Adding Color to 3D Scene Reconstruction for Augmented Reality Headset Experiences Using an iPhone

Philip Turner

June 3, 2021

Abstract

In October 2020, Apple released the iPhone 12 Pro, which had a LiDAR scanner that enabled it to reconstruct the 3D shape of its surrounding scene. Coupling the Google Cardboard VR viewer with scene reconstruction allows the real world to be rendered in VR, replicating the experience of using an AR headset such as Hololens. In order to enhance this AR headset experience, I created software that mapped color in a video stream to areas on a reconstructed scene mesh, and retained that color when the mesh expanded and changed shape as the LiDAR scanner gathered new data. By adding color to scene reconstruction, the experience became more realistic, allowing the user to view the color of their surroundings in their peripheral vision and in areas occluded from the camera's view. This software runs on an iPhone, providing users with an accessible way to experience an AR headset.

Table of Contents

Introduction

Augmented reality (AR) headsets are currently not affordable. The most widespread AR headset, Microsoft Hololens 2, costs $3,500 (Bohn, 2019). The high price tag makes AR headset experiences inaccessible to the general public, limiting the use of headsets to the corporate enterprise environment. I wanted to give the general public an affordable way to try out an AR headset, so I created software that replicates one using an iPhone and the $25 Google Cardboard VR viewer.

In October 2020, I created software similar to StarBoard, a leaked Apple developer tool (Diaz, 2019). StarBoard allows Apple employees to test out apps for an AR headset using an iPhone. By displaying real-time video of the user's surroundings in two separate views, they could simulate the stereoscopic rendering of an AR headset, in a handheld AR experience. My software passed these two views into the Google Cardboard VR viewer, fully replicating the experience of using an AR headset. This concept has been implemented in the "Passthrough" feature on Oculus VR headsets, which allows a user to navigate their environment without removing their headset (Hawthorne, 2021).

I realized it was possible to develop software that can do better than passing 2D video through to VR. In October 2020, Apple released the iPhone 12 Pro, which had a LiDAR scanner (Wilson, 2020). This hardware scanned the nearby scene up to five meters away, allowing the world's 3D shape to be reconstructed in the form of a triangle mesh. Apple intended for the mesh to be rendered on top of real-time video in a handheld AR experience, so they never added color to it.

After buying the iPhone 12 Pro Max in November, the 3D mesh generated by Apple's scene reconstruction algorithm was incorporated into my software. The purpose of using scene

reconstruction was to represent the real world as well as possible in VR. Reconstructing reality through the 3D mesh instead of 2D video allowed for depth perception, making the reconstructed world "pop out" in 3D, in the way that makes geometry in VR seem realistic.

Since the mesh did not have any color, I had to render it twice - once as a mesh of triangles and once as a wireframe mesh. This approach outlined the mesh's triangles so that the user could experience the depth perception effect. The triangles were colored based on their distance to the device's camera - an effect Apple first implemented in their sample code (Apple, 2020, 17:25).

In late January 2021, I began a project that would add color to this mesh. This project took two months and 450 hours of work, culminating on the last day of March.

Key Concepts

Before explaining the process of creating the color tracking algorithm, the reader must be familiar with several key concepts.

The first concept is serial versus parallel programming. Serial computations iterate over a massive set of tasks one at a time, while parallel computations process several tasks at the same time. Modern computers can run up to five billion operations per second - a rate called the clock speed. This rate does not scale directly with power consumption. After reaching one billion operations per second, doubling the clock speed more than doubles power consumption. Using a very high clock speed could cause a computer to overheat. An alternative approach to increasing processing power is to use a low clock speed while running several computations in parallel. A GPU (graphics processing unit) uses hundreds to thousands of low-frequency processing units. A CPU (central processing unit) uses a small amount of high-frequency units. While a CPU often has an order of magnitude less processing power than a GPU, it can consume just as much energy. Thus, it is preferable to run as many calculations as possible on a GPU.

Creating software for GPUs is a much more difficult process than creating software for CPUs, as it requires a developer with a lot of experience programming. In addition, it is more vulnerable to security flaws than CPU software because it requires low-level access to hardware to maximize performance. For example, WebGPU (a modern framework for accessing a GPU in a browser application) speeds up tasks that run on a GPU by giving the software more control over the hardware than WebGL (Malyshau, 2020). It currently is not secure enough for widespread use.

Not all calculations can be parallelized. In order to be parallelized, some aspect of a computation must be independent from all other computations in a workload. Compiling code

and parsing HTML to render web pages is extremely difficult, if not impossible, to parallelize. Working with large lists is possible to parallelize, but this approach requires a massive amount of creativity from the developer. In addition, extensive knowledge about the properties of the data in the lists is gathered to maximize performance. Rendering 3D graphics and processing images are the easiest tasks to parallelize.

Until recently, powerful GPUs were always on separate pieces of silicon than a CPU, and a CPU had to communicate with them through a wire. This communication was slow, so GPUs managed memory separate from a CPU's memory. This structure prevented CPUs and GPUs from operating on the same data and switching contexts quickly.

Integrated GPUs are placed on the same chip as the CPU, massively reducing the cost of communication between the CPU and GPU. They reduce latency from milliseconds to microseconds, allowing the CPU and GPU to use the same memory.

Apple's most recent mobile chips have extremely powerful integrated GPUs. For example, the A14 chip on the iPhone 12 is capable of one trillion single-precision floating point operations per second, which was only achievable by a laptop or desktop GPU until a few years ago.

The massive amount of processing power in the A14's GPU is necessary for processing the reconstructed world mesh to allow for color tracking. The mesh is stored in massive lists, and list operations are difficult, but possible to parallelize. Some of this processing runs more quickly on the CPU than the GPU, and using the same memory for both processors allows them to quickly read results from each other's calculations.

Another important concept is memory bandwidth. Most often, the time taken to complete a massively parallel calculation is not determined by the number of arithmetic calculations that

can be performed, but by the amount of data that can be transferred between main memory and the CPU or GPU each second. On the A14 chip used for this research, this limit was 42.8 billion bytes per second. Since several parts of the chip might access an area of memory at the same time, complex circuit logic is required to coordinate memory accesses, making each memory access take several clock cycles to complete. A way to alleviate bandwidth and latency bottlenecks is to store commonly referenced pieces of data in a small cache close to the CPU or GPU. This cache takes less clock cycles to access and can prevent memory reads from accessing main memory, conserving bandwidth.

