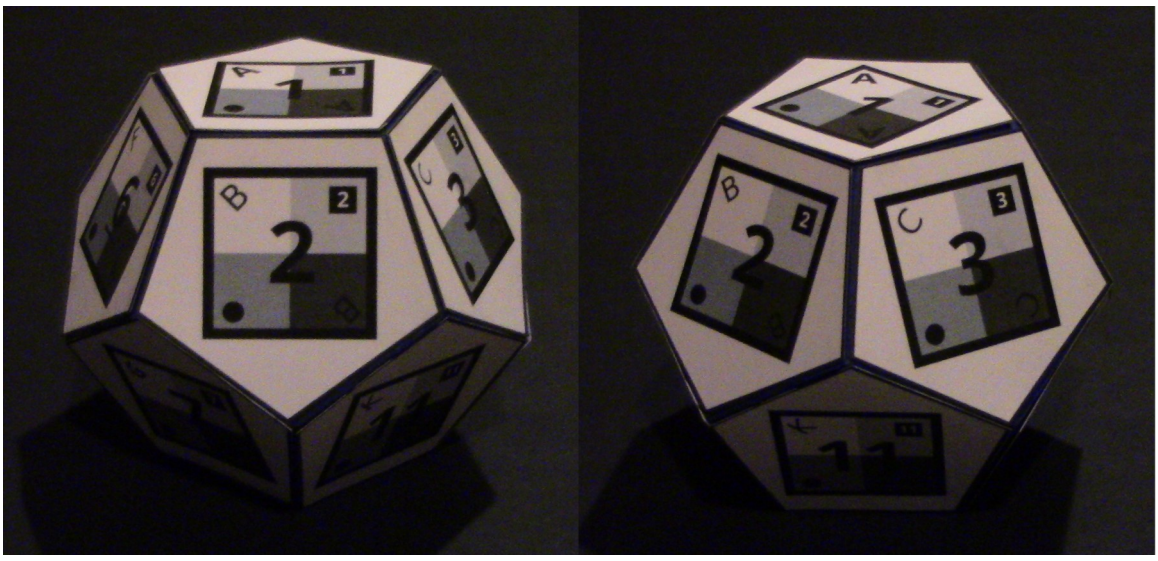

Figure 4: The DM viewed straight (left) and the marker viewed turned (right).

Lastly, the aforementioned tests were performed twice with different 2D marker sets to observe the effects of different maximum luminance values on the surfaces of the DM. One test consisted of sampling only from points on the marker that are white. The second set of tests had the program sample from markers that were divided into four quadrants, with the top-left corner being white, the top-right being 75% gray, the bottom-left being 50% gray, and the bottom-right being 25% gray (See Figure 5).

Each phase of the test begins by calculating the physical light vector. To do this, the position of the lamp’s bulb relative to the DM is regarded as a set of Cartesian coordinates, with the center of the DM being the coordinate system’s origin. The X, Y, and Z elements of the light vector are obtained by measuring the length between the DM and the lamp’s light-bulb along each axis. This vector is normalized to get the physical light vector (PLV).

