REAL-TIME OBJECT REMOVAL IN AUGMENTED REALITY

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Tyler Dahl

June 2018

COMMITTEE MEMBERSHIP

TITLE:	Real-Time Object Removal in Augmented Reality
AUTHOR:	Tyler Dahl
DATE SUBMITTED:	June 2018
COMMITTEE CHAIR:	Christian Eckhardt, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Maria Pantoja, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Franz J. Kurfess, Ph.D. Professor of Computer Science

ABSTRACT

Real-Time Object Removal in Augmented Reality

Tyler Dahl

Diminished reality, as a sub-topic of augmented reality where digital information is overlaid on an environment, is the perceived removal of an object from an environment. Previous approaches to diminished reality used digital replacement techniques, inpainting, and multi-view homographies. However, few used a virtual representation of the real environment, limiting their domains to planar environments.

This thesis provides a framework to achieve real-time diminished reality on an augmented reality headset. Using state-of-the-art hardware, we combine a virtual representation of the real environment with inpainting to remove existing objects from complex environments.[1]

Our work is found to be competitive with previous results, with a similar qualitative outcome under the limitations of available technology. Additionally, by implementing new texturing algorithms, a more detailed representation of the real environment is achieved.

[1]Source code is provided at https://github.com/tydahlwave/Thesis

ACKNOWLEDGMENTS

Thanks to:

- My parents, Derek and Michelle, for their constant support, love, and belief that I can accomplish whatever I set my mind to
- My sister, Courtney, for her constant support, love, and encouragement
- God, for placing me at Cal Poly and giving me the strength to finish strong
- My friends and relatives, for their prayers and encouragement throughout this process
- Zoe Wood, for igniting my love of computer graphics and pushing me to pursue challenging projects
- Christian Eckhardt, for establishing the Mixed Reality Lab and providing me with the tools necessary to complete this thesis
- Cal Poly, for an amazing computer science program
- Andrew Guenther, for uploading this template which saved considerable time

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

Our perception of reality is being reshaped. Much like the industrial revolution forever changed society through the rise of machines, the digital revolution is changing society through the rise of visual, interactive data. This data is accessible through interaction with 2D screens in the form of smartphones, tablets, computers, and TVs. However, new developments in virtual reality make it feasible to interact with this data in 3D space.

Virtual reality (VR) allows us to become completely immersed in a fully artificial, digital environment [42]. Current VR technology displays digital content onto a head-mounted device with two screens, one for each eye, where a stereoscopic image is displayed to trick the brain into perceiving depth. Users can interact with virtual objects through controllers specifically designed for such interaction. The experiences users have within the virtual environment are often described as highly immersive. Sensations range from experiencing virtual movement to altitude panic [13].

The primary focus of recent investigations in virtual reality have been the side effects of this high level of immersion, specifically motion sickness. If a user moves within the virtual environment while their physical body remains at rest, their brain registers a certain acceleration and tries to compensate, leading to an uncomfortable disorientation. The same sensation is felt in the opposite scenario, when e.g. sitting in a static environment while the body senses movement such as carsickness and seasickness. This can result in nausea, dizziness, and fatigue, reducing time spent in the virtual environment [46].

Motion sickness can be prevented by moving a user's position in the virtual environment in sync with their physical body. This allows the user to control their virtual counterpart in a natural way by walking around their real environment. However, this presents several limitations, such as the user colliding with physical objects and a virtual world constrained by the size of the real environment. Other strategies have been developed to reduce motion sickness, such as teleporting a person's virtual counterpart between locations and walking in place. These strategies allow users to experience the fun and immersion of virtual reality without the unwanted side effects. Another technology, called augmented reality, provides a similar experience to virtual reality but without the motion sickness.

Augmented reality (AR) allows us to interact with digital content while also remaining aware of the real environment. Current AR technology displays digital content onto a see-through display, overlapping real world information with digital content. In the simplest form of augmented reality, digital information is displayed on billboards within the user's field of view. Digital content cannot interact with the real world. This form of augmented reality became popular through the introduction of heads-up displays (HUD). Originally developed to prevent military pilots from looking down to view instrument data, HUDs have also been used in popular modern products such as Google Glass.

Recent advancements in computer vision and depth sensing are enabling a more advanced form of augmented reality, where digital content is not confined to a static location. These advancements allow the pinpointing of real object positions and properties (size, color) for a fluid interaction between virtual and real elements.

Mixed reality (MR), an advanced form of augmented reality, is the anchoring and interaction of digital content with the real world. Digital content can collide with and hide behind real objects. As users navigate their environment, digital content stays in place, mimicking the behavior of real objects. This form of augmented reality provides an immersive experience similar to virtual reality.

Though this technology is still in its infancy, industry-leading tech companies are actively pursuing its development [32]. Mobile devices are dominating the market in the near-term with head-mounted devices predicted to dominate in the future. In 2015, the first head-mounted mixed reality device, the HoloLens, was released by Microsoft. In 2017, mobile mixed reality experiences became common with the introduction of ARKit and ARCore by Apple and Google respectively. The future is one where digital content seamlessly blends with our real world.

The primary vision touted by futurists is a pair of augmented reality glasses powerful enough to replace smartphones. To realize this future, technology needs to shrink and faster algorithms need to be developed. Consumers want a device that enhances their life without the need to change their habits. In order to seamlessly blend digital content with the real world, digital content ought to not just be added to the world, but real content ought to be digitally removed.

Diminished reality (DR), a sub-topic of augmented reality, is the perceived removal of an object from an environment. DR is achieved by displaying digital content in front of an object, where the digital content emulates the region behind the object. This causes the user to perceive that the object no longer exists. The need for DR is most easily seen when considering the virtual refurnishing of a house interior. When viewing virtual furniture, the virtual content ought to match as closely as possible what the real environment will look like after refurnishing. If virtual furniture is placed on top of existing furniture, overlapping content will reduce the realism of the experience. However, it would be time-consuming to require the user to physically remove existing furniture prior to viewing virtual furniture. Instead, we can provide a realistic refurnishing with minimal effort by digitally removing the existing furniture prior to placing virtual furniture. Other applications of DR include hiding trash or graffiti from view, obscuring secret agents from webcams, seeing through walls on a construction site, seeing a patient's internal organs during surgery, or removing unwanted objects in photographs.

For this thesis, we chose to research diminished reality. With the goal of eventual consumer adoption, an ideal diminished reality solution is one that works in any environment without complex setup. Our research question is therefore: "Can a diminished reality solution be developed with current hardware which works in any environment in real-time, with no prior knowledge of the environment?"

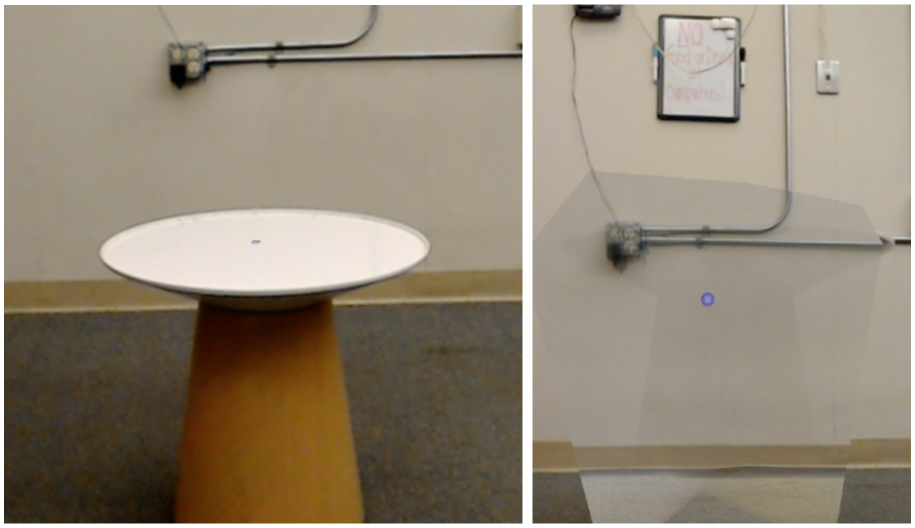


Figure 1.1: Diminishing a table with our solution. Left image is the table before diminishing. Right image is after the table is removed.

Our solution is a diminished reality prototype created on the Microsoft HoloLens which successfully removes an object in an unseen environment in real-time. See Figure 1.1. Our contribution to the field of diminished reality is a pipeline that can be used as a basis upon which future diminished reality solutions can be developed. Each section of the pipeline can be replaced with newer technologies and algorithms as they become available to enhance the diminished result.

This thesis is organized as follows. Chapter 2 discusses the background and related work that has been conducted within the field of diminished reality. Chapter

3 provides an overview of the design of our diminished reality pipeline. Chapter 4 discusses the hardware and software platforms available to create a diminished reality prototype as well as our specific implementation. Chapter 5 discusses the results of our solution in various environments and compared to similar work in diminished reality. Chapter 6 discusses potential future enhancements to our solution. Chapter 7 concludes our work.

Chapter 2

BACKGROUND

The term ”diminished reality” was first coined in 1999 by Mann in his concept of mediated reality which included AR, MR, and DR [35]. In practice, diminished reality is achieved when applying the following three steps to a video feed: region selection, region tracking, and region removal. The following sections will explore various techniques that have been explored for each step.

2.1 Region Selection

Before an object can be removed, there must first be a way to specify the object. Techniques used by previous diminished reality approaches range from completely manual selection to completely automated selection. The more accurate the selection, the more realistic the removal.

2.1.1 Manual

Manual region selection requires the user to specify the exact pixels to be removed. This allows for extremely accurate results, where only pixels of the object are removed. However, specifying every pixel of a complex object is a tedious process. Zokai et al. alleviate this problem by allowing the user to specify a bounding box as the region instead of selecting every pixel [58]. This method reduces the manual selection time, but also causes more pixels to be removed if the desired region is not rectangular. Manual selection methods cannot account for regions that change shape over time, as it requires the region to be respecified by the user.

2.1.2 Semi-Automatic

Semi-automatic region selection methods require the user to provide minimal input prior to generating the selected region. Both region expansion and region shrinking algorithms exist. Herling and Broll use a region shrinking method to fit the selected region to the contour of the dominant object within an initial circle drawn by the user [28]. Additional padding was added to account for errors in contour detection. Simpson used a region expansion method to segment an object from a mesh given a 3D point on the object [47]. The selected object was found by traversing triangles connected to the given point until a plane was detected. Semi-automatic selection methods allow complex shapes to be quickly selected and provide impressive results.

2.1.3 Automatic

Automatic region selection requires no real-time input from the user. Region selection is performed immediately and automatically, sometimes using offline input. Nakajima et al. use automatic region selection by removing all objects of a chosen category specified offline [38]. Objects of the chosen category are detected using a neural network and segmented from a global point cloud. Automatic selection methods allow immediate diminished results, but are not practical in all scenarios. Some user input is preferred.

2.2 Region Tracking

After selecting the region, it needs to be tracked between frames. When the selected region and the camera are stationary, tracking is not necessary. However, to maintain an accurate removal, tracking is necessary if either the camera or the selected object moves. To track an object between frames, either 2D features or 3D positions are used.

2.2.1 2D Features

Region tracking with 2D features involves the detection and matching of small patterns within pairs of images. Features that can be tracked include edges, corners, blobs, ridges, and other shapes learned by neural networks. Using the difference between tracked features within two images, a homography is calculated and used to transform one image to the perspective of the other. See Figure 2.1. This transform is also used to update the selected region's position in each frame. Many diminished reality methods use 2D image features to update the selected region per frame. For example, Herling and Broll used 2D features to detect the contour of an object and update the selected region to match the contour every frame [28].

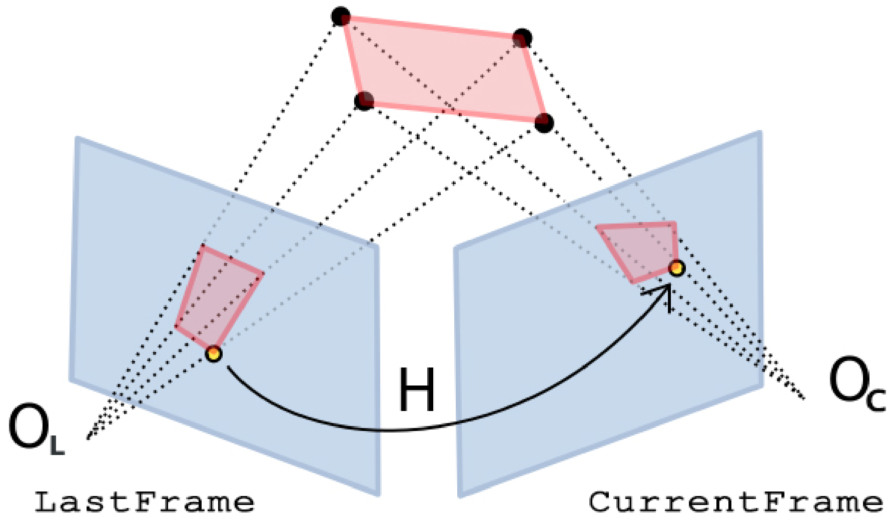


Figure 2.1: A homography between two frames taken from different perspectives [55].

2.2.2 SLAM

Region tracking with 3D positions is performed through simultaneous localization and mapping (SLAM) techniques. SLAM techniques use RGB plus depth (RGB-D) data per pixel to create a map of the environment and calculate the camera's position within the map. See Figure 2.2. Visual-SLAM techniques use only RGB data per pixel and calculate depth from changes in 2D features over time. Visual-SLAM is less accurate than SLAM, but doesn't require a sensor capable of acquiring depth data. The 3D positions of the selected object are then transformed to screen space every frame to obtain the updated selected region. Kawai et al. used visual-SLAM to track an object's position within an environment and inpaint detected planes [30]. SLAM techniques are less often used in diminished reality due to the need for specialized sensors and computationally expensive algorithms.

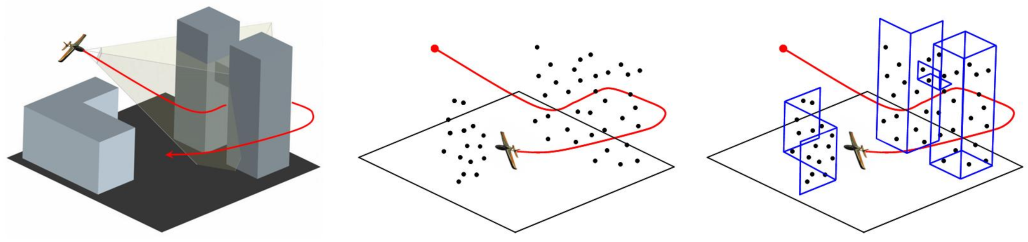


Figure 2.2: Simultaneous localization and mapping (SLAM) [34].

2.3 Region Diminishing

The most important step to achieve diminished reality is the removal of the selected region. This requires coloring pixels in front of the selected region with data from another source. First, the selected region is converted into a binary grayscale mask, with pixels in the selected region colored white and the rest of the image colored black. This mask is then fed into an algorithm with the original image of the environment.

The result is the original image where the selected region is diminished. There are three categories of object removal: replacing, inpainting, and see-through.

2.3.1 Replacing

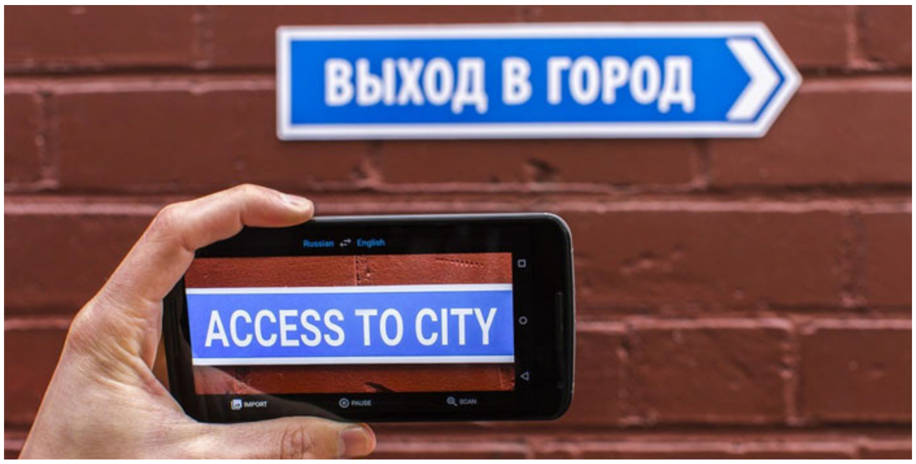


Figure 2.3: Google Translate app replaces text with a different language [41].

Object removal techniques that cover or replace the selected region are the simplest form of diminished reality. The selected region is diminished by hiding it behind a virtual object large enough to cover the entire region. An example of this is implemented in the Google Translate iOS and Android app [1]. The app replaces text in any picture taken by the user with text of a chosen language. A virtual rectangle is displayed in front of the existing text to hide it and then text of the chosen language is displayed in front of the virtual rectangle. See Figure 2.3. This approach works well when the user does not need to see the background behind the selected object.

2.3.2 Inpainting

Inpainting is a popular object removal technique. Originating as a form of art restoration centuries ago, it involves filling a region in an image using similar content within the same image. Inpainting relies on the idea that patterns are common in nature and often repeated. By repeating nearby patterns in front of the selected region, the object will appear to vanish. See Figure 2.4. In 2010, Herling and Broll demonstrated the first real-time inpainting algorithm capable of running on commodity hardware with simple setup, named PixMix [27, 28]. In 2017, Iizuka et al. further showed that it was possible to use a neural network to learn and repeat patterns from similar images to provide a realistic diminished result [29].