Finally, one must understand two similar algorithms that use parallel processing on a GPU to accelerate tasks traditionally done on a CPU. The first is the "compact" algorithm, which shrinks a large list with several empty elements that must be removed (Fig. 1). The second is the "allocate" algorithm, which expands a list with several elements that must be duplicated (Fig. 2). This explanation only describes the "compact" algorithm, although both algorithms coordinate the copying of list elements in the same way.

The "compact" algorithm is quite complex, so it is best to first describe a simpler, but less efficient approach. This approach shrinks a list using a single CPU core, without any parallel processing. It uses a counter that represents both the size of the new list and the location of the next element to be added to the new list.

The CPU scans each element in the old list from beginning to end. Each time the CPU encounters a valid (non-empty) element, it increases the counter by one and adds the element to the new list at the position described by the counter. For example, if the new list does not have any elements yet, its counter is zero. After finding the first valid element, the counter increases to one, so the CPU copies that element to the first position in the new list. The CPU repeats this

process until all of the old list is scanned, and every valid element is copied to the new list. This process is extremely slow because it does not use parallel processing.

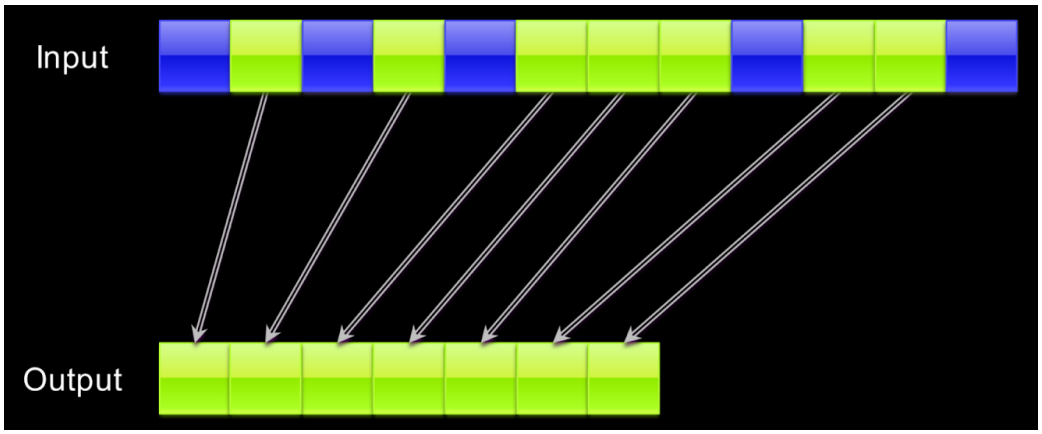

Figure 1. The "compact" algorithm removes blue elements from a list, creating a new list with only green elements. (Harris, 2010)

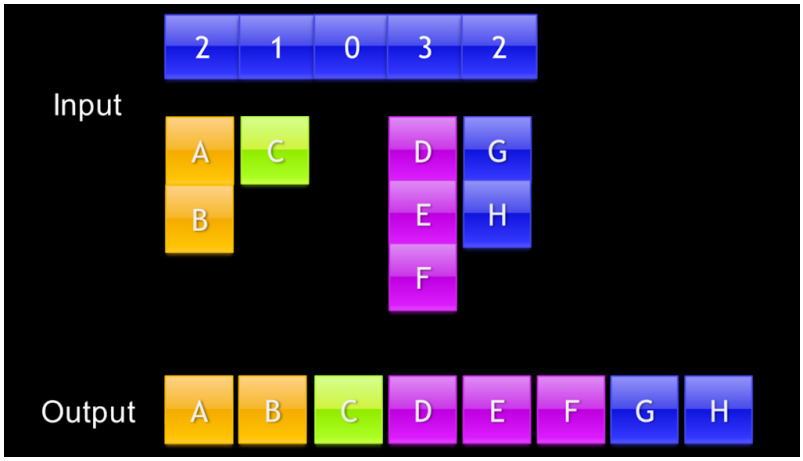

Figure 2. The "allocate" algorithm creates a new list that may be larger than the original list. This algorithm reserves a variable amount of memory in the new list for every element in the old list. (Harris, 2010)

A slightly better approach delays the copying of elements until a later stage, which occurs on the GPU. The largest bottleneck on single-threaded CPU operations is often the length of time

taken for memory reads and writes, not arithmetic calculations, so this approach would make scanning the list happen more quickly. After the CPU scans the list and finds the new positions of all valid elements, the GPU copies all valid elements from the old list to the new list. Since main memory can handle tens of reads or writes per clock cycle, this approach fully utilizes main memory bandwidth, while the previous approach does not.

The best approach does both the scanning and copying in parallel. The scanning starts out by splitting the list into small groups. For example, a list of 40,000 elements is split into 4,000 groups of 10, and the GPU finds the number of valid elements in each small group, entirely in parallel. Then, those 4,000 groups are divided into 1,000 groups, whose sizes are calculated by adding the sizes of smaller groups. Each successive stage uses 1/4 as many groups, until the GPU finds the size of the entire list.

Then, the GPU finds every group's offset, which means the position of its first valid element within the new list. The algorithm finds offsets of the largest groups first and works down to the smallest groups. In the first stage, which processes the largest grouping of elements, the first group starts at the first position in the new list. The second group's first element is placed after the first group's last element. If the first group has 1,000 valid elements and starts at position one, then the second group starts at position 1,001 in the new list. If the second group has 700 valid elements, then the third group starts at position 1,701. This addition continues until the offset of every large group is found.

The process then repeats for the next subdivision, where groups are 1/4 as large. Instead of using position one as the initial offset, it uses the offset of the larger group that encapsulates a set of smaller groups. For example, the number 1,701 is used as the starting point for the four smaller groups that make up the third large group. The GPU continues calculating offsets until it

reaches the smallest group size. Then, it copies elements from the old list to the new list. During copying, the algorithm switches to an approach similar to the serial algorithm. If an element in a small group is valid, it is copied to the new list and the group's offset increases by one, just like the counter in the serial approach.

The "compact" and "allocate" algorithms are powerful because they can parallelize many list-related tasks that would otherwise happen serially. For example, culling geometry is an operation that removes many elements from a large list and shrinks the list before sending it to a GPU for rendering. Culling geometry happens because in 3D graphics, a large amount of geometry is often outside of the user's frame of view, and processing that geometry wastes time and energy on a GPU. This geometry must be culled (prevented from being processed). Using the "compact" algorithm, one can shrink a large list of 3D objects to include only those inside the user's frame of view, and do so entirely on a GPU. This approach saves time and energy, as calculations done on a GPU use less energy than calculations done on a CPU.

