Virtual Walkthrough of 3D Captured Scenes in Web-based Virtual Reality

Austin Chen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-1
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-1.html

January 9, 2017

Virtual Walkthrough of 3D Captured Scenes in Web-based Virtual Reality

by

Austin Luke Chen

A thesis submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Engineering - Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Avideh Zakhor, Chair
Professor Ren Ng

Fall 2016

The thesis of Austin Luke Chen, titled Virtual Walkthrough of 3D Captured Scenes in Web-based Virtual Reality, is approved:

Chair _____ Date january 9, 2017

_____ Date January 9, 2017

_____ Date _____

University of California, Berkeley

Abstract

Virtual Walkthrough of 3D Captured Scenes in Web-based Virtual Reality

by

Austin Luke Chen

Master of Science in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Avideh Zakhor, Chair

In the last few years, virtual reality head mounted displays (VR HMDs) have seen an explosion of consumer interest. However, virtual content that utilizes the capabilities of these new HMDs is still rather scant. In this thesis, we present a streamlined application that allows users to virtually navigate through photorealistic rendered 3D scenes that were captured by a human-operated backpack system. Our app provides users the ability to look around captured 360-degree panoramas by orienting their headsets, traverse between panoramas by selecting target destinations, visualize normal and depth data by controlling a pancake cursor, and take distance measurements among any two points in space. We discuss the various technical challenges of building the first web-based VR app compatible with the Android-based Daydream VR headset and controller. We also outline techniques for monitoring and optimizing the performance of our web-based VR app, which allowed us to achieve a 5x increase in framerate over our original naive implementation.

1 Introduction

In recent years, a variety of new virtual reality head mounted displays (VR HMDs) have been commercially introduced. From powerful, dedicated HMDs such as the Oculus Rift, to cheap containers making use of existing mobile devices such as Google Cardboard, the amount of hardware available for 3D visualization is unprecedented. To date, however, most of the applications come in the form of consumer entertainment such as games and videos [1]. Applications that leverage VR functionality for professional purposes exist [2], but remain few and far between.

One promising area for VR to disrupt is that of indoor visualization. Specifically, the construction industry has much to gain from being able to view and catch errors at an early stage. Gordon et. al. [3] showed promising results in using 3D laser scans of buildings under construction to find and correct errors before project completion, but have not applied their work to VR visualizations. Johansson [4] demonstrates a novel technique for accelerating the rendering of Building Information Models (BIMs) on the Oculus Rift, but few indoor scenes are stored as a single complex mesh to render.

There are various existing solutions for reality capture in outdoor settings. One well-known example is Google's Street View Car [5], which automatically captures panoramas and distances as it drives along the streets. Bing Streetside [6] and Apple Maps [7] also rely on similar systems. However, 3D indoor mapping is more challenging due to the requirements for higher precision, as well as the lack of GPS positioning indoors.

Research groups and companies have developed various different indoor capture systems. The capture systems able to rapidly scan large areas at a time can be divided into two main categories: cart-based, and backpack-based. Cart-based systems are built on wheels, and automatically capture scenes as they are carted around the building. Examples of cart-based systems on the market include Applanix's Trimble Indoor Mobile Mapping Solution [8] and NavVis's M3 Trolley [9]. However, the drawback of cart-based systems are that they require relatively smooth ground to traverse over; they struggle with obstacles and stairs. Backpack systems are designed instead to be more lightweight, and are worn on the back of an operator who walks along the interior of a building. Leica's Pegasus Backpack [10] is one such example; however, it does not generate panoramas suitable for viewing in VR. The Video and Image Processing (VIP) Group at U.C. Berkeley has devised a backpack system that captures the interior of a building as a set of indoor panoramas alongside depth information [11, 12, 13, 14, 15], which was used to collect the datasets displayed in our application.

(a) (b) (c)

Figure 1.1: Different indoor capture systems on the market. (a) Applanix's Trimble Indoor Mobile Mapping Solution. (b) NavVis's M3 Trolley. (c) Leica's Pegasus Backpack.

1.1 Project Overview

Our overarching goal for this project was to develop a more immersive interface to allow users to explore and understand an indoor environment without having to be physically present. Existing desktop solutions are built on 2D interfaces – screens and mouse input. We aimed to use the stereoscopic nature of VR headsets to allow users to better visualize the 3D positions of objects in a scene, and gyroscopic controllers to better measure the distances between them. We also sought a method of navigation through a potentially large virtual area that would not tire the user, especially as prolonged VR use can cause motion sickness [16]. As a low display refresh rate is also linked to VR sickness [17], we wanted our solution to be as performant as possible.

In this thesis, we present our solution for exploring the VIP backpack's captured panoramas on one of the newest VR systems on the market, the Google Daydream. In Section 2, we describe the main hardware and software components on which we have built our VR viewer. The rationale for choosing Daydream is outlined in Section 2.1, and for choosing the VIP backpack system in 2.2. In Section 3, we describe the process of integrating our components to create a fully functional VR experience. In Section 4, we outline the steps taken to improve our application's framerate and thus reduce visible lag. Finally, in Section 5, we describe future areas to explore and build upon.

2 Technical Choices

Among the wealth of available options for VR HMDs and indoor reality capture systems, we chose the Google Daydream VR and VIP's backpack system. In this section, we describe our choices of hardware and software in more detail, as well as the rationale behind the choices.