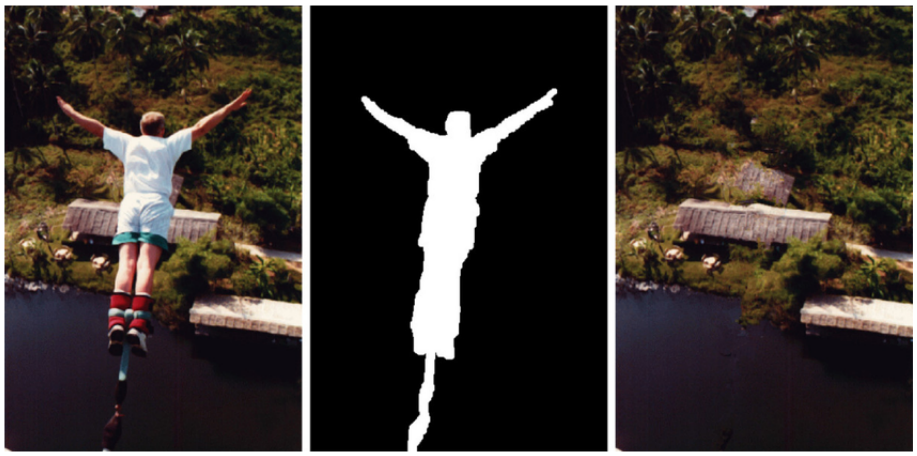


Figure 2.4: Inpainting a bungee jumper [40].

A common challenge faced by inpainting techniques is complex background geometry. Most methods assume a planar background, leading to unrealistic perspective distortions when moving the camera within a complex environment. See Figure 2.5. However, methods exist that remedy this situation. In 2016, Kawai et al. extracted and inpainted multiple planes from an environment to accurately simulate more complex environments [30]. A limitation of inpainting is that it cannot recover information for objects which are completely hidden behind the selected object. For this, a see-through object removal technique must be used.

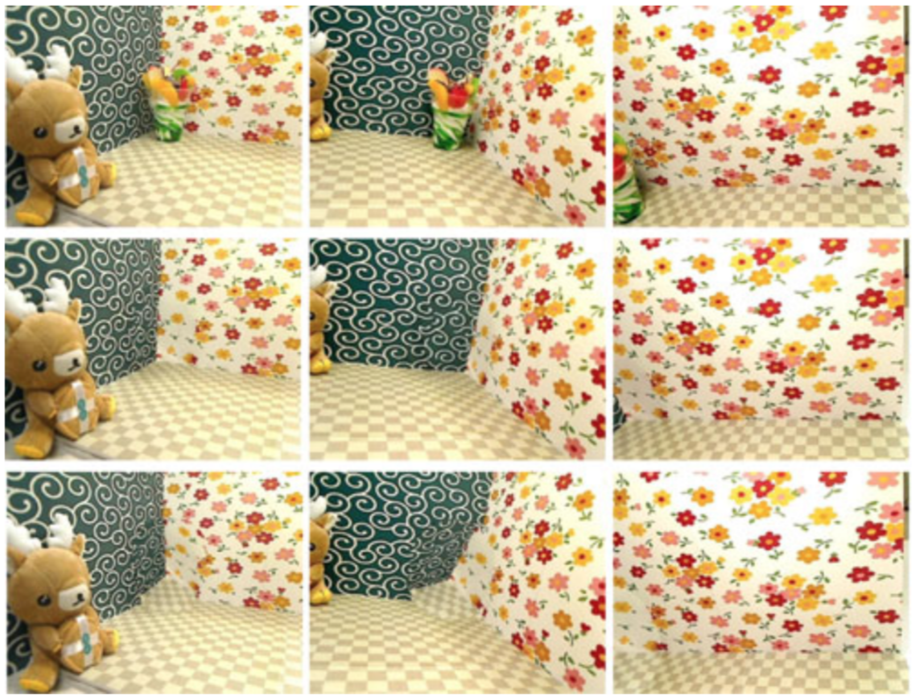


Figure 2.5: Comparison of inpainting from different viewpoints [30]. Top: input images. Middle: inpainting multiple planes. Bottom: inpainting a single plane.

2.3.3 See-through

The final category of object removal is see-through techniques. These techniques provide the user with a realistic view of the environment behind a removed object. Unlike replacement and inpainting, see-through techniques have knowledge of the content behind the selected object. The easiest way to accomplish this is with multiple cameras as demonstrated by Zokai et al. [58]. One camera views the selected object and another camera views the region behind the selected object. See Figure 2.6. Then a homography is calculated and used to transform the output of the camera viewing the background to the perspective of the camera viewing the selected object. The selected region is cropped from the background camera and displayed on the main camera. This provides an accurate and live view of the content behind the diminished object.

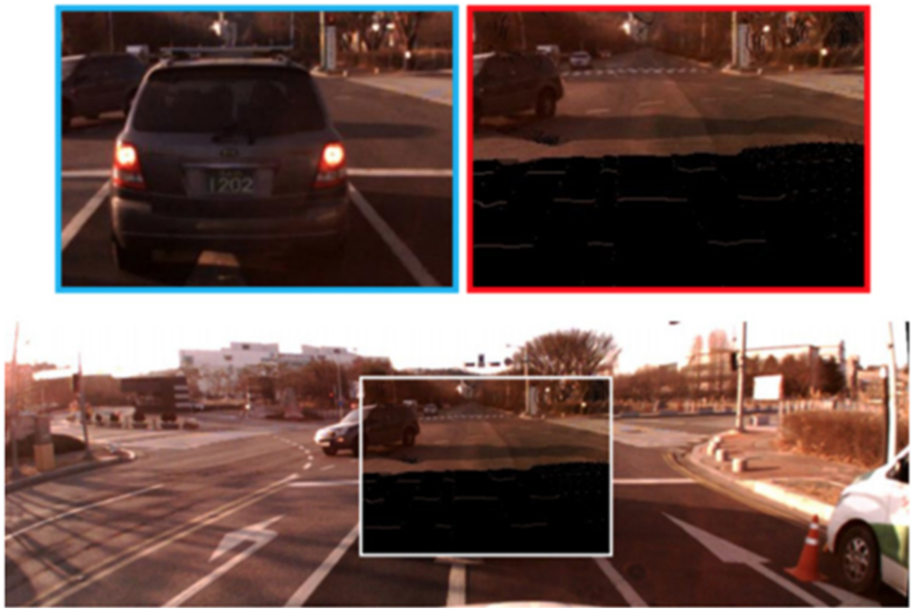


Figure 2.6: Seeing through a car using multiple cameras [43]. Top left image is from the back car's perspective. Top right image is from the front car's perspective.

Other approaches use a single camera from multiple viewpoints or previous images of the environment. Yokoi and Fujiyoshi removed a professor from a lecture video using a stationary camera where the subject was moving [56]. Previous frames were used to fill the region containing the professor in the current frame. A method proposed by Li et al. used a corpus of internet images taken at the same location to construct a map of the environment offline and used this reconstruction to display content behind the selected object [33]. Recent approaches have used SLAM to construct a textured map of the environment as a single device moves throughout the environment. Simpson used this technique to remove a small table using a Google Tango device [47]. However, perceived content behind the removed object will become invalid if the background content changes and a recent view of the background cannot be obtained.

A challenge faced by see-through techniques is obtaining data for directly adjacent surfaces. It is not possible to setup a camera to view the region connecting a couch to the floor. However, previous images of the environment without the selected object can be used. When this data is not available, inpainting techniques ought to be used to fill the region as realistically as possible.

Each object removal method has its benefits and challenges. Replacing the diminished region with a virtual object is simple, yet does not allow the user to see content behind the object. Inpainting techniques generate convincing backgrounds for which no data exists, but cannot recreate content that is completely hidden by the selected object. See-through techniques allow the user to see content hidden by the selected object, but require knowledge of the background content, which may not be available. However, combining inpainting and see-through techniques compensates for the limitations of each and achieves a more realistic diminished result.

2.4 Proposed Solution

The diminished reality solution proposed by this paper uses semi-automatic object selection, SLAM for object tracking, and a combination of inpainting and see-through diminished reality techniques for object removal. Specifically, we use the HoloLens, an advanced augmented reality platform, to create a virtual representation of the real environment. We then texture this virtual environment using one of two texturing algorithms. The object is selected using a 3D point specified by the user which is used to find nearby vertices belonging to the object. Tracking is achieved using the updated camera position and spatial mapping obtained from the HoloLens. Object removal is achieved by displaying the textured virtual environment behind the selected object and by inpainting regions where no texture data exists.

2.5 Related Work

Some similar approaches exist, which also make use of virtual representations of the real environment and inpainting. The first is the method proposed by Kawai et al. where visual-SLAM is used to reconstruct background planes and inpainting is used on each individual plane [30]. Previous diminished reality solutions assumed a planar background and suffered from perspective distortion errors when the background was composed of multiple planes. By finding multiple background planes, the method proposed by Kawai et al. provides better results in a wider range of environments. However, visual-SLAM only tracks 2D features within a series of RGB images and requires diverse textures to work well. When the scene contains surfaces with minimal texturing, this approach suffers. Instead, an RGB-D sensor combined with SLAM can be used to track 3D features and reconstruct geometry even in scenes with limited texturing.

A method proposed by Simpson reconstructs complex scene geometry using SLAM with an RGB-D camera on a Google Tango device [47]. Scene geometry is textured as the user moves around the environment. To diminish an object, the scene geometry excluding the selected object is rendered in front of the object. The selected object is assumed to be connected to a large planar surface, and any holes created when removing the selected object from the surface are inpainted. However, this approach maintains limited detail in its texturing of the environment and provides a low-resolution result of the diminished region. Real-time performance is achieved after an initial removal process of approximately 10 seconds.

Another method proposed by Nakajima et al. also reconstructs scene geometry using SLAM with an RGB-D camera [38]. Scene geometry is obtained and textured as the user moves around the environment. An automatic object selection method is used which relies on a convolutional neural network to recognize categories of objects within its 3D reconstruction of the environment. An object of the chosen category is automatically and immediately removed from the scene, providing a diminished view of the object from the start. However, this method does not rely on inpainting or other filling techniques to color regions where no 3D reconstruction exists. Thus, the diminished region is black until the user moves around the environment and views regions behind the object. This method provides high-resolution results and runs in real-time. However, it was tested on a system containing 125 GB of RAM, which is not representative of commodity hardware.

Our solution reconstructs complex scene geometry, runs in real-time with limited memory constraints, provides high-resolution results, and inpaints regions where scene geometry is nonexistant. The following chapter will discuss the architecture of our solution.

Chapter 3

ARCHITECTURE

In this chapter, we provide a high-level overview of the design of the diminished reality solution proposed in this paper and describe each stage in our diminished reality pipeline. The next chapter will go into detail on how each stage was implemented. The minimum requirements of a diminished reality solution are object selection, object tracking, and object removal. However, due to the nature of our implementation, other important components include obtaining a spatial mapping of the environment, texturing the spatial mapping, and performing post-processing after various stages of the pipeline. The following sections will give a brief overview of each stage.

3.1 Spatial Mapping

The first stage in our diminished reality pipeline is to obtain a spatial mapping of the environment. A spatial mapping is a detailed representation of real-world surfaces in the environment around the user [57]. This detailed representation is a collection of 3D points relative to the camera's original position when starting the application. These points are stored in a spatial data structure, making queries of specific points extremely efficient. Additionally, color information is often stored for each point. The density of the 3D points within an environment varies per system and is restricted by hardware capabilities. Higher density provides a more accurate representation of the real environment but at the cost of higher processing requirements.

A spatial mapping of the environment allows us to save background information while using only a single device. As the user moves throughout the environment, the spatial mapping is updated and made more accurate. See-through object removal is then performed by displaying the background information that has been saved in the spatial mapping in front of the selected object and mapping each pixel to a color.

3.2 3D Post-Processing

The second stage in our diminished reality pipeline is to perform post-processing on the spatial mapping obtained from the previous stage. This step is necessary because the spatial mapping is often noisy and contains abnormalities. Post-processing of the spatial mapping entails removing disconnected points, smoothing surfaces, filling holes, and replacing flat surfaces with planes. Performing these operations on the spatial mapping allow us to account for errors in the mapping process and provide a better approximation of the real environment. Due to our choice of hardware, the spatial mapping automatically undergoes smoothing and removal of disconnected points. We further explore the filling of holes and replacing of flat surfaces with planes.

3.3 Texturing

The third stage in our diminished reality pipeline is to texture the spatial mapping. A textured spatial mapping is important to provide realistic see-through object removal. This is accomplished by saving color information pertaining to the real environment and mapping each rendered pixel of the spatial mapping to a color. Color information is either directly stored within the spatial mapping or stored separately and mapped to each pixel at runtime. Our approach explores two texturing methods and their tradeoffs: projective texture mapping and sparse voxel octree texture mapping. Projective texture mapping supports high levels of detail by storing high-resolution images taken from different viewpoints. Sparse voxel octree texture mapping supports efficient mapping between colors and pixels by storing color in a spatial data structure.

3.4 Object Selection and Tracking

The fourth stage in our diminished reality pipeline is to select the object to diminish and track it per frame. We use a semi-automatic object selection technique, which requires the user to provide minimal input. The appropriate object is then automatically selected. In our implementation, the user selects a single point in the spatial mapping and the system automatically determines which object to remove by finding nearby vertices. Using this technique, the user does not need to specify the exact 3D volume to be diminished, which is time-consuming. Once the object has been selected, it is ready to be removed.

Object tracking is accomplished by tracking 2D or 3D features between frames. In our solution, we use static objects and use SLAM to constantly update our spatial mapping, which provides us with stable 3D positions of objects. After selecting the 3D volume to be diminished, no further action is necessary to track it.

3.5 Object Removal (Diminishing)

The fifth stage in our diminished reality pipeline is to diminish the selected object and remove it from view. To create the illusion that the object has vanished, pixels are displayed in front of the real object representing colors of the environment behind the object. As the user moves throughout the environment, these pixels are updated with colors from the appropriate background. In our solution, the textured virtual environment from behind the object is rendered in front of the object, causing the user to see through it. The next stage performs post-processing to further enhance the diminished result.

3.6 2D Post-Processing

The sixth and final stage in our diminished reality pipeline is to perform 2D post-processing on the final diminished region to make it blend in better with its surroundings. By fading the transparency of the edges of the diminished region, color smoothly transitions between the virtual environment and the real environment, preventing any discrepancies between color at the edges of the region.

The most important 2D post-processing step in our solution is to apply inpainting to the parts of the diminished region that do not have any textured spatial mapping information. This arises when the user has not viewed the region behind the object or when spatial mapping fails to capture areas that are far from the user. Inpainting allows us to realistically fill these areas with color from surrounding pixels.

Chapter 4

IMPLEMENTATION

In this chapter, we discuss the hardware and software used to develop our diminished reality solution and why each was chosen. We then take a deep dive into our diminished reality pipeline and see how each stage was implemented.

4.1 Hardware

When implementing a mixed reality application, it is important to carefully consider the hardware that will be used, as each comes with its own benefits and challenges. Choosing one technology might make implementation easier while choosing another might provide more accurate results. Throughout the development of this solution, three different hardware platforms were explored: Google Tango, Microsoft HoloLens, and Intel RealSense. The following sections discuss each piece of hardware, their tradeoffs, and why we ended up settling with the Microsoft HoloLens. Other potential hardware is also briefly touched on.

4.1.1 Google Tango

The first platform we explored was Google Tango. Google Tango was an augmented reality platform developed by the Advanced Technology and Projects division at Google that enabled smartphones and tablets with specialized sensors to map their surroundings. This platform was originally released in June 2014, but a commercial smartphone capable of running the framework wasn't released until late 2016 with the introduction of the Lenovo Phab 2 Pro. A second smartphone, the Asus Zenfone AR, was released in August 2017. Prior to this, developers and researchers could purchase a tablet development kit from the Google Tango team that contained the required sensors and internal processing capable of running the framework. The tablet development kit was discontinued in May 2017 [48].

Originally, this platform was a good choice for implementing diminished reality. Hardware costs were roughly $500 to acquire a smartphone capable of running the framework. The framework also supported automatic texturing of the environment during the spatial mapping stage. However, in mid-2017, both Apple and Google released augmented reality frameworks, ARKit and ARCore respectively, which enabled most smartphones with no specialized sensors to run augmented reality experiences. These frameworks do not provide a spatial mapping of the surrounding environment, but instead detect vertical and horizontal planes, which is enough to enable simple augmented reality experiences. These frameworks received wide adoption by consumers and quickly dwarfed the popularity and adoption rates of Google Tango. By December 2017, Google announced the termination of Google Tango beginning in March 2018 [21]. As of May 2018, the Google Tango site and related documentation is no longer available. This development shifted our focus to other platforms.