Process

Late January

In order to add color to 3D scene reconstruction, an algorithm can not just assign color to triangles in the world mesh after each video frame. The mesh updates multiple times per second, and the algorithm needs to transfer color data across each mesh update. Not all color data can be directly transferred because not all areas in the new mesh line up with areas in the old mesh.

After storing the color of as many triangles in the mesh as possible, one can combine stored color with more accurate video color to maximize the quality of the scene's reconstructed color. Video would be used to color in triangles visible to the camera. Stored color would be used for triangles outside of the camera's frame of the view or obstructed from the camera's view by other triangles.

Before adding color to the scene mesh, one first needs to create it. Apple's scene reconstruction algorithm stores the scene mesh as a group of smaller submeshes, each of which has a different coordinate system. The vertices in each submesh must be transformed to one common coordinate system, then combined into one large mesh. When vertices fall at the boundaries of submeshes, they are duplicated, with a copy present in each submesh at the boundary. This is problematic, as transforming them to a common coordinate system leaves each duplicated vertex at a slightly different location, causing discontinuities in the mesh. This problem is solved by detecting extremely close vertices and merging references to them, which requires searching a massive list for groups of close vertices.

To maximize the chance that duplicate vertices are correctly identified, one needs to maximize the accuracy of vertex positions after transforming them between coordinate systems. To accomplish this, high-precision calculations are used during the transformation. The A14

chip's GPU can not process 64-bit (double-precision) floating point numbers, which are more precise than 32-bit (single-precision) floating point numbers. To circumvent this lack of capability, ideas were used from the work of Polok and Smrz (2016) to emulate double-precision calculations on the GPU.

Both eliminating duplicate vertices and matching nearby parts of the mesh before transferring color data required sorting vertices into volume sectors. The first method of sorting assigned a hash (semi-unique number) based on each vertex's location in a 3D grid and then sorted a list of references to vertices by comparing those hashes. At the end of the sort, nearby vertices would have the same hash and appear next to each other in the list, allowing nearby vertices to be found and tested. However, this sort had to happen eight times to capture all matches, and it did not allow for geometry in two different versions of the world mesh to be matched.

The sort had to be repeated because dividing space into sectors does not capture all groups of potentially duplicated vertices. For example, a pair of duplicated vertices can fall on opposite sides of a sector boundary, meaning the pair will not be identified. In order to make sure this pair is identified, one must translate all sectors by a small amount so that the boundary does not fall between the pair, and repeat the sort (Fig. 3).

The sort repeats three additional times in 2D, and seven times in 3D (Fig. 4). The sort repeats several times because one translation can not capture all vertex matches. For example, a horizontal translation does not capture vertex pairs separated by a vertical boundary. In addition, a rare configuration of nearby vertices requires a shift in multiple directions (Fig. 5).

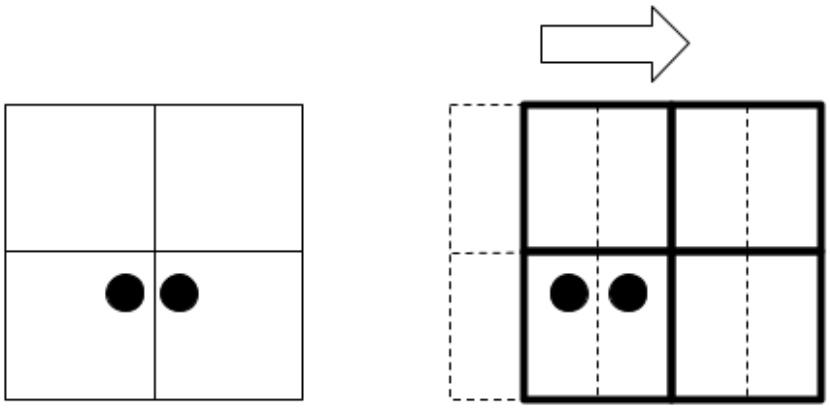

Figure 3. Two nearby vertices do not fall into the same sector in the initial subdivision. When the sectors are shifted, the vertices are grouped into the same sector, allowing for their closeness to be tested.

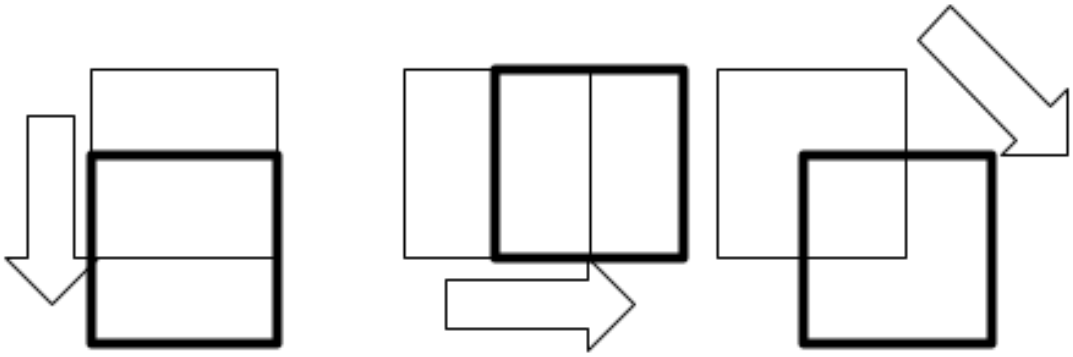

Figure 4. Each of the three translations required to capture all nearby vertex pairs in 2D. The 3D analogue results in seven unique translations.

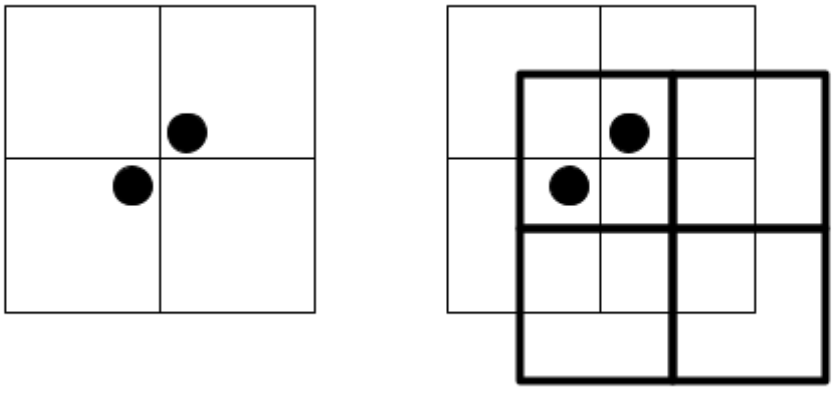

Figure 5. The nearby vertices depicted here do not fall into the same sector with only horizontal or only vertical translations, but they fall into the same sector with a shift in both directions.

