
PROJECTING PHYSICAL OBJECTS INTO A VIRTUAL SPACE USING THE KINECT AND OCULUS RIFT

A Project

By Shaun P. Bond

Presented to the Faculty of the University of Alaska Fairbanks

In Partial Fulfillment of the Requirements of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Fairbanks, Alaska

April 2015

PROJECTING PHYSICAL OBJECTS INTO A VIRTUAL SPACE USING THE KINECT AND OCULUS RIFT

A Project

By Shaun P. Bond


RECOMMENDED:

 2015-04-21

Dr. Orion Lawlor, Advisory Committee Chair Date


 2015-04-21

Dr. Glenn Chappell, Advisory Committee Member Date

 4/21/15

Dr. Jon Genetti, Advisory Committee Member Date

APPROVED:

 4/21/15

Dr. Jon Genetti, Dept. Head, Computer Science Date

ABSTRACT

Virtualized Reality as a field of research has been increasing over the last couple of decades. Initially, it required large camera arrays, expensive equipment, and custom software to implement a virtualized reality system. With the release of the Kinect and the Oculus Rift development kits, however, the average person now has the potential to acquire the hardware and software needed to implement a virtualized reality system. This project explores the possibility of using the Kinect and Oculus Rift together to display geometry based on real-world objects in a virtual environment.

ACKNOWLEDGEMENTS

Special thanks to Dr. Orion Lawlor, Dr. Glenn Chappell, and Dr. John Genetti for their guidance, technical support, and assistance during my completion of the Master's program and this project report. I would also like to thank Connie Huizenga for always being helpful, the Department of Computer Science at the University of Alaska Fairbanks for providing me with this opportunity, all of my professors for being awesome professors and imparting their knowledge to me, my mother, Anne Holladay, for her financial assistance and support, and Ayse Yeager for her personal support.

TABLE OF CONTENTS

Abstract 3

Acknowledgements 4

Table of Figures..... 6

Introduction..... 7

Hardware 9

Converting Kinect Depth to 3D Geometry..... 12

Rendering With the Oculus Rift..... 15

Integrating Oculus Rift and Kinect 17

Performance 21

Future Research 22

References 23

Appendix A: Source code..... 25

Appendix B: GLSL vertex shader 38

Appendix C: GLSL Geometry shader 39

Appendix D: GLSL Fragment Shader 40

TABLE OF FIGURES

Figure 1: VEVI Interface (Hine, 1994).....	7
Figure 2: The Dome at Carnegie Mellon University (Copyright Carnegie Mellon, 1995).....	8
Figure 3: 3rd person perspective point cloud (Property of Johnathan Ota).....	8
Figure 4: Kinect for Xbox 360 (Photograph: Michal Czerwonka/Getty Images)	9
Figure 5: IR dot pattern emitted by the Kinect	10
Figure 6: Distortion caused by lenses in Oculus Rift.....	11
Figure 7: Chromatic aberration.....	11
Figure 8: Near-infrared LEDs in Oculus Rift (Terndrup, 2014)	11
Figure 9: Output of unmodified <code>cpp_view</code>	12
Figure 10: Stephane Magnenat's depth to distance conversion (OpenKinect, 2013).....	13
Figure 11: Dr. Lawlor's code for 2D to 3D projection.....	13
Figure 12: Modified <code>cpp_view</code> showing generated point cloud.....	14
Figure 13: Algorithm for generating triangle strip.....	14
Figure 14: Modified <code>cpp_view</code> showing triangle strip with discarded triangles.....	15
Figure 15: Simple cube rendered in stereo to standard display	16
Figure 16: Real-world geometry rendered on the Oculus Rift	17
Figure 17: RGB image applied to geometry without correction	18
Figure 18: FOV and distance between cameras accounted for when applying texture	19
Figure 19: Scene rendered with geometry averaged over 2 Kinect frames.....	20
Figure 20: Scene rendered with virtual lighting removed from real-world geometry.....	20
Figure 21: Performance of Rendering and Kinect image processing.....	21

INTRODUCTION

Virtualized reality can be viewed as the complement to augmented reality. In augmented reality, computer generated visuals are integrated with real-world visuals. In virtualized reality, a computer generated representation of real-world objects is viewed in a virtual reality environment.

Virtualized reality has applications in many fields. It has been applied in research involving remotely controlled robotics (Hine, 1994; Goza, 2004), entertainment (Carnegie Mellon, 2001), and physical and mental rehabilitation (Thin, 2012). It also has potential for other applications such as microscopic visualization, training, engineering, architecture, and others.

Virtualized reality has been a field of research since as early as 1994, when Hine et al. developed a system to aid in the remote navigation of an underwater vehicle. This system used sensory data from the vehicle's stereo cameras, and positional data to reconstruct the underwater environment. It also supported two display modes, one of which was a VR headset. Once the environmental data was collected, the operator could navigate from within the virtual environment relatively free from communication latency (Hine, 1994).

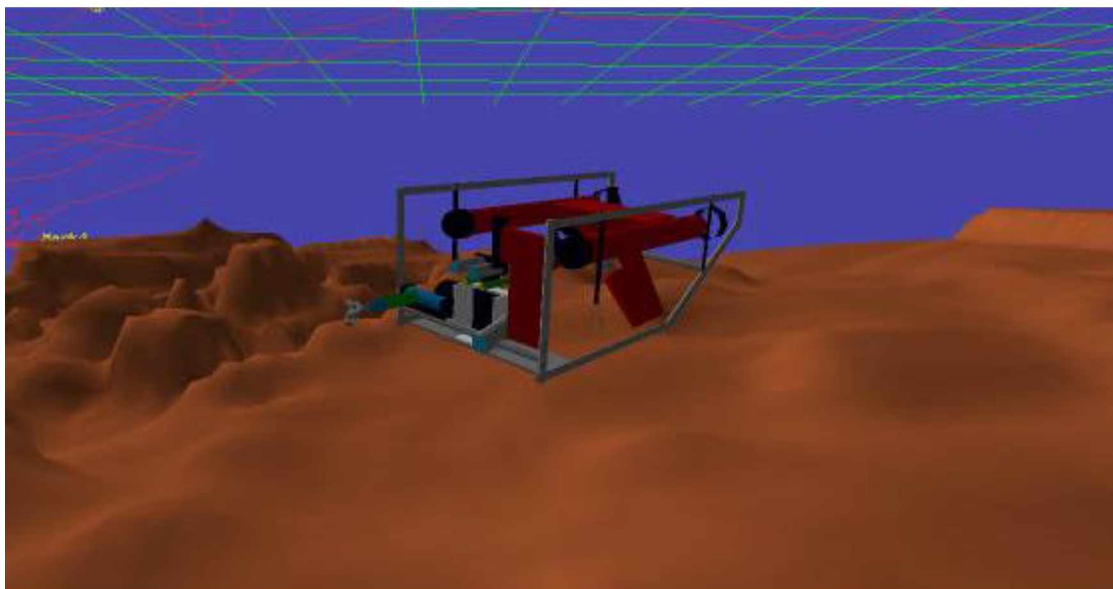


FIGURE 1: VEVI INTERFACE (HINE, 1994)

In 1995, Kanade et al. developed a virtualized reality system utilizing a large camera array (Kanade, 1995). This camera array recorded video from various perspectives, which was later used to construct a 3-dimensional video. This allowed the viewer to select an arbitrary viewpoint during playback (Kanade, 1997). Later, in 2001, a version of this system was used to record a Super Bowl event (Carnegie Mellon, 2001).

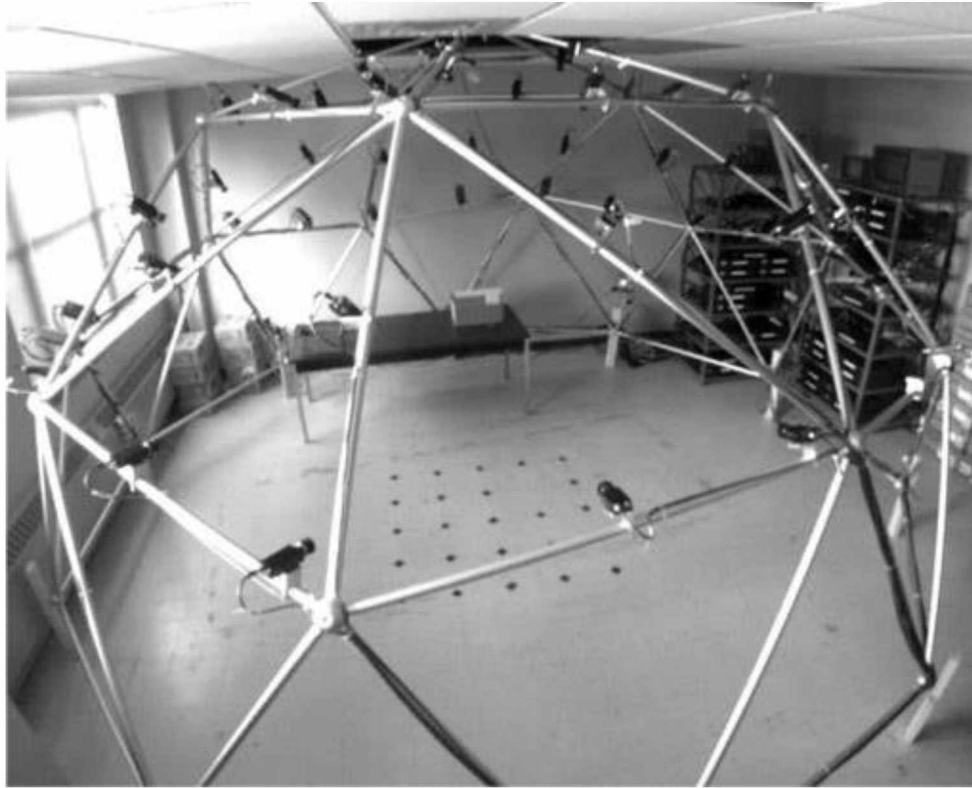


FIGURE 2: THE DOME AT CARNEGIE MELLON UNIVERSITY (COPYRIGHT CARNEGIE MELLON, 1995)

More recently, some students at Carnegie Mellon University created a virtualized reality system that allows a person to explore the world from a third-person perspective (Ota, 2012). This system utilizes the Kinect's depth sensor to gather environmental information, and the Oculus Rift head mounted display to view it. The Kinect is a motion-sensing input device developed by Microsoft, and the Oculus Rift is a virtual reality display device. The Kinect is mounted at a fixed position relative to the user, giving the third-person perspective to the viewer.



FIGURE 3: 3RD PERSON PERSPECTIVE POINT CLOUD (PROPERTY OF JOHNATHAN OTA)

I wanted to explore whether real-time virtualized reality was feasible on commodity hardware. My goals for this project were to use the Microsoft Kinect to generate a 3D triangle mesh of real-world objects, place the mesh in a virtual environment, and display it to the Oculus Rift in real-time. Some projects I have found produce a triangle mesh in real-time, but do not display it in stereoscopic VR (Izadi, 2011). Others (Ota, 2012), display data from the Kinect in stereoscopic VR, but use a simple point-based rendering approach instead of a continuous triangle mesh. My project differs from these in that it does both.

HARDWARE

The Kinect is a device developed by the Microsoft Corporation to allow motion input for video games, but researchers have found many other uses for its depth detection and motion tracking capabilities. Among these are 3D reconstruction of objects (Izadi, 2011), translation of sign language (Chen, 2013), and hands free control of operating room computers (Tan, 2012). The original Kinect was designed for use with the Xbox 360, but there are newer versions available for Xbox One and personal computers. I used the original Xbox 360 version in my project, so the hardware and software specifications in this paper refer to that version.