4.1.2 Microsoft HoloLens

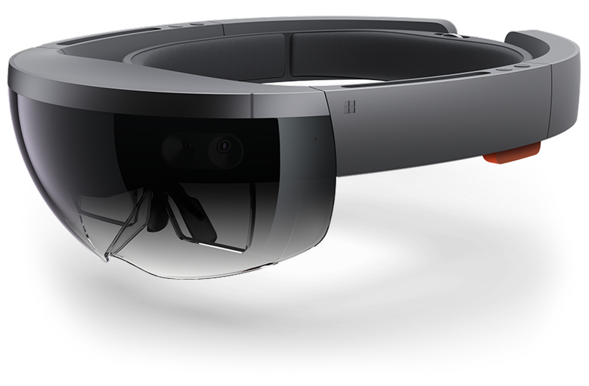
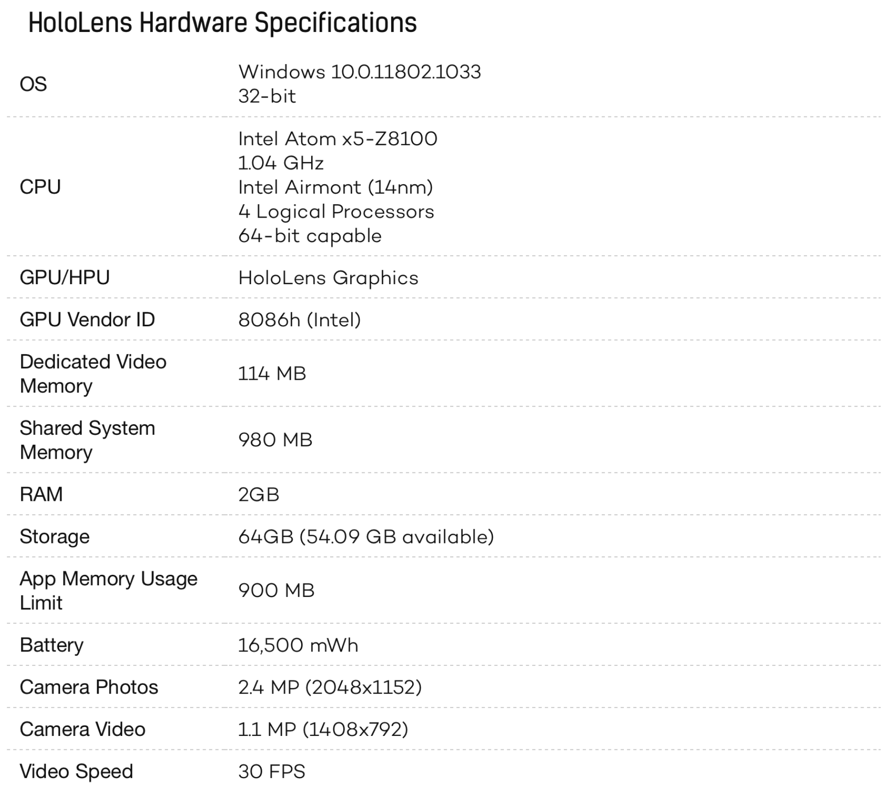


Figure 4.1: The Microsoft HoloLens [12].

The second platform we explored was the Microsoft HoloLens. See Figure 4.1. Microsoft HoloLens is a mixed reality headset developed and manufactured by Microsoft. It was the first device to support Microsoft's Windows Mixed Reality framework, which enables developers to create apps for VR and AR devices running Windows 10. It also makes use of RGB-D cameras and technology similar to its predecessor, the Kinect, to create a 3D map of the environment. The development edition of the HoloLens was released in March 2016 and continues to be the most advanced mixed reality headset that developers can purchase [39]. However, priced at \$3000, it still hasn't seen wide adoption by developers. There is no consumer version of the headset as of May 2018 and there has not been an update to the hardware since its release. However, software updates have been released that enhance its spatial mapping capabilities and user interface. The HoloLens' primary forms of input are hand gestures, voice commands, and a small, single-click remote. See Figure 4.2 for additional hardware specifications [44].

With advanced spatial mapping capabilities, the HoloLens is a good candidate for diminished reality. It generates a mesh of the environment in real-time and allows the user to interact with virtual holograms placed throughout the environment within a 30 degree field of view [31]. The lack of color information provided in the environment mesh is a limitation for our use case. However, this is compensated by exploring various texturing methods to obtain and map color information to the spatial mapping. Termination of the Google Tango platform further solidified our decision to use this platform. The future of augmented reality is not on our smartphones, but on our heads. The eventual goal is to make the technology small enough to fit into a normal pair of glasses.



HoloLens Hardware Specifications

OS	Windows 10.0.11802.1033 32-bit
CPU	Intel Atom x5-Z8100 1.04 GHz Intel Airmont (14nm) 4 Logical Processors 64-bit capable
GPU/HPU	HoloLens Graphics
GPU Vendor ID	8086h (Intel)
Dedicated Video Memory	114 MB
Shared System Memory	980 MB
RAM	2GB
Storage	64GB (54.09 GB available)
App Memory Usage Limit	900 MB
Battery	16,500 mWh
Camera Photos	2.4 MP (2048x1152)
Camera Video	1.1 MP (1408x792)
Video Speed	30 FPS

Figure 4.2: HoloLens hardware specifications [44].

4.1.3 Intel RealSense

The third platform we explored was the Intel RealSense depth camera. Specifically, we looked at the Intel RealSense D415 and D435 Depth Cameras that Intel launched in January 2018. See Figure 4.3. These cameras capture not just RGB data, but also depth data per pixel. Using these depth images, spatial mapping can be performed on an environment. Previously, developers and hobbyists used the Microsoft Kinect to gather RGB images plus depth data (RGB-D images). However, Microsoft terminated the Kinect in October 2017 and hobbyists were left with few great options. Intel's new depth cameras became the perfect replacement for the Kinect, especially at their \$150 and \$180 price point respectively. They boast a maximum depth resolution of 1280x720 at 90 FPS compared to the Kinect's 512x524 at 30 FPS [45]. See Figure

Figure 4.3: Intel RealSense Depth Camera D400-Series [9].

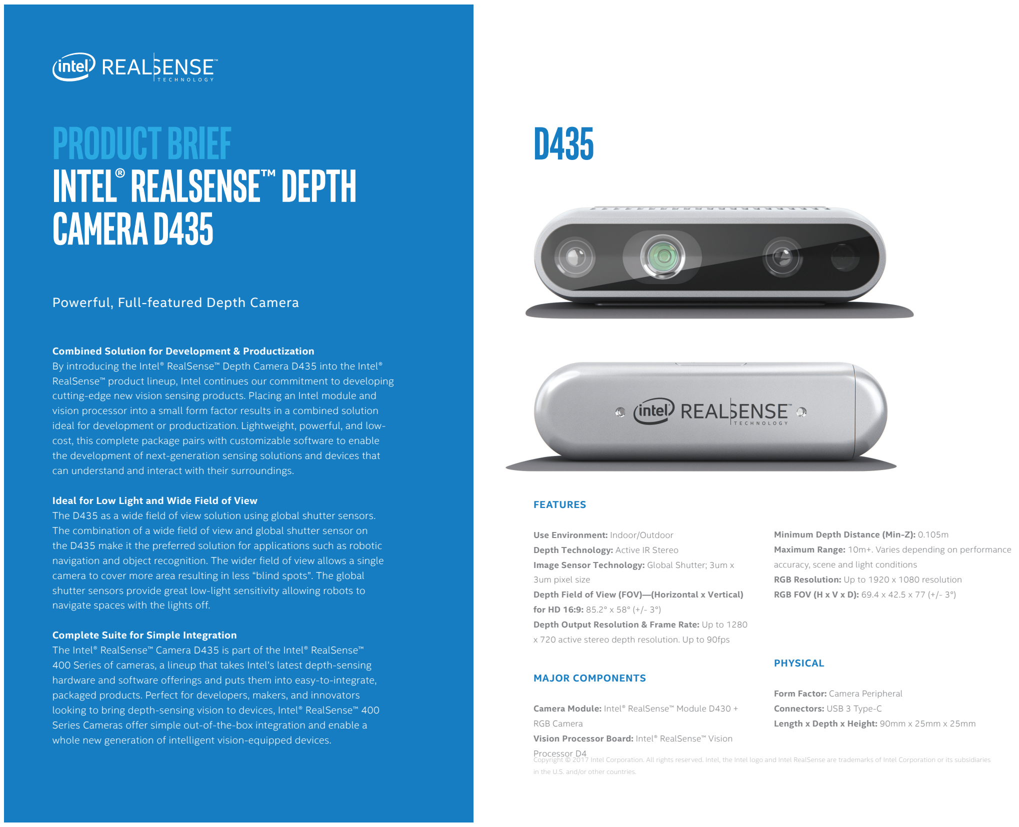
4.4 for additional hardware specifications [17].

The Intel RealSense D400 Depth Cameras were incredibly popular upon launch, such that Intel incurred shipping delays of several months from the overwhelming amount of orders. Upon hearing the announcement from Intel, we too ordered one of their cameras. Specifically, we ordered the Intel RealSense D435 Depth Camera, which had a wider field of view than the D415 (85 vs 63 degrees). Unfortunately, due to shipping delays, we were not able to acquire the camera until mid-March 2018.

One benefit the Intel RealSense cameras pose is access to raw depth data, which allows for more efficient processing of data than is possible using a spatial mapping provided by a third party framework. However, the developer is required to implement the algorithms to perform SLAM and generate the spatial mapping. Due to shipping delays by Intel and already obtaining access to a HoloLens, we chose to pursue development on the HoloLens.

4.1.4 Other Hardware

Other platforms that exist include the Microsoft Kinect, the Meta 2, and the upcoming Magic Leap One. See Figure 4.5. The Kinect was a depth sensor developed


PRODUCT BRIEF
INTEL® REALSENSE™ DEPTH CAMERA D435

Powerful, Full-featured Depth Camera

Combined Solution for Development & Productization
By introducing the Intel® RealSense™ Depth Camera D435 into the Intel® RealSense™ product lineup, Intel continues our commitment to developing cutting-edge new vision sensing products. Placing an Intel module and vision processor into a small form factor results in a combined solution ideal for development or productization. Lightweight, powerful, and low-cost, this complete package pairs with customizable software to enable the development of next-generation sensing solutions and devices that can understand and interact with their surroundings.

Ideal for Low Light and Wide Field of View
The D435 as a wide field of view solution using global shutter sensors. The combination of a wide field of view and global shutter sensor on the D435 make it the preferred solution for applications such as robotic navigation and object recognition. The wider field of view allows a single camera to cover more area resulting in less “blind spots”. The global shutter sensors provide great low-light sensitivity allowing robots to navigate spaces with the lights off.

Complete Suite for Simple Integration
The Intel® RealSense™ Camera D435 is part of the Intel® RealSense™ 400 Series of cameras, a lineup that takes Intel’s latest depth-sensing hardware and software offerings and puts them into easy-to-integrate, packaged products. Perfect for developers, makers, and innovators looking to bring depth-sensing vision to devices, Intel® RealSense™ 400 Series Cameras offer simple out-of-the-box integration and enable a whole new generation of intelligent vision-equipped devices.

D435

FEATURES

Use Environment: Indoor/Outdoor
Depth Technology: Active IR Stereo
Image Sensor Technology: Global Shutter; 3um x 3um pixel size
Depth Field of View (FOV)—(Horizontal x Vertical) for HD 16:9: 85.2° x 58° (+/- 3°)
Depth Output Resolution & Frame Rate: Up to 1280 x 720 active stereo depth resolution. Up to 90fps
Minimum Depth Distance (Min-Z): 0.105m
Maximum Range: 10m+. Varies depending on performance accuracy, scene and light conditions
RGB Resolution: Up to 1920 x 1080 resolution
RGB FOV (H x V x D): 69.4 x 42.5 x 77 (+/- 3°)

MAJOR COMPONENTS

Camera Module: Intel® RealSense™ Module D430 + RGB Camera
Vision Processor Board: Intel® RealSense™ Vision Processor D4

PHYSICAL

Form Factor: Camera Peripheral
Connectors: USB 3 Type-C
Length x Depth x Height: 90mm x 25mm x 25mm

Copyright © 2017 Intel Corporation. All rights reserved. Intel, the Intel logo and Intel RealSense are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

Figure 4.4: Intel RealSense Depth Camera D435 Specifications [17].

by Microsoft for its gaming consoles, the Xbox 360 and the Xbox One. Microsoft provided APIs that enabled full body tracking and even hand tracking in real-time. These technologies were further refined for the Microsoft HoloLens, which provides a spatial mapping of an environment. The Kinect was officially terminated in October 2017, when Microsoft announced they would stop producing the units [54].

The Meta 2 is another popular mixed reality headset. Created by the startup, Meta, it projects images onto a see-through display in front of the user. According to online reviews, the holograms displayed by the Meta 2 are not completely stable as the user moves around the environment compared to the HoloLens which has very accurate tracking. However, the Meta 2 boasts a much wider field of view, at 90 degrees compared to the HoloLens' 30 degrees. Priced at $949, it is also considerably cheaper than the HoloLens [51]. We did not pursue development with the Meta 2

Figure 4.5: Other hardware platforms. From left to right: Kinect v2, Meta 2, Magic Leap One [5, 11, 6].

due to acquiring access to a HoloLens and due to the HoloLens' enhanced spatial mapping and tracking capabilities.

The most interesting mixed reality platform is the Magic Leap One, created by the secretive company, Magic Leap. Founded in 2010, Magic Leap has developed its advanced mixed reality technology in secret, only showing the technology to a select few, including investors, under strict NDAs. In 2014, Magic Leap raised $540 million in venture funding with Google leading the pack [36]. As of May 2018, the company has raised over $2 billion and is valued at over $6 billion. The company also has the founder of Alibaba and the CEO of Google as board members [10]. In December 2017, Magic Leap finally gave the world a glimpse of its mixed reality platform, the Magic Leap One. In early 2018, they released a developer portal online and announced that the developer edition of the Magic Leap One would ship in late 2018 [18]. With billions of dollars invested, and support from industry leaders like the CEO of Google, Magic Leap is set to transform the mixed reality industry, which has remained relatively stagnant for the last two years. This would be an ideal platform for diminished reality, but due to it not being released, the HoloLens is the next best thing.

4.2 Development Environment

Equally important as the hardware chosen when implementing a mixed reality application is the development environment in which to create such application. With our chosen hardware platform of the Microsoft HoloLens, three development environments were available: Unity, Unreal Engine and DirectX. The following sections discuss each development environment, their tradeoffs, and why we chose to develop with Unity.

4.2.1 Unity

Unity is a cross-platform game engine developed by Unity Technologies [19]. Since its inception in 2005, it has quickly become one of the leading two game engines that developers use to create amazing games. It supports content creation on most platforms, including Windows Mixed Reality platforms like the Microsoft HoloLens. Large game development studios such as Activision often develop their own game engines, but for hobbyists and smaller game studios, Unity is a great tool that saves significant development time.

Unity provides a set of APIs which support complex graphics concepts. Due to Unity's abstraction of core graphics APIs, less time and code is needed by developers to create amazing content. Unity also supports development of applications for the HoloLens through the use of the Windows Mixed Reality APIs. Unity further provides a holographic emulator which allows applications to run on the computer rather than on the HoloLens so developers can perform tests prior to uploading their applications to the device. Uploading applications to the HoloLens takes time, so providing an emulator significantly speeds up the development process. Developers can also connect to the HoloLens remotely via the Holographic Remoting app on the HoloLens. This allows content to be streamed between the computer and the HoloLens. This allows the developer to see the output of an application in Unity on the device without actually uploading the application. These tools make Unity the development platform of choice when developing for the HoloLens. Microsoft has also published several tutorials focused on developing HoloLens applications with Unity.

4.2.2 Unreal Engine

The Unreal Engine is a cross-platform game engine developed by Epic Games [20]. Since its inception in 1998, it has continued to evolve and has become one of the leading two game engines preferred by game developers. It supports content creation on most platforms, including popular virtual reality and augmented reality platforms like the Oculus Rift, HTC Vive, and Magic Leap One. However, it does not officially support the Windows Mixed Reality platform. Instead, it supports SteamVR, which allows developers to create virtual reality games for Steam, a software distribution platform used to download and play games. Through SteamVR, developers can also interface with the Windows Mixed Reality APIs and create mixed reality content that can run on Windows Mixed Reality devices such as the HoloLens [50].

Unity and Unreal Engine both dramatically speed up development time of a game. Yet Microsoft has not published tutorials focused on developing HoloLens applications with Unreal Engine. Due to the support of Unity by Microsoft and the lack of immediate support of the Windows Mixed Reality platform by the Unreal Engine, Unity is the clear development platform of choice for the HoloLens.

4.2.3 DirectX

DirectX is a set of APIs created by Microsoft to support video processing and game programming on Windows platforms [2]. The X is a stand-in for different APIs including Direct3D, Direct2D, DirectSound, and many more. These APIs allow developers to integrate with low-level graphics hardware with fine-grain control. This also means that developers need to write more code to tell the graphics hardware exactly how to handle their data. Experienced developers can use these APIs to write more efficient algorithms than would be possible with a conventional game engine like Unity and Unreal Engine. DirectX was first released in 1995, has seen significant changes since its inception, and is currently at revision 12. Microsoft's Windows Mixed Reality platform supports applications created in DirectX and some tutorials have been provided.

When considering the software platform for this project, we also considered the amount of time it would take to develop a working solution. Due to our inexperience with DirectX APIs and the increased amount of code necessary when working with DirectX, we chose not to pursue development on this platform. Instead, we chose to work with Unity, which is equally supported by Microsoft when creating applications for the HoloLens. Unity also uses the DirectX APIs under-the-hood, but provides a software abstraction to make development faster.

4.3 Diminished Reality Pipeline

Upon choosing hardware and software development platforms, we developed a diminished reality solution following a pipeline consisting of six stages, occurring in order.

1. Spatial Mapping
2. 3D Post-Processing
3. Texturing
4. Object Selection and Tracking
5. Object Removal (Diminishing)
6. 2D Post-Processing

The following sections discuss our specific implementation of each stage.