The repetitiveness of the first approach at sorting vertices made it already eight times slower than a similar method that handled border conflicts. However, there was also another performance problem. This method was a merge sort, which has O(nlog(n)) complexity. This term means that as the size of the list grows, the duration of the sort grows at a slightly faster rate. A merge sort occurs in several stages. In the first stage, pairs of list elements are put in increasing order. In the second stage, groups of four are sorted, and the process repeats with increasing powers of two. It is named a "merge sort" because it merges smaller sorted lists into larger sorted lists. However, with each stage, the number of groups halves, making later stages less parallelizable.

Since the last stages work with a small number of groups, they either underutilize GPU parallelism or run more quickly on the CPU. For example, the number of vertices in the mesh is close to 48,000, and the A14's GPU can be thought of as 1024 processing units, which should not be confused with GPU arithmetic logic units (each of which represents two processing units). The first stage sorts pairs of vertices, so it runs 24,000 independent tasks. The second stage has 12,000 independent tasks. When the number of tasks falls below 1024, the sort can't fully utilize the GPU processing units. However, each stage requires the same number of calculations because each stage processes the entire list. As the number of tasks decreases, each task takes longer to complete. When the number of tasks reaches ten, the sort runs more quickly on the CPU, which has less processing units than the GPU, but runs calculations at a higher frequency.

With the merge sort approach, the sort took an extremely long amount of time. The first few stages made full use of the GPU processing power and took hundreds of microseconds. The next few stages, which had less than 1024 tasks, still took just as much time. This behavior

occurred because the most severe performance bottleneck was not the number of computations, but rather the amount of memory bandwidth.

On the A14 chip, 42.8 billion bytes of data can be transferred across memory each second. In contrast, the GPU can execute one trillion four-byte arithmetic operations per second. The hashes used to identify grid sectors were four bytes long, which means the GPU could theoretically do one trillion comparisons a second (although the actual number was much less, as the GPU also had to do multiple calculations to prepare for each comparison). However, only 10.7 billion hashes could be transferred over memory per second. Each comparison involved at least one read and one write, meaning that the bandwidth limited the speed of comparisons to around 5.4 billion per second.

This constraint meant that even when the GPU's processing power was underutilized, stages still took the same amount of time. In fact, the duration of each stage only started to double when the number of GPU processing units used reached 50. The stages that ran on the GPU took an elapsed time of seven milliseconds. Repeating that seven additional times yields 56 milliseconds - a massive amount of time and energy for any operation that happens frequently.

Using that much energy would cause the phone to overheat quickly, causing the system to throttle frame rate less than 10 minutes into every AR session. In addition, the merge sort did not produce an organized volume hierarchy, so the GPU could not traverse 3D space or locate similar sectors in different meshes. So, an entirely different approach was used to sort the world mesh's vertices.

Early - Mid February

A new algorithm sorted vertices into volume sectors and subdivided each sector into smaller ones, repeating this process until each sector contained a small number of vertices. Sectors included vertices slightly outside of their boundaries, solving the problem of sector boundaries preventing vertex matches. This solution meant that a vertex could be assigned to multiple sectors. The data about which vertices fell into which sectors was stored in one list of vertex references and in numbers describing where each sector's references started and ended.

The first few stages of the sort generated a sparse octree. An octree divides 3D sectors into eight smaller ones, and a sparse octree does not store data for sectors that are not occupied. The algorithm started by dividing the world into eight octants, based around the origin of the world space coordinate system. These octants are the 3D analogue to the four quadrants of the XY plane.

Each octant was assigned a size based on the farthest distance any of its vertices deviated from the world's origin. Octants most often had different sizes, but the sectors participating in each round of subdivision had to have the same size. So, a second stage broke down large octants into smaller sectors, stopping when the sectors were two meters wide. In a room with the dimensions of 3m x 3m x 2.5m, this resulted in an average of 16 sectors.

The volume hierarchy was stored in a data structure based on a linked list. A linked list stores data as a chain of references, where each reference points to a new link in the chain, and the final link does not point to anything. The sparse octree's volume hierarchy was stored in a data structure where each link pointed to up to eight additional links. It could be expanded into an array of sectors along with their paths inside the structure and could be reconstructed from

that array. This hierarchy allowed the algorithm to subdivide only a selected group of sectors and restructure the octree after each subdivision.

The final sector size of two meters was chosen because it produced a small number of sectors, and every sector contained less than 65,000 vertices. This size allowed the GPU to use 16-bit integers instead of 32-bit integers for several calculations, completing twice as many calculations per clock cycle while halving bandwidth consumption, without a serious risk of an integer overflow. An integer overflow occurs when a number exceeds the maximum size its data type can hold, and wraps around to zero or a negative number. In the case of unsigned 16-bit integers, that limit is $2^{16} - 1$, or 65535.

This stage used the "allocate" algorithm several times, as sorting vertices into new sections of an octree expands the list of vertex references. Since a vertex might fall near the boundary of multiple sectors, a reference to it can be duplicated. This means the number of references in the list increases each stage. Before each spatial subdivision, one has no way of knowing how large the final list will be. So, the CPU intervenes after the first half of the "allocate" algorithm, where the size of the new list is known. After the CPU ensures enough memory is allocated to fit the final list, the GPU completes the list expansion.

On most desktop GPUs, switching contexts between the CPU and GPU takes a long time. On integrated GPUs such as the A14's GPU and the Intel UHD Graphics 630, this takes around 20 microseconds, which is fast, but still is not negligible. So, interactions between the CPU and GPU were optimized to minimize time spent waiting. For example, the calculation of offsets in the "allocate" algorithm could proceed while the CPU ensured the final list was large enough to fit all vertex references. After the CPU and GPU completed their work, the GPU could fill the final list. In addition, the task of sorting the world mesh happened on a different thread than the

one that controlled the user interface, allowing the sort to take as much time as needed without dropping the frame rate.

If the sort subdivided space using an octree for every stage, it would take several stages to reach sectors a few centimeters wide, and traversing the octree would take too many computations and memory accesses. So, the last stages of the sort jumped two orders of magnitude in two stages, using an entirely different method to subdivide space into a sparse volume hierarchy.

The first of these stages subdivided each two-meter sector into an 8x8x8 grid of 25 cm sectors. It incremented counters using atomic operations, which allow independent processing units to modify small areas of memory in a safe and deterministic way. Several processing units could increment a counter for each subsector and receive the counter's previous value, which represented a vertex's offset within the subsector. After this was done, each vertex was assigned its positions within the sectors it belonged to, while the GPU could find sectors that did not have any vertices. Then, the "compact" algorithm created a list of references to only the sectors that contained vertices. This process repeated, dividing the 25 cm sectors into 8x8x8 grids of 3.125 cm sectors.