The Kinect device consists of an infrared (IR) emitter, IR receiver, and color (RGB) camera. The IR emitter and receiver are located approximately 7.5 cm apart, and the IR emitter projects a predefined dot pattern (Freedman, 2012). This dot pattern bounces off of IR reflective objects and is acquired by the IR receiver. The Kinect analyzes the parallax shift and distortion of the pattern to build a two dimensional matrix of depth values, hereafter referred to as the depth image. The dot pattern can be seen in figure 5. The Kinect also has a motor in its base that allows it to adjust its vertical tilt.

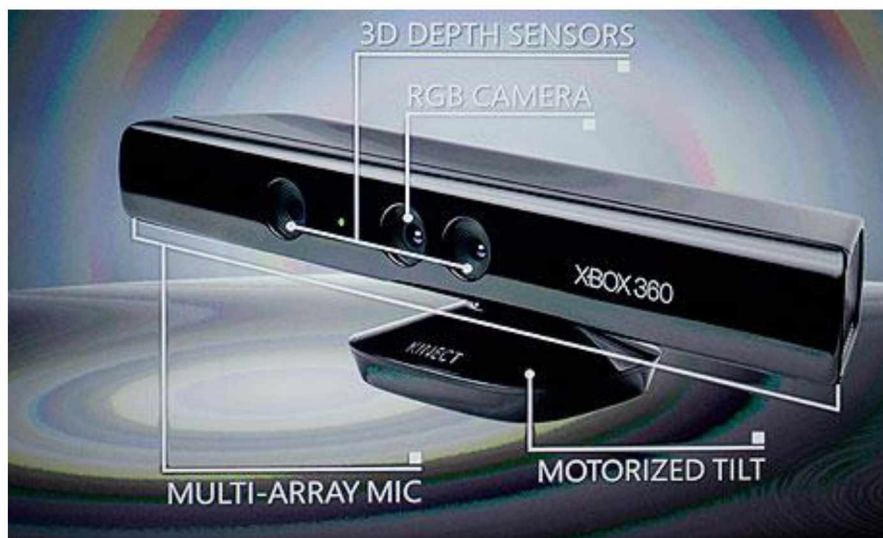


FIGURE 4: KINECT FOR XBOX 360 (PHOTOGRAPH: MICHAL CZERWONKA/GETTY IMAGES)



FIGURE 5: IR DOT PATTERN EMITTED BY THE KINECT

The Oculus Rift is a virtual reality headset in development by Oculus VR. As of this paper, they have released two versions. These are referred to as Development Kit 1 (DK1), and Development Kit 2 (DK2). The DK2 possesses a higher resolution and reduced latency over the DK1, as well as the addition of positional tracking. The DK2 is the version I used for this project, so the specifications here refer to that version.

The DK2 has a single display and two eyepieces. The display has a 1920 x 1080 pixel resolution, and 2 milliseconds of persistence. The low persistence provides better immersion and reduced motion sickness, compared with earlier VR headsets, by reducing motion blur. The optical lenses in the eyepieces have a 100-degree field of view, which allows for greater peripheral vision. However, the large field of view causes distortion of the image, which needs to be corrected. Another problem caused by the lenses is chromatic aberration, which must also be accounted for when rendering. These effects can be seen in figures 6 and 7. In order to achieve position and orientation tracking, the DK2 possesses an internal accelerometer, magnetometer, and gyroscope, as well as an external position tracking camera. The internal sensors update 1000 times per second, while the position tracking camera updates 60 times per second. All of the sensors communicate with the computer over a USB 2.0 connection, while the display is transmitted over an HDMI connection.

Stereoscopic vision is achieved by drawing a scene from two different perspectives, that of the left eye and that of the right eye. Each of these images is drawn on one half of the DK2 display, and the eyepieces are focused on their respective halves. Chromatic aberration and lens distortion are handled in software prior to displaying the scene. The image is predistorted and chromatic aberration is applied in reverse. When the image passes through the lens, the effects are undone. This avoids the necessity for expensive hardware solutions.

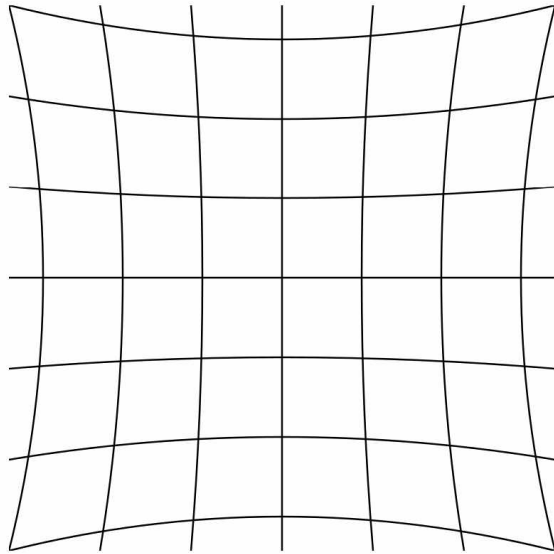


FIGURE 6: DISTORTION CAUSED BY LENSES IN OCULUS RIFT



FIGURE 7: CHROMATIC ABERRATION

The DK2 handles position tracking by using near-infrared LEDs placed on the headset. Each of these LEDs emits at a different frequency and intensity (Terndrup, 2014). This allows the position tracking software to identify which LED a signal came from, and calculate the position of the headset. The distance of the headset from the camera can be determined by the relative spacing of the LEDs. The vertical and horizontal position can be found in a straightforward manner using the 2D position of an LED and its distance from the camera.

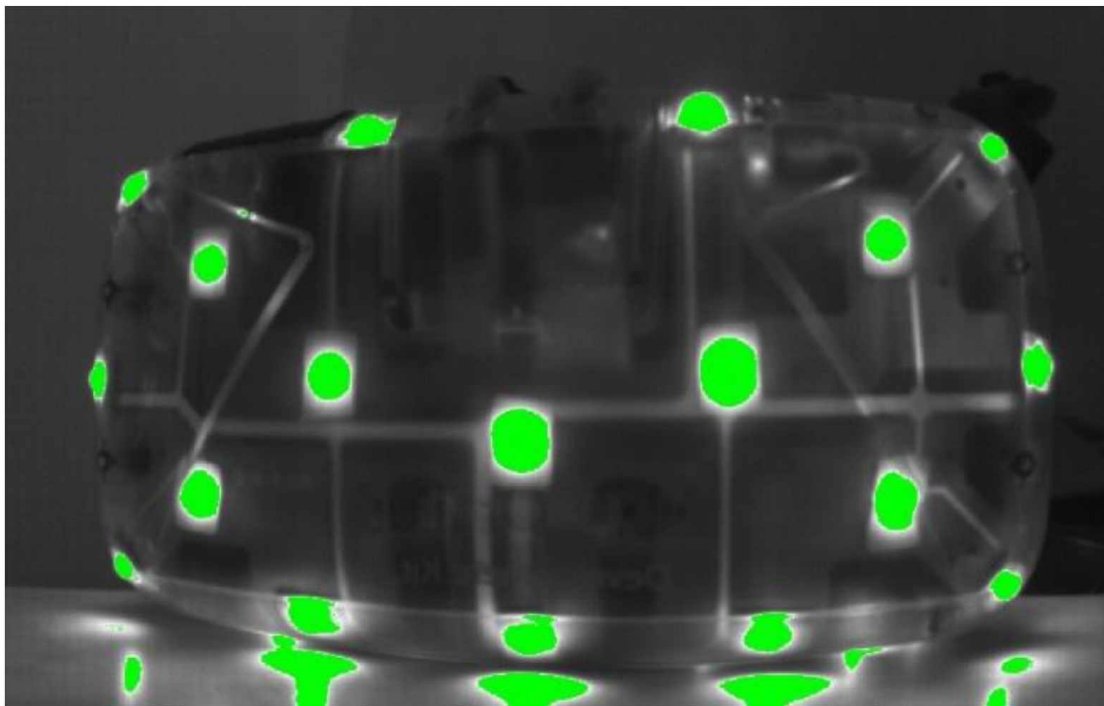


FIGURE 8: NEAR-INFRARED LEDs IN OCULUS RIFT (TERNDRUP, 2014)

CONVERTING KINECT DEPTH TO 3D GEOMETRY

My first step for this project was to produce a 3D geometry from a Kinect depth image. To interface with the Kinect, I used an open source library called Libfreenect. Libfreenect defines the FreenectDevice base class, which is responsible for communicating with the Kinect device. A user then derives a class from FreenectDevice and overloads functions to receive video frames and perform any desired computation. FreenectDevice also contains functionality enabling the user to control the Kinect motor and LEDs.

The Kinect supports RGB, YUV, Bayer, and infrared (IR) image formats for its color video stream. RGB and Bayer can be returned in 640x480 or 1280x960 pixel resolutions. YUV and IR formats can only be returned in 640x480 pixel resolution. The user can request any of these formats and resolutions with Libfreenect. I used the 640x480 RGB resolution, which allows a maximum of 30 frames per second.

Because the focus of this project was doing the actual transformation, rather than building the infrastructure, I opted to start with an existing program that would handle all of the setup and receive images from the Kinect. The program I selected for this purpose is called `cpp_view`, a demo program that is part of the Libfreenect project. It receives frames from both the RGB and depth cameras, does some basic colorization of the depth image, and displays both images to the screen. Figure 9 shows the output of the original program. It also supports switching between the various image formats, adjusting the tilt of the Kinect, and operating the LEDs on the Kinect.

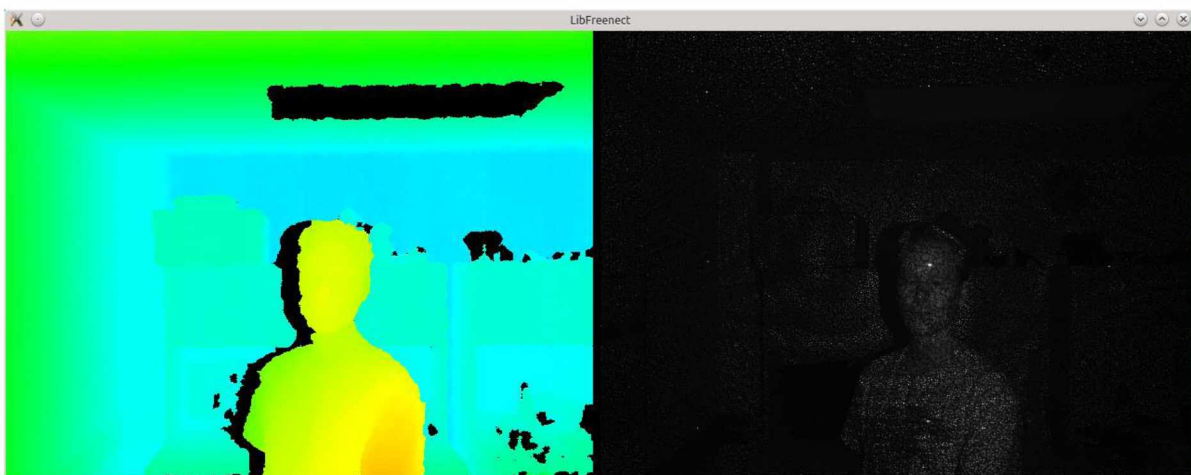


FIGURE 9: OUTPUT OF UNMODIFIED `CPP_VIEW`. LEFT: DEPTH STREAM. RIGHT: RGB STREAM SHOWING IR.