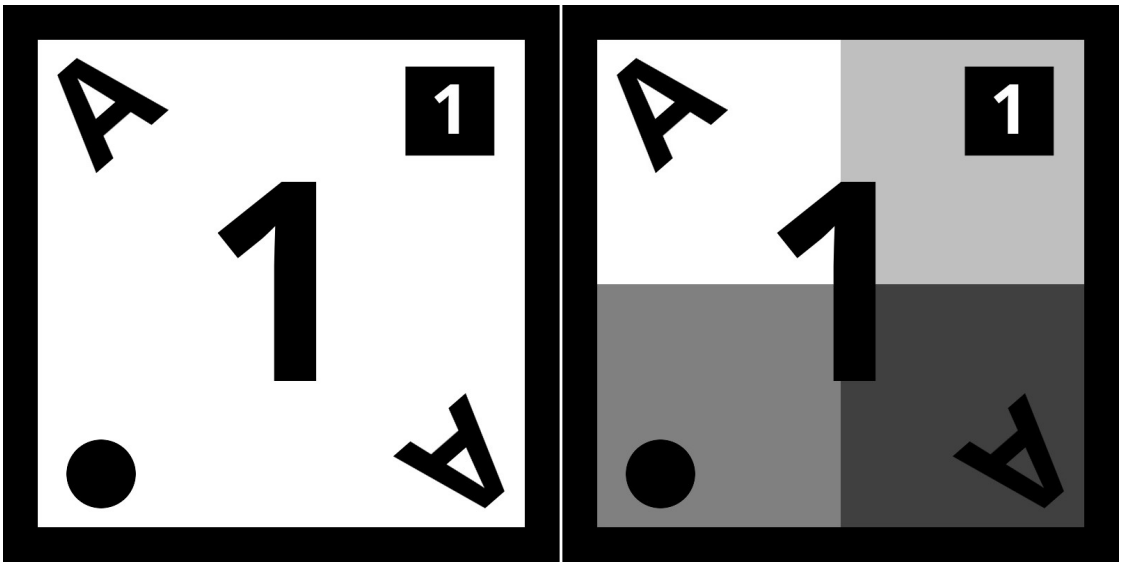

Figure 5: Both versions of the side 1 marker.

Unfortunately, the PLV cannot be directly compared to the light vector in the program. The coordinate system of the program is oriented relative to the camera and not the arbitrary Cartesian system of the experiment, meaning the light vector generated by the program is relative to the camera as well. To account for this, a 2D "world" marker was placed in front of the camera and aligned so that it was square with the imaginary axes of the experiment. The marker's transformation matrix then was extracted and the inverse of which was found. The light vector then can be padded out with zeros to make a 4D direction vector and that then is multiplied by the inverse matrix to find the light vector in world space. Because the light vector is a direction vector and not a position vector, the position of the world does not affect the outcome.

Figure 6: The world orientation marker centered and aligned with the testing area. In this case the world marker is one of the default markers that comes with ARToolkit.

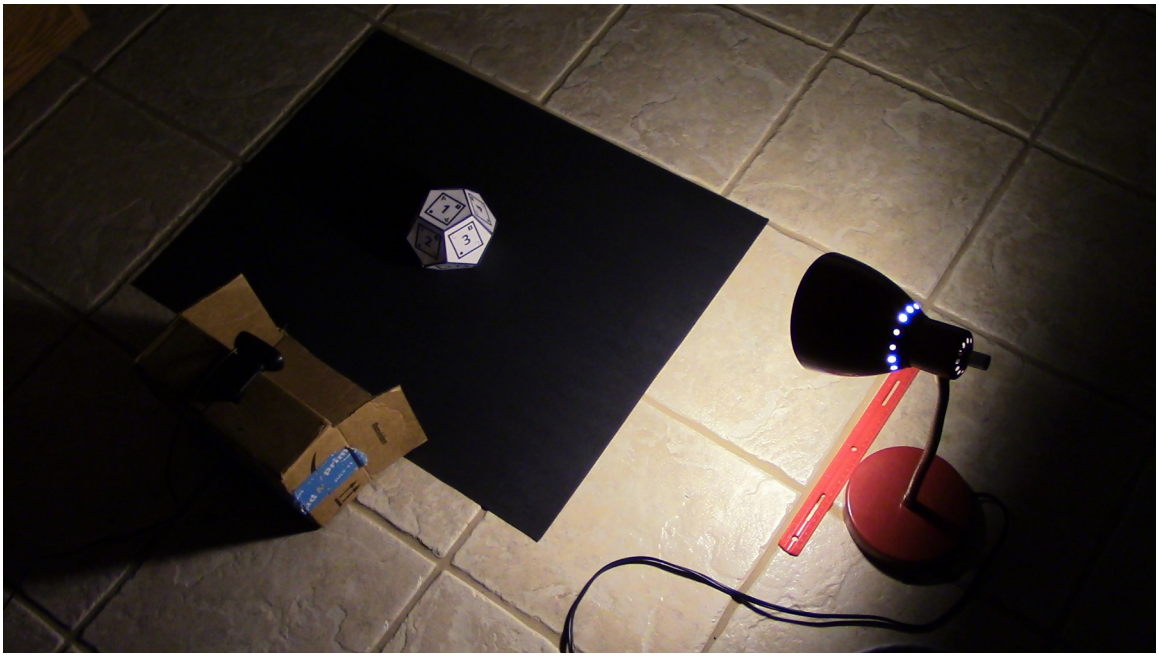

Figure 7: The experimental set up. The web camera is mounted on top of the cardborad box.