4.3.1 Spatial Mapping

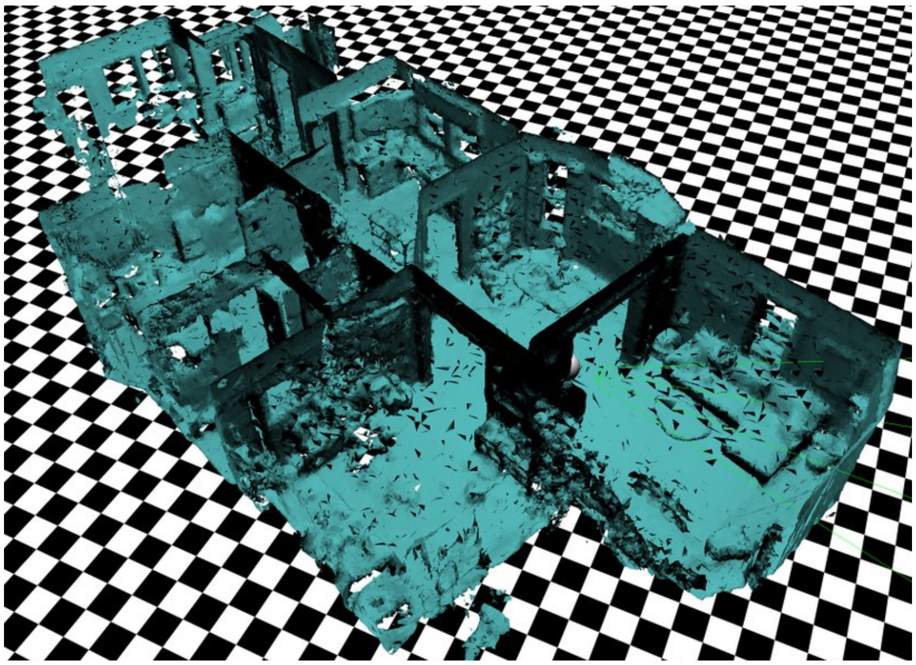


Figure 4.6: Triangle mesh of an environment obtained via spatial mapping [7].

For the spatial mapping stage, we used the internal representation of the environment provided by the HoloLens. Rather than provide developers with a set of points containing color and depth information, the HoloLens processes this raw depth data behind-the-scenes and provides developers with a mesh of the environment composed of triangles. See Figure 4.6. The density of the triangle mesh can be changed through an internal variable, named maxTrianglesPerCubicMeter. According to documentation, low, medium, and high resolutions are available at 100, 500, and 1000 triangles per cubic meter respectively [52]. From testing, we found that medium resolution was ideal. A spatial mapping can be obtained many times per second with low resolution, but with poor quality. At medium resolution, the spatial mapping is moderately detailed and can be obtained 1-2 times per second. At high resolution, the spatial mapping is only slightly better than at medium resolution and takes several seconds to obtain the mesh. See Figure 4.7.

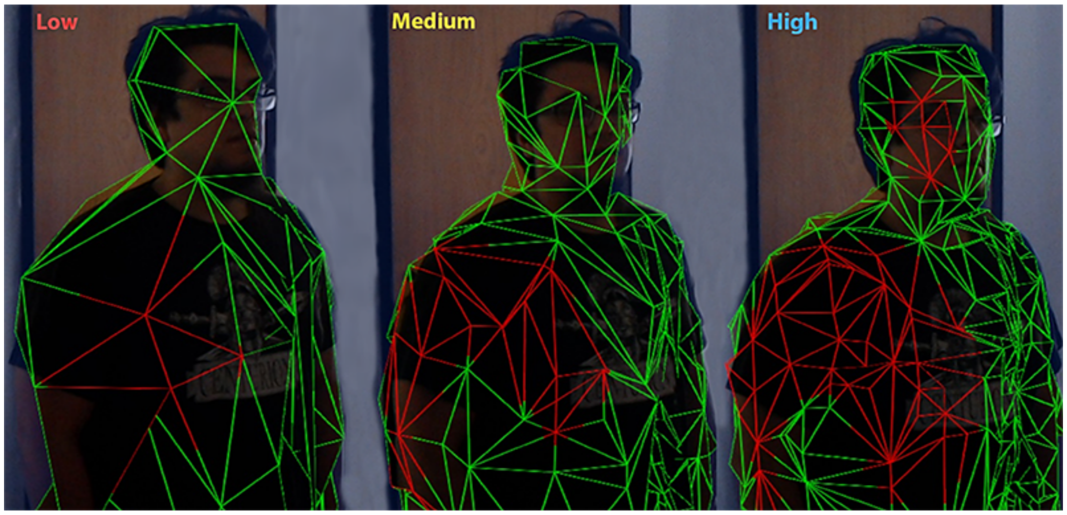


Figure 4.7: Low, medium, and high triangle densities [16].

To obtain the spatial mapping mesh from the HoloLens, we first create an instance of the SurfaceObserver class, set its observation area, and then request a mesh of the environment. The SurfaceObserver class is one of the many classes provided to interface with the Windows Mixed Reality APIs. The SurfaceObserver converts the HoloLens' raw depth data into a triangle mesh usable by developers, which is an expensive process. This triangle mesh is called a Surface, and many Surfaces can be created per observation area. The HoloLens scans the environment within 0.8-3.1 meters in front of it in a 70-degree cone. A Surface is then constructed from what the HoloLens scans over several frames. If the observable area is small, it will only be composed of a couple Surfaces. If the observable area is large, it will contain many Surfaces and become computationally expensive to update and render all Surfaces. To reduce processing time, a mesh is only constructed when the developer requests an update. The developer can request a mesh once or can assign a method to be called when the SurfaceObserver finds new, updated, or removed surfaces. Each Surface is also given an identifier so the developer can efficiently save, update, and delete old

Surfaces [3].

Microsoft provides an additional set of APIs and tutorials for development with Windows Mixed Reality called the Mixed Reality Toolkit [37]. The toolkit includes classes to help manage the spatial mapping of an environment as well as performing post-processing operations on the spatial mapping. Using these classes significantly saved development time and allowed easy access to the Surfaces obtained from the spatial mapping.

4.3.2 3D Post-Processing

In the 3D post-processing stage, several operations are performed to make the Surfaces obtained from the spatial mapping more accurate, such as removing disconnected points, smoothing surfaces, filling holes, and replacing flat surfaces with planes. The HoloLens automatically performs smoothing and removal of disconnected points during creation of the spatial mapping mesh. Further refinements are performed to manually remove remaining holes in the environment and to improve mesh visibility.

To find planes in the environment, we use the PlaneFinding class provided by the Mixed Reality Toolkit, which detects horizontal and vertical planes. Using this class, we obtain a list of planes in the environment, create two triangles to represent each plane, and then iterate through the Surfaces, removing any old vertices that overlap with the new plane. This process reduces the total number of triangles rendered, increasing performance a small amount. It further removes errors in the spatial mapping by using a flat surface to approximate walls, floors, and tables. Another benefit of plane finding is that any holes in the mesh that were identified as part of a plane will be filled, making the mesh even more accurate. This is especially important for the sides of an object touching a plane. The spatial mapping process cannot see the region between two touching surfaces and thus will not map it. Instead, objects appear to be part of the floor and walls instead of separate from them. See Figure 4.8. Plane finding remedies this issue by creating a surface between the object and the plane. This further improves object selection by making a clear distinction between the floor and an object.

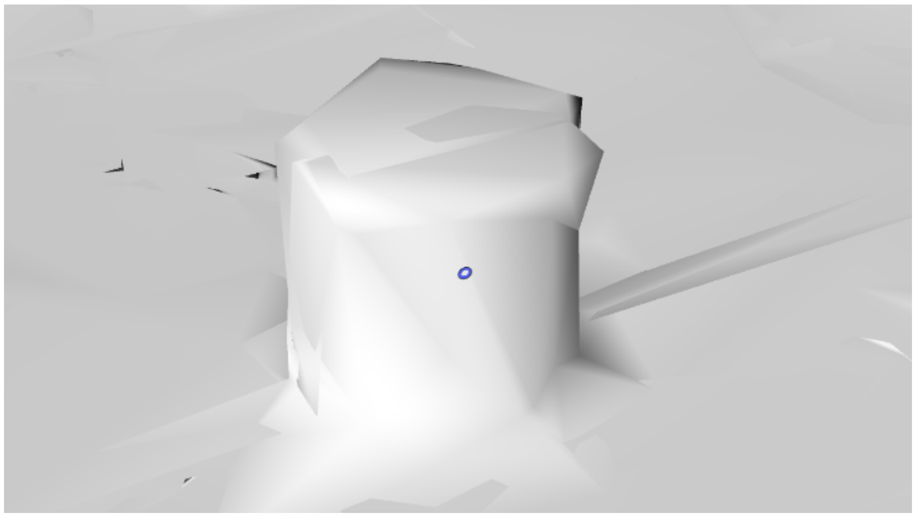


Figure 4.8: An object connected to the floor before post-processing.

Additional operations performed on the mesh include modifying its default shading. Lighting was removed in order to save on processing time. However, removal of lighting on a mesh that was arbitrarily colored white caused the entire mesh to appear the same intensity of white and the contours of the mesh could not be observed. Thus, we implemented a shader to color the mesh using each vertex's normal value, plus a black checkerboard pattern so that the real environment could also be seen - black pixels are considered transparent by the HoloLens. Each checkerboard square is 0.1 meters, which allows for easy measurement of real world surfaces. This proved sufficient to observe the contour and shape of the spatial mapping mesh. See Figure 4.9. In order to obtain the checkerboard normal shading, we added the vertex's world-space x, y, and z coordinates (each rounded down to nearest integer) and modded the result by 2. If the result was 0, black was drawn. If the result was 1, the normal color was drawn.

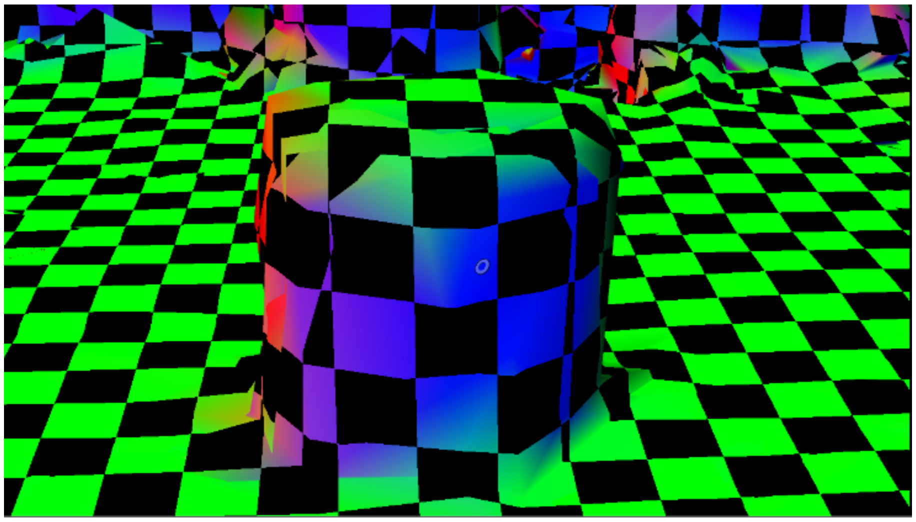


Figure 4.9: Checkerboard normal shading.

4.3.3 Texturing

The third stage in our pipeline is the texturing of the spatial mapping mesh. This stage is necessary to obtain a realistic virtual representation of the real environment. Due to the lack of color provided in the HoloLens' spatial mapping, manual capture of environment textures is necessary. For this process, we utilize the HoloLens' RGB camera to take pictures of the environment at runtime and store this color data along with the location where each image was take within the environment. The user triggers the capture of an image by performing an air-tap gesture or through the use of the HoloLens' remote. Upon capturing an image, audio feedback is provided to the user to indicate completion of the image capture. The air-tap gesture is performed by quickly moving one's pointer finger down and then back up while also within view of the front-facing HoloLens sensors. The resulting image is then displayed on the spatial mapping mesh so the user can see which areas have been textured and which have not. In order to texture the environment mesh, two techniques are explored: projective texture mapping and sparse voxel octree texture mapping.

Projective Texture Mapping

Projective texture mapping is the projection of a texture onto a mesh, much like the projection of a movie onto a screen in real life. In our case, we use multiple projectors to project textures onto different sections of the mesh. This allows us to paint the virtual environment with colors from the real world. See Figure 4.10.

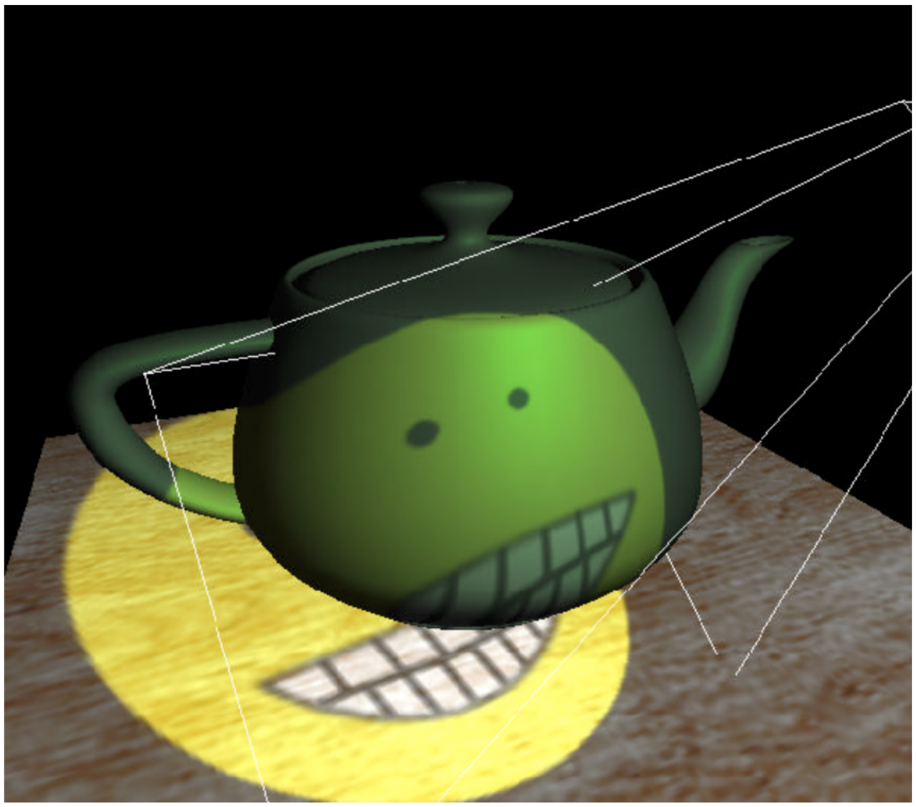


Figure 4.10: Projective texture mapping with a single projector [25].

Projective texture mapping on the HoloLens involves acquiring a picture from the HoloLens' RGB camera, saving the image along with the camera's view and projection matrices, sending the texture and camera information to the shader responsible for texturing the spatial mapping mesh, and then calculating the texture coordinates for each pixel in view of the main camera by transforming the pixel to the coordinate space of the projector. The following paragraphs explain this process in more detail.

Resolution	Horizontal Field of View
1280x720	45 degrees
2048x1152	67 degrees
1408x792	48 degrees
1344x756	67 degrees
896x504	48 degrees

Table 4.1: HoloLens camera resolutions, supported at 30, 24, 20, 15, and 5 fps [53].

To acquire an RGB image from the HoloLens' camera, we first create an instance of the PhotoCapture class provided by the Windows Mixed Reality framework. We then specify camera properties such as resolution, pixel format, and hologram opacity. The HoloLens supports five different resolutions that can be acquired at various framerates [53]. See Table 4.1. We use the default pixel format of BGRA32, hologram opacity of 0.0f, and resolution of 1280x720. Photo mode is then asynchronously enabled for the PhotoCapture object and an image is asynchronously captured upon calling the method to capture an image. The resulting image is provided to the developer in a callback, wrapped in an instance of a PhotoCaptureResult class. We then copy the image data from the PhotoCaptureResult into a Texture2D to more easily use the data throughout our application.

The next step is to save the view and projection matrices from the camera's perspective when capturing the image along with the Texture2D. This allows us to transform pixels from the main camera's coordinate space into the coordinate space of the projector. The result of the photo capture callback provides an instance of a PhotoCaptureFrame, which contains methods to obtain the view and projection matrices used by the camera to generate the image. These matrices are extracted and stored in a struct along with the texture and sent to the shader for the spatial mapping mesh.

Sending data to shaders in Unity is straightforward. We attach a script to an object, obtain a reference to the material used by the spatial mapping mesh, and then call the material's SetTexture() and SetMatrix() methods with the name of the variable in the shader as the first parameter and the data as the second parameter. In the shader, we define a Property for each texture and each matrix, create a sampler2D for each texture and a float4x4 for each matrix, and use the values in the fragment shader. Unity shaders can use the standard vertex and fragment shader stages, or can use a surface shader stage which compiles into a vertex and fragment shader stage. We chose the former method due to our familiarity with OpenGL vertex and fragment shaders.

The final step is to use the view and projection matrices sent to the shader to transform a pixel from the current camera's perspective into the perspective of the projector. To do this, we pass the vertex's world location from the vertex shader to the fragment shader, which automatically interpolates it. We then multiply the projector's view and projection matrices by the interpolated world position and divide the result by the w component of the vector to obtain the coordinate from the projector's perspective in the range [-1,1] in the x and y axes. We then apply a viewport transformation (add 1 and then divide by 2) to obtain the coordinate in the range [0,1] in the x and y axes. If the coordinate is outside this range, then the projector did not see it. For any pixel whose coordinate is within this range, we sample the projector's texture using the x and y components of this coordinate to get the color for the pixel. All other pixels use the default checkerboard normal shading described in the previous stage of the pipeline.