Initially, I would get terminal errors intermittently when starting the program. I would also receive a terminal error consistently when exiting the program. After reviewing the code, I noticed that video streams from the Kinect were being initialized during the startup sequence, but were never stopped. Stopping the video streams in the destructor of the class derived from FreenectDevice solved both problems. It is my belief that the Kinect was attempting to continue writing to a buffer after it had been destroyed.

In order to familiarize myself with the program and how it handled images, I modified the code so that it would display a gray-scale coloring of the depth image instead of a color one. I then added code to the same routine that calculates the 3D position of each pixel in the depth image and stores it in a vertex array. To assist with this, I used some code written previously by Dr. Lawlor to project a 2D point into 3D space (see figure 11). The Kinect does not record depth in any standard unit. Fortunately, Stephane Magnenat (OpenKinect, 2013) developed an algorithm that converts the depth value to meters with less than 3% error, as measured during my previous work with the CyberAlaska project.

```
distance = 0.1236 * tan(rawDisparity / 2842.5 + 1.1863)
```

FIGURE 10: STEPHANE MAGNENAT'S DEPTH TO DISTANCE CONVERSION (OPENKINECT, 2013)

```
vec3 loc(int x,int y) const {
    return dir(x,y)*depth(x,y);
}
vec3 dir(int x,int y) const {
    return vec3((x-w*0.5)*pixelFOV, (h*0.5-y)*pixelFOV, 1);
}
float depth(int x,int y) const {
    uint16_t disp=depthi[y*w+x];
    if (disp>= INVALID_DEPTH) return 0.0;
    return 0.1236 * tan(disp / 2842.5 + 1.1863) - 0.037;
}
```

FIGURE 11: DR. LAWLOR'S CODE FOR 2D TO 3D PROJECTION

I needed to verify that my code was really producing a 3D point cloud with reasonable values. For the purposes of this project, it is more important that the results are visually convincing than it is that they are accurate, so I decided that the easiest way to verify was to display the point cloud along with the colorized depth image, as seen in figure 12. In order to do this, I replaced the RGB image with the point cloud, but this presented some difficulties. The Kinect uses a downward pointing y-axis, while OpenGL uses an upward pointing y-axis. The original code corrects this by flipping the projection coordinates, but Dr. Lawlor's code, mentioned previously, corrects the y-axis for the point cloud when the point coordinates are calculated. This resulted in the point cloud being displayed upside down. I corrected this issue by using two different viewports with two different projections. This also allowed me to display the point cloud in perspective, yielding a more convincing image, while not subjecting the depth image to perspective distortion.

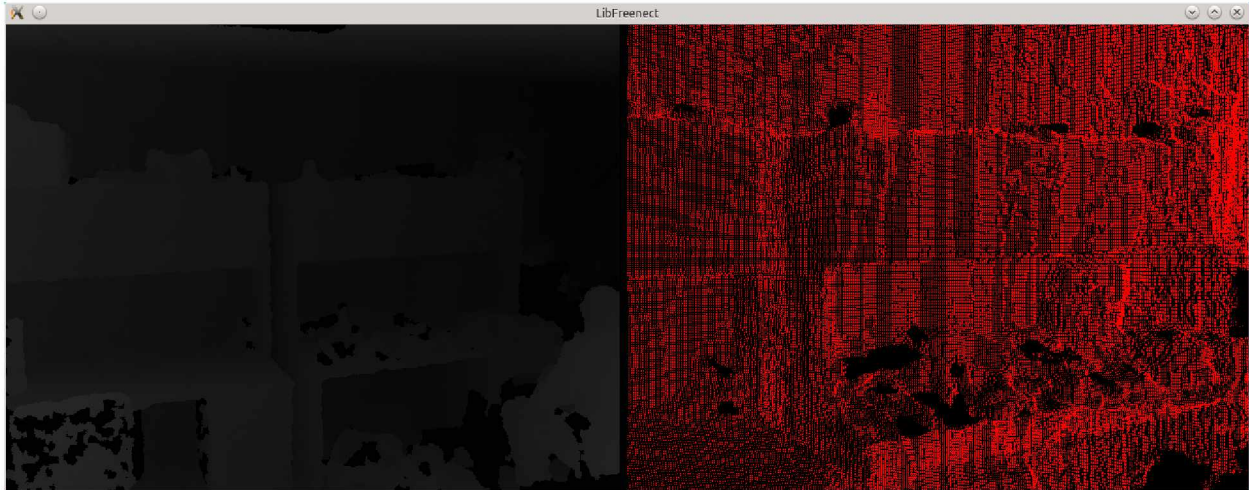


FIGURE 12: MODIFIED CPP_VIEW SHOWING GENERATED POINT CLOUD

Initially, I connected all of the points into a single triangle strip using a simple algorithm illustrated in figure 13. Unfortunately, the Kinect does not always know the depth at a given pixel. In those cases, it will return the value 2047, marking it as too far, too close, or otherwise unknown. Because I assumed that every pixel was a point so that I could implement this simplified triangle strip algorithm, my code assigned coordinates to invalid pixels as well. By default, Dr. Lawlor's code gave the coordinates (0, 0, 0) to pixels with an invalid depth. The consequence of this was that the triangle strip contained triangles connected to the origin, producing an image that was obviously not correct. Rather than attempt to implement an algorithm that did not include these invalid points in the triangle strip, and could have potentially been far more complex, this problem can be solved efficiently using a geometry shader. The coordinates (0, 0, 0) are the location of the Kinect device itself. Therefore, no point seen by the Kinect can have those coordinates. This makes it a reasonable way to represent invalid points. Using this, I was able to write a geometry shader that checks each triangle for a point containing those coordinates. If the shader finds such a point, it discards the whole triangle. This produces an object that is much more visually convincing, leaving holes where the Kinect is unable to obtain information. Figure 14 shows the resulting geometry. This also has the advantage that it runs on the graphics card, so it does not use any more of the CPU's resources, and it is able to exploit the GPU's massive parallelization.

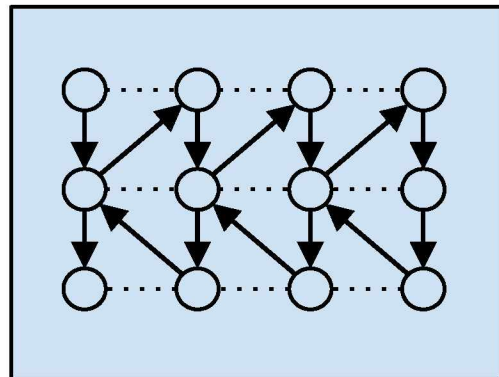


FIGURE 13: ALGORITHM FOR GENERATING TRIANGLE STRIP. CIRCLES REPRESENT VERTICES, ARROWS REPRESENT THE ORDER IN WHICH THEY ARE ADDED, AND DOTTED LINES REPRESENT THE IMPLICIT COMPLETION OF A TRIANGLE.

One other drawback of this algorithm is that it produces a triangle strip that is not well suited to spherical surfaces and some other shapes. However, this did not seem to make a noticeable impact on the appearance of the geometry, probably because of the extremely small size of the individual triangles. Therefore, I did not feel it was worth my time to create a more complicated algorithm.

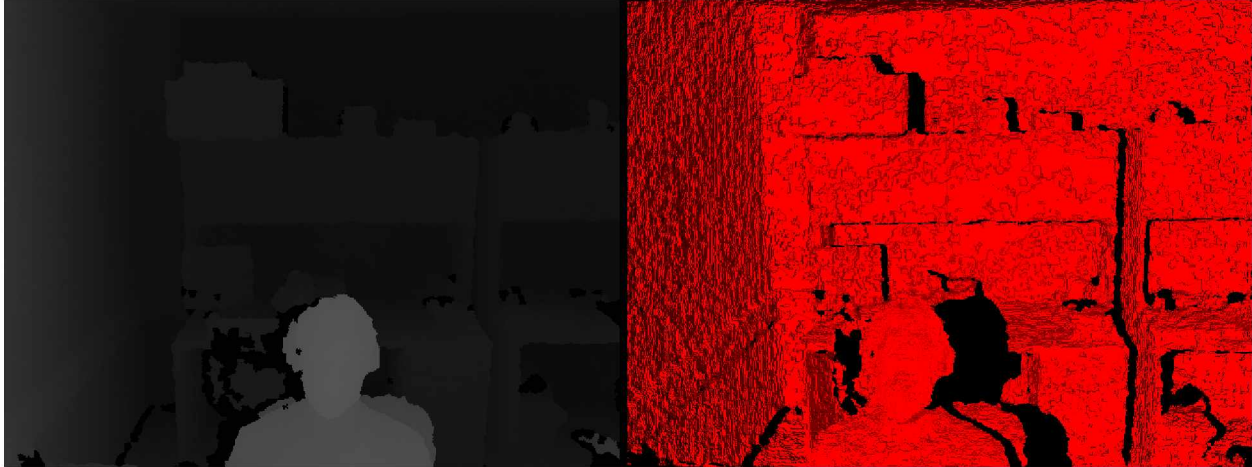


FIGURE 14: MODIFIED CPP_VIEW SHOWING TRIANGLE STRIP WITH DISCARDED TRIANGLES

RENDERING WITH THE OCULUS RIFT

The second phase of my project involved using the Oculus Rift to display a simple stereoscopic cube. I chose a simple cube for this phase to eliminate any complexities that might have been introduced by the code that handles the Kinect image processing and generated geometry. The API I used to interface with the Oculus Rift is the Oculus SDK, which includes LibOVR, a configuration utility, and demo application. Oculus SDK and LibOVR are developed by Oculus VR and have experimental Linux support at the time of this writing.

LibOVR is a state-driven API. There are several general steps involved in setting up the Oculus Rift. LibOVR must be initialized before anything else can be done. The library handles this itself when the initialization function is called. A head mounted display object must then be created. This object contains information about the headset and its capabilities, and it allows the software to interact with the head mounted display. The next step is to configure the head tracking and rendering settings. The application is responsible for creating a render texture using either Direct3D or OpenGL, and for passing a handle to LibOVR along with the graphics API used, texture size, and viewport. LibOVR supports the use of a single texture for both eyes, or a separate texture for each eye. In the prior case, one would define two viewports, and in the latter case, one would define two textures. LibOVR must be called before and after rendering the scene, and the application must render the scene once for each eye. The library contains various functions to assist with this.

As in the previous stage, I began with existing code in order to reduce development time. This time, I used a program I had written previously for a simple OpenGL app. I modified this code to render a basic cube in the standard way. Once I had that working, I included some basic LibOVR functionality. At this point I only set up the head mounted display so that I could

access sensor data. I did not initialize the rendering configuration because I was not ready to render to the Oculus Rift yet. I then used the head mounted display's reported interpupillary distance to shift the camera and render two images in stereo. I displayed both of these images side-by-side to the standard display so that I could visually verify that stereo rendering was working before introducing possible complexities with the head mounted display.



FIGURE 15: SIMPLE CUBE RENDERED IN STEREO TO STANDARD DISPLAY

The next step was to display the cube on the headset. For this step, I needed to initialize the LibOVR rendering configuration. Actually getting LibOVR to do the rendering was as simple as replacing my call to swap the display buffers with the library's equivalent. Because I set it up in the rendering configuration, LibOVR handled distortion correction and chromatic aberration correction for me. The final step in this stage was to incorporate head tracking. This involved querying the head mounted display for its position and orientation, and constructing view and projection matrices using that information.

Problems in this stage largely arose from a lack of documentation and simple demo programs. Oculus VR had written a user guide for the SDK, but it was not a very detailed guide. It described the basic steps involved in setting up and using the library, but described each step in isolation, with references to functions whose behavior was not described. Furthermore, there was no code reference. I spent a lot of time reading through the implementation, seeing how others used the library in examples, and experimenting. It was also difficult to apply the code in the examples to my project because all of the examples written for Linux were rather complex.