CHAPTER V

OBSERVATIONS AND ANALYSIS

Section 5.1: Notable Issues Encountered During Research

Several issues were encountered during the development of the test implementation. One of the most persistent issues when writing the test program was *ARToolkit*'s insufficient documentation. Many of the functions and structures have no description of their purpose or how to use them. While *ARToolkit* does come with several example programs to examine, sifting through source code to determine the proper usage of a function is a tedious process and is far from foolproof.

Another issue encountered when dealing with *ARToolkit* is its assumption that the developer is using fix-function pipeline *OpenGL*. Attempts to integrate modern *OpenGL* (i.e. shader-based) created conflicts with *ARToolkit*'s background rendering function, which takes the image provided (typically the most recent image obtained from the camera) and writes it to the render buffer. Fortunately, despite *ARToolkit*'s render function not working, the function to retrieve a frame from the camera still functioned properly. In the end, a new shader-based background manager had to be written from scratch.

As stated in section 4.1.2, the program was originally intended to use natural feature tracking markers. NFT markers were originally considered for a number of reasons. First of all, NFT is more robust than the standard threshold-based tracking. Once the initial pose is determined, the program can track the marker even if part of it is obscured or the marker is bent slightly. NFT is

also less sensitive to differences in light and shadow; threshold markers may stop working if a shadow is cast on them.

Unfortunately, there were several limitations to *ARToolkit*'s NFT system. Firstly, it is significantly slower than threshold-based tracking, making it much more difficult to get the marker to register in the first place. Secondly, it is much harder to set up and implement than threshold-based tracking: about two or three weeks of the test program's development (and three or four flowcharts) were dedicated to deciphering the spaghetti code of *ARToolkit*'s NFT example project.

ARToolkit's implementation of NFT also has the peculiar quality of being lower-left justified, meaning the pose obtained from NFT markers is centered on the lower-left corner of the marker. This proved problematic when attempting projected-sampling, as it was very difficult to get a projected face to align with its corresponding physical face.

The final nail in the coffin for NFT was its difficulties with multi-marker tracking. The method this research produced requires the ability to detect multiple markers simultaneously, or at the very least—assuming projected-sampling works properly (see previous paragraph)—the ability to distinguish between individual markers. Sadly, *ARToolkit*'s NFT does neither. NFT results produced by *ARToolkit* do not contain any sort of ID for the marker detected. More over, even when NFT markers are registered to completely different handles, *ARToolkit* is very likely to confuse two markers for one another.

Part of the problem stems from how error values are handled for NFT results. For threshold markers, the error value *ARToolkit* produces is bounded on the interval [0, 1] with one being the utmost confidence in the result. *ARToolkit*'s NFT system behaves completely differently: an error rating of zero represents high confidence in the result and the error score has no upper limit. This would not be too problematic if it weren't for the fact that NFT scores are incredibly

inconsistent depending on factors such as lighting conditions and image resolution, meaning setting an error tolerance value is not feasible.

Tests have shown that threshold markers can be mistaken for one another by *ARToolkit* as well, but the fact that the error scores are not only bounded, but also largely unaffected by image resolution and lighting conditions means the error score produced by *ARToolkit* is consistent enough that an error tolerance value can be used to discard false positives. For this research, the program was set to discard any result with an error score less than 0.65.

Another library specific issue was GLM's matrix decomposition function. The test program decomposes the transformation matrices obtained from *ARToolkit* so that the major elements of the pose (translation, rotation, and scale) can be manipulated individually. GLM provides a matrix decomposition function in its experimental extensions, but halfway through development it stopped working as intended. The rotation obtained from the function was reversed, meaning that every time the marker was rotated to the left, the virtual object would rotate to the right and so on. The cause of this error was never found, as even using older versions of the code, where the bug originally was not present, exhibited the errant behavior. Fortunately, matrix decomposition is not a difficult process to implement and a custom decomposition function was written.