2.1 Daydream VR

Google's Daydream VR system [18], released in 2016, consists of two main components: the VR headset Daydream View, and the handheld controller. The VR headset is an evolution of Google's previous VR system, Google Cardboard [19], which is known for its low cost of materials relative to existing headsets. Both the View and Cardboard headsets save on costs of displays and sensors by instead drawing on the capabilities of a smartphone. They split the high-resolution displays of modern smartphones into two sides, and with the help of built-in lenses, provide stereoscopic 3D by displaying two different images to the user's two eyes. The phone's built-in gyroscope, accelerometer, and compass suffice to track the orientation of the user's head, allowing for greater immersion as the user pans around the simulated scene. Although the Daydream system is optimized for a handful of new phones, most Android systems can be configured to work with it; in this thesis, we developed and tested on a last-generation Nexus 6P.

New to the Daydream system, however, is the handheld controller. The Daydream View is designed to be strapped to the user's head, unlike Cardboard, which requires the user to hold the device against their face. This leaves the user's hands free to interact with the scene through a custom controller. This controller, roughly the size of a pocketknife, is designed to pair with the phone via Bluetooth Low Energy, and provides many controls not previously available in Cardboard, as shown in Figure 2.2. There is the main button at the front of the controller, whose area also doubles as a small touchpad. Two other minor buttons are located on the top side: one customizable to developer applications, and one for returning to the home screen. A volume up/down rocker is also located along the right side of the controller. Together, these controls allow the user many new degrees of freedom to act in the virtual world being shown to them.

Our choice of the Daydream over other VR HMDs was motivated by several factors.

Figure 2.1: The Daydream View headset and controller.

Foremost, the handheld controller included with the Daydream system allows users to intuitively interact with the virtual scene around them; other smartphone-based HMDs such as Google Cardboard [19] or Samsung Gear VR [20] do not have similar controllers. Meanwhile, many high-end HMDs such as the Oculus Rift [21], Playstation VR [22], or HTC Vive [23] do come with handheld controllers, with high-precision tracking and features such as haptic feedback not available in the Daydream controller. However, these HMDs require powerful external systems to run: a desktop PC in the case of the Oculus Rift and HTC Vive, and the Playstation 4 for the Playstation VR. The cost of the external system plus the headset itself can run 2 to 3 times the cost of the Daydream plus phone. Additionally, the Daydream headset incorporates a cloth material into its chassis rather than the more common plastic, allowing prolonged VR sessions due to its lighter weight and softer fabric. Thus, we decided on the Daydream as an affordable and comfortable option that still supports controller-based interaction.

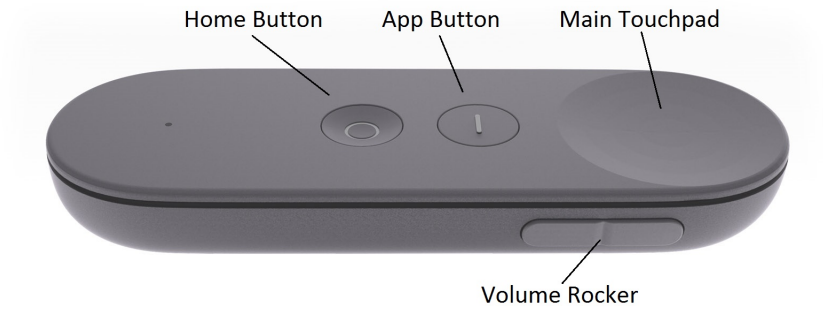


Figure 2.2: Various controls available on the Daydream controller.

Figure 2.3: An operator wearing the VIP Lab's backpack system.

2.2 Backpack-based 3D Capture System

Over the past decade, the VIP lab at U.C. Berkeley has developed a backpack-based 3D reality capture system [12, 13, 14] that is now being brought to market by Indoor Reality [24]. This backpack acquisition system allows for rapid capture of the interior of a building. It is worn by a human operator who walks around a floor, capturing 360-degree visual panoramas using 5 cameras, and depth using LiDAR. Together, the captured images and geometry information are processed [11, 15] and displayed in a web-based viewer, which allows users to explore the panorama in an interface similar to Google's Street View. This viewer is built using Three.js, a Javascript wrapper around WebGL, the standard web library for 3D graphics. There are four key features of the web viewer to highlight:

- The 360-degree panoramas are each converted into 6 faces of a cube, which are arranged in the scene to surround the virtual camera at a distance. This provides the visual illusion of being at the location where the backpack captured the image.
- The geometrical information from the LiDAR capture is converted into a mesh reconstruction of the building interior. From this mesh, we generate depth and normal information. Then, when the user's mouse moves across the scene, we render a small rectangle ("pancake") at the user's cursor location, at an appropriate distance and angle. On a desktop, the user can then rotate their viewpoint by dragging this pancake across the web viewer.
- The user can also select the distance measurement tool, which allows them to select two points in the space of the web viewer. The app then calculates the distance between the selected locations, and renders it as a persistent arrow in 3D space along with the measured distance.
- Along the floor of the scene, blue circles ("bubbles") are rendered, showing all the locations that a 360-degree panorama was captured at. By clicking on one of these bubbles, the user can switch over to that panorama. Repeating this process allows the user to get different views of the entire building.

2.3 Android and Web Hybrid

To integrate functionality from the Indoor Reality web viewer into the Daydream VR system, we elected to build out a hybrid app between Java and Javascript. Android provides a Java API for programmatic access to the Daydream system, but the existing business logic of the web viewer is written in Javascript, an unrelated programming language despite any naming similarities. We started with a thin native Android client written in Java, in order to capture the state of the controller paired via Bluetooth. In Java, the controller recalculates its internal state at a high frequency, detecting its orientation and status of touchpad and

various buttons. We needed to convert this controller state into Javascript, so that we could trigger web viewer logic based on the user's interactions with the controller. This would allow us to interoperate with the existing Javascript data and logic to construct the virtual scene. We also needed to modify the web viewer so that it would render two frames at offset camera locations, one on each half of the phone, to provide the stereoscopic 3D effect.