INTEGRATING OCULUS RIFT AND KINECT

My goal for the third and final stage of this project was to display the geometry produced by the Kinect in stereoscopic 3D in real-time. However, I had some extra time at the end of the project to explore a couple of related problems. I will discuss those here as well.

Integrating the image processing code with the rendering code went smoothly. The image processing code was already written as an independent component that functioned in its own thread. The only change I needed to make after combining the code was to get the list of vertices from the Freenect device and tell OpenGL to display the vertices as a triangle strip instead of the cube. While the geometry displayed correctly, it was not oriented or positioned correctly. I fixed these problems by manually correcting the coordinate system. The result can be seen in figure 16. This would not be a suitable solution for released software, however, because simply moving the Kinect physically would require code changes to correct the geometry. Initially, the position tracking was not working, either. This was a hardware issue rather than a software one, though. The position tracking camera and the Kinect happened to be on the same USB bus. Because both devices send uncompressed video, the USB bus did not have enough bandwidth to support them, and the operating system dropped the position tracking camera. Ensuring that the two devices were on separate buses resolved the problem.

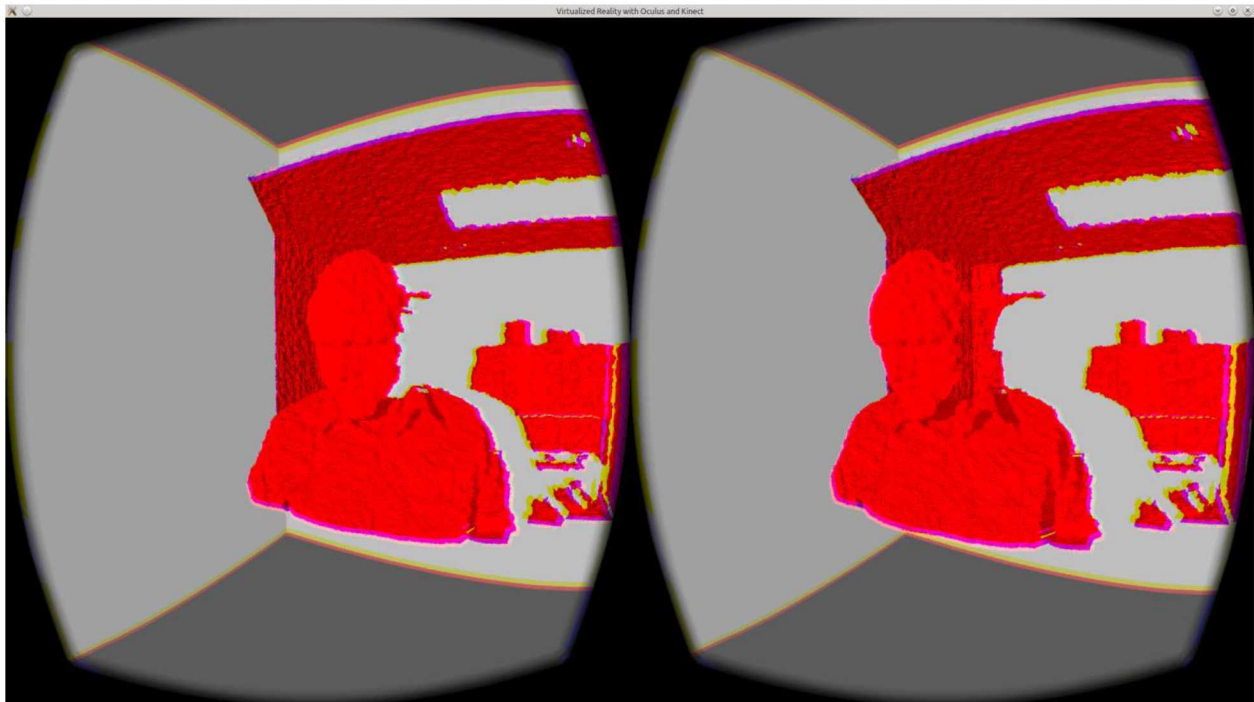


FIGURE 16: REAL-WORLD GEOMETRY RENDERED ON THE OCULUS RIFT

The first additional problem I explored was that of coloring the geometry. For this to be of practical use, some kind of appropriate coloring needs to be applied to whatever geometry is produced. The Kinect comes equipped with an RGB camera, so I thought to use this to acquire the colors for the image. For my first attempt, I simply colored the geometry using a one to one correspondence between the pixels in the depth image and the RGB image. As shown in figure 17, however, the colors did not line up correctly using this method, and the disparity was worse toward the extremities of the object. There are a couple reasons for this. The first is that the depth camera and RGB camera have different fields of view. This means that the RGB camera sees more of the scene than the depth camera does. In addition, the RGB camera and depth camera point in slightly different directions, which vary from device to device. There is a device-dependent series of transformations that one can perform to project a depth pixel into the color image, but simply adjusting for the difference in field of view and distance between IR and RGB cameras produces reasonable results.

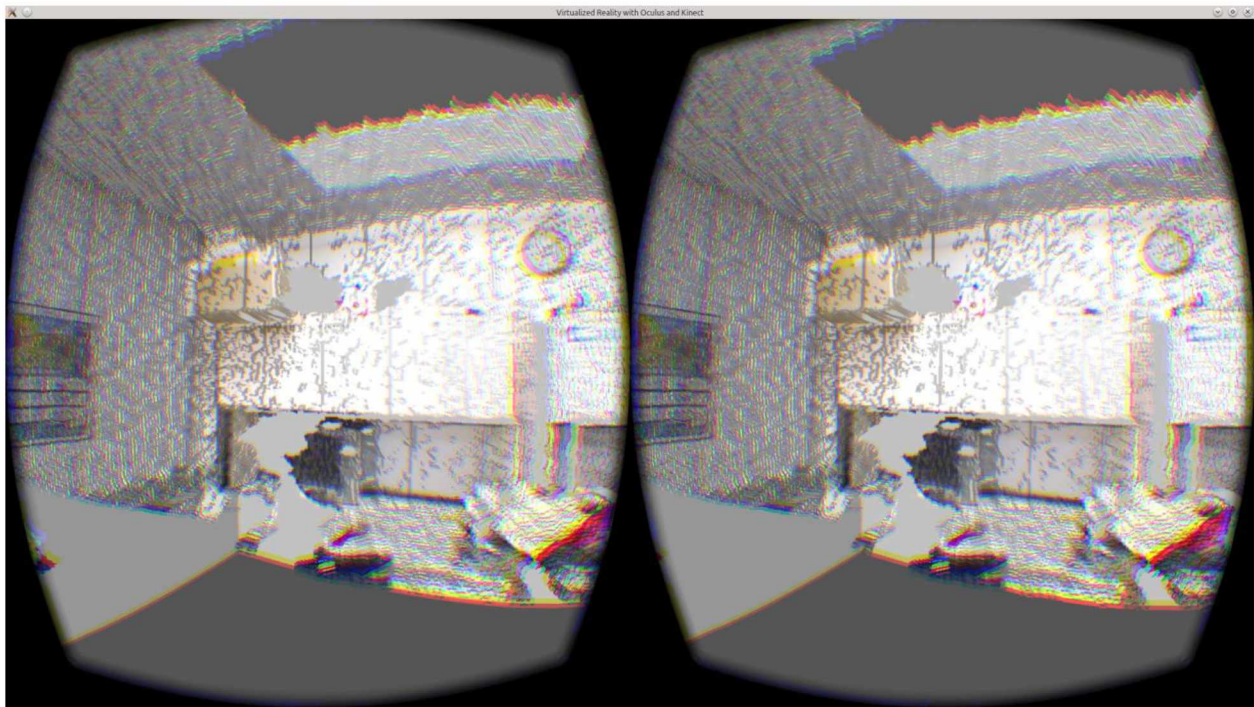


FIGURE 17: RGB IMAGE APPLIED TO GEOMETRY WITHOUT CORRECTION

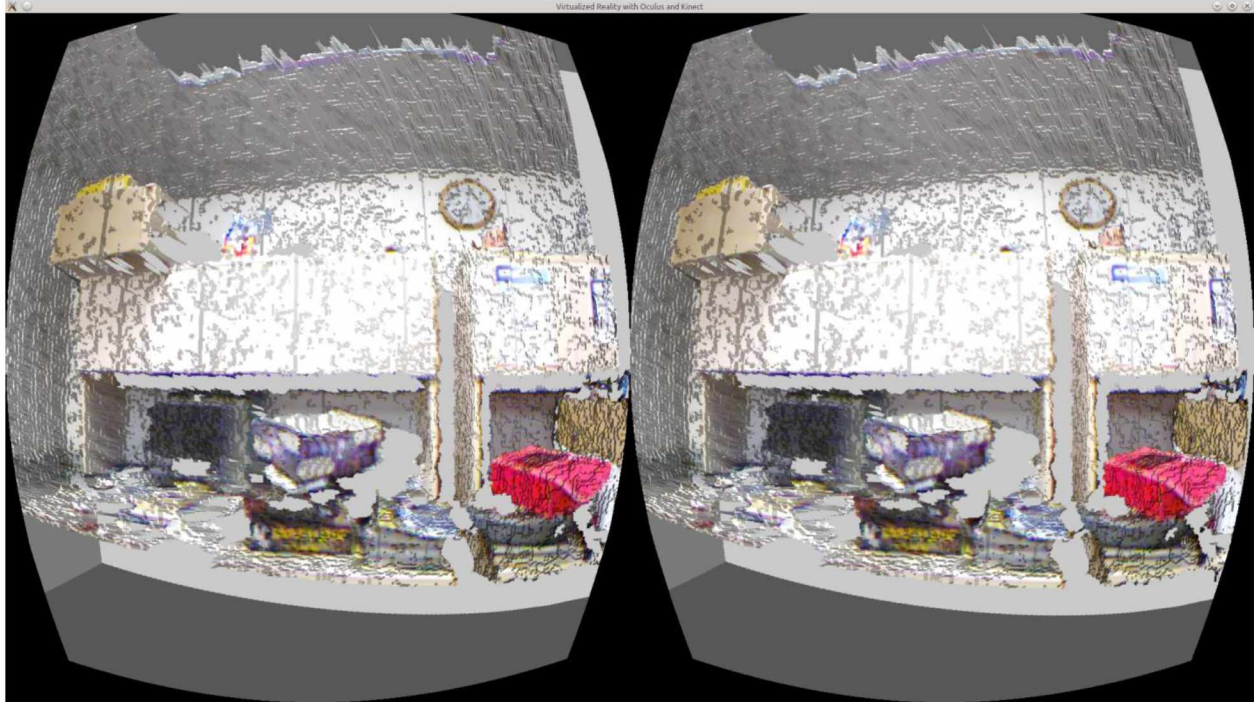


FIGURE 18: FOV AND DISTANCE BETWEEN CAMERAS ACCOUNTED FOR WHEN APPLYING TEXTURE

The second issue I wanted to explore was how to deal with depth measurement jitter in the generated object. Because the depth image from the Kinect is imprecise, and the object is regenerated every frame, the object surface appears to be constantly shifting. I explored three simple ideas to see how they would affect the appearance.

The first idea was to sample a pixel from every other row and column. This generates half as many points which are twice as far apart. The result is that slopes between points tend to be smaller, but the magnitude of deviations does not change. In the end, this method had little effect.