Late February - Early March

At this point, the software could sort the mesh into sufficiently small sectors, where each sector had references to several neighboring vertices. Each vertex could also locate the sector whose bounds contained it and scan that sector's vertices, allowing it to compare itself to nearby vertices. Since the software could locate groups of close vertices, it could eliminate duplicate vertices.

However, to transfer color data between two different meshes, one needed to find pairs of similar vertices in two different spatial subdivisions. This process required storing both the new and old mesh's volume hierarchies. Then, two-meter sectors in the new mesh were matched to two-meter sectors in the old mesh with the same locations. This way, one could climb up the new mesh's hierarchy (from 3.125 cm to two meters), cross meshes using the links between two-meter sectors, and climb down the old mesh's hierarchy, finding sectors in the old mesh that corresponded to sectors in the new mesh.

In order to find an adequate number of pairs of close vertices between the old and new meshes, the tolerance (maximum distance for a pair to be considered a match) had to be extremely high. This caused the amount of overlap between adjacent sectors to be large, so most vertices were classified as belonging to more than one sector. The final value for tolerance when comparing vertices in the new and old meshes was 1.2 cm, and a 4 mm tolerance was used to find duplicate vertices in the same mesh. The 1.2 cm tolerance resulted in each vertex being included in an average of 5.4 adjacent sectors.

After removing duplicate vertices, a small number of triangles were degenerate, which meant one or more of their sides had zero length. Degenerate triangles were formed when two of a triangle's vertices were so close that they were considered duplicates and merged. These

triangles were removed using the "compact" algorithm, reducing the amount of computations required to process the mesh.

To transfer color data between meshes, matches between triangles, not just vertices, had to be found. The transferring of color data was delayed until as late as possible, to ensure that the transferred color was as recent as possible. However, links between triangles in the new and old mesh were generated immediately after the new mesh was sorted.

A triangle match was found when all three vertices in a new triangle were within the 1.2 cm tolerance of vertices in an old triangle. When multiple matches were found, the closest one was selected. Closeness was determined by adding the squared distances between vertices, with smaller values indicating better matches.

There were always triangles that did not have a match. This would make sense if new areas of the scene were being scanned, because some parts of the new mesh would not have any analogue in the old mesh. However, triangles still lacked matches when no new volume was scanned because Apple's scene reconstruction algorithm rearranged triangles and refined the mesh as it gathered more data. In addition, the removal of extremely close vertices altered the mesh. Without any mechanism to compensate for this, large portions of the new mesh would not have any color data. Later on, this problem was mitigated by transferring low-quality color data to triangles that did not have a match.

At this point, matching similar triangles only allowed simple color data to be transferred. Each triangle had only one color, which was determined by sampling the color of the triangle's center when it was visible to the camera. However, that did not provide enough detail to reconstruct the world's color to an acceptable quality. This is where high-resolution color tracking became necessary.

Mid March

In order to store high-resolution color data, each triangle was subdivided into a grid of texels (texture pixels), and each texel's color was tracked independently. Processing color data this way was costly, so only the texels that overlapped the triangle were stored in memory. Storing only the necessary texels was achieved by splitting the grid into columns. The size and location of each column of overlapping texels was stored, which was more efficient than storing the location of each individual texel. The triangle's longest side was used as the horizontal axis of the grid's coordinate system, causing every column of texels to start at the same vertical position (Fig. 6). This meant that each column's location could be described using only its horizontal position.

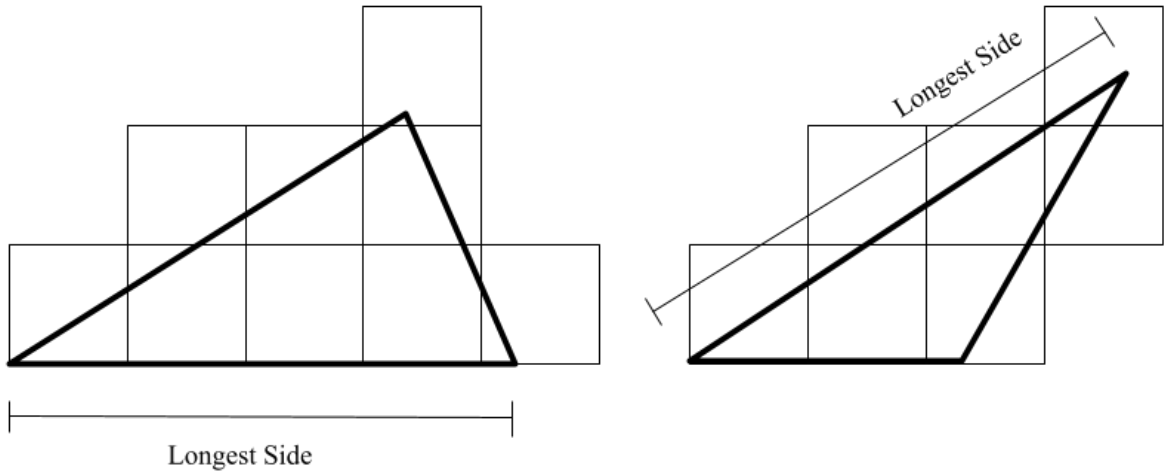

Figure 6. Left: a texel grid with the triangle's longest side used as the horizontal axis. Right: a texel grid where the triangle's longest side is not used as the horizontal axis, causing the rightmost column to start at a different height than the other columns.

Each texel was stored in a buffer - a large list where each element has the same size. However, each texel was located using its column, and columns had different sizes. To solve this, two buffers were used to store the texels. The first buffer stored the color data, while the second buffer stored the columns' locations in the first buffer. Each element in the first buffer described

one texel, while each element in the second buffer described one column. One could locate any texel by locating the column it belonged to and knowing the texel's position within that column.

When coloring in a pixel on the phone's screen using high resolution color, the pixel's color was calculated by averaging four adjacent texels. Although it was possible to locate these texels using their columns' positions within the buffer, doing so was too costly. So, a triangle's color data was copied to a massive texture when it was rendered to the screen. In the texture, each column was given the same number of texels, and each triangle was given the same number of columns, meaning each triangle's color data occupied the same amount of memory.