At this point, we have achieved a simple form of projective texture mapping. However, our projector paints its texture onto all triangles within its view frustum, whether or not they are hidden. See Figure 4.12. In addition to this, our projector suffers from back projection, where the texture is applied to triangles behind the projector as well as in front of it. See Figure 4.11. Finally, when using multiple projectors, some type of blending must be implemented when images overlap in order to prevent obvious edges and to correct color discrepancies. See Figure 4.13. The following paragraphs discuss how we address each of these issues.

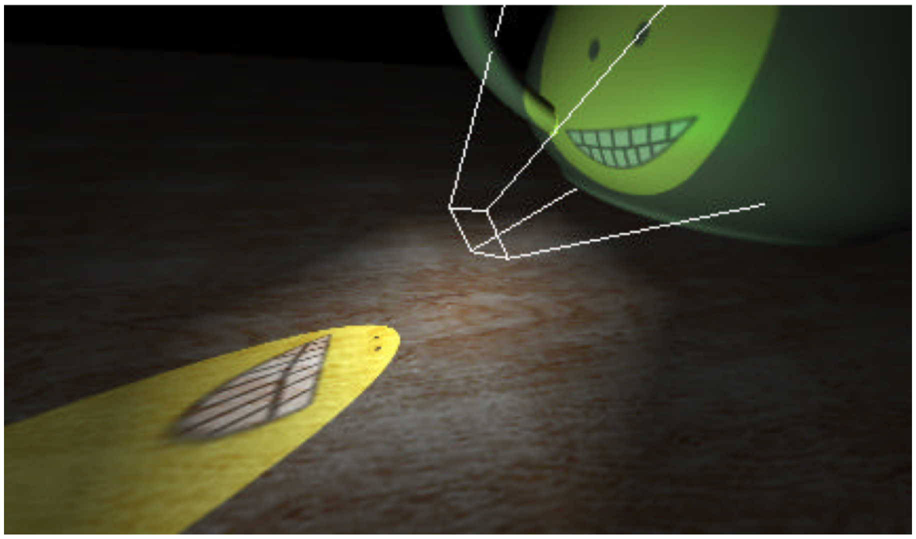


Figure 4.11: The problem of back/reverse projection [25].

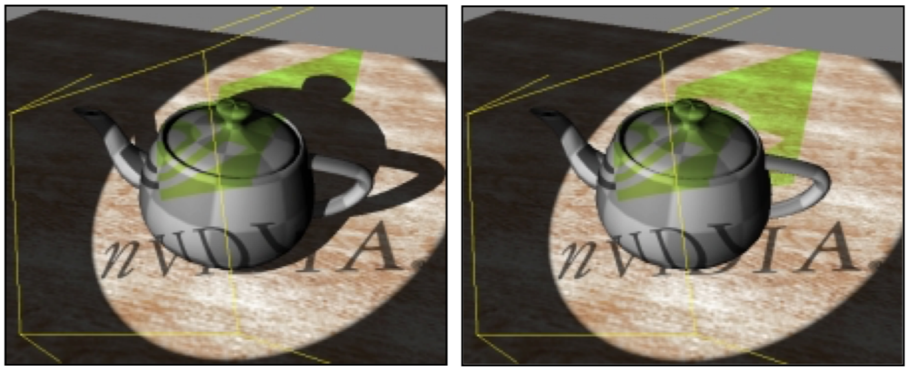


Figure 4.12: The problem of projecting onto triangles that the projector can't see [26].

Back projection is the simplest issue to resolve. To prevent back projection, we do not sample the projector's texture if the transformed coordinate's z value is negative.

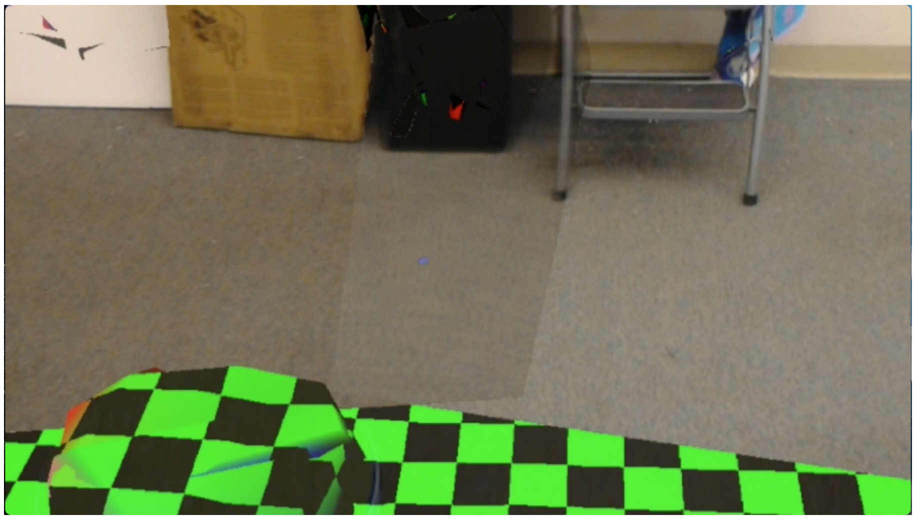


Figure 4.13: Blending error on overlapping images.

To allow multiple projectors, we send each projector's texture and matrices to the same shader. From there, each pixel is transformed into each projector's perspective to determine if the pixel is within the projector's view frustum and should derive its color from the given projector. If two or more projectors overlap, their color values are combined using a method similar to the one proposed by Debevec et al. in 1996 [23]. Each projector's color value is assigned a weight, where the sum of all weights is 1 and each weight is in the range [0,1]. The weights are inversely proportional to the magnitudes of the angles between the normalized view vector and projection vectors. See Figure 4.14. Using this method, the contribution of each texture is dependent upon viewing angle, unless it is the only texture for a given pixel. Additionally, to prevent hard edges along the border of the textured region, we increase the alpha component of the color near the edges to make it smoothly transition to transparent.

Projecting a texture only onto visible triangles is resolved through the use of shadow mapping techniques. Shadow mapping techniques determine whether a pixel is visible to a light source by storing a depth map of the scene from the perspective of each light source. We use this technique to determine if a pixel is visible to a

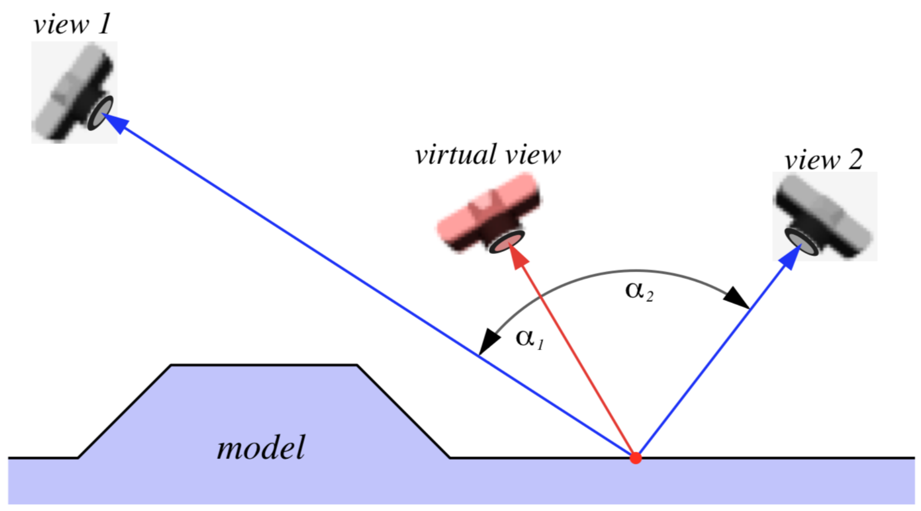


Figure 4.14: The weighting function used in view-dependent texture mapping. Weights w1 and w2 are inversely proportional to the magnitude of the angles a1 and a2 [23].

projector by storing a depth map per projector. However, depth data is not provided with the HoloLens' RGB camera output. Instead, we use the spatial mapping mesh that was created from the Spatial Mapping stage of the pipeline and render it into a buffer for later use. For this, Unity provides the RenderBuffer class. Unity allows a Camera game object to render the scene into a RenderBuffer object instead of rendering it to a display. To use this functionality, we create a new Camera object as a child of the main camera to allow it to match the main camera's position and orientation at all times. We further set the render target of this new Camera object to a RenderBuffer instance with resolution of 1280x720 to match the RGB image output from the HoloLens. When rendering to the depth buffer in Unity, a Camera's depth texture mode must be set to Depth or DepthNormals. We also set the depth buffer's precision to 32, which is the highest resolution setting, allowing 32 bits of information per pixel. The result is a Camera game object capable of rendering the scene into a buffer from the main camera's perspective.

The next step is to use these depth buffers to determine which projectors are visible to each pixel. An issue we ran into was that Unity only allows a single camera's depth buffer to be bound and readable by a shader, yet we needed to access depth data from multiple projectors within the same shader. To circumvent this constraint, we render the scene per projector with an extra shader that converts the depth texture into an RGB texture with grayscale values. Upon obtaining the new depth texture per projector, we send the textures to the spatial mapping shader along with projector-specific view and projection matrices used to render the depth image, which are different than the HoloLens' RGB camera's view and projection matrices. Within the shader, we transform each pixel's position in the same way as before, but using the new depth-specific view and projection matrices. This allows us to compare the depth values in each projector's depth map to the current pixel's depth values seen from the main camera's perspective. If the computed depth value is smaller than the value stored in the depth map, then the pixel is visible for the given projector. If the computed depth value is larger than the value in the depth map, then it must have been hidden when the projector's depth map was created. Using these values, we ensure pixels are only textured if they are visible to a projector. See Appendix A for code.

The final result is a spatial mapping mesh that is textured with camera images and smoothly blended when images overlap. See Figure 4.15.

An advantage of this approach is that detailed texture information is preserved. The closer an image is taken to a surface, the more detailed the texture will be. This approach also avoids the need to keep track of a texture atlas that maps texture coordinates to triangle vertices.

A limitation of this approach is poor scaling in computational complexity. The calculation of a pixel's color scales linearly with the number of images that are taken. This is due to the comparison of every pixel with every projector. However, no noticeable difference was observed in the use of 1 image vs 10 images.

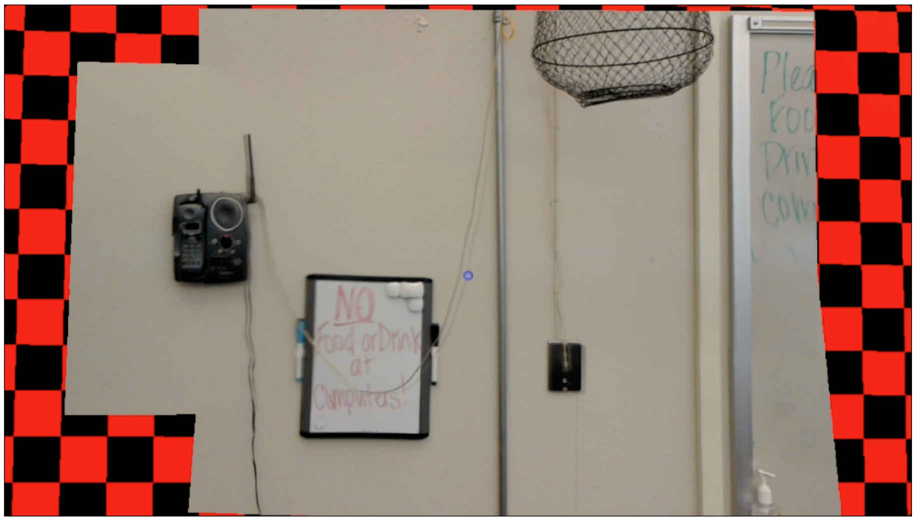


Figure 4.15: Spatial mapping textured with three overlapping images of the real environment.

Sparse Voxel Octree on GPU

Our second approach is texture mapping with sparse voxel octrees (SVO). Rather than use triangles to represent geometry within a scene, volume elements called voxels are used. A voxel represents a location on a regularly sampled three-dimensional grid where each voxel stores some kind of data specified by the developer. Voxels are often visualized as 3D cubes tightly packed within a larger volume. See Figure 4.16. A voxelization of a scene is the conversion from a triangle representation to a voxel representation.

In our implementation, we use a sparse voxelization of scene data. In contrast to a dense voxelization, where most or all of a volume is filled with voxels, a sparse voxelization does not fill empty space with voxels. This saves a significant amount of memory as most scenes contain vast amounts of empty space.

To improve performance of spatial queries, we store our scene voxelization in an octree. An octree begins with a root node, or voxel, which encompasses the entire scene. This node is split into 8 child nodes, which represent different equal-sized

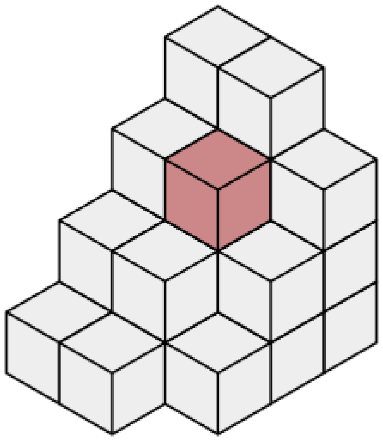


Figure 4.16: A volume element known as a voxel [8].

volumes within the root node. Each of these nodes is further split into child nodes until a desired level of detail is achieved. See Figure 4.17. The nodes at the lowest level contain the actual data while higher-level nodes contain averages of their children's data. When using a sparse voxel octree representation, a node is only split into eight children if data is found that belongs to a voxel within the node's volume.

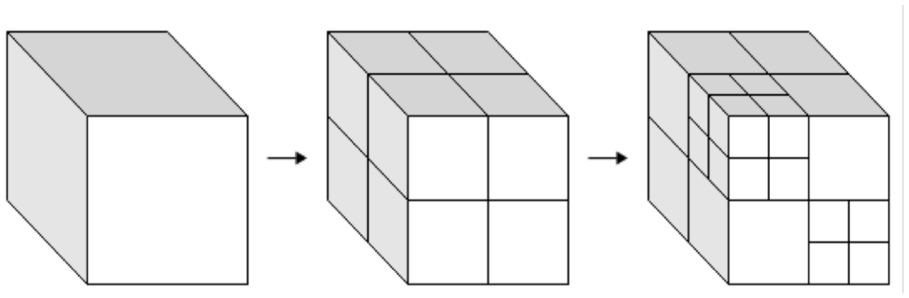


Figure 4.17: An octree of varying resolutions [4].

In our case, we use a SVO to store color data from pictures taken at different locations throughout the environment. This is accomplished by corresponding each pixel to a voxel by using the pixel's 3D position within the scene. However, pixels only represent colors at 2D screen coordinates. To obtain a pixel's 3D position, the depth of the pixel within the scene is needed. We obtain this depth information by rendering the spatial mapping mesh into a depth buffer from the viewpoint of the camera every time an image is taken. Using this depth data, we then unproject the 2D pixel to a 3D coordinate, map the 3D coordinate to a voxel, and store the color data within the voxel.

Typical projection of a 3D point to a 2D pixel coordinate occurs through these steps:

1. Transform the point from world space to normalized device coordinates (values in range [-1,1]) by multiplying the view and projection matrices with the point.
2. Apply the perspective divide by dividing each component (x,y,z) of the point by the point's w component.
3. Convert the point to the range [0,1].
4. Transform the point into screen space coordinates by multiplying the x and y components by the screen width and height respectively.

Unprojection occurs through a similar process, but in opposite order:

1. Divide the pixel's x and y components by the screen's width and height respectively.
2. Convert the point to normalized device coordinates (values in range [-1,1]).
3. Set the point's z value from the depth data, converted to the range [0,1].
4. Transform the point to world space by multiplying by the inverse perspective and inverse view matrices.

After obtaining the 3D position for a pixel, we determine which voxel it belongs to and fill that voxel with the color of the pixel. After all pixels have been stored in the sparse voxel octree, we use the same process to extract these values from the SVO to color the environment.

Before diving into the details of our implementation, we first discuss the structure of our SVO. In our implementation, an octree consists of a 2D array of int values, one dimension representing voxels and the other dimension representing data within a voxel. Nodes in the octree are also represented as voxels. Due to this structure, voxels contain 8 ints representing child node indices, 3 ints representing RGB color values, and 1 int representing a temporary value used in construction of the SVO. Therefore, the total size of a voxel is 12 ints, or 48 bytes.

Using this information, space limitations of our SVO are calculated. Image resolution is 1280x720 and each pixel in the image corresponds to at most 1 voxel. By using an octree data structure, for every eight voxels at a specific level of detail, there is one parent voxel. This approximates to 1.15 times the space requirements without an octree. However, use of an octree improves spatial queries from O(n^3) to O($log(n)^3$), n being the number of voxels in one dimension. To calculate the upper bound of the total space required by our SVO, we use this equation:

$$ImageCount * ImageResolution * VoxelSize * OctreeFactor \tag{4.1}$$

When only using a single image, the upper bound of the total space required is 1280 * 720 * 48 * 1.15 = 50,872,320 bytes = 50.87 MB. However, video memory on the HoloLens is limited to 114 MB [44]. Thus, we restrict our SVO to a maximum of 100 MB. We further restrict our voxel size to 1x1x1 cm to support more images. Using larger voxel sizes causes many pixels to map to the same voxel, which reduces the total number of voxels, thereby also decreasing the resolution of the texturing. Pixels from overlapping images will also map to the same voxel, saving additional space. In practice, at least 8 images can be captured and used to texture the environment.