The second idea was to average the generated points over several Kinect frames. The idea is that it will not affect points that are stable, while the points that experience jitter will tend toward an average value. This value may still not be correct, but should generally be more stable. I only averaged over two frames in my test, but it resulted in generally smoother edges and more subdued jitter, as figure 19 shows. It is likely that averaging over a greater number of frames would produce better results, but this would also increase latency in the video. A second drawback of this method is that points that are invalid in one of the frames being averaged, therefore having coordinates $(0, 0, 0)$, will pull the value of the other frame well away from its real position, creating visual artifacts.

The third method yielded the best results for the least effort. This method involved removing computer-generated lighting from the objects generated by the Kinect, and can be seen in figure 20. Because the objects are colored with the real-world image, lighting from the real-world is apparent on the object, creating subtle shading differences that help the viewer to recognize three dimensional shape. Additionally, because there is no jitter in the real-world, it is

not apparent when viewing the object directly due to the lack of shading. Jitter is still apparent at the edges, though, because of deformation of the object silhouettes. The obvious drawback of this method is that the real-world lighting may not match the computer-generated lighting in the scene.

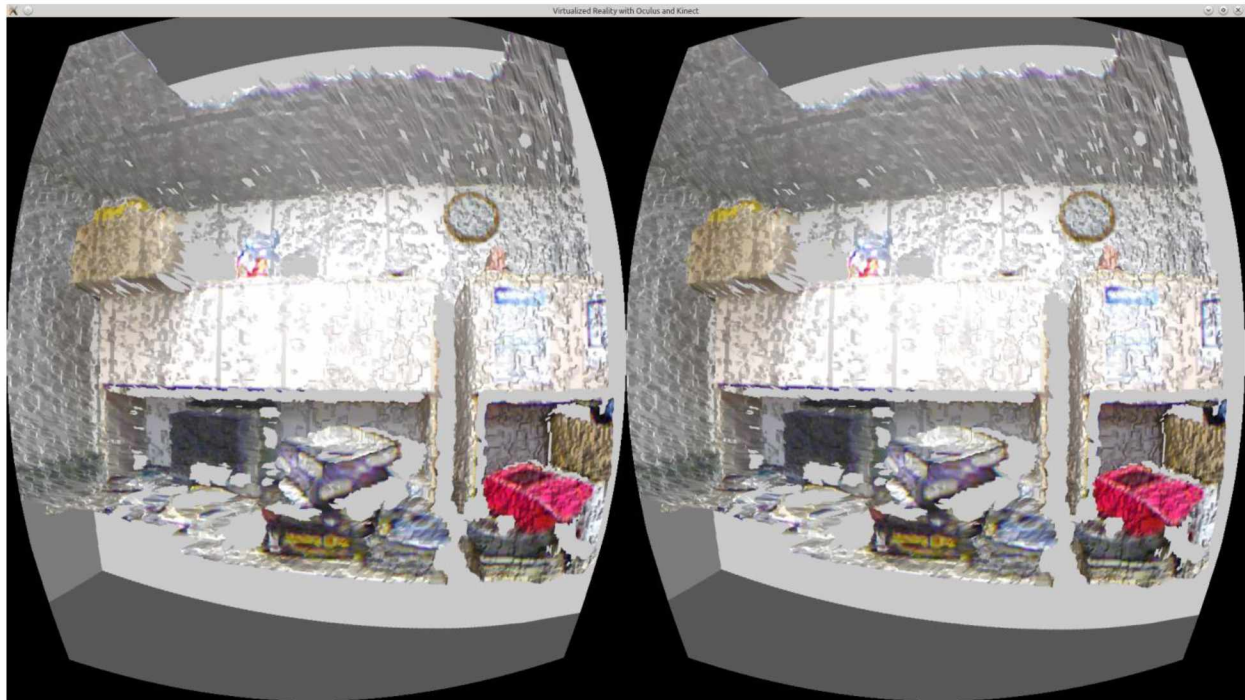


FIGURE 19: SCENE RENDERED WITH GEOMETRY AVERAGED OVER 2 KINECT FRAMES

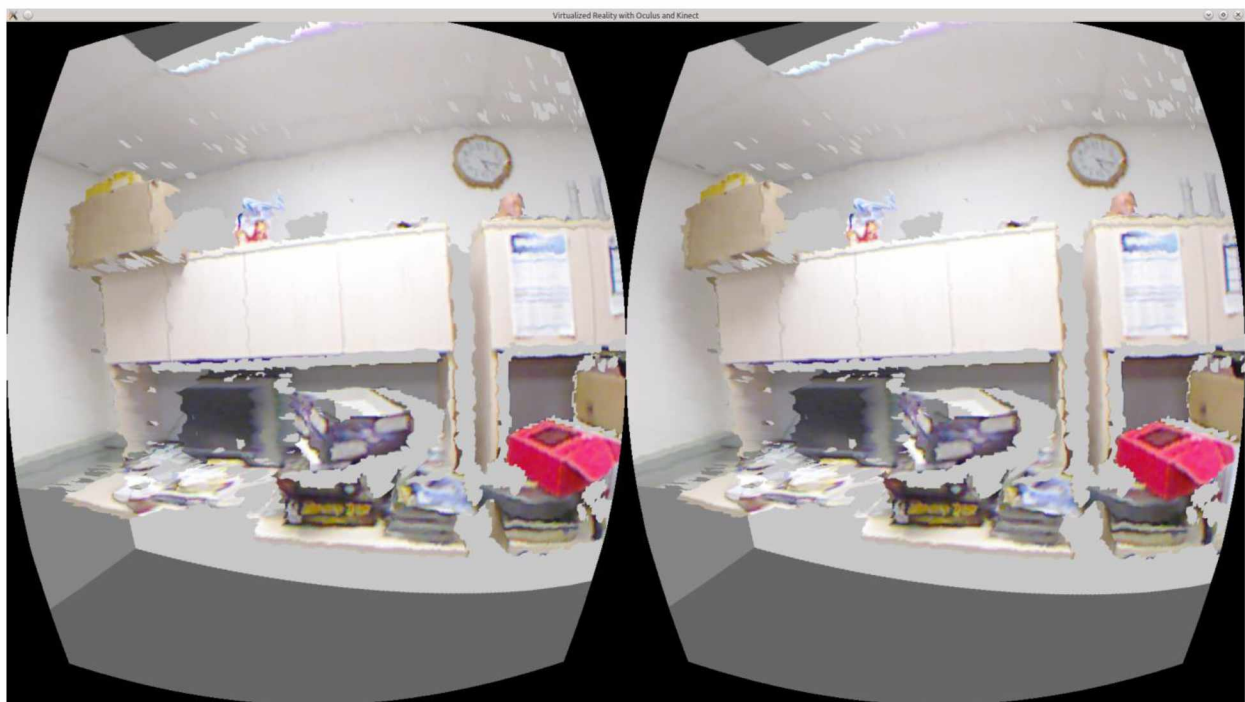


FIGURE 20: SCENE RENDERED WITH COMPUTER-GENERATED LIGHTING REMOVED FROM REAL-WORLD GEOMETRY

PERFORMANCE

As I was implementing this project, I recorded some performance metrics in case I needed to improve the processing or rendering speed to meet my goal of real-time interaction. Without the Oculus as part of the system, I was able to process frames from the Kinect at close to 28 frames per second (fps) when I compiled the code with full optimizations. I was also able to display frames at a rate of around 36 fps. This was acceptable because the Kinect's max frame rate is 30 fps.

When I added the Oculus to the system, the display rate improved, but the Kinect processing rate dropped rather dramatically. This is surprising because the Oculus imposes no extra load on the Kinect. Without access to the core implementation, or time for additional tests, I can only hypothesize that this shift in performance is due to the Oculus Rift using a multithreaded implementation that was stealing CPU time from the thread the Kinect was running in.

Otherwise, the system performed as one would expect. With the extra processing of the RGB image from the Kinect for coloring the object, the performance again dropped somewhat. I could only process frames from the Kinect at around 12 fps at this point, which was feeling rather choppy. However, sampling every other row and column, instead of every pixel, drastically improved the performance, and did not result in a significant drop in image quality.

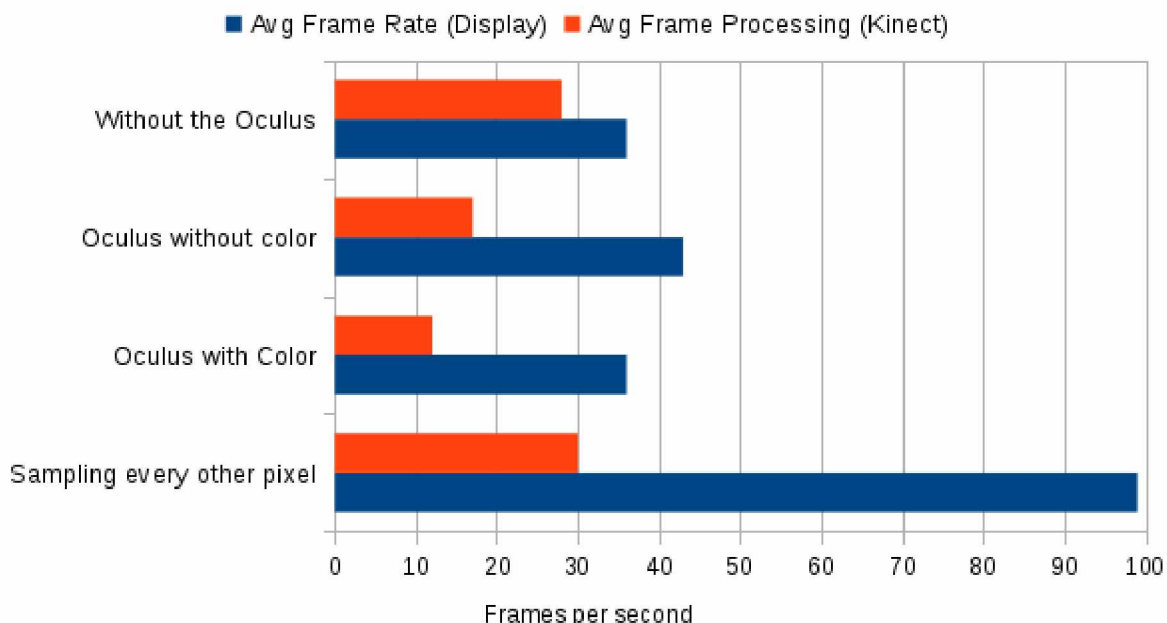


FIGURE 21: PERFORMANCE OF RENDERING AND KINECT IMAGE PROCESSING

FUTURE RESEARCH

There is a lot of potential for future research extending the scope of this project. A few ideas follow:

- Automatically orienting Kinect generated geometry. The geometry is in the same units used by the Oculus rift for positional tracking, but it is oriented relative to the Kinect. Since the Kinect can be located at an arbitrary position and orientation, there should be some way to calibrate it so that objects detected will be in the proper space. For example, if we could automatically identify the Oculus headset in the Kinect image, we could correlate the Oculus and Kinect coordinate systems.
- Using multiple Kinects, or persistent objects, to generate full 3D objects. A single Kinect can only see objects from one side. Additionally, it will cast IR shadows, preventing it from seeing objects behind other objects. Collecting multiple images and combining them to acquire a more complete image would do a lot to enhance the experience. When the same object is visible in more than one Kinect, it should be possible to automatically compute the coordinate system transformations between them to make the images of the object align.
- Reducing or eliminating jitter. More experiments could be done with averaging of frames, but I feel that any real progress here will require some level of object persistence. If objects are only updated when data is missing or new, then jitter could be eliminated. This would be a difficult problem that delves into computer vision issues, because you would need a way to determine when a non-stationary object has changed and how.
- Differentiation of objects. Currently, everything seen by the Kinect is treated as a single object unless the depth discrepancy with neighboring objects passes some threshold. This is a very simplistic algorithm that may separate a single object into two or more. A better algorithm would enhance the experience greatly, and may allow objects to be virtually manipulated when combined with object persistence. One possibility could be to include color data in classifying object connectivity, or to use persistence or multiple Kinects to determine when there is a discontinuity between objects.