A more general design issue was determining how to set the illumination threshold during the light estimation phase. The general assumption was that the average luminance values of each face would be bimodal, with unlit faces' and illuminated faces' average luminance values being clustered near their respective groups. As stated in section 3.4.1, the threshold was originally intended to be set dynamically. This was intended to help account for low-light conditions.

Several options were considered, using the mean, median, or finding the halfway point between the highest and lowest luminance values. The mean and median were dismissed early on, as an imbalance in the number of dark and lit faces could skew the threshold too far in one direction,

thus labeling faces that are ostensibly illuminated as dark and vice versa. For this reason, the half-way method was initially chosen for determining the threshold. An issue with this method, however, is if no low-light face is detected, then the program would set the threshold too high. This meant there had to be a cutoff point for how bright a face could be for it to be considered viable for use in the threshold calculation, which ultimately defeated the point of attempting a dynamic threshold in the first place. As such, the idea was abandoned and the illumination threshold is now specified in the configuration file.

Section 5.2: Experiment Results

Section 5.2.1: Frame Rate Results

The results of the frame rate tests were quite promising. Under all testing configurations, the program managed to exceed the 30 FPS target frame rate. The number of frames per second for when the DM was not in frame remained a consistent 60 across almost all configurations, up to and until the highest resolution of 1280×960, at which point it dipped down to an average FPS of 53.6. The average frame rates for when the DM was in frame were lower across all test cases. Much like before, the frame rates had roughly the same value for the resolutions below 1280×960, averaging 54.77 FPS, after which the FPS dropped sharply to an average frame-rate of only 38.5. The average difference in performance that having the DM in frame was 14%.

In an effort to better understand which process was consuming the most resources, an additional test was run. The previous tests were all conducted with four passes of marker detection. This last test was run at the highest resolution used in the previous test, 1280×960, but was configured to only perform a single pass over the input image. This dramatically improved the frame-rate. While only performing a single tracking pass, the program ran at a steady 60 FPS regardless of whether the DM was present. Taking this into consideration, it is fairly safe to conclude that most of the program's processing time is spent on the tracking phase, and that light estimation only takes up a fraction of the time spent calculating each frame.

An odd trend in the data is that there is a slight increase in the frame-rate as the resolution increases from 640×480 to 960×720. This holds true regardless of whether the DM was in frame or not, though it was far less pronounced when the DM was not in view of the camera. This is somewhat counter-intuitive, as one would assume that more pixels means more data to process. One possible explanation for this trend is that the image the camera captures is of a set size and the program has to down-sample the camera-frame to the resolution that the program outputs to, adding a time-consuming extra step on each frame. The smaller the resolution used by *ARToolkit*, the more the program has to downscale the image, and thus the more time this step takes. Alternatively, it is possible that the issue lies in the rendering function, as the window across all testing conditions was set to 800×600, meaning image scaling was occurring during the display routine. It may also be some combination of the two.

It should be noted that the frame rate is capped at 60, which would explain why this effect is only noticeable when the marker is within view. This is most likely a function of *GLUT* or the OS, as the frame rate that was hard-coded into the program was 120.

Section 5.2.2: Light Estimation Results

Unlike the frame-rate tests, the results of the light estimation tests were mixed. On average, the estimated light vectors had an absolute error of 67.48°, which means an average percent error of 37.49%. The median absolute error value of the all of the tests was 60.01° (33.34%). The standard deviation of error scores was 33.49° (19%). The distribution of the errors follows the standard bell curve for the most part, save for one outlier (see figure 8).

Several trends become apparent when looking at how individual variables affect the results. Firstly, between the two DM poses, orienting a face toward the camera produced less accurate estimations than orienting an edge toward the camera. On average, poses that oriented the face toward the camera resulted in 6.38% greater error value. The average error for the gray DM when viewed straight was 43.16% while the error when viewed turned was only31.63%. The

average error for white markers when viewed straight was 38.19% while the average error of the white DM when turned was 36.97%.This may give the impression that the white markers were fairly close in terms of accuracy, but in most cases that was not the case. The straight-view results have a standard deviation of 9.89% while the turned-view results had a standard deviation of 35.65%, indicating either a more scattered grouping of results or an outlier. In this case it is the latter: the white marker's turned error scores included an average error of 100% for when the light was placed behind the DM, greatly skewing the average. When the greatest error scores for both poses are discarded, the error when the DM was viewed straight was 34.86% while the turned error dropped all the way down to 21.21%, indicating that in most cases, viewing one of the faces of the DM head-on tends to produce less accurate results.