3 | Building the Viewer

Having settled on a hybrid approach, we began the process of implementing it. To our knowledge, all other existing Daydream apps are built either in Java, with native OpenGL bindings, or with a 3D game engine such as Unity [25]. In this section, we describe the process and techniques we used to develop the first web-based Daydream VR application.

3.1 Prototyping

To evaluate the feasibility of building a Daydream VR app with the features of Indoor Reality's web viewer, we began by prototyping a web app. We wished to control the amount of complexity being introduced at a time, as the full Indoor Reality web viewer contains a great deal of functionality designed for the optimizing the desktop web experience, such as bird's eye view overlays of the building floorplan and annotation of features through keyboard input. Among existing web technologies, the one most suitable for accomplishing our goals was WebVR [26]. WebVR is a newly-emerging standard that allows web clients to provide the same VR experience that native apps provide, with specifications for VR presentation, head orientation tracking, and controller input.

We started with a native Android application which occupies the entire screen, border to border, with an Android component that is intended to display web content, a WebView. Although Android WebViews do not yet support the full WebVR specification, a polyfill is available to bridge the APIs not yet available [27]. The polyfill, along with Three.js support and some UI widgets, are bundled together in a framework known as WebVR Boilerplate [28]. After enabling Javascript and Document Object Model (DOM) storage for the Android client WebView, we were able to successfully connect the Android web client to a local server hosting the prototype web app.

In order to test out the features that the Indoor Reality web viewer depended, we added two features to the prototype web app. First, we rendered a cube textured with a panoramic image around the virtual camera in the Three.js scene. Since the virtual camera serves

Figure 3.1: The final system in Fullscreen Mode. The green circle in the middle is the pancake, located where the user is pointing the controller.

as a starting point for the virtual left and right eyes, this effectively placed the user into the center of the simulated scene. Second, we instantiated a long rod in the scene, which was a virtual version of the Daydream controller. By reading the controller state in Java, forming an appropriate Javascript function call as a string, and then invoking that function against the prototype web app, we could pass along the current state of the controller. In the prototype code, we would align the rod object to the orientation of the controller, provided as a quaternion to protect against gimbal lock. Then, the WebVR framework would render the rod object from two different angles, and thus the rod would appear to the user in 3D, in the same orientation as the Daydream controller they were holding.

3.2 Final System

With the web app prototype demonstrating the ability to access the required functionality from the Daydream controller, we began in earnest developing a full-fledged VR app with the capabilities of the Indoor Reality viewer. At a high level, clients using Three.js must specify the geometry and material for various 3D objects in a scene, and then specify a particular camera from which to render those objects. Therefore the first objective in implementing VR was to replace the existing render loop of the Indoor Reality viewer with the WebVR loop to render what the two eyes would see. By combining the screen parameters of the phone, read from a repository of measured dimensions, with the lens locations of the VR headset, standardized across all Cardboard and Daydream headsets, we place two virtual cameras

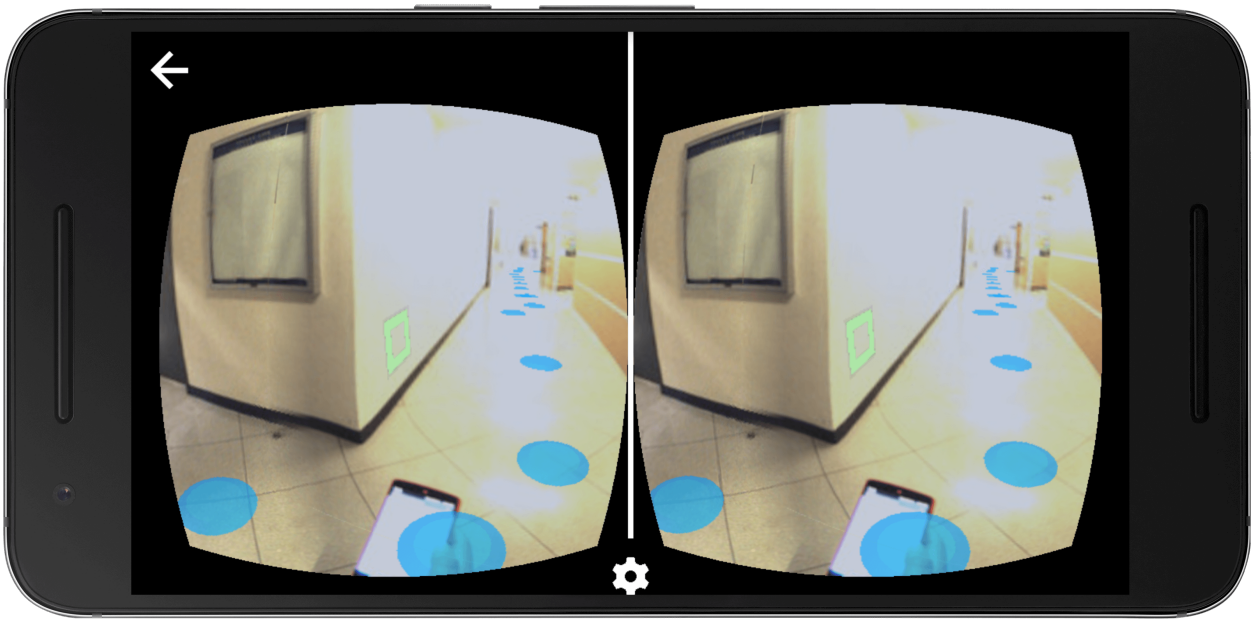


Figure 3.2: Entering VR Mode causes the screen to be rendered for each eye. Note that the pancake (now rectangular) is slightly offset between the left and right images, which provides the stereoscopic effect.

into the scene and show the results side by side, as shown in Figure 3.2. The accelerometer, gyroscope, and compass data are also fused to provide the orientation of the user's head, which is translated into the orientation of the virtual cameras.