REFERENCES

- (Carnegie Mellon, 2001) Carnegie Mellon. "Carnegie Mellon Goes to the Super Bowl." *Carnegie Mellon Goes to the Super Bowl*. Jan. 2001. Web. 8 Apr. 2015.
- (Chen, 2013) Chen, Xilin, Hanjing Li, Tim Pan, Stewart Tansley, and Ming Zhou. "Kinect Sign Language Translator Expands Communication Possibilities." *Microsoft Research*. 30 Oct. 2013. Web. 8 Apr. 2015.
- (Freedman, 2012) Freedman, Barak, Alexander Shpunt, Meir Machline, and Yoel Arieli. "Depth Mapping Using Projected Patterns." Prime Sense Ltd., assignee. Patent US 8150142 B2. 3 Apr. 2012. Print.
- (Goza, 2004) Goza, S.M., R. O. Ambrose, M. A. Diftler, and I. M. Spain. "Telepresence Control of the NASA/DARPA Robonaut on a Mobility Platform." *CHI '04 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 623-629. ACM. 25 Apr. 2004. Print.
- (Hine, 1994) Hine, B. P., et al. "The Application of Telepresence and Virtual Reality to Subsea Exploration." *Proc. ROV '94. The 2nd Workshop on Mobile Robots for Subsea Environments*, May 1994. Print.
- (Hine, 1995) Hine, B., P. Hontalas, Terrence W. Fong, L. Piguet, E. Nygren, and A. Kline. "VEVI: A Virtual Environment Teleoperations Interface for Planetary Exploration." *SAE 25th International Conference on Environmental Systems*, July 1995. Print.
- (Izadi, 2011) Izadi, Shahram, et al. "KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera." *Microsoft Research*. ACM Symposium on User Interface Software and Technology, Oct. 2011. Web. 8 Apr. 2015.
- (Kanade, 1995) Kanade, Takeo, Peter Rander, and P. J. Narayanan. "Virtualized Reality: Concepts and Early Results." *IEEE Workshop on the Representation of Virtual Scenes*, pp. 69-76. IEEE. 24 June 1995. Print.

- (Kanade, 1997) Kanade, Takeo, Peter Rander, and P. J. Narayanan. "Virtualized Reality: Constructing Virtual Worlds from Real Scenes." *IEEE Multimedia, Immersive Telepresence* Vol. 4, Iss. 1, pp. 34-47. IEEE. Jan. 1997. Print.
- (OpenKinect, 2013) OpenKinect Project. "Imaging Information." *OpenKinect*. 12 Nov. 2013. Web. 17 Apr. 2015.
- (Ota, 2012) Ota, Jonathan. "Virtualized Reality." *Virtualized Reality - Jonathan Ota*. 2012. Web. 8 Apr. 2015.
- (Tan, 2012) Tan, Justin H., Cherng Chao, Mazen Zawaideh, Anne C. Roberts, and Thomas B. Kinney. "Informatics in Radiology: Developing a Touchless User Interface for Intraoperative Image Control during Interventional Radiology Procedures." *RadioGraphics*. RSNA. 12 Sept. 2012. Web. 20 Apr. 2015.
- (Terndrup, 2015) Terndrup, Matthew. "Reverse Engineering the Oculus Rift DK2 Provides Brilliant Insight into Inner Workings." *Road to VR*. 8 Oct. 2014. Web. 8 Apr. 2015.
- (Thin, 2012) Thin, Alasdair G. "A Game-Based Virtualized Reality Approach for Simultaneous Rehabilitation of Motor Skill and Confidence." *Advances in Human-Computer Interaction* Vol. 2012. Hindawi Publishing Corporation. 8 Oct. 2012. Web. 8 Apr. 2015.

APPENDIX A: SOURCE CODE

```
/* Author: Shaun Bond (samuraicodemonkey@gmail.com)
 * Date: 4-20-2015
 * Source code is available at:
 * https://projects.cs.uaf.edu/redmine/projects/virtualized_reality_oculus_kinect
 *
 * This code is licensed to you under the terms of the Apache License, version
 * 2.0, or, at your option, the terms of the GNU General Public License,
 * version 2.0. See the APACHE20 and GPL2 files for the text of the licenses,
 * or the following URLs:
 * http://www.apache.org/licenses/LICENSE-2.0
 * http://www.gnu.org/licenses/gpl-2.0.txt
 *
 * If you redistribute this file in source form, modified or unmodified, you
 * may:
 * 1) Leave this header intact and distribute it under the same terms,
 *    accompanying it with the APACHE20 and GPL20 files, or
 * 2) Delete the Apache 2.0 clause and accompany it with the GPL2 file, or
 * 3) Delete the GPL v2 clause and accompany it with the APACHE20 file
 * In all cases you must keep the copyright notice intact and include a copy
 * of the CONTRIB file.
 *
 * Binary distributions must follow the binary distribution requirements of
 * either License.
 */

// NOTE: Oculus daemon must be running (may have to be manually started)
// NOTE: Kinect and Oculus position tracking camera must be on different buses

#include "libfreenect.hpp"
#include "lib/glsprog.h"
#include <pthread.h>

#include <iostream>
using std::cerr;
using std::cout;
using std::endl;

#include <string>
using std::string;

#include <vector>
using std::vector;

#include <algorithm>
using std::copy;

#include <iomanip>
using std::setw;
using std::fixed;
using std::setprecision;

#include "lib/vec4.h"
#include "OVR.h"

#include "OVR_CAPI_GL.h"
using OVR::Matrix4f;
using OVR::Vector3f;

#include "Service_NetClient.h" // So I can query Oculus service for HMD errors
```

```

#if defined(__APPLE__)
#include <GLUT/glut.h>
#include <OpenGL/gl.h>
#include <OpenGL/glu.h>
#else
#include <GL/glut.h>
#include <GL/gl.h>
#include <GL/glu.h>
#endif

// Global constants
const short IMG_WIDTH = 640;
const short IMG_HEIGHT = 480;
const short PXL_SIZE = 3;
const short DIMENSIONS = 3;
const short INVALID_DEPTH = 2047;
const char ESC = 27;

// Global variables
int saved_x = 0;
int saved_y = 0;

// Dimensions of render texture
unsigned texture_w = 0;
unsigned texture_h = 0;

vector<unsigned> indices; // Vertex indices for triangle strip.
vector<float> texCoords; // Texture coordinates for triangle strip.

ovrHmd hmd = NULL;
ovrEyeRenderDesc eyeRenderDesc[2];
ovrGLTexture eyeTextures[2];

GLuint hide_invalid_vertices = 0;
GLuint gl_rgb_tex;
GLuint eye_tex[2];
GLuint frame_buffers[2];

double freenect_angle(0);
int window(0);
int g_argc;
char **g_argv;

/*
Coordinate system:
Origin is Kinect's IR receiver
+X faces to the left (from Kinect's point of view)
+Y is up
+Z is away from Kinect

Assumes that depth image is 640 x 480

This class is curtesy of Dr. Orion Lawlor. Used here in original form.
*/
class kinect_depth_image {
public:
kinect_depth_image(const uint16_t *d_)
: depthi(d_), w(IMG_WIDTH), h(IMG_HEIGHT)
{

```

```

    pixelFOV=tan(0.5 * (M_PI / 180.0) * 57.8)/(w*0.5);
}

/* Return depth, in meters, at this pixel */
float depth(int x,int y) const {
    uint16_t disp=depthi[y*w+x];
    if (disp>= INVALID_DEPTH) return 0.0;

    //From Stephane Magnenat's depth-to-distance conversion function:
    return 0.1236 * tan(disp / 2842.5 + 1.1863) - 0.037; // (meters)
}

/* Return 3D direction pointing from the sensor out through this pixel
   (not a unit vector) */
vec3 dir(int x,int y) const {
    // Ypix = -Ydist / (pixelFOV*Depth) + .5h
    return vec3((x-w*0.5)*pixelFOV, (h*0.5-y)*pixelFOV, 1);
}

/* Return 3D location, in meters, at this pixel */
vec3 loc(int x,int y) const {
    // Project view ray out for that pixel
    return dir(x,y)*depth(x,y);
}

private:
    const uint16_t *depthi;
    int w, h;          /* dimensions of image */
    float pixelFOV; /* Unit-depth field of view offset per X or Y pixel */
};

/* Borrowed this class from cppview.cpp. Used here in original form. */
class Mutex {
public:
    Mutex() {
        pthread_mutex_init( &m_mutex, NULL );
    }
    void lock() {
        pthread_mutex_lock( &m_mutex );
    }
    void unlock() {
        pthread_mutex_unlock( &m_mutex );
    }

    class ScopedLock
    {
    public:
        ScopedLock(Mutex & mutex)
        : _mutex(mutex)
        {
            _mutex.lock();
        }
        ~ScopedLock()
        {
            _mutex.unlock();
        }
    };
};

private:
    pthread_mutex_t m_mutex;
};

```

```

/* Borrowed this class from cppview.cpp. Used here in a heavily modified form */
class MyFreenectDevice : public Freenect::FreenectDevice {
public:
    enum DisplayMode {POINTS, TRIANGLES};

    MyFreenectDevice(freenect_context *_ctx, int _index)
    : Freenect::FreenectDevice(_ctx, _index),
      m_buffer_video(freenect_find_video_mode(FREENECT_RESOLUTION_MEDIUM,
                                              FREENECT_VIDEO_RGB).bytes),
      m_new_rgb_frame(false),
      m_new_vertices(false),
      m_display_format(TRIANGLES),
      m_depth_frames(0)
    {}

    ~MyFreenectDevice() {
        stopVideo();
        stopDepth();
    }

    // Do not call directly even in child
    void VideoCallback(void* _rgb, uint32_t timestamp) {
        Mutex::ScopedLock lock(m_rgb_mutex);
        uint8_t* rgb = static_cast<uint8_t*>(_rgb);
        copy(rgb, rgb+getVideoBufferSize(), m_buffer_video.begin());
        m_new_rgb_frame = true;
    };

    // Do not call directly even in child
    // Recieves a depth image for processing.
    // Stores grayscale image in m_buffr_depth, with greater distance = darker.
    // Stores 3d vertex for each pixel in m_vertices.
    // Sets m_new_depth_frame and m_new_vertices to true.
    void DepthCallback(void* _depth, uint32_t timestamp) {
        Mutex::ScopedLock vertexLock(m_vertex_mutex);

        uint16_t* depth = static_cast<uint16_t*>(_depth);
        kinect_depth_image img(depth);

        // Move last frame into m_vertices.
        m_vertices.clear();

        // Convert every other row and every other column into vertices.
        for( unsigned int yy = 0 ; yy < IMG_HEIGHT ; yy+=2) {
            for( unsigned int xx = 0 ; xx < IMG_WIDTH ; xx+=2) {

                // Get 3d coordinates of pixel in meters
                vec3 vertex = img.loc(xx, yy);

                // Push vertex onto vertex array.
                m_vertices.push_back( vertex.x );
                m_vertices.push_back( vertex.y );
                m_vertices.push_back( vertex.z );
            }
        }

        m_new_vertices = true;
        m_depth_frames += 1;
    }

    // If no new rgb frame
    // Returns false
    // buffer remains unchanged.