However, not all triangles needed the same amount of memory. For example, if the largest triangle needed 200 texels, every triangle would be given 200 texels to store its color data, even if it only needed a small fraction of that amount. To mitigate this problem, triangles were split into two different groups based on size. Small triangles were stored in a different texture than large triangles. Small triangles were each given an 8x8 area in their texture, although only the innermost 6x6 texels could be used to store color data. Large triangles were given a 16x16 texel area, but only the innermost 14x14 texels could be used. The slight reduction in usable area happened because when a sample point was close to the edge of a triangle, the outer edge of the texels were sampled during rendering. As a result, some of a pixel's color could be calculated from texels outside of the triangle's footprint (Fig. 7). If nothing were done to handle this scenario, the edges of triangles might show color from nearby triangles. So, one layer of texels was filled in around every triangle, to ensure that all rendered colors were valid.

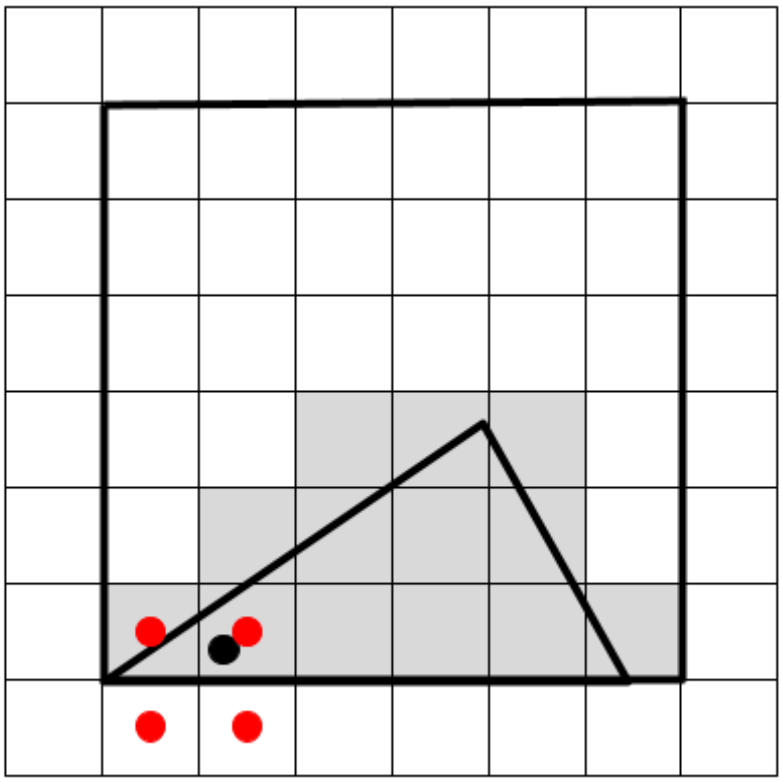

Figure 7. The black dot represents a sample point close to the triangle's edge. It averages the color taken from four nearby texels, with centers indicated by red dots. If there were no padding, these samples would access another triangle's texels.

To balance color quality with the cost of tracking a large number of texels, an optimal texel size had to be found. When texels were 4 mm wide, most triangles had less than 50 texels (Fig. 8). When texels were 8 mm wide, most triangles had less than 15 texels (Fig. 9). The larger 8 mm texel size was chosen because it required less computations to process, and it made most triangles fit within the 6x6 texel footprint for small triangles. In addition, it minimized the chance of triangles having more than 255 texels, allowing the GPU to use 8-bit integers instead of 16-bit integers for some data describing column locations within the color data buffer, which halved bandwidth consumption. Bandwidth was also conserved by only transferring color data

between the textures and buffers when absolutely necessary. In addition, color updates happened at most 10 times per second, instead of 60 times per second, to conserve energy.

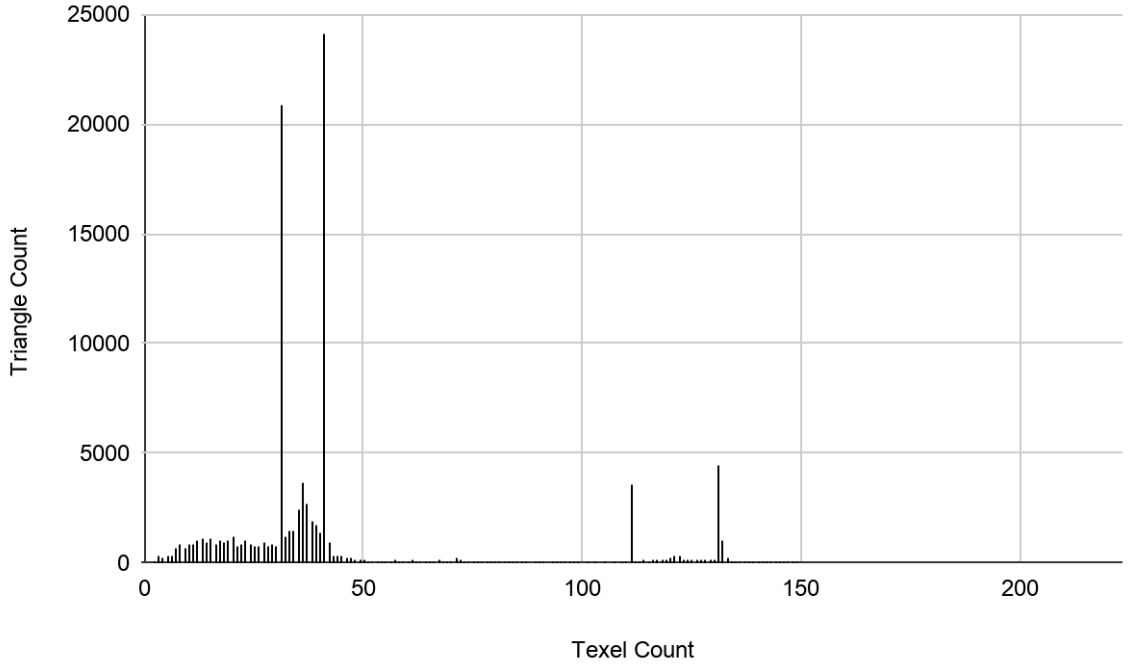

Figure 8. Distribution of triangles with respect to number of texels, where each texel is 4mm x 4mm.