Next, our objective was to translate the Daydream controller orientation into its VR analogue. The input device that the Indoor Reality desktop application expects is a mouse, with a 2D screen coordinate to translate into a 3D ray. This 3D ray is then raycast and used to determine the corresponding point on the 3D mesh that the mouse is pointing at, and thus the scale and orientation of the pancake cursor. We also determine whether the user is clicking on a bubble by the same raycasting method. Now that we have a Daydream controller for input, we can skip the step of converting the 2D mouse coordinates, and directly use the orientation of the Daydream controller as the direction of the ray to cast. The starting position of the ray remains the same: the position of the camera. By replacing all 2D mouse inputs with 3D controller orientations, we are able to render the target of the controller as a pancake as shown in Figure 3.1.

Finally, we wished to map the various Daydream controller inputs to the functionality of the Indoor Reality viewer. As the Daydream controller only exposes a polling API at 60 FPS, we began by keeping track of the previous and current state of the controller at every update on the native Android client. Then, we could detect a transition from a button being depressed to released as a "click event" for that button. Depending on which button was clicked, we would fire an appropriate Javascript payload into the WebView, which would be interpreted by the VR app and executed as some action.

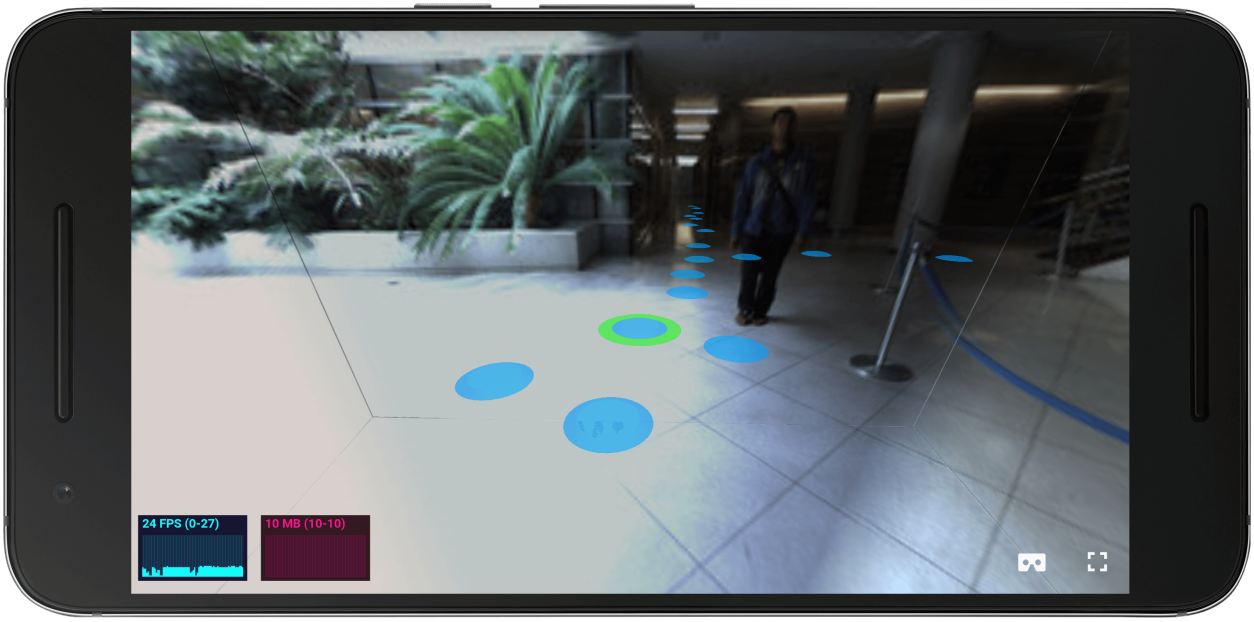


Figure 3.3: Hovering the pancake cursor over a bubble. The user can choose to navigate to that location by clicking on the main touchpad of the controller.

First, we assigned the main button on the Daydream controller as navigation between different 360-degree panorama locations. When the user points the controller to a bubble on the ground and clicks the main button, we overwrite the current cubemap images and geometrical information with that of the destination bubble, providing an illusion of point-to-point teleportation in the building scene. This allows users to quickly traverse large distances without prolonged, sickness-inducing motion. Figure 3.3 provides an example of user navigation between panoramas.

Next, we assigned the standalone app button to be used for distance measurement. In the process, we also simplified the distance measurement workflow for users. In the Indoor Reality desktop viewer, the user needed to make 3 mouse clicks: 1) select the distance measurement tool, 2) select the start point, and 3) select the end point. In the VR viewer, however, the user can skip step 1), and simply select the start and end points to measure. The user is then presented with a 3D arrow between the endpoints, and an accompanying tag denoting the real-life distance between them in meters, as shown in Figure 3.4.