```



```

// m_new_rgb_frame remains unchanged.
// Otherwise
// Returns true
// buffer will contain rgb image acquired from Kinect.
// Sets m_new_rgb_frame to false;
bool getRGBFrame(vector<uint8_t> &buffer) {
    Mutex::ScopedLock lock(m_rgb_mutex);

    if (!m_new_rgb_frame)
        return false;

    buffer.swap(m_buffer_video);
    m_new_rgb_frame = false;
    return true;
}

// If no new depth frame
// Returns false
// buffer remains unchanged.
// m_new_vertices remains unchanged.
// Otherwise
// Returns true
// buffer will contain 3d vertices acquired from depth image.
// Sets m_new_vertices to false;
bool getVertices(vector<float> &buffer) {
    Mutex::ScopedLock lock(m_vertex_mutex);

    if (!m_new_vertices)
        return false;

    buffer.swap(m_vertices);
    m_new_vertices = false;
    return true;
}

// Returns the currently set display format.
DisplayMode getDisplayMode() {
    return m_display_format;
}

// Returns the number of frames which have been processed.
unsigned getFrames() {
    return m_depth_frames;
}

// Toggles display mode between 3d point cloud, and kinect video.
void toggleDisplayMode() {
    if (m_display_format == TRIANGLES)
        m_display_format = POINTS;
    else if (m_display_format == POINTS)
        m_display_format = TRIANGLES;
}

private:
vector<uint8_t> m_buffer_video;
vector<float> m_vertices;
Mutex m_rgb_mutex;
Mutex m_vertex_mutex;
bool m_new_rgb_frame;
bool m_new_vertices;
DisplayMode m_display_format;
unsigned m_depth_frames;

```

```

};

Freenect::Freenect freenect;
MyFreenectDevice* device;

// This function is called every frame to track FPS statistics.
void calculateFPS ()
{
    // Variables used for various framerate calculations
    static const unsigned NUM_FRAMES = 30;
    static unsigned frames = 0;
    static double saved_time[NUM_FRAMES] = {0};
    static double avg_fps = 0;
    static double max_fps = 0;
    static double min_fps = DBL_MAX;

    // Used to access time NUM_FRAMES frames ago
    unsigned index = frames % NUM_FRAMES;

    // Get elapsed time in seconds and record the current time
    double curr_time = glutGet(GLUT_ELAPSED_TIME) / 1000.0;
    double elapsed_time = curr_time - saved_time[index];
    saved_time[index] = curr_time;

    // Calculate current fps averaged over NUM_FRAMES frames for stability,
    // average fps over total execution time, and min and max fps.
    // First NUM_FRAMES frames don't yield accurate information.
    double fps = NUM_FRAMES / elapsed_time;
    if( frames > NUM_FRAMES - 1 ) // At least NUM_FRAMES frames have passed
    {
        float x = 1.0 / (frames - (NUM_FRAMES - 1)); // 1 / number of fpses in average
        avg_fps = (1 - x) * avg_fps + x * fps;

        if( fps > max_fps ) max_fps = fps;
        if( fps < min_fps ) min_fps = fps;

        // Here is some console output for user
        device->updateState();
        cout << "\r demanded tilt angle: " << setw(5) << freenect_angle
            << " device tilt angle: " << setw(5)
            << device-> getState().getTiltDegs()
            << fixed << setprecision(2)
            << " fps: " << setw(6) << fps
            << " avg fps: " << setw(6) << avg_fps
            << " min fps: " << setw(6) << min_fps
            << " max fps: " << setw(6) << max_fps
            << " kinect fps: " << setw(6) << device->getFrames() / curr_time;
        cout.flush();
    }

    ++frames;
}

// Sets up rendering parameters for kinect image vertices
void setUpVertices(void* vertices)
{
    // Send vertices to the graphics card
    glVertexPointer(3,
                   GL_FLOAT,
                   3*sizeof(float),

```

```

        vertices );

    glEnableClientState(GL_VERTEX_ARRAY);
}

// This function is responsible for rendering the scene every frame
void DrawGLScene()
{
    // Set up buffers for images and point cloud
    static vector<uint8_t> rgb(IMG_WIDTH * IMG_HEIGHT * PXL_SIZE);
    static vector<float> vertices(IMG_WIDTH * IMG_HEIGHT * DIMENSIONS / 4);

    calculateFPS();

    // Start rendering. This allows libOVR to track timing information
    // for things like predictive position tracking, which helps with rendering.
    ovrHmd_BeginFrame(hmd, 0);

    // set viewport
    glViewport(0, 0, texture_w, texture_h);

    // Get the offset of each eye from center.
    ovrVector3f hmdToEyeViewOffset[2];
    hmdToEyeViewOffset[0] = eyeRenderDesc[0].HmdToEyeViewOffset;
    hmdToEyeViewOffset[1] = eyeRenderDesc[1].HmdToEyeViewOffset;

    // Position and orientation of each eye will be stored in eyePoses.
    ovrPosef eyePoses[2];
    ovrTrackingState hmdState;
    ovrHmd_GetEyePoses(hmd, 0, hmdToEyeViewOffset, eyePoses, &hmdState);

    if(!(hmdState.StatusFlags & ovrStatus_PositionTracked))
        cout << endl << "No position tracking" << endl;

    if(!(hmdState.StatusFlags & ovrStatus_PositionConnected))
        cout << endl << "Position tracker not connected" << endl;

    // Get the geometry.
    device->getVertices(vertices);
    setUpVertices(&vertices.front());

    // Setup the texture to place on geometry.
    device->getRGBframe(rgb);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, gl_rgb_tex);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, IMG_WIDTH, IMG_HEIGHT,
                0, GL_RGB, GL_UNSIGNED_BYTE, rgb.data());

    // Render the scene for each eye
    for(int index = 0; index < ovrEye_Count; ++index)
    {
        ovrEyeType curr_eye = hmd->EyeRenderOrder[index];

        // Bind framebuffer
        glBindFramebuffer(GL_FRAMEBUFFER, frame_buffers[curr_eye]);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // Get a projection matrix from LibOVR.
        Matrix4f projection = ovrMatrix4f_Projection(eyeRenderDesc[curr_eye].Fov, .01,
                                                    100, false);
    }
}

```

```

// Calculate left handed up vector and forward vector for eye
Matrix4f view = Matrix4f(eyePoses[curr_eye].Orientation);
OVR::Vector3f up      = view.Transform(OVR::Vector3f(0, -1, 0));
OVR::Vector3f forward = view.Transform(OVR::Vector3f(0, 0, -1));

// Get view matrix from LibOVR.
// Orientation + position in left handed system.
view = OVR::Matrix4f::LookAtLH(eyePoses[curr_eye].Position,
                               OVR::Vector3f(eyePoses[curr_eye].Position) +
                               forward,
                               up);

// Set projection matrix.
glMatrixMode(GL_PROJECTION);
glLoadMatrixf(&projection.Transposed().M[0][0]);

// Set view matrix.
glMatrixMode(GL_MODELVIEW);
glLoadMatrixf(&view.Transposed().M[0][0]); // Camera position/orientation
// .5 meters from position tracking camera is a good distance to call center
glTranslatef(0, 0, .5); // Move World
// Rotate World

// Draw cube (virtual room)
glPushMatrix();
// Position tracking camera is 1.2m high (0 y coordinate),
// virtual floor is 1.5m below 0 y, move up by difference.
glTranslatef(0, .3, 0); // Move "room"
// Rotate "room"

glColor4f(0.5, 0.5, 0.5, 1.0);
glutSolidCube(3.8);
glPopMatrix();

// Draw Kinect geometry
glPushMatrix();
// Transform Kinect geometry
// Negate Z because image is behind
// Kinect is positioned .5 meters above position tracking camera,
// reduced by .1 meters due to angle of camera.
glTranslatef(-.5, .4, 1.6); // Move geometry
// Rotate geometry

glScalef(1, 1, -1);

glColor4f(1, 0, 0, 1);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);

if (device->getDisplayMode() == MyFreenectDevice::POINTS)
{
// Draw point cloud
glDrawArrays(GL_POINTS, 0, IMG_WIDTH * IMG_HEIGHT/4 );
}
else if (device->getDisplayMode() == MyFreenectDevice::TRIANGLES)
{
// Draw triangle strip
glUseProgram(hide_invalid_vertices);
glDrawElements(GL_TRIANGLE_STRIP, indices.size(), GL_UNSIGNED_INT,
               &indices.front());
glUseProgram(0);
}

glDisableClientState(GL_TEXTURE_COORD_ARRAY);
glPopMatrix();

```



```

    }

    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glBindTexture(GL_TEXTURE_2D, 0);

    // Tell LibOVR to display the rendered scene.
    ovrHmd_EndFrame(hmd, eyePoses, &eyeTextures[0].Texture);
}

// This is executed when there is no input.
void idleFunc()
{
    // Written by Glenn G. Chappell
    static int error_count = 0;
    if (GLenum err = glGetError())
    {
        ++error_count;
        std::cerr << "OpenGL ERROR " << error_count << ": "
                  << gluErrorString(err) << endl;
    }

    glutPostRedisplay();
}

// This handles keyboard keypresses.
void keyPressed(unsigned char key, int x, int y)
{
    switch (key)
    {
        case ESC: // Shutdown program
            glutDestroyWindow(window);

            // Clean up hmd and oculus VR library.
            ovrHmd_Destroy(hmd);
            ovr_Shutdown();

            cout << endl << "Finished" << endl << endl;
            break;
        case 'v': // Toggle display mode between point cloud and triangle strip.
            cout << endl << endl << " Changing display mode to: ";
            device->toggleDisplayMode();
            if(device->getDisplayMode() == MyFreenectDevice::POINTS)
                cout << "POINTS" << endl;
            else if(device->getDisplayMode() == MyFreenectDevice::TRIANGLES)
                cout << "TRIANGLES" << endl;
            break;

        // Change verticle tilt angle of Kinect.
        case 'w':
            freenect_angle++;
            if (freenect_angle > 30)
            {
                freenect_angle = 30;
            }
            break;
        case 's':
        case 'd':
            freenect_angle = 0;
            break;
        case 'x':
            freenect_angle--;
    }
}

```

```

        if (freenect_angle < -30)
        {
            freenect_angle = -30;
        }
        break;
    case 'e':
        freenect_angle = 10;
        break;
    case 'c':
        freenect_angle = -10;
        break;
    default: ;
}

device->setTiltDegrees(freenect_angle);
}

// Tracks the current mouse position when
// the mouse button is held and the mouse is moved.
void clickAndDrag(int x, int y)
{
    saved_x = x;
    saved_y = y;
}

// Tracks the current mouse position when mouse is moved.
void mouseMove(int x, int y)
{
    saved_x = x;
    saved_y = y;
}

// Generate texture coordinate array for triangle strip assuming every pixel is a
// vertex
void generateTextureCoords()
{
    const float fovCorrection = .92185;
    const float offset = (1 - fovCorrection) / 2;
    for( unsigned yy = 0; yy < IMG_HEIGHT; yy+=2 ) {
        for( unsigned xx = 0; xx < IMG_WIDTH; xx+=2 ) {

            texCoords.push_back(float(xx) / IMG_WIDTH * fovCorrection + offset);
            texCoords.push_back(float(yy) / IMG_HEIGHT * fovCorrection + 1.5 *
                                offset);
        }
    }

    glTexCoordPointer(2, GL_FLOAT, 0, &texCoords.front());
}

// Initialize rendering variables, and set up shaders.
void InitGL(unsigned int tex_w, unsigned int tex_h)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClearDepth(1.0);

    // Compile shaders into a program.
    glewInit();
    string vShader = "shaders/invalids_v.glsl";