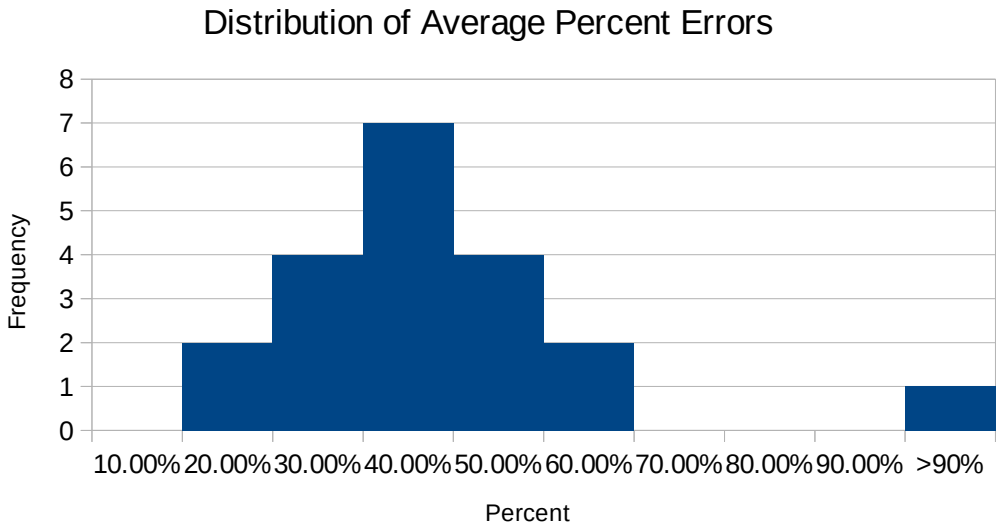

Figure 8: The distribution of the tests' percent errors.

This may be surprising at first, as one of the general assumptions in this method is that the more of the DM that is visible, the more information the program can gather and thus the more accurate the light estimation. There is another factor at work, however: *ARToolkit*. *ARToolkit*'s marker tracking notably works best when viewing markers at a slight angle; viewing a marker head-on

tends to cause the virtual objects projected onto it to jitter and shake. As such, the results from the pose tests may say more about *ARToolkit* than they do about the method itself.

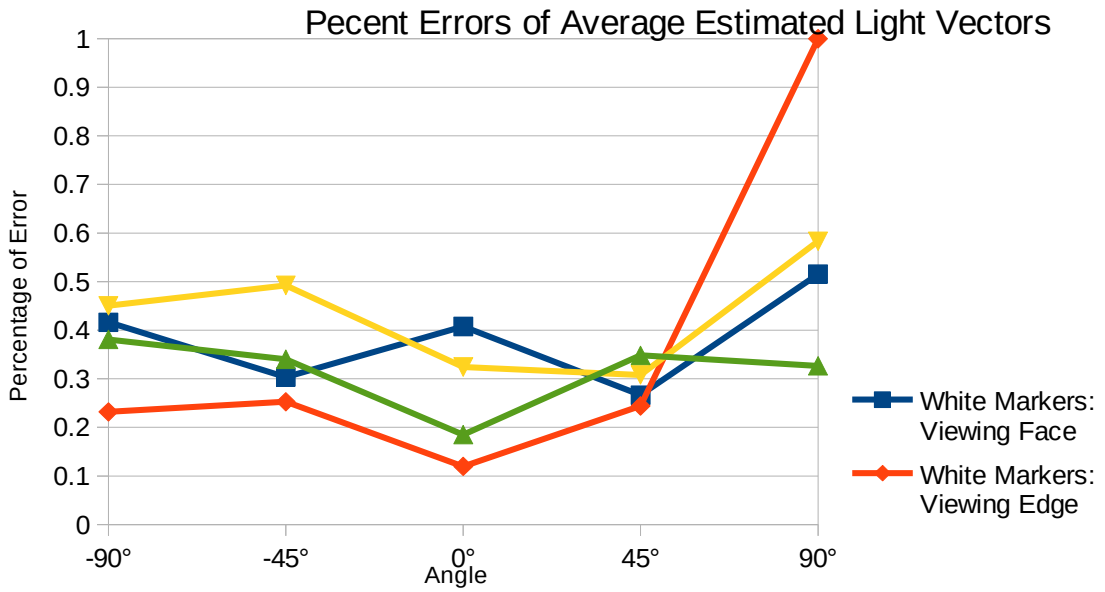

Figure 9: A graph of the average percent error for different marker poses and light positions. Using standard polar coordinates, -90° indicates when the lamp was in front of the DM, 0° the right, and positive 90° behind the DM.