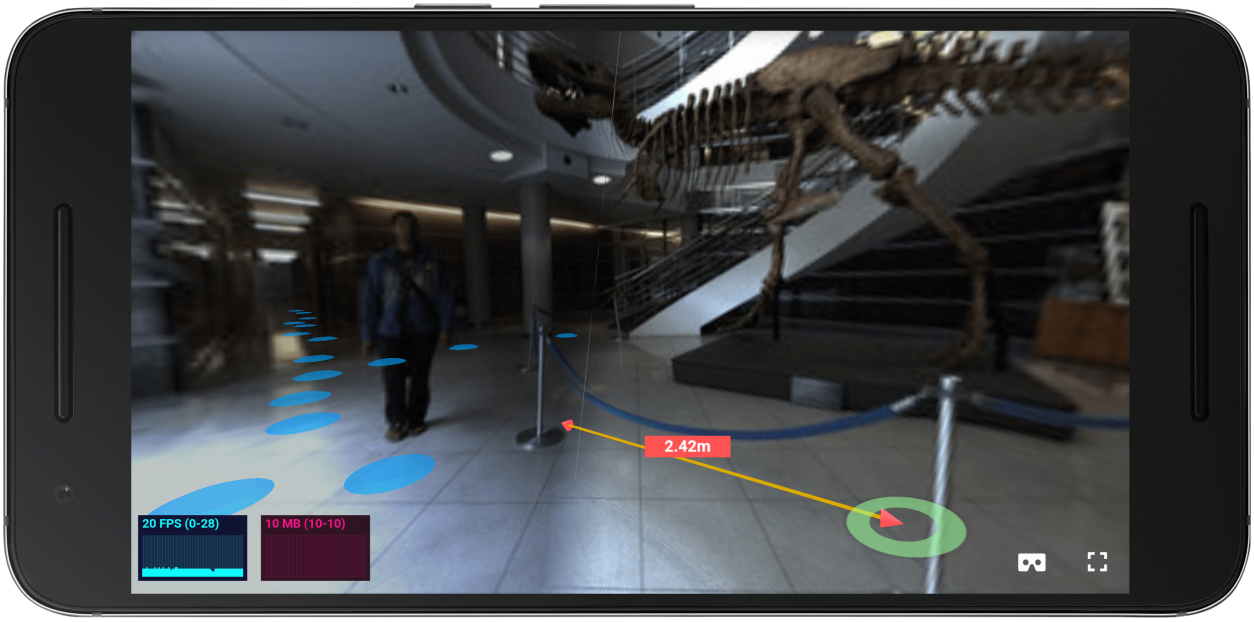


Figure 3.4: Distance measurement in progress. The user clicked the app button while the pancake was at the far post; now the other end of the arrow follows the pancake. Clicking again will finish the measurement.

4 Performance

A major consideration of any VR application is to have smooth user-perceived performance. This is important for the visual consistency of a scene, as well as for reducing the possibility of experiencing motion sickness. Whereas other VR systems such as the Oculus Rift, HTC Vive, and Playstation VR can offload the rendering computations to external desktops or gaming consoles with powerful processing units, the Daydream headset is limited to the computational resources of the smartphone placed within. Although smartphone processors have greatly improved in performance in the last few years, there remains a wide gulf between them and desktop processors, which translates into a disparity in performance. Moreover, as a web-based application, we would incur additional overhead relative to native VR apps. As a result, the steps we took to optimize the performance of the Indoor Reality viewer were paramount to providing a good user experience.

4.1 Monitoring and Profiling

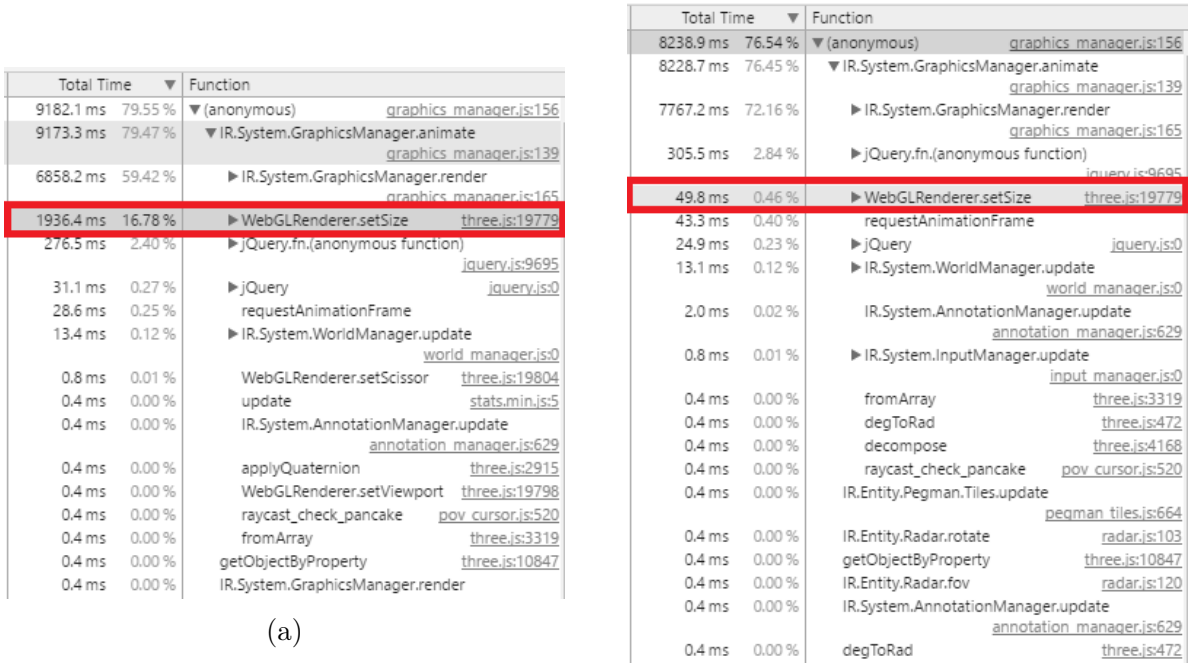
The core metric we used to evaluate performance of our VR app was the number of frames rendered and displayed on a second by second basis, also known as frames per second (FPS). A minimum of 10 to 12 FPS is necessary to provide the user a perception of motion; most films and TV shows are recorded at 24 FPS, and traditional 2D Android apps target 60 FPS to provide a smooth user experience [29]. Though high-end VR systems like the Oculus Rift, HTC Vive, and Playstation VR can support a refresh rate of 90 to 120 hz [21, 23, 22], the Daydream system is limited by the refresh rate of the supplied phone – in our case, 60 hz. Correspondingly, we set our performance target at the maximum capacity of 60 FPS.