Our implementation of sparse voxel octree texture mapping consists of two phases:

constructing the sparse voxel octree and texture mapping with the built sparse voxel octree. The SVO is only constructed once and then used per frame to texture the pixels seen by the main camera. Construction occurs after all images of the environment have been taken. This ensures spatial mapping is complete and a consistent depth map is used for all images. Texture mapping is then performed per pixel per frame, where each pixel is mapped to a voxel within the SVO and its color determined from the voxel.

During the construction phase, we build the SVO level by level, starting with the largest voxel sizes at the top layer of the octree. This supports construction on the GPU and allows our SVO to be built as quickly as possible.

We construct the SVO on the GPU per level of the octree in two steps: flagging and building. The first step involves flagging all nodes that contain pixels for a given level of the octree. The second step involves creating the next level of the octree based on the flagged nodes. The octree itself is allocated prior to construction to the maximum size that will fit in video memory (100 MB).

Prior to performing the flagging step, we first convert our image data into a structured buffer for easier processing. Each struct in the buffer contains 3 ints representing an RGB color, 3 floats representing a 3D position, and 1 int representing the index of the last node traversed in the octree for this pixel. There is a one-to-one mapping of pixel to struct, making this an ideal candidate for parallelization. To convert the image data into a structured buffer, we use a compute shader. Compute shaders allow arbitrary batch computations to be performed on the GPU instead of the CPU, taking advantage of many cores that can perform operations simultaneously. Our compute shader runs per pixel and takes an RGB image, depth image, camera matrices, and empty structured buffer as input. It then calculates the 3D position of the pixel and creates a struct with the data mentioned above. The node index is set to 0. See Appendix B for code.

After converting the images to structured buffers, we perform the flagging and building steps incrementally for each level of the octree. We also use compute shaders to parallelize these steps. For the flagging step, a compute shader is run for every struct in our structured buffer. The position for each struct is used to traverse the SVO up to the current octree level, where the node containing the position is flagged by setting the node's index within its parent node to 1. Additionally, the color for the struct is added to the node and the node's temporary variable is incremented and used to count the total number of pixels within the node. The count of pixels is stored so we can later compute the average color for each node. We further set the temporary variable within the struct to the node's index to speed up octree traversal in later iterations by skipping already-flagged levels of the octree. Upon completion of the flagging step, the node indices of the octree level currently being processed will either be 0 or 1 to indicate whether they contain pixels or not. See Appendix B for code.

After flagging the nodes in the current level of the octree, we build the next level of nodes. For this step, a compute shader is run for every node in the current level of the octree. If the node's index within its parent node is 0, then the compute shader exits early. If the node's index is 1, then the compute shader synchronously gets the next available unused voxel index in the preallocated octree array. The next available unused voxel index will always be a multiple of the size of a voxel and will be constantly increasing. Upon synchronously reading this value, the compute shader will synchronously increase this value by the size of one voxel. The synchronization of compute shaders guarantees that each will get a unique index into the octree array and that no collisions occur. The build step is skipped for the final level of the octree. See Appendix B for code.

After all levels of the octree have been built, a final compute shader pass is used to average the colors for each node. During the flagging step, a pixel's RGB color was added to the node it was found within and a pixel counter on the node was incremented. We now divide the RGB values per node by the total pixel count per node to obtain the average color of all pixels within the node. See Appendix B for code.

After constructing the SVO, we use it every frame to sample the texture per pixel of the spatial mapping. We first send the SVO to the GPU so it can be accessed within the spatial mapping shader. We then render the spatial mapping into a depth texture for the main camera and send it to the spatial mapping shader. Using the main camera's depth texture, each pixel's 3D coordinate is calculated. Finally, the SVO is traversed using the 3D coordinate until a sufficiently small voxel is found. The color stored in this voxel is returned as the pixel's color. If no sufficiently small voxel is found, the default color is used.

An advantage of SVO texture mapping is logarithmic performance scaling in contrast to linear scaling with projective texture mapping. The size of the SVO increases with each image taken, but the time it takes to find the voxel for a given pixel increases logarithmically. Another advantage is that the SVO is built level by level, which allows early stopping if memory constraints become an issue while maintaining a usable SVO of lower resolution.

A disadvantage of SVO texture mapping is the large amount of memory used in the creation of the SVO. Additionally, due to memory constraints on the HoloLens, voxel sizes must be large to allow texture mapping of an entire scene. However, large voxel sizes reduce the amount of detail that can be captured in the SVO.

4.3.4 Object Selection and Tracking

The fourth stage in our diminished reality pipeline is selecting the object to be removed and tracking it per frame. In our implementation, object tracking is automatic due to the SLAM being performed during the spatial mapping stage. SLAM continuously determines the camera's position relative to the origin of the environment and all 3D locations are mapped relative to this origin as well. Once an object has been selected, its 3D location within the spatial mapping is all that is needed to locate the object per frame.

In order to select the object to be removed, we use a semi-automated process, where the user provides a point in 3D space associated with an object and then the object is automatically determined from their input. This is in contrast to manual techniques where the user draws the entire outline of the region they want removed and automated techniques where an algorithm determines the object to remove without user input. User input is provided once and then the object is tracked per frame.

To obtain user input, we make use of the HoloLens' ability to track hand gestures and head movement. Specifically, we use the built-in air-tap hand gesture, where a user makes a fist, points their index finger, and then quickly moves their index finger down and back up. Performing this action in front of the HoloLens triggers a callback in code, where we perform custom actions. Upon detection of this hand gesture, we access the HoloLens' position and orientation in 3D space and cast a ray from the center of the HoloLens into the virtual scene along the viewing direction. The first point that intersects the ray is used as the selected user input point. Feedback is provided to the user in the form of a small circle in the center of their display to show which point they are currently looking at, as well as a sound effect when the air-tap gesture is recognized. This allows the user to provide more accurate input and to know when their input has been recognized.

After obtaining the user's selected point, we automatically determine the object to remove. We define the object as the collection of vertices within a set distance from the selected input point and the triangles containing such vertices. We acquire these vertices by looping through the spatial mapping mesh and create a copy of every triangle that contains at least one vertex within the defined radius. We also ignore triangles belonging to the floor and walls. We limit object selection to objects larger than 1 ft in diameter due to the HoloLens' spatial mapping not being capable of accurately capturing the geometry of smaller objects.

Limitations of this object selection approach include only selecting a portion of a large object and selecting multiple objects that are in close proximity. Since the radius is a fixed size (1 meter in our implementation), other objects within this close proximity can accidentally be captured as part of the selected object. Also, objects that are larger than the set radius will not be captured in their entirety unless the radius is expanded. For best results, objects that are spatially separated from other objects by at least 0.5 meters and that fit within the set radius ought to be chosen.

4.3.5 Object Removal (Diminishing)

The fifth stage in our diminished reality pipeline is the perceived removal or diminishing of the selected object. To diminish the selected object, we display pixels in front of it that look like the region behind it. We accomplish this by removing the selected object's vertices from the spatial mapping mesh and then render the textured spatial mapping only within the region of the selected object. This allows the user to see the real world everywhere except in front of the selected object, where they instead see the virtual textured environment behind the object.

Removing the selected object from the virtual mesh of the environment is similar to the automated object selection process, except we remove triangles instead of copying them. This involves iterating over the vertices in the spatial mapping and removing any triangles that contain vertices within a specified radius (1 meter) of the originally selected point by the user. In our implementation, this step is combined with the automated object selection process of the previous stage to improve performance.

After removing the selected object from the virtual copy of the environment, we clean up the geometry of our selected object and create an approximate representation of it with no concavities, called a convex hull. See Figure 4.18. This approximation is used to alleviate spatial mapping errors which cause the selected region to be too small or to entirely miss parts of the object. The spatial mapping is not detailed enough to capture small objects on top of surfaces and may also contain irregularities within the mesh. See Figure 4.19. By approximating the selected object with a convex hull, we increase the likelihood of capturing the entire bounds of the real object. We also expand the convex hull by 0.05 meters to better capture corners and edges of an object. We use an open source implementation of the quick hull algorithm for generating a convex hull on GitHub called MIConvexHull, created by DesignEngrLab [24].

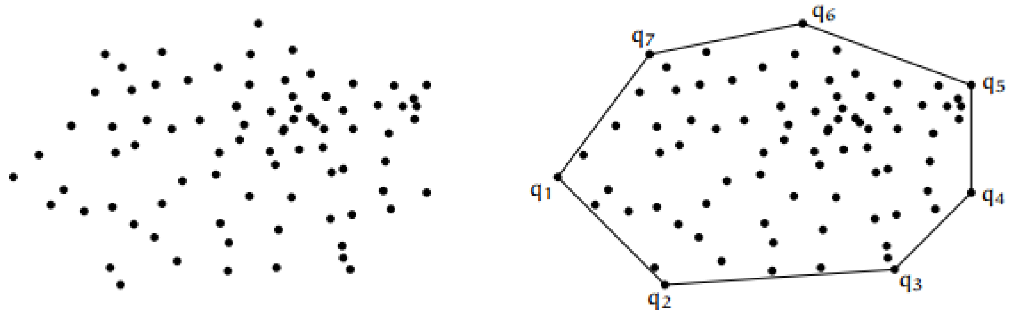


Figure 4.18: Convex hull generation from a collection of points. Left is the collection of points. Right is the convex hull around the collection of points.

After creating a convex hull of our selected object geometry, we use a stencil buffer to render the virtual environment only within the region of the selected object's convex

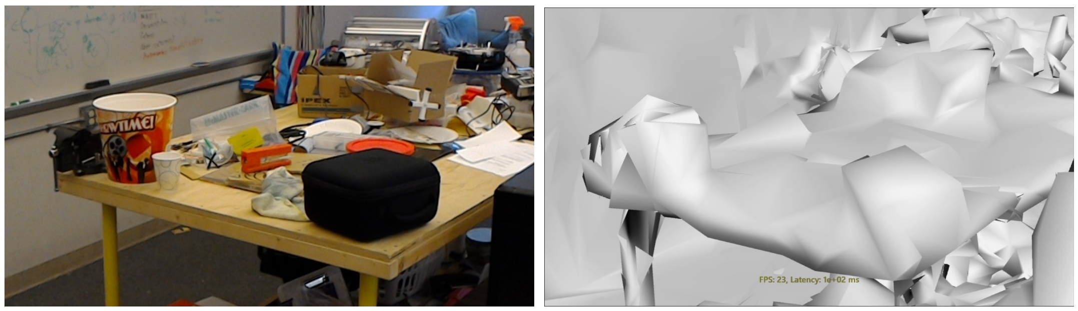


Figure 4.19: Spatial mapping of small objects on a table. Left is the original image. Right is the spatial mapping at 500 triangles per cubic meter.

hull. The stencil buffer is a special per-pixel integer buffer often used to render a scene into a specific region of the screen. In our implementation, we make use of the stencil buffer by performing two render passes. The first pass renders the selected object without color and sets the stencil buffer for each pixel that contains the object to 2. The second pass renders the virtual environment, but only in pixels where the stencil buffer is set to 2 from the previous render pass. This ensures the virtual environment is only rendered in pixels that contain the selected object. The final result is an image of the real environment with pixels from the virtual environment rendered in front of the selected real object.

4.3.6 2D Post-Processing

The final stage in our diminished reality pipeline is 2D post-processing on the resulting image from the previous stage to enhance the realism of the diminished result. Specifically, we perform inpainting to fill regions where the virtual environment is untextured and perform alpha blending near the edges of the diminished region to hide obvious color discrepancies.

Untextured regions appear when the user has not viewed the surfaces behind the selected object and when spatial mapping fails to generate geometry for surfaces far from the user. During these scenarios, inpainting is used to fill in the holes.

Specifically, we use the OpenCV implementation of inpainting, provided by Enox Software through the Unity asset, "OpenCV for Unity" [14]. OpenCV provides two methods of inpainting, the first being an implementation of a 2001 paper by Bertalmio et al. [22] and the second being an implementation of a 2004 paper by Telea [49] [15]. Both papers focus on quickly inpainting small regions within an image. See Figure 4.20. These inpainting techniques allow us to run inpainting at real-time rates and fill holes for every frame that requires it.

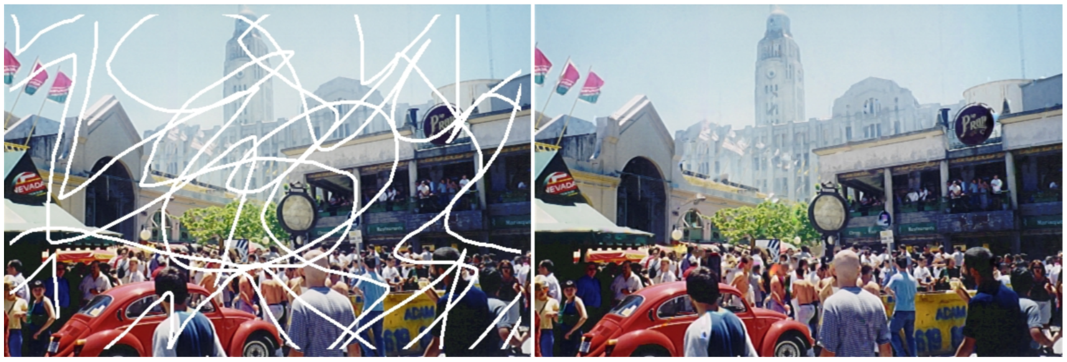


Figure 4.20: Inpainting of small strokes in an image [22].

In our implementation, we use the OpenCV inpainting method based on the 2004 paper by Telea. Inpainting is performed by providing the algorithm with two images: a source image and a mask image. The mask image is a grayscale image used to determine which pixels to inpaint within the source image. White pixels within the mask image are inpainted while black pixels are not. To support easy creation of a mask image, we modify the spatial mapping shader to display bright green (00FF00) when no texture information exists for a pixel. We then use a shader to create the mask image by converting green pixels to white and all other pixels to black. This mask image is fed into the inpainting algorithm along with the original image provided by the previous stage of the pipeline. The resulting image is one where the green pixels have been replaced with colors similar to their surroundings. See Figure 4.21.

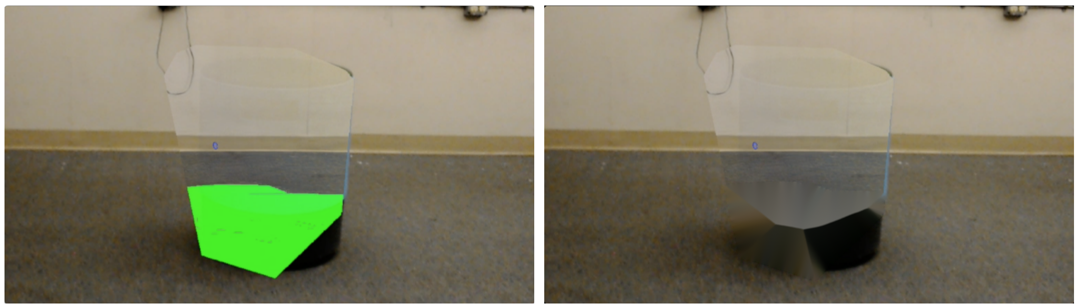


Figure 4.21: Inpainting green pixels.

The final post-processing step we perform is fading the edges of the diminished region. When diminishing a region within an image, lighting inconsistencies often cause seams to appear at the edges which degrade the realism of the experience. To resolve this, we linearly fade the alpha component from 1 to 0 for all pixels within 20 pixels from an edge based on distance from the edge. Distance to edge for every pixel is determined by passing the image mask into the OpenCV distanceTransform() method, which labels every non-black pixel based on its Euclidean distance to the nearest black pixel. We then pass this distance mask image into a shader to modify the alpha values of pixels near edges of the diminished region. Any pixels with an alpha value of 0 are rendered as completely transparent by the HoloLens whereas alpha values of 1 are rendered as completely opaque. The result is a smooth transition between the diminished region and reality.

Chapter 5

RESULTS

This thesis demonstrates the perceived removal of an object in an augmented reality context. Unity 2017.4.3f1 was used to develop our solution. Computation solely takes place on the HoloLens. Images were captured using the Mixed Reality Capture tool on the Windows Device Portal. We first compare our results internally. Both texture mapping algorithms are analyzed as well as our results in simple and complex environments. Then we compare our results to previous work.

5.1 Internal Comparisons

The first qualitative comparison we make is our solution with itself in various environments and with different texture mapping algorithms. We compared our texture mapping algorithms in a simple environment consisting of a single well-defined object, a ground plane, and a wall plane. Both texture mapping approaches use the same images captured of the real environment. Projective texture mapping provided high resolution results but suffered from linear scaling in time and space complexity with the number of images taken. Sparse voxel octree texture mapping provided lower resolution results but maintained logarithmic scaling in time complexity and constant space complexity due to pre-allocating the SVO. Further, the result of projective texture mapping was immediately viewable after each image was taken, whereas our other approach required all images to be taken prior to constructing the sparse voxel octree. Comparisons are summarized in Table 5.1.

We then compared our solution in simple vs complex environments. A simple environment was defined to be an environment consisting of a single well-defined object

Projective Texture Mapping	SVO Texture Mapping
High resolution	Low resolution
Linear time complexity	Logarithmic time complexity
Linear space complexity	Constant space complexity
View immediately	View after taking all images

Table 5.1: Texture mapping comparisons.

with ground and wall planes. A complex environment was defined to be an environment consisting of multiple objects with many planes. Our solution performed best in the simple environment. When multiple objects existed in the environment, object selection was less accurate due to accidental selection of nearby objects. Further, the large number of planes in the complex environment required many images to be taken to perform texture mapping, reducing performance and increasing the chance of overlapping images. However, our solution provided convincing results in simple environments and adequate results in complex environments. See Figure 5.1.