```

```

string gShader = "shaders/normals_g.glsl";
string fShader = "shaders/invalids_f.glsl";
hide_invalid_vertices = makeShaderProgramFromFiles(vShader, gShader, fShader);

// Create a texture for coloring Kinect geometry.
glGenTextures(1, &gl_rgb_tex);
glBindTexture(GL_TEXTURE_2D, gl_rgb_tex);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
generateTextureCoords();

// Create textures for each eye, and framebuffers for drawing to the textures.
glGenTextures(2, eye_tex);
glGenFramebuffers(2, frame_buffers);
GLuint render_buffers[2];
glGenRenderbuffers(2, render_buffers);

// For position tracking.
ovrHmd_ConfigureTracking(hmd, ovrTrackingCap_Orientation |
                        ovrTrackingCap_MagYawCorrection |
                        ovrTrackingCap_Position, 0);

// Configure OVR rendering.
ovrGLConfig apiConfig;
apiConfig.OGL.Header.API = ovrRenderAPI_OpenGL;
apiConfig.OGL.Header.BackBufferSize = OVR::Sizei(hmd->Resolution.w,
                                                hmd->Resolution.h);

apiConfig.OGL.Header.Multisample = 1;
apiConfig.OGL.Disp = NULL;

ovrHmd_ConfigureRendering(hmd,
                        &apiConfig.Config,
                        hmd->DistortionCaps,
                        hmd->DefaultEyeFov,
                        eyeRenderDesc);

// Set up render textures for each eye, and pass information to LibOVR.
for(int eye = 0; eye < 2; ++eye)
{
    // Make empty texture with correct size.
    glBindTexture(GL_TEXTURE_2D, eye_tex[eye]);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tex_w, tex_h, 0, GL_RGBA,
                GL_UNSIGNED_BYTE, NULL);

    // Attach texture to render buffer.
    glBindFramebuffer(GL_FRAMEBUFFER, frame_buffers[eye]);
    glBindRenderbuffer(GL_RENDERBUFFER, render_buffers[eye]);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, tex_w, tex_h);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
                          eye_tex[eye], 0);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                              GL_RENDERBUFFER, render_buffers[eye]);

    // Give texture handles to LibOVR.
    eyeTextures[eye].OGL.Header.API = ovrRenderAPI_OpenGL;
    eyeTextures[eye].OGL.Header.TextureSize = OVR::Sizei(tex_w, tex_h);
    eyeTextures[eye].OGL.Header.RenderViewport = OVR::Recti(0, 0, tex_w, tex_h);
    eyeTextures[eye].OGL.TexId = eye_tex[eye];
}

// Bind default texture and frame buffers for safety.
glBindTexture(GL_TEXTURE_2D, 0);

```

```

glBindFramebuffer(GL_FRAMEBUFFER, 0);
glBindRenderbuffer(GL_RENDERBUFFER, 0);

// Enable simple lighting
glEnable(GL_LIGHTING);
glEnable(GL_COLOR_MATERIAL);
glEnable(GL_LIGHT0);
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

glEnable(GL_DEPTH_TEST);
}

// Generate index array for triangle strip assuming every pixel is a vertex
void makeIndexArray()
{
    unsigned height = IMG_HEIGHT/2; // Only using every other pixel in depth image.
    unsigned width = IMG_WIDTH/2; // Only using every other pixel in depth image.

    for( unsigned yy = 0; yy < height-1; ++yy ) {
        for( unsigned xx = 0; xx < width-1; ++xx ) {

            unsigned x;
            if( yy%2 == 1 ) x = width - xx - 1; // Odd rows go backwards
            else x = xx; // Even rows

            // Push back vertical pairs of vertices.
            indices.push_back(yy * width + x);
            indices.push_back((yy+1) * width + x);
        }

        // Only add one at the end of each row,
        // the start of the next row will add the other.
        if( yy%2 == 0 )
            indices.push_back((yy * width) + (width - 1));
        else
            indices.push_back(yy * width);
    }
}

// Handles OpenGL in separate thread.
void *gl_threadfunc(void *arg)
{
    cout << "GL thread" << endl;

    // Initialize Oculus VR library.
    if( !ovr_Initialize() )
    {
        cout<< "Failed to Initialize OVR" << endl;
        exit(1);
    }

    cout << "libOVR initialized." << endl;

    // Create an HMD object with data about the head mounted display.
    hmd = ovrHmd_Create(0);
    if( !hmd )
    {
        cout << OVR::Service::NetClient::GetInstance()->Hmd_GetLastError(0) << endl;
        ovr_Shutdown();
        exit(1);
    }
}

```



```

cout << "HMD created." << endl;

// If the hmd capabilities does not include extended desktop mode.
if( !(hmd->HmdCaps & ovrHmdCap_ExtendDesktop) )
{
    cout << "Not in extended desktop mode." << endl;
    ovrHmd_Destroy(hmd);
    ovr_Shutdown();
    exit(0);
}

makeIndexArray();

// Initialize glut and create window with oculus HMD display size
glutInit(&g_argc, g_argv);
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_ALPHA | GLUT_DEPTH);
glutInitWindowSize(hmd->Resolution.w, hmd->Resolution.h);
glutInitWindowPosition(0, 0);

window = glutCreateWindow("Virtualized Reality with Oculus and Kinect");

// Register glut callback functions
glutDisplayFunc(&DrawGLScene);
glutIdleFunc(&idleFunc);
glutKeyboardFunc(&keyPressed);
glutMotionFunc(&clickAndDrag);
glutPassiveMotionFunc(&mouseMove);

texture_w = hmd->Resolution.w/2;
texture_h = hmd->Resolution.h;

InitGL(texture_w, texture_h);

glutMainLoop();

return NULL;
}

int main(int argc, char **argv)
{
    device = &freenect.createDevice<MyFreenectDevice>(0);
    if( device )
    {
        // Start Kinect processing.
        device->startVideo();
        device->startDepth();

        // Start Rendering in separate thread.
        gl_threadfunc(NULL);
    }
    else
    {
        cerr << "Failed to create Freenect Device." << endl;
    }

    return 0;
}

```

APPENDIX B: GLSL VERTEX SHADER

```
/* Author: Shaun Bond (samuraicodemonkey@gmail.com)
 * Date: 4-20-2015
 * Source code is available at:
 * https://projects.cs.uaf.edu/redmine/projects/virtualized_reality_oculus_kinect
 *
 * This code is licensed to you under the terms of the Apache License, version
 * 2.0, or, at your option, the terms of the GNU General Public License,
 * version 2.0. See the APACHE20 and GPL2 files for the text of the licenses,
 * or the following URLs:
 * http://www.apache.org/licenses/LICENSE-2.0
 * http://www.gnu.org/licenses/gpl-2.0.txt
 *
 * If you redistribute this file in source form, modified or unmodified, you
 * may:
 * 1) Leave this header intact and distribute it under the same terms,
 *    accompanying it with the APACHE20 and GPL20 files, or
 * 2) Delete the Apache 2.0 clause and accompany it with the GPL2 file, or
 * 3) Delete the GPL v2 clause and accompany it with the APACHE20 file
 * In all cases you must keep the copyright notice intact and include a copy
 * of the CONTRIB file.
 *
 * Binary distributions must follow the binary distribution requirements of
 * either License.
 */

// So I can use things like gl_ModelViewProjectionMatrix to save time/effort.
// Ideally, the shaders should use modern glsl specification.
#version 150

//out vec4 v_color;
out vec3 vertex;
out vec2 tex_coords;

void main() {
    vertex = gl_Vertex.xyz; // Unmodified coordinates passed to geometry shader

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    tex_coords = gl_MultiTexCoord0.st;
}
```

APPENDIX C: GLSL GEOMETRY SHADER

```
/* Author: Shaun Bond (samuraicodemonkey@gmail.com)
 * Date: 4-20-2015
 * Source code is available at:
 * https://projects.cs.uaf.edu/redmine/projects/virtualized_reality_oculus_kinect
 *
 * This code is licensed to you under the terms of the Apache License, version
 * 2.0, or, at your option, the terms of the GNU General Public License,
 * version 2.0. See the APACHE20 and GPL2 files for the text of the licenses,
 * or the following URLs:
 * http://www.apache.org/licenses/LICENSE-2.0
 * http://www.gnu.org/licenses/gpl-2.0.txt
 *
 * If you redistribute this file in source form, modified or unmodified, you
 * may:
 * 1) Leave this header intact and distribute it under the same terms,
 * accompanying it with the APACHE20 and GPL20 files, or
 * 2) Delete the Apache 2.0 clause and accompany it with the GPL2 file, or
 * 3) Delete the GPL v2 clause and accompany it with the APACHE20 file
 * In all cases you must keep the copyright notice intact and include a copy
 * of the CONTRIB file.
 *
 * Binary distributions must follow the binary distribution requirements of
 * either License.
 */

const int SIZE = 3;
const float INVALID = 0;

in vec3 vertex[SIZE]; // Incoming from vertex shader
in vec2 tex_coords[SIZE]; // Incoming from vertex shader

out vec2 uv;
//out vec3 surface_normal; // Used for virtual lighting

void main() {
    // Find 2 sides of triangle
    vec3 vector1 = vertex[1] - vertex[0];
    vec3 vector2 = vertex[2] - vertex[0];

    // Uncomment the following for virtual lighting
    //vec3 normal = cross(vector1, vector2);

    // Only draw triangles that do not have a "long" side
    if( length(vector1) < .1 && length(vector2) < .1 ) {

        for(int i = 0; i < SIZE; ++i) {
            gl_Position = gl_in[i].gl_Position;
            //surface_normal = normal; // Used for virtual lighting
            uv = tex_coords[i];

            EmitVertex();
        }
    }
}
```

APPENDIX D: GLSL FRAGMENT SHADER

```
/* Author: Shaun Bond (samuraicodemonkey@gmail.com)
 * Date: 4-20-2015
 * Source code is available at:
 * https://projects.cs.uaf.edu/redmine/projects/virtualized_reality_oculus_kinect
 *
 * This code is licensed to you under the terms of the Apache License, version
 * 2.0, or, at your option, the terms of the GNU General Public License,
 * version 2.0. See the APACHE20 and GPL2 files for the text of the licenses,
 * or the following URLs:
 * http://www.apache.org/licenses/LICENSE-2.0
 * http://www.gnu.org/licenses/gpl-2.0.txt
 *
 * If you redistribute this file in source form, modified or unmodified, you
 * may:
 * 1) Leave this header intact and distribute it under the same terms,
 *    accompanying it with the APACHE20 and GPL20 files, or
 * 2) Delete the Apache 2.0 clause and accompany it with the GPL2 file, or
 * 3) Delete the GPL v2 clause and accompany it with the APACHE20 file
 * In all cases you must keep the copyright notice intact and include a copy
 * of the CONTRIB file.
 *
 * Binary distributions must follow the binary distribution requirements of
 * either License.
 */

// direction of light source, hard coded
const vec3 source = vec3(0,0,1);

uniform sampler2D texture; // RGB image from kinect

in vec2 uv; // Texture coordinates to sample
//in vec3 surface_normal; // Used for virtual lighting

void main() {
    // Uncomment the following for virtual lighting
    //vec3 s_normal = normalize(surface_normal);
    //float intensity = dot(s_normal, source);

    // Uncomment the following for no virtual lighting
    float intensity = 1.0;

    vec4 color = texture2D(texture, uv);
    gl_FragColor = vec4(color.xyz * intensity + color.xyz * .5, 1);
}
```