To benchmark the performance of the app, we needed to get a view into the FPS at any point in time, and also track its changes as a result of user input and operations. Here, we used stats.js [30], a dashboard written by the creator of Three.js for monitoring the performance of Javascript. This dashboard provides a real-time graph of frames rendered over time, allowing us to immediately detect the latency changes caused by new pieces of code. From this, we saw that the initial implementation of the app rendered at a dismal 5 to 7 FPS, which corresponded to low user immersion and even possible nausea.

Although stats.js provides an overview of current performance, it does not detail where the CPU cycles are being spent. For this, we used the Chrome browser's Javascript profiler, which allows us to record the Javascript function calls being made in between two points in time, and display the cumulative CPU time spent within each function. The WebView on which our Android client displays the VR app is an extension of the Chrome browser, and thus we can hook into the function calls being made through the Chrome remote debugging tool [31]. This allows us to look for bottlenecks in our Javascript rendering path and optimize them, increasing the overall FPS performance of our app. For example, profiling quickly exposed a bug in the WebVR Boilerplate code that was causing the Three.js render function to be called twice in every instance of the render loop; each of which would occupy 40% of CPU time. Eliminating one of the render calls thus brought a 66% improvement in rendering speed and thus FPS. Another usage of CPU profiling was to expose performance differences between full-screen mode and VR mode, and identify bottlenecks in the latter. Figure 4.1 demonstrates our usage of this method to identify a discrepancy of CPU time between the modes. Because the size of the WebGL canvas changes between Fullscreen and VR Mode, we discovered that the application would waste processing power by repeatedly resizing the renderer. By modifying the code to resize the renderer exactly once, we were able to cut down CPU usage by roughly 16%.

4.2 Raycasting

Another performance bottleneck that we optimized on was the frequency of raycasting the controller orientation. This is particularly important because the test for intersection



Total Time		Function	
9182.1 ms	79.55 %	▼(anonymous)	graphics_manager.js:156
9173.3 ms	79.47 %	▼IR.System.GraphicsManager.animate	graphics_manager.js:139
6858.2 ms	59.42 %	▶IR.System.GraphicsManager.render	graphics_manager.js:165
1936.4 ms	16.78 %	▶WebGLRenderer.setSize	three.js:19779
276.5 ms	2.40 %	▶jQuery.fn.(anonymous function)	jquery.js:9695
31.1 ms	0.27 %	▶jQuery	jquery.js:0
28.6 ms	0.25 %	requestAnimationFrame	
13.4 ms	0.12 %	▶IR.System.WorldManager.update	world_manager.js:0
0.8 ms	0.01 %	WebGLRenderer.setScissor	three.js:19804
0.4 ms	0.00 %	update	stats.min.js:5
0.4 ms	0.00 %	IR.System.AnnotationManager.update	annotation_manager.js:629
0.4 ms	0.00 %	applyQuaternion	three.js:2915
0.4 ms	0.00 %	WebGLRenderer.setViewport	three.js:19798
0.4 ms	0.00 %	raycast_check_pancake	pov_cursor.js:520
0.4 ms	0.00 %	fromArray	three.js:3319
0.4 ms	0.00 %	getObjectByProperty	three.js:10847
0.4 ms	0.00 %	IR.System.GraphicsManager.render	

(a)

Total Time		Function	
8238.9 ms	76.54 %	▼(anonymous)	graphics_manager.js:156
8228.7 ms	76.45 %	▼IR.System.GraphicsManager.animate	graphics_manager.js:139
7767.2 ms	72.16 %	▶IR.System.GraphicsManager.render	graphics_manager.js:165
305.5 ms	2.84 %	▶jQuery.fn.(anonymous function)	jquery.js:9695
49.8 ms	0.46 %	▶WebGLRenderer.setSize	three.js:19779
43.3 ms	0.40 %	requestAnimationFrame	
24.9 ms	0.23 %	▶jQuery	jquery.js:0
13.1 ms	0.12 %	▶IR.System.WorldManager.update	world_manager.js:0
2.0 ms	0.02 %	IR.System.AnnotationManager.update	annotation_manager.js:629
0.8 ms	0.01 %	▶IR.System.InputManager.update	input_manager.js:0
0.4 ms	0.00 %	fromArray	three.js:3319
0.4 ms	0.00 %	degToRad	three.js:472
0.4 ms	0.00 %	decompose	three.js:4168
0.4 ms	0.00 %	raycast_check_pancake	pov_cursor.js:520
0.4 ms	0.00 %	IR.Entity.Pegman.Tiles.update	pegman_tiles.js:664
0.4 ms	0.00 %	IR.Entity.Radar.rotate	radar.js:103
0.4 ms	0.00 %	getObjectByProperty	three.js:10847
0.4 ms	0.00 %	IR.Entity.Radar.fov	radar.js:120
0.4 ms	0.00 %	IR.System.AnnotationManager.update	annotation_manager.js:629
0.4 ms	0.00 %	degToRad	three.js:472