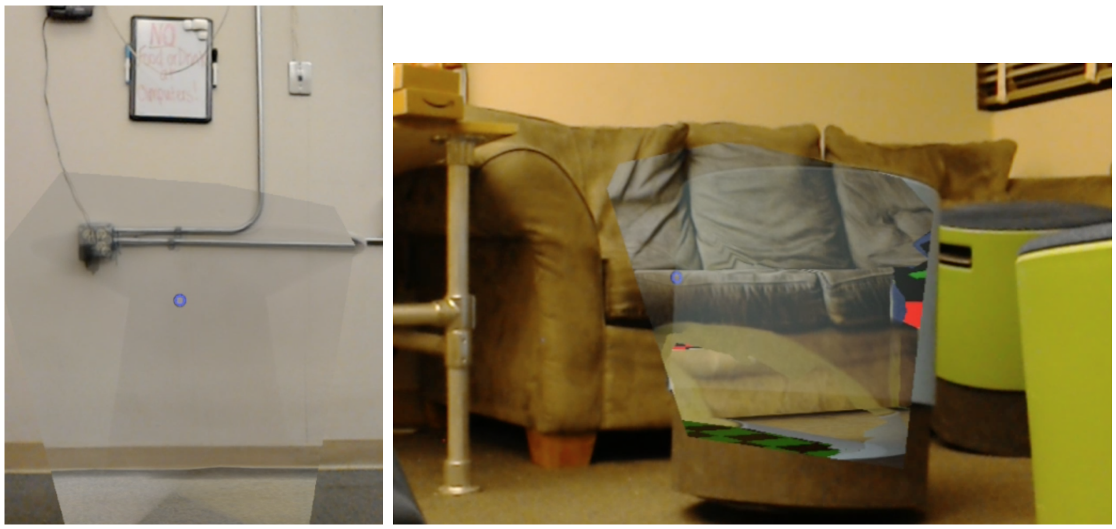


Figure 5.1: Simple vs complex environment. Left is a simple environment containing a single table. Right is a complex environment containing many small stools.

5.2 Previous Work Comparisons

The second qualitative comparison we make is that of previous work vs our solution. Much research has been conducted in the field of diminished reality, but few have used a virtual representation of the environment to accomplish diminished reality in arbitrarily complex environments. Two such examples exist; the first method was proposed by Simpson and the second by Nakajima et al. Simpson achieved diminished reality using a Google Tango device, which mapped the environment in much the same way the HoloLens does, yet also provided texturing. Simpson's results can be seen in Figure 5.2. The spatial mapping obtained by Simpson is more dense than the one obtained via the HoloLens and thus provides more accurate object selection. However, our method provides higher resolution texturing of the environment.

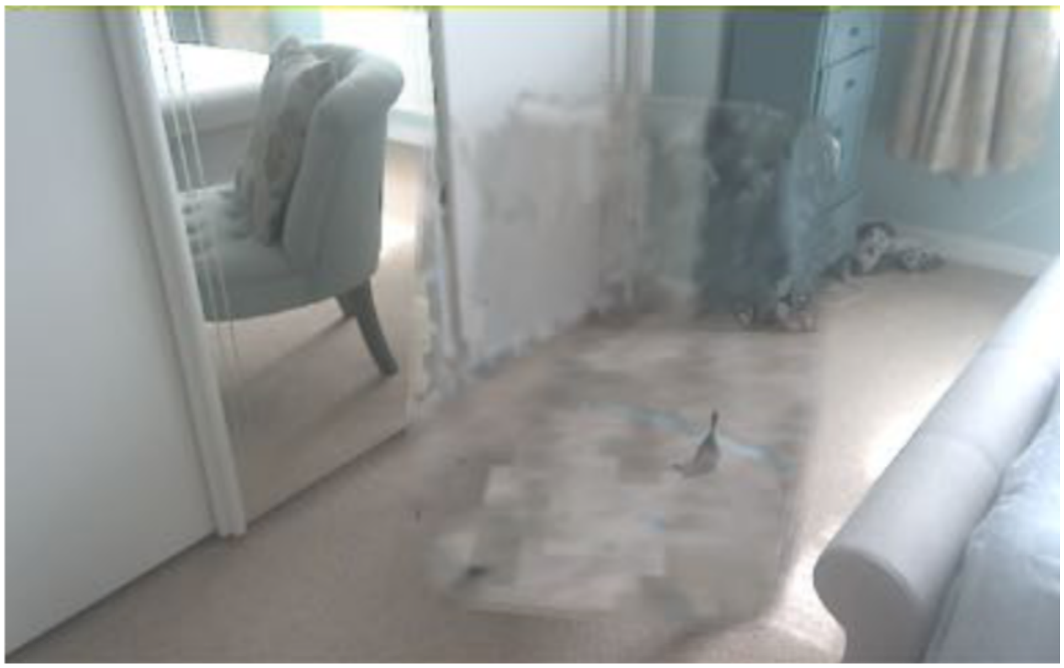


Figure 5.2: Result of removing a chair from the diminished reality method proposed by Simpson [47].

The method proposed by Nakajima et al. uses an RGB-D camera with SLAM to construct a textured point cloud of the environment. They also use a state-of-the-art segmentation algorithm and a novel object recognition approach to recognize objects within the point cloud and automatically select them without user input. Their method achieves real-time results on a system with 125 GB of RAM. Our method achieves real-time results on a system with 2 GB of RAM. Their method is also capable of diminishing the object immediately, before background geometry has been observed. However, they do not provide any means of filling missing regions in their point cloud and instead display black pixels. See Figure 5.3. Our method fills these empty regions with inpainting. Both our method and the method proposed by Nakajima et al. maintain high quality texturing of the environment, although Nakajima et al. handles borders of the diminished region better than ours.

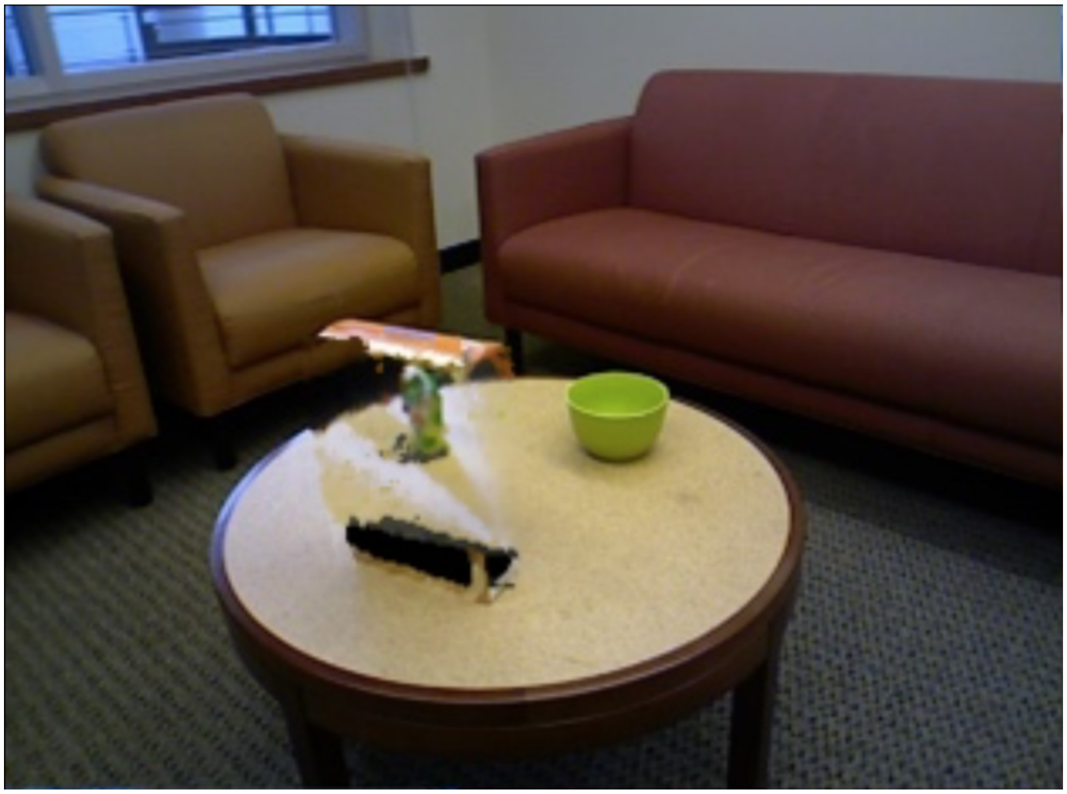


Figure 5.3: Result of removing a cereal box from the diminished reality method proposed by Nakajima et al [38].

After comparing our work qualitatively in various environments and with previous work, it is found to be competitive with previous work and maintains high-quality results.

It is worth noting that the HoloLens uses an additive display to render holographic content to the user, causing black colors to appear transparent and white colors to appear very vibrant. This prevents any diminished object from perfectly matching the color of its surroundings since it is only possible to diminish it by displaying pixels in front of it, which will appear more vibrant than the real environment. However, the results displayed in this paper are captured via the Mixed Reality Capture tool on the Windows Device Portal, which does not suffer from this effect. Any lighting discrepancies are artifacts of the texturing process and the exposure of the HoloLens camera at the time of the image capture.

Chapter 6

FUTURE WORK

Some challenges in the field of diminished reality include removing shadows and maintaining consistent lighting. Shadows are particularly difficult as both the object being removed may cast a shadow on its surrounding geometry and other geometry may cast shadows on the selected object. To properly remove shadows from surrounding geometry and propagate shadows within the removed region, knowledge of the lighting for the scene is necessary. Creating an accurate model of scene lighting is an ongoing research topic. Some consumer products have started adopting lighting estimation algorithms such as Apple's ARKit framework which provides directional lighting estimation. Beyond shadows, consistent lighting is also important for maintaining specular highlights and propagating lighting gradients across the diminished region. True diminished reality will need to accurately determine the current lighting conditions and understand how lighting affects an environment. Color bleeding is also difficult to approximate and can lead to less than realistic results when removing an object.

Although our diminished reality solution provided high-quality results, a number of items would be interesting to explore in future versions. First, our solution used an older implementation of inpainting that has been improved upon in recent years. Exploring Herling and Broll's PixMix inpainting method would provide more realistic inpainting results while maintaining real-time performance. Second, storing textures in an environment map when the depth of an environment is too far for spatial mapping to capture. When viewing content that is far away, little change is observed while moving throughout an environment. This would allow detailed textures to be used even when no spatial mapping data exists. Third, providing user control over the selection radius. In our implementation, the selection radius is a fixed size, which limits the types of objects that can be removed. Implementing a technique such as a pinch and drag gesture to modify radius size would be simple and provide users with more choice over which object to remove. Finally, diminishing moving objects rather than solely static objects. Our solution does not support removal of dynamic content, but doing so is plausible. Due to our use of a virtual representation of the real environment, displaying such an environment in front of a moving object would obscure it from view. The difficulty lies in identifying the region containing the moving object. Structure from motion techniques could be used for this.

Chapter 7

CONCLUSION

This thesis provides a diminished reality framework that operates in real-time on a state-of-the-art augmented reality headset. No external processing is necessary. To the best of our knowledge, this is the first diminished reality solution implemented on the HoloLens. Using a combination of inpainting and a textured virtual representation of an environment obtained at runtime, high-quality object removal is achieved. We further outline a six-stage pipeline which can be applied to other diminished reality solutions. Each stage represents a feature that can be upgraded or replaced to achieve an enhanced diminished result. Our work is found to be competitive with related work in the field and operates in a wide range of environments previously constrained to planar geometries.

BIBLIOGRAPHY

[1] Google Translate. Version 5.14.0, Google, Inc. Apple App Store, `https://itunes.apple.com/us/app/google-translate/id414706506?mt=8`. Accessed May 2018.

[2] DirectX. NVIDIA. `https://developer.nvidia.com/directx`. Accessed May 2018.

[3] HoloLens - Introduction to the HoloLens, Part 2: Spatial Mapping. `https://msdn.microsoft.com/en-us/magazine/mt745096.aspx`. Accessed May 2018.

[4] Image of an octree. Matter and Form. `https://matterandform.desk.com/customer/en/portal/articles/2107547-meshing---octree-degree-samples-per-node-explained`. Accessed May 2018.

[5] Image of Kinect v2. Extreme Tech. June 2014. `https://www.extremetech.com/gaming/183913-microsoft-begs-devs-to-pay-attention-to-kinect-2-for-windows-even-after-xbox-one-de-bundling`. Accessed May 2018.

[6] Image of Magic Leap One. Spiria Digital Blog. December 2017. `https://www.spiria.com/en/blog/tech-news-brief/magic-leap-one`. Accessed May 2018.

[7] Image of spatial mapping. SharpGIS. April 2016. `http://www.sharpgis.net/post/2016/04/10/Using-HoloLens-Spatial-Mapping-to-occlude-objects`. Accessed May 2018.

[8] Image of voxels. Wikipedia Commons. https://commons.wikimedia.org/wiki/File:Voxels.svg. Accessed May 2018.

[9] Intel RealSense Depth Camera D400-Series. Intel. https://software.intel.com/en-us/realsense/d400. Accessed May 2018.

[10] Magic leap. Crunchbase. https://www.crunchbase.com/organization/magic-leap#section-overview. Accessed May 2018.

[11] Meta 2. Meta Company. http://www.metavision.com. Accessed May 2018.

[12] Microsoft HoloLens. Microsoft. https://www.microsoft.com/en-us/hololens. Accessed May 2018.

[13] Oculus miffed: when vr is so immersive you fall flat on your face. The Guardian. Nov 2016. https://www.theguardian.com/technology/2016/nov/30/oculus-vr-immersive-fall-face-plant-virtual-reality.

[14] OpenCV for Unity. Enox Software. Unity Asset Store. https://assetstore.unity.com/packages/tools/integration/opencv-for-unity-21088. Accessed May 2018.

[15] OpenCV: Image Inpainting. OpenCV Documentation. https://docs.opencv.org/3.4/df/d3d/tutorial_py_inpainting.html. Accessed May 2018.

[16] Spatial mapping collider - level of detail. Unity Documentation. https://docs.unity3d.com/Manual/SpatialMappingCollider.html. Accessed May 2018.

[17] Stereo - intel realsense. Intel. Jan 2018.
`https://realsense.intel.com/stereo/`.

[18] Stories — magic leap. Magic Leap. `https://www.magicleap.com/stories`.
Accessed May 2018.

[19] Unity company facts. Unity. `https://unity3d.com/public-relations`.
Accessed May 2018.

[20] What is unreal engine? Unreal Engine.
`https://www.unrealengine.com/en-US/what-is-unreal-engine-4`.
Accessed May 2018.

[21] B. Barbee and D. Cogen. Tango vs. ARCore: Which is the future of augmented reality on Android?, Dec 2017.
`https://www.digitaltrends.com/mobile/tango-vs-arcore-theunlockr/`.

[22] M. Bertalmio, A. L. Bertozzi, and G. Sapiro. Navier-stokes, fluid dynamics, and image and video inpainting. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–I. IEEE, 2001.

[23] P. E. Debevec, C. J. Taylor, and J. Malik. Modeling and rendering architecture from photographs: A hybrid geometry-and image-based approach. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 11–20. ACM, 1996.

[24] DesignEngrLab. Miconvexhull. GitHub.
`https://github.com/DesignEngrLab/MIConvexHull`.

[25] C. Everitt. Projective texture mapping. *White paper, NVidia Corporation*, 4, 2001.

[26] C. Everitt, A. Rege, and C. Cebenoyan. Hardware shadow mapping. *White paper, nVIDIA*, 2, 2001.

[27] J. Herling and W. Broll. Advanced self-contained object removal for realizing real-time diminished reality in unconstrained environments. In *Mixed and Augmented Reality (ISMAR), 2010 9th IEEE International Symposium on*, pages 207–212. IEEE, 2010.

[28] J. Herling and W. Broll. Pixmix: A real-time approach to high-quality diminished reality. In *Mixed and Augmented Reality (ISMAR), 2012 IEEE International Symposium on*, pages 141–150. IEEE, 2012.

[29] S. Iizuka, E. Simo-Serra, and H. Ishikawa. Globally and locally consistent image completion. *ACM Transactions on Graphics (TOG)*, 36(4):107, 2017.

[30] N. Kawai, T. Sato, and N. Yokoya. Diminished reality based on image inpainting considering background geometry. *IEEE transactions on visualization and computer graphics*, 22(3):1236–1247, 2016.

[31] O. Kreylos. On the road for vr: Microsoft hololens at build 2015, san francisco. Doc-OK. May 2015. `http://doc-ok.org/?p=1223`.

[32] S. Levy. The race for ar glasses starts now. Wired. Dec 2017. `https://www.wired.com/story/future-of-augmented-reality-2018/`.

[33] Z. Li, Y. Wang, J. Guo, L.-F. Cheong, and S. Z. Zhou. Diminished reality using appearance and 3d geometry of internet photo collections. In *Mixed and Augmented Reality (ISMAR), 2013 IEEE International Symposium on*, pages 11–19. IEEE, 2013.

[34] R. Lind. Path planning for optimal mapping of unknown, urban environments. `http://web.mae.ufl.edu/~rick/rick_pro/slam`. Accessed May 2018.

[35] S. Mann. Mediated reality. *Linux Journal*, 1999(59es):5, 1999.