Where the light was positioned also had an effect on the accuracy of the estimation. As to be expected, the estimated light vectors were the least accurate when the light was placed behind the DM, with an average error of 60.63%. This includes the aforementioned instance when the estimated vector had an error of 100%. Interestingly, while the white DM's average error was 100% when turned, the gray DM produced the lowest error while turned of all of the poses for that light position. The most accurate results came from when the light was placed directly to the right of the DM, with an average error of 25.92%. The other light positions and their average error values were in front of the DM at 36.98% error, in front and to the right at 34.72% error, and behind and to the right at 29.18% error.

The last variable to be examined is the color of the 2D markers on the surface of the DM. Surprisingly, white markers and gray markers performed almost equally. The average error for

the estimated light vectors of the white DM was 37.58%, while gray markers had an average error of 37.39%.

Based on informal observations during testing, having light estimation provided a greater sense of visual cohesion than using a constant light vector (see figure 10). That said, there is no denying that an average error of 67.48° is hardly impressive. What the program boasts in efficiency, it lacks in accuracy. Ultimately, the method is perhaps best thought of as a "quick-and-dirty" approach to real-time light estimation.

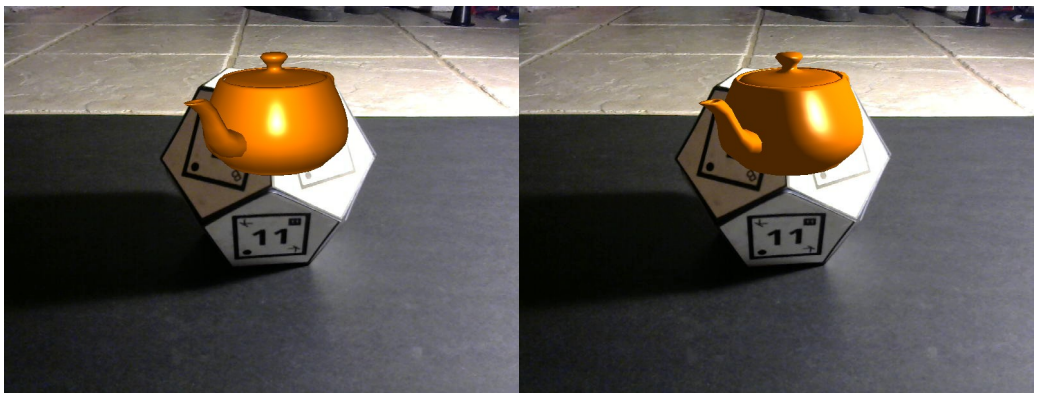

Figure 10: The test program running without light estimation (left) and with light estimation (right).

Section 5.2.3: Marker Threshold

A final observation was the effect the black and white threshold for marker detection had on the results. During marker detection, the image is run through a thresholding algorithm to convert the image into black and white. This contrast makes it easier to detect the marker. Unfortunately, to get the most stable result, i.e. those with the least amount of visual jitter, the threshold level had to be manually calibrated. While *ARToolkit* has an option to automatically find the optimal threshold value, this had to be disabled to implement multi-pass marker detection. The black and white DM tended to work well with lower threshold values, generally around 30-45, while the gray DM usually produced the most stable results around 50-60.

CHAPTER VI

CONCLUSIONS AND FUTURE RESEARCH

This research has produced a working inverse lighting algorithm that uses the surface illumination of a dodecahedral object to make inferences about the lighting conditions of the surrounding real-world environment. Moreover, the algorithm does so in real-time and without the need for an initial calibration phase. The light estimation algorithm works quickly, running at over 30 frames per second and in many cases at or near 60 FPS, especially when not using optional settings such as multi-pass marker detection. While a visible improvement over no light estimation, the estimated light vectors produced by the algorithm have a high margin of error. This is especially true for instances where the light is directly behind the dodecahedral marker.