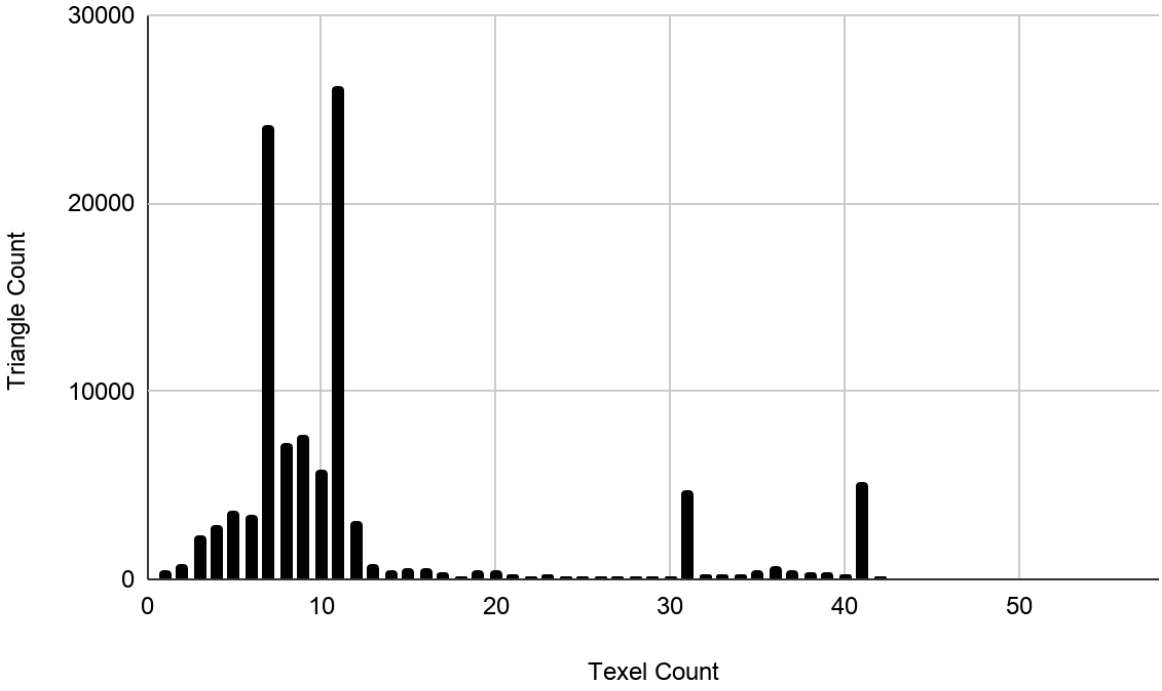

Figure 9. Distribution of triangles with respect to number of texels, where each texel is 8mm x 8mm.

Texel colors were updated inside the texture instead of the buffer, because the following render pass would only read data in the texture. The color update started by doing an intermediate render pass, which only involved triangles within the camera's point of view. Triangles were rendered onto a 1024x768 texture where each pixel stored a reference that identified a triangle. That way, one could figure out which triangle was visible to the camera at a particular point in the video frame. If part of a triangle was not visible to the camera, then updating that part's color would give it a color belonging to a triangle in front of it, which is incorrect. So, the GPU iterated over every texel in the triangle, testing whether each individual texel was visible to the camera and thus could be assigned a valid color.

Texels obstructed from the camera's view by the user's hand were prevented from updating their color. This happened because Apple's scene reconstruction algorithm did not handle small and moving objects well. So, Apple created a human body segmentation algorithm, which produced a texture where each pixel's value indicated whether it was likely part of a human body. This texture could be used to filter people out of the world mesh.

Although the user's hand was not part of the mesh, it was visible to the camera. If nothing were done about this fact, triangles behind the hand would incorrectly receive color from the hand. So, every pixel in the intermediate texture that corresponded to part of a human body was overwritten with an invalid triangle reference, preventing any triangles from registering their texels as visible there (Fig. 10).

Figure 10. The user's hand is filtered out, preventing most texels covered by the hand from updating their color. This made the user's hand appear almost invisible.

The video frame's color and the high-resolution triangle color data were both stored in the YCbCr color space. This color space allows brightness (luma) to be stored at double the resolution of hue (chroma), meaning that the apparent resolution of the triangle color was twice the resolution of the stored texels. The apparent resolution of texels was 4 mm, while the actual resolution was 8 mm, conserving energy while maintaining a high color quality.

The luma and chroma were stored in different textures in both the video frame and triangle texels. In the triangle luma textures, small triangles had a memory allocation of 16x16 texels and a footprint of 12x12 texels, while large triangles had corresponding values of 32x32 and 28x28 texels. This structure gave the luma two texels of padding on all sides, so luma could be sampled through bicubic filtering when rendering to the phone's screen. Bicubic filtering is more accurate at scaling to a higher resolution than standard bilinear filtering (Fig. 11).

Conventional texture sampling finds the location of a sample point in a texture, and combines the values of the closest pixels. However, when the sampling frequency is much lower than the resolution, undesired patterns form, called *aliasing* (Fig. 12). To combat this problem, a technique called mipmapping creates versions of the texture at lower resolutions, called mipmap

levels. When the sampling frequency is lower than the original resolution, it samples one of the lower-resolution levels.

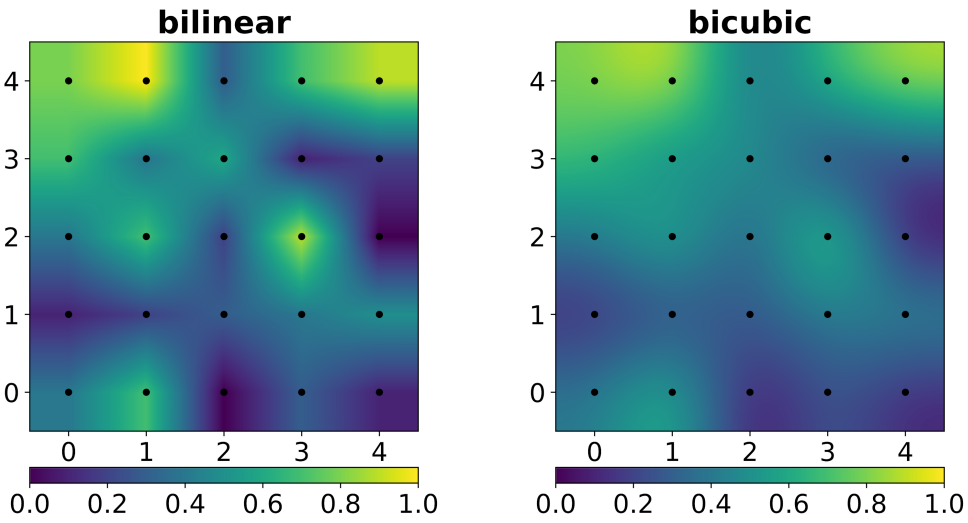

Figure 11. Left: linear filtering, right: bicubic filtering. Bicubic filtering requires 16 samples in a 4x4 grid around the sample point, while bilinear filtering requires four samples in a 2x2 grid. Although more costly, bicubic filtering produces smoother upscaling. By Zykure (2016) - Own work, CC BY-SA 4.0