[36] J. Markoff. Real-life illness in a virtual world. The New York Times. July 2014. https://www.nytimes.com/2014/07/15/science/taking-real-life-sickness-out-of-virtual-reality.html.

[37] Microsoft. Microsoft/MixedRealityToolkit-Unity. GitHub. https://github.com/Microsoft/MixedRealityToolkit-Unity.

[38] Y. Nakajima, S. Mori, and H. Saito. Semantic object selection and detection for diminished reality based on slam with viewpoint class. In *Mixed and Augmented Reality (ISMAR-Adjunct), 2017 IEEE International Symposium on*, pages 338–343. IEEE, 2017.

[39] J. Odom. What's the difference between hololens, meta 2 magic leap? Next Reality. Dec 2017. https://next.reality.news/news/whats-difference-between-hololens-meta-2-magic-leap-0181804/.

[40] T. Ogawa and M. Haseyama. Image inpainting based on sparse representations with a perceptual metric. *EURASIP Journal on Advances in Signal Processing*, 2013(1):179, 2013.

[41] P. Pachal. Google translate gets smarter with language detection, word lens, Jan 2015. Mashable. https://mashable.com/2015/01/14/google-translate-word-lens/#0Qx40TE9hsqC.

[42] Quora. The difference between virtual reality, augmented reality and mixed reality. Forbes. Feb 2018. https://www.forbes.com/sites/quora/2018/02/02/the-difference-between-virtual-reality-augmented-reality-and-mixed-reality/#1c7733e62d07.

[43] F. Rameau, H. Ha, K. Joo, J. Choi, K. Park, and I. S. Kweon. A real-time augmented reality system to see-through cars. *IEEE transactions on visualization and computer graphics*, 22(11):2395–2404, 2016.

[44] D. Rubino. Microsoft hololens - here are the full processor, storage and ram specs. Windows Central. May 2016. `https://www.windowscentral.com/microsoft-hololens-processor-storage-and-ram`.

[45] Shravan. Intel realsense d435: Intels answer to kinect? The Stack Underflow Blog. Jan 2018. `https://thestackunderflowblog.wordpress.com/2018/01/29/intel-realsense-d435-intels-answer-to-kinect/`.

[46] M. SICKNESS and A. J. Benson. Medical aspects of harsh environments, volume 2, chapter 35, motion sickness. 2003.

[47] R. Simpson. Real-time mobile object removal using google project tango. Master's thesis, University of Glasgow, April 2017.

[48] Tango Moderator. Tango Tablet Development Kit Discontinuation, Feb 2017. Google Plus post. `https://plus.google.com/ProjectTango/posts/RzK7Ey5V56Y`.

[49] A. Telea. An image inpainting technique based on the fast marching method. *Journal of graphics tools*, 9(1):23–34, 2004.

[50] Teriander. Windows' acer mixed reality support? Unreal Engine Forums. Sept 2017. `https://forums.unrealengine.com/development-discussion/vr-ar-development/1360963-windows-acer-mixed-reality-support`.

[51] Tested. Projections, episode 9: Meta 2 augmented reality glasses! YouTube. April 28, 2017. `https://www.youtube.com/watch?v=_cmPFsBOquk`.

[52] Unity Technologies. Unity - Scripting API: XR.WSA.SurfaceData.trianglesPerCubicMeter. `https://docs.unity3d.com/ScriptReference/XR.WSA.SurfaceData-trianglesPerCubicMeter.html`.

[53] wguyman, M. Zeller, E. Cowley, and B. Bray. Locatable camera - mixed reality — microsoft docs. March 2018. `https://docs.microsoft.com/en-us/windows/mixed-reality/locatable-camera`.

[54] M. Wilson. Exclusive: Microsoft has stopped manufacturing the kinect. Co. Design. Oct 2017. `https://www.fastcodesign.com/90147868/exclusive-microsoft-has-stopped-manufacturing-the-kinect`.

[55] C. Yin, S. Yang, X. Yi, Z. Wang, Y. Wang, B. Zhang, and Y. Tang. Removing dynamic 3D objects from point clouds of a moving RGB-D camera. In *Information and Automation, 2015 IEEE International Conference on*, pages 1600–1606. IEEE, 2015.

[56] T. Yokoi and H. Fujiyoshi. Generating a time shrunk lecture video by event detection. In *Multimedia and Expo, 2006 IEEE International Conference on*, pages 641–644. IEEE, 2006.

[57] M. Zeller, K. Baker, and B. Bray. Spatial Mapping - Mixed Reality, March 2018. Microsoft Docs. `https://docs.microsoft.com/en-us/windows/mixed-reality/spatial-mapping`.

[58] S. Zokai, J. Esteve, Y. Genc, and N. Navab. Multiview paraperspective projection model for diminished reality. In *Mixed and Augmented Reality, 2003.*

Proceedings. The Second IEEE and ACM International Symposium on, pages 217–226. IEEE, 2003.

APPENDICES

Appendix A

PROJECTIVE TEXTURE MAPPING SHADER

```
Shader "Custom/ProjectiveTextureMapping" {
    Properties{
        _ColorTexArray("Color Texture Array", 2DArray) = "" {}
        _DepthTexArray("Depth Texture Array", 2DArray) = "" {}

        _Count("Number of snapshots to use.", Int) = 1
        _MainCamPos("Main Camera Position", Vector) = (0, 0, 0,
            1)
        _ShaderType("The type of material to display [0=default
            , 1=depth texture , 2=depth projection]", Int) = 1
    }
    SubShader{
        Tags{ "RenderType" = "Opaque" }
        LOD 200

        Stencil{
            Ref 2
            Comp equal
            Pass keep
        }

```

```
Pass{
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    #pragma target 4.0

    #include "UnityCG.cginc"

    // Can only have a maximum of 16 sampler2D textures
    // Otherwise this error occurs: "maximum ps_4_0
        sampler register index (16) exceeded at line 76 (
        on d3d11)"
    UNITY_DECLARE_TEX2DARRAY(_ColorTexArray);
    UNITY_DECLARE_TEX2DARRAY(_DepthTexArray);
    uniform float4x4 _VPArray[2];
    uniform float4x4 _DVPArray[2];
    uniform float4 _PosArray[2];

    uniform int _Count;
    uniform float4 _MainCamPos;
    uniform int _ShaderType;

    struct v2f {
        float4 pos : SV_POSITION;
        float4 vertex: VERTEX;
        float4 worldPos: VERTEX1;
    float4 normal: NORMAL;
};

v2f vert(appdata_base v) {
    v2f o;
    o.vertex = v.vertex;
    o.pos = UnityObjectToClipPos(v.vertex); //
        Equivalent to mul(UNITY_MATRIX_MVP, float4(pos
        , 1.0))
    o.worldPos = mul(UNITY_MATRIX_M, v.vertex);
    o.normal = mul(UNITY_MATRIX_M, v.normal);
    return o;
}

// Calculate how much of the texture should be
    visible given the current viewing angle
float viewingAngleVisibility(float4x4 mvp, float4
    projectorPos, float4 worldPos, float4 normal) {
    float3 projectorDir = normalize(projectorPos -
        worldPos);
    float3 viewDir = normalize(_MainCamPos - worldPos
        );
    float contribution = (dot(viewDir, projectorDir)
        + 1) / 2;
    return pow(contribution, 10);
}
```

```
float4 getProjectorColor(v2f input, int arrayIndex)
  {
  float4 color = float4(0, 0, 0, 0);

  float4x4 mvp = mul(_VPArray[arrayIndex],
    UNITY_MATRIX_M);
  float4x4 dmvp = mul(_DVPArray[arrayIndex],
    UNITY_MATRIX_M);
  float4 projectorPos = _PosArray[arrayIndex];

  // Unproject 3D point to snapshot location
  float4 proj = mul(mvp, input.vertex);
  // Prevent backprojection
  if (proj.z < 0) return color;
  // Apply perspective divide
  proj = ((proj / proj.w) + 1) / 2;
  // Clamp texture; don't allow it to repeat
  if (proj.x < 0 || proj.x > 1 || proj.y < 0 ||
    proj.y > 1) return color;

  float distToEdge = max(abs(proj.x - 0.5), abs(
    proj.y - 0.5)) * 2;

  // Find contribution for each snapshot based on
    its viewing angle
  float contribution = _Count >= arrayIndex ?
    viewingAngleVisibility(mvp, projectorPos,
    input.worldPos, input.normal) : 0;

  // Get depth value of pixel at unprojected point
  float4 depthProj = mul(dmvp, input.vertex);
  depthProj = ((depthProj / depthProj.w) + 1) / 2;
  float4 depth = UNITY_SAMPLE_TEX2DARRAY(
    _DepthTexArray, float3(depthProj.xy / 2,
    arrayIndex));
  // Prevent shadow artifacts (shadow acne) by
    setting a bias/offset
  float bias = 0.03f;
  // Only apply texture if pixel is not hidden in
    snapshot
  // Invert proj.z since depth is from 1-0 instead
    of 0-1
  if (abs((1 - depthProj.z) - depth.r) < bias) {
    color = UNITY_SAMPLE_TEX2DARRAY(_ColorTexArray
        , float3(proj.xy / 2, arrayIndex));
  } else {
    contribution = 0;
  }

  // Store contribution in alpha component
  color.a = contribution * (1 - distToEdge);

  return color;
}
```

```
fixed4 frag(v2f i) : SV_Target {
    float4 c = float4(0, 0, 0, 0);

    float projectorColors[2];

    float totalContribution = 0;
    for (int index = 0; index < _Count; index++) {
        float4 projectorColor = getProjectorColor(i,
            index);
        projectorColors[index] = projectorColor;
        totalContribution += projectorColor.a;
        c += float4(projectorColor.rgb *
            projectorColor.a, projectorColor.a);
    }

    // If pixel is not completely colored by a
        texture, color the rest of it white
    float checkerboard = ((int)((i.worldPos.x + 100.0
        f) / 0.1) + (int)((i.worldPos.y + 100.0f) /
        0.1) + (int)((i.worldPos.z + 100.0f) / 0.1)) %
         2;
    float4 defaultColor = float4(checkerboard * i.
        normal.r, checkerboard * i.normal.g,
        checkerboard * i.normal.b, 1);

    // Normalize the contribution from all projectors
```

```
            if (totalContribution > 0) {
                c /= totalContribution;
            } else {
                c = defaultColor;
            }

            // Reset alpha component
            c.a = 1;

            return c;
        }

        ENDCG
    }
  }
  FallBack "Diffuse"
}
```

Listing A.1: Determine pixel color from projectors.

Appendix B

SPARSE VOXEL OCTREE TEXTURE MAPPING SHADERS

```
    // Each #kernel tells which function to compile; you can
    have many kernels
#pragma kernel CSMain

struct SVOPixelData {
    float3 pos;
    uint3 color;
    int nodeIndex;
};

RWStructuredBuffer<SVOPixelData> pixelData;

Texture2DArray<float4> _ColorTexArray;
Texture2DArray<float4> _DepthTexArray;
float4x4 _VPArray[2];
float4x4 _DVPArray[2];
float4x4 _InverseVPArray[2];
float4x4 _InverseDVPArray[2];

// Convert pixel and depth data into structs for easier use
    by SVO.
// To calculate the 3D position of a pixel, we have to
    unproject a pixel from the color
```

```
// texture to the near plane, then project it into the depth
   map to acquire the depth value,
// and then unproject again from the color texture using the
   correct depth value.
// The same point may be located at a different pixel in the
   depth and color textures since
// both textures were created using different view and
   projection matrices.
[numthreads(8,8,1)]
void CSMain (uint3 id : SV_DispatchThreadID) {
   int dataIndex = (id.z * 1280 * 720) + (id.y * 1280) + id.x
      ;
   pixelData[dataIndex].color.x = (uint)(_ColorTexArray[id].x
      * 255.0);
   pixelData[dataIndex].color.y = (uint)(_ColorTexArray[id].y
      * 255.0);
   pixelData[dataIndex].color.z = (uint)(_ColorTexArray[id].z
      * 255.0);
   //pixelData[dataIndex].color = _ColorTexArray[id] * 255.0f
      ;
   pixelData[dataIndex].nodeIndex = 0;

   // Calculate position of color pixel on near plane
   int tex = id.z;
   int row = id.y;
   int col = id.x;
```

```
    // Use depth value of 0 to unproject point onto the near
       plane
    float4 ndc = float4((col / (float)1280) * 2 - 1, (row / (
       float)720) * 2 - 1, 0, 1);
    float4 worldPos = mul(_InverseVPArray[tex], ndc);
    worldPos /= worldPos.w;

    // Project into depth map to get depth value
    worldPos.w = 1;
    float4 ndc_depth = mul(_DVPArray[tex], worldPos);
    ndc_depth /= ndc_depth.w;
    uint3 depthPixelPos = uint3(ndc_depth.x * 1280, ndc_depth.
       y * 720, tex);
    float depthValue = _DepthTexArray[depthPixelPos].r;

    // Calculate the real position using depth value
    ndc.z = 1-depthValue;
    float4 realWorldPos = mul(_InverseVPArray[tex], ndc);
    realWorldPos /= realWorldPos.w;
    pixelData[dataIndex].pos = realWorldPos.xyz;
}
```

Listing B.1: Convert pixels to structs.

```
    // Each #kernel tells which function to compile; you can
    have many kernels
#pragma kernel CSMain

```

```
struct SVOPixelData {
    float3 pos;
    uint3 color;
    int nodeIndex;
};

struct SVONode {
    int children[8];
    float3 pos;
    uint3 color;
    uint pixelCount;
};

RWStructuredBuffer<SVOPixelData> pixelData;
RWStructuredBuffer<SVONode> octree;
int octreeSize;

[numthreads(64,1,1)]
void CSMain (uint3 id : SV_DispatchThreadID) {
    // Traverse octree
    float3 pixelPos = pixelData[id.x].pos;
    int oldNodeIndex = pixelData[id.x].nodeIndex;
    SVONode oldNode = octree[oldNodeIndex];
    int childIndex = (pixelPos.z >= oldNode.pos.z) * 4 + (
        pixelPos.y >= oldNode.pos.y) * 2 + (pixelPos.x >=
        oldNode.pos.x);

    // Continue traversing until at lowest level of octree
    while (oldNode.children[childIndex] > 0) {
        oldNodeIndex = oldNode.children[childIndex];
        oldNode = octree[oldNodeIndex];
        childIndex = (pixelPos.z >= oldNode.pos.z) * 4 + (
            pixelPos.y >= oldNode.pos.y) * 2 + (pixelPos.x >=
            oldNode.pos.x);
    }

    // Flag the child node
    octree[oldNodeIndex].children[childIndex] = -1;

    // Update the pixelData's nodeIndex
    pixelData[id.x].nodeIndex = oldNodeIndex;

    // Increment pixel count
    InterlockedAdd(octree[oldNodeIndex].pixelCount, 1);

    // Add color to node
    InterlockedAdd(octree[oldNodeIndex].color.r, pixelData[id.
        x].color.r);
    InterlockedAdd(octree[oldNodeIndex].color.g, pixelData[id.
        x].color.g);
    InterlockedAdd(octree[oldNodeIndex].color.b, pixelData[id.
        x].color.b);
}
```

Listing B.2: Flag nodes in the current LOD of the octree containing pixels.

```
    // Each #kernel tells which function to compile; you can
    have many kernels
#pragma kernel CSMain

struct SVONode {
    int children[8];
    float3 pos;
    uint3 color;
    uint pixelCount;
};

RWStructuredBuffer<SVONode> octree;
RWStructuredBuffer<uint> octreeMetadata;
// [0] == lastLODIndex
// [1] == lastNodeIndex
// [2] == currentLOD
int octreeSize;
float currentVoxelSize;

[numthreads(64,1,1)]
void CSMain (uint3 id : SV_DispatchThreadID) {
    int lastLODIndex = octreeMetadata[0];
    uint nodeOffset = id.x / 8;
    uint childIndex = id.x % 8;
    int nodeIndex = lastLODIndex + nodeOffset;
    // Do nothing if node is not flagged
    if (octree[nodeIndex].children[childIndex] == 0) return;

    // Get last node in octree and increment global counter
    int lastNodeIndex;
    InterlockedAdd(octreeMetadata[1], 1, lastNodeIndex);

    // Set child node to the new node index
    int newNodeIndex = lastNodeIndex + 1;
    octree[nodeIndex].children[childIndex] = newNodeIndex;

    // Update the new node's position
    float newVoxelRadius = currentVoxelSize / 4.0;
    octree[newNodeIndex].pos.z = octree[nodeIndex].pos.z +
        newVoxelRadius * ((int)(childIndex / 4) * 2 - 1);
    octree[newNodeIndex].pos.y = octree[nodeIndex].pos.y +
        newVoxelRadius * ((int)((childIndex / 2) % 2) * 2 - 1);
    octree[newNodeIndex].pos.x = octree[nodeIndex].pos.x +
        newVoxelRadius * ((int)(childIndex % 2) * 2 - 1);
}
```

Listing B.3: Build the next LOD of the octree.

```
    // Each #kernel tells which function to compile; you can
    have many kernels
#pragma kernel CSMain

struct SVONode {
    int children[8];
    float3 pos;
    uint3 color;
    uint pixelCount;
};

RWStructuredBuffer<SVONode> octree;
int octreeSize;

// Average the color of each node.
[numthreads(64,1,1)]
void CSMain (uint3 id : SV_DispatchThreadID) {
    uint pixelCount = octree[id.x].pixelCount;
    // Prevent indexing out of bounds
    if ((int)id.x >= octreeSize) return;
    octree[id.x].color = octree[id.x].color / pixelCount;
}
```

Listing B.4: Average the color value for each node.