This method's greatest advantages are its speed, ease of use, and that it works in real-time. Its primary disadvantage is its low accuracy. One possible issue is instability in marker detection and tracking. Whether or not the finicky nature of the tracking phase is a product of the method itself, or if it is simply a limitation of the tools used warrants further investigation. With all of these factors taken into consideration, this method would be useful for instances where real-time light estimation is needed and in which performance and simplicity of setup are crucial, with the accuracy of the estimation only needing to be "good enough".

There are many possible topics for future research. The first and most obvious one would be to implement the algorithm using an AR API other than *ARToolkit* to determine whether or not

some of the accuracy issues noted—especially those observed while one side of the DM was directly facing the camera—were the result of the algorithm or the tools used to implement it. Another possible topic for future research is, instead of averaging the preliminary light vectors, simply generating a list of light vectors and using all of them to simultaneously light the object. Yet more possible topics are using different methods of calculating the weights of preliminary light vectors, using a different polyhedron for light estimation, identifying multiple light sources, and determining the color of the incoming light.

REFERENCES

[1] *Pokémon Go*. Niantic, 2016.

[2] S. B. Knorr and D. Kurz, "Real-Time Illumination Estimation from Faces for Coherent Rendering," presented at the IEEE International Symposium on Mixed and Augmented Reality, Munich, Germany, 2014, pp. 113–122.

[3] M. Kanbara and N. Yokoya, "Real-time Estimation of Light Source Environment for Photorealistic Augmented Reality." IEEE, 2004.

[4] P. Supan, I. Stuppacher, and Michael Haller, "Image Based Shadowing in Real-Time Augmented Reality," *International Journal of Virtual Reality*, vol. 5, no. 3, pp. 1–10, 2006.

[5] C.B. Madsen, T. Jensen, and M.S. Andersen, "Real-Time Image-Based Lighting for Outdoor Augmented Reality under Dynamically Changing Illumination Conditions," presented at the International Conference on Graphics Theory and Applications, Setúbal, Portuga, 2006, pp. 364–371.

[6] C. Madsen and R. Laursen, "Scalable GPU-Based Approach to Shading and Shadowing for Photo-Realistic Real-Time Augmented Reality," presented at the International Conference on Computer Graphics Theory and Applications, Barcelona, Spain, 2007, pp. 252–261.

[7] G. Finlayson, C. Fredembach, and M.S. Drew, "Detecting Illumination in Images," presented at the IEEE 11th International Conference on Computer Vision, Rio de Janeiro, Brazil, 2007.

[8] G. Nakano, I. Kitahara, and Y. Ohta, "Generating Perceptually-Correct Shadows for Mixed Reality," in *Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*, Cambridge, United Kingdom, 2008.

[9] M. Aittala, "Inverse lighting and photorealistic rendering for augmented reality," *The Visual Computer*, vol. 26, no. 6, pp. 669–678, Apr. 2010.

[10] Z. Noh and M.S. Sunar, "Soft Shadow Rendering based on Real Light Source Estimation in Augmented Reality," *Advances in Multimedia-An International Journal (AMIJ)*, vol. 1, no. 2, p. 26, 2010.

[11] S. Pessoa et al, "Photorealistic Rendering for Augmented Reality: A Global Illumination and BRDF Solution," presented at the Virtural Reality Conference, Boston, MA, US, 2010, pp. 3–10.

[12] B.F. Jenson et al., "Simplifying Real-Time Light Source Tracking and Credible Shadow Generation for Augmented Reality using ARToolkit," *Medialogy*, 2010.

[13] X. Jiang, A.J. Schofield, and J.L. Wyatt, "Shadow Detection based on Colour Segmentation and Estimated Illumination," presented at the BMVC, Dundee, UK, 2011.

[14] X. Chen, K. Wang, and X. Jin, "Single image based illumination estimation for lighting virtual object in real scene," presented at the 12th International Conference on Computer-Aided Design and Computer Graphics, San Jose, CA, USA, 2011.