(b)

Figure 4.1: Profiling revealed a discrepancy between the two modes; 16.78% of CPU was being consumed by spurious calls to `WebGLRenderer.setSize()`. (a) CPU profile trace in VR Mode. (b) CPU profile trace in Fullscreen Mode.

with the bubbles in the scene is a CPU-intensive operation. Operations that take up many CPU cycles will reduce the amount of resources for rendering, and thus lower the VR app's perceived performance. Originally, we raycasted from the controller orientation every time the controller position was updated in the Android Java code. Because Android client runs at 60 FPS, this meant that we were attempting to raycast at the same frequency – 60 times a second, or every 16.67 milliseconds. The consequence was that raycasting the controller was taking up about 30% of the phone's CPU time, as profiled by Chrome Remote Debugger. Moreover, many of these raycasts were unnecessary. The VR app itself would only render at around 15 FPS, meaning that 3 out of every 4 raycasts were discarded without ever being displayed to the user. The first improvement in this area was to tie the raycasting operation to the render loop, rather than the Android client. This meant that we were only raycasting on demand, reducing raycasting operations to 5% of CPU time. We were able to further improve this by configuring the VR app to only raycast the controller orientation while the controller touchpad is actively being touched. As a result, the user now has the option of improving performance by simply not touching the touchpad, which then skips the pancake render and raycast intersection test.

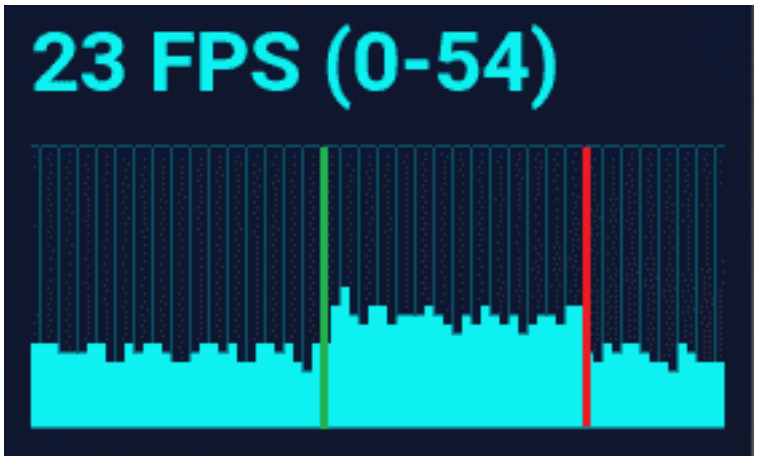


Figure 4.2: stats.js chart of framerate over time. The green line is when the tile cubemap was removed; the red line was when it was added back.

4.3 Simplifying Textures

Other performance gains were found by simplifying the cubemap being displayed to the user. The desktop Indoor Reality viewer has access to plenty of processing power to display high resolution images, and would prefer to incrementally load them so as to only consume network resources on demand. This led to the development of a tiling system which splits each face of the cubemap into an 8×8 grid, where grid tiles are only requested when they are viewed by the user. However, checking the user's field of view to determine which grid tile to load comes at a performance penalty. As an operation that is called within the main render loop, it consistently would take up 10% of CPU time, as the check would be called even after all tiles were loaded. Removing the incremental tile checking functionality allows us to bypass that CPU load.

More significant gains in performance were realized by altogether removing the tiled cubemap geometry from the scene. The tile cubemap led to an allocation of $8 \times 8 \times 6 = 384$ meshes to be rendered and intersected against. By simplifying the cubemap to be composed of exactly 6 meshes instead, one for each face, we reduce the costly calls to draw and check for ray intersection on the cubemap geometry objects. Figure 4.2 demonstrates the significant improvement in rendering speed as a result of this change, doubling the average framerate from roughly 20 to 40 FPS. In order to maintain the user perception of image resolution and quality in accordance with the reduction of cubemap complexity, we wrote a script to merge the cubemap tile textures into a single texture per cube face. This improvement to rendering time by means of cubemap simplification does come at the cost of increased texture load time, especially over slower networks. However, appropriate choices of image compression and resolution allow us some degree of control over how much load time is acceptable in the initial load and subsequent scene transitions.

5 Future Work

Our VR app correctly implements all the essential features of the Indoor Reality web viewer, but more work can be done to improve the overall experience. Foremost, additional performance improvements could further reduce the latency that is still perceptible. As the majority of CPU time is now being consumed by the Three.js render function, we would need to dive deeply into the current Indoor Reality desktop viewer and look to cut down on our object, geometry, material, and shader allocations, reusing instead of reallocating wherever possible. Although we have achieved a base frame rate of 50 FPS, any spikes in latency caused by expensive temporary operations can pull the user out of immersion, and in the worst case, induce motion sickness.

There are also a few remaining features in the viewer that may be mapped to unused Daydream controller inputs. For example, we could conceivably use the touchpad to calibrate the level of zoom that the camera is set to, or set the volume up/down buttons to toggle between different kinds of distance measurement, such as major- or minor-axis aligned measurement. Additionally, it may be possible to use other features of the phone itself to add capabilities to the VR app. Most promising would be to use the microphone to allow tool selection and text input in an natural user interface not constrained by the number of physical buttons that a user can manipulate.

Finally, we could extend the current viewer to support other VR platforms. As our viewer is based on standard web technologies, it would be possible to configure it to run on the Oculus Rift, HTC Vive, Samsung Gear VR, or almost any VR platform that supports WebGL [26]. Since the physical interface presented to the user differs from platform to platform, however, some amount of design would be required to determine how the user would manipulate the controls in each system.

Bibliography

[1] Soojeong Yoo and Callum Parker. "Controller-less Interaction Methods for Google Cardboard". In: *Proceedings of the 3rd ACM Symposium on Spatial User Interaction*. ACM. 2015, pp. 127–127.

[2] Alexander Badju and David Lundberg. "Shopping Using Gesture-Driven Interaction". In: (2015).

[3] Chris Gordon, Frank Boukamp, Daniel Huber, Edward Latimer, Kuhn Park, and Burcu Akinci. "Combining reality capture technologies for construction defect detection: a case study". In: *EIA9: E-Activities and Intelligent Support in Design and the Built Environment, 9th EuropIA International Conference*. 2003, pp. 99–108.

[4] Mikael Johansson. "Efficient stereoscopic rendering of building information models (BIM)". In: *Journal of Computer Graphics Techniques (JCGT)* 5.3 (2016).

[5] Google. *Google Street View*. 2017. URL: `https://www.google.com/streetview/understand/`.

[6] Microsoft. *Bing Streetside*. 2017. URL: `https://www.microsoft.com/maps/streetside.aspx`.

[7] Apple. *Apple Maps*. 2017. URL: `https://maps.apple.com/vehicles`.

[8] Applanix. *Trimble Indoor Mobile Mapping System*. 2017. URL: `http://www.applanix.com/products/timms-indoor-mapping.html`.

[9] NavVis. *M3 Trolley*. 2017. URL: `http://www.navvis.com/products/m3-trolley/)`..

[10] Leica. *Pegasus Backpack*. 2017. URL: `http://leica-geosystems.com/products/mobile-sensor-platforms/capture-platforms/leica-pegasus-backpack`.

[11] Christian Frueh, Siddharth Jain, and Avideh Zakhor. "Data processing algorithms for generating textured 3D building facade meshes from laser scans and camera images". In: *International Journal of Computer Vision* 61.2 (2005), pp. 159–184.

[12] Timothy Liu, Matthew Carlberg, George Chen, Jacky Chen, John Kua, and Avideh Zakhor. "Indoor localization and visualization using a human-operated backpack system". In: *Indoor Positioning and Indoor Navigation (IPIN), 2010 International Conference on*. IEEE. 2010, pp. 1–10.

[13] George Chen, John Kua, Stephen Shum, Nikhil Naikal, Matthew Carlberg, and Avideh Zakhor. "Indoor localization algorithms for a human-operated backpack system". In: *3D Data Processing, Visualization, and Transmission*. Citeseer. 2010.

[14] Nicholas Corso and Avideh Zakhor. "Indoor localization algorithms for an ambulatory human operated 3d mobile mapping system". In: *Remote Sensing* 5.12 (2013), pp. 6611–6646.

[15] Eric Turner et al. "Watertight Floor Plans Generated from Laser Range Data". In: (2013).

[16] Roy A Ruddle. "The effect of environment characteristics and user interaction on levels of virtual environment sickness". In: *Virtual Reality, 2004. Proceedings. IEEE*. IEEE. 2004, pp. 141–285.

[17] S Jennings, G Craig, L Reid, and R Kruk. "The effect of visual system time delay on helicopter control". In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*. Vol. 44. 13. SAGE Publications. 2000, pp. 69–72.

[18] Google. *Daydream View*. 2017. URL: `https://vr.google.com/daydream/headset/`.

[19] Google. *Google Cardboard*. 2017. URL: `https://vr.google.com/cardboard/`.

[20] Samsung. *Samsung Gear VR*. 2017. URL: `http://www.samsung.com/global/galaxy/gear-vr/`.

[21] Oculus. *Oculus Rift*. 2017. URL: `https://www3.oculus.com/en-us/rift/`.

[22] Sony. *PlayStation VR*. 2017. URL: `https://www.playstation.com/en-us/explore/playstation-vr/`.

[23] HTC. *HTC Vive*. 2017. URL: `https://www.vive.com/us/`.

[24] Avideh Zakhor. *Indoor Reality - Mapping Indoors One Step at a Time*. 2017. URL: `http://indoorreality.com/`.

[25] Unity. *Unity - Google Daydream*. 2017. URL: `https://unity3d.com/partners/google/daydream`.

[26] *WebVR*. 2017. URL: `https://webvr.info/`.

[27] Google. *WebVR Polyfill*. Github. 2017. URL: `https://github.com/googlevr/webvr-polyfill`.

[28] Boris Smus. *WebVR Boilerplate*. Github. 2017. URL: `https://github.com/borismus/webvr-boilerplate`.

[29] Colt McAnlis. *Android Performance Patterns: Why 60fps?* Youtube. 2015. URL: `https://www.youtube.com/watch?v=CaMTIgxCSqU`.

[30] Recardo Cabello. *Stats.js*. Github. 2017. URL: `https://github.com/mrdoob/stats.js/`.

[31] Google. *Chrome Devtools: Remote Debugging*. 2017. URL: `https://developers.google.com/web/tools/chrome-devtools/remote-debugging/`.