[15] I. Arief, S. McCallum, and J.Y. Hardeburg, “Realtime Estimation of Illumination Direction for Augmented Reality on Mobile Devices,” presented at the Color and Imaging Conference, 2012, vol. 2012.
[16] L. Gruber et al., “Efficient and Robust Radiance Transfer for Probeless Photorealistic Augmented Reality,” presented at the Virtual Reality 2014 IEEE, Minneapolis, Minneapolis, MN, USA, 2014.
[17] D. Kamboj and W. Liu, “Improved Variance Cut Algorithm for Light SourceEstimation in Augmented Reality,” *International Journal of Computer Applications*, vol. 85, no. 19, Jan. 2014.
[18] N. Michiels et al., “Interactive Augmented Omnidirectional Video with Realistic Lighting,” presented at the International Conference on Augmented and Virtual Reality, 2014, pp. 247–263.
[19] K. Rohmer, “Interactive near-field illumination for photorealistic augmented reality on mobile devices,” presented at the 2014 IEEE International Symposium on Mixed and Augmented Reality (ISMAR), 2014, pp. 29–38.
[20] M. Borg, M.M. Paprocki, and C.B. Madsen, “Perceptual Evaluation of Photo-Realism in Real-Time 3D Augmented Reality,” presented at the GRAPP 2014: International Conference on Computer Graphics Theory and Applications, Lisbon, Portugal, 2014.
[21] H. Alhajhamad, M. S. Sunar, and H. Kolivand, “Automatic Estimation of Illumination Features for Indoor Photorealistic Rendering in Augmented Reality,” presented at the International Conference on Intelligent Software Methodologies, Tools, and Techniques, Naples, Italy, 2015.
[22] F. Einabadi and O. Grau, “Discrete Light Source Estimation from Light Probes for Photorealistic Rendering,” presented at the BMVC, Swansea, UK, 2015.
[23] P. Kán, J. Unterguggenberger, and H. Kaufmann, “High-Quality Consistent Illumination in Mobile Augmented Reality by Radiance Convolution on the GPU,” presented at the International Symposium on Visual Computing, Las Vegas, Las Vegas, Nevada, US, 2015, pp. 574–585.
[24] P.E. Buteau and H. Saito, “Retrieving Lights Positions Using Plane Segmentation with Diffuse Illumination Reinforced with Specular Component,” presented at the IEEE International Symposium on Mixed and Augmented Reality, Fukuoka, Japan, 2015.
[25] K.E. Soulier, M.N. Selzer, and M.L. Larrea, “Real-Time Estimation of Illumination Direction for Augmented Reality with Low-Cost Sensors,” presented at the XXII Congreso Argentino de Ciencias de la Computación, 2016.
[26] M. Kasper et al., “Light Source Estimation with Analytical Path-tracing,” *arXiv*, Jan. 2017.
[27] Graham Sellers, Richard S Wright Jr., and Nicholas Haemel, *OpenGL Superbible: Comprehensive Tutorial and Reference*, 7th ed. Boston, MA, US: Addison-Wesley Professional, 2015.
[28] *ARToolkit 5*. Daqri, 2016.
[29] *OpenGL*. Kronos Group, 1992.
[30] Milan Ikits, Marcelo Magallon, and Nigel Stewart, *OpenGL Extension Wrangler*. 2015.
[31] Mark Kilgard, *OpenGL Utility Toolkit 3.7*. 1998.
[32] *OpenGL Mathematics*. GL-Truc Creation, 2016.
[33] Jesse Beder, *yaml-cpp*. 2016.
[34] Eric Lengyel, “Open Game Exchange,” 29-Jan-2017. [Online]. Available: OpenGEX.org. [Accessed: 20-Feb-2018].

VITA

Glen Kemp Straughn

Candidate for the Degree of

Master of Science

Thesis: REAL-TIME INVERSE LIGHTING FOR AUGMENTED REALITY USING DODECAHEDRAL MARKER

Major Field: Computer Science

Biographical:

Education:

Completed the requirements for the Master of Science in Computer Science at Oklahoma State University, Stillwater, Oklahoma in August, 2018.

Completed the requirements for the Bachelor of Science in Computer Science at University of Science and Arts of Oklahoma, Chickasha, Oklahoma in 2014.