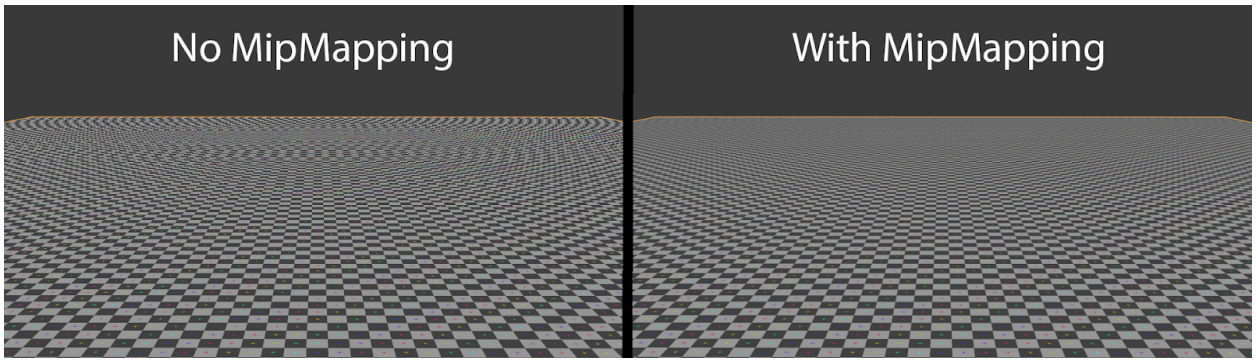

Figure 12. Left: undesired patterns form in the middle of the rendered image because the sampling frequency is lower than the checkerboard pattern's frequency. Right: mipmapping mitigates aliasing by using a lower-resolution texture when sampling at a lower frequency.

When viewing a texture from an extreme angle, the rate of sampling is vastly different along different directions. Using simple mipmapping causes the resolution to be blurred in the direction of high-frequency sampling. A modified version of mipmapping called *anisotropic mipmapping* alleviates this. It takes in two different sampling rates and produces an output that is only blurred in one direction (Fig. 13).

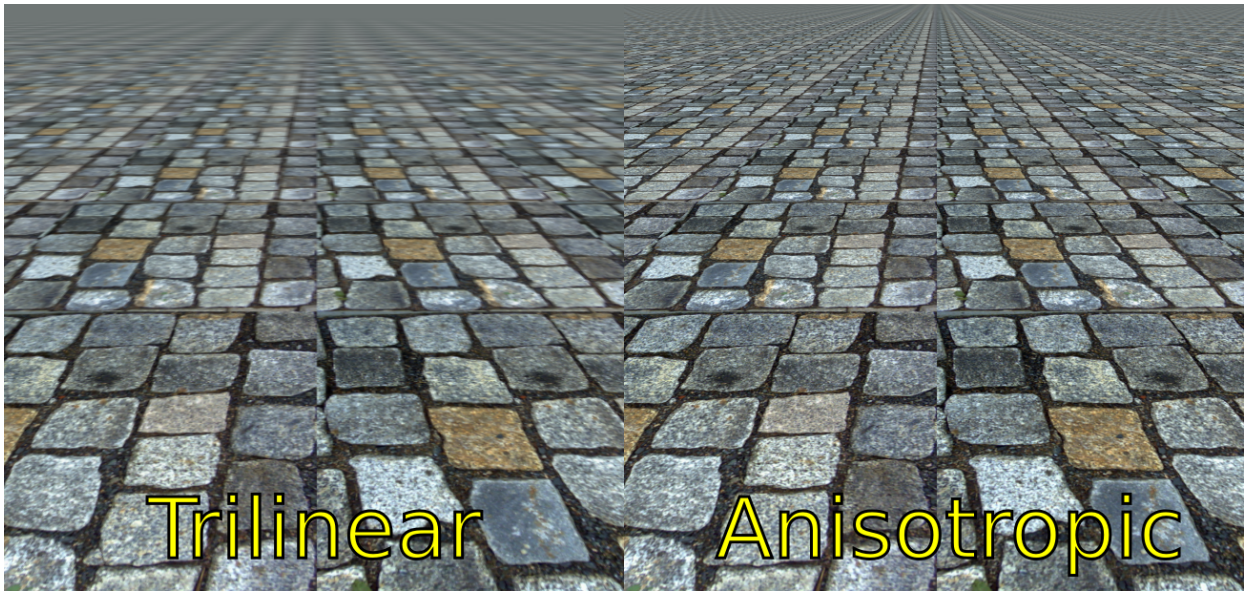

Figure 13. The difference between the two methods of mipmapping is most apparent at the top of the images. On the left, trilinear filtering mixes values at two different mipmap levels, and blurs detail in all directions. On the right, anisotropic filtering preserves detail in the direction of the highest sampling frequency. By Thomas (2010) - derivative work: Lampak - Cobblestones.jpg, CC BY-SA 3.0

When texels sample from the video frame to update their color, they often cover several pixels in either direction. Their massive footprint means their sampling rate is much lower than the video resolution, causing aliasing. In addition, triangles are sometimes viewed from steep angles. To maximize the quality of reconstructed color, anisotropic mipmapping is needed. This requires calculating each triangle texel's footprint on the video frame's texture.

When generating mipmap levels, each level has exactly half the resolution of the previous one. To generate perfect mipmaps, the original texture's resolution should be a multiple of a large

power of two. The resolution of the video frame was 960x720 (chroma) and 1920x1440 (luma). These values meant that the number of mipmap levels was five for chroma and six for luma.

When triangles were farther away, the footprint of a texel's projection onto the video frame was smaller. For the original video resolution, the average footprint of a triangle's texel matched the size of a video pixel at 6.6 meters away. Smaller resolutions were more appropriate for closer triangles, whose texel footprint occupied a larger part of the video frame. The largest mipmap level in the original video textures could support triangles 21 cm away (luma) and 42 cm away (chroma). However, many triangles in the world mesh were closer to the user than 42 cm, so the video frame had to be resized to maximize the number of mipmap levels.

The video frame was resized to 768x576 (chroma) and 1536x1152 (luma), allowing for two more mipmap levels. This changed the smallest supported distance to 3.9 cm (luma) and 7.8 cm (chroma) and reduced the number of pixels by 36%. The change also allowed the first mipmap level for each texture to be generated while resizing, minimizing the cost of the resizing operation.

Late March

The last weeks of the project were spent resolving problems that decreased color quality. The first problem was motion blur. Since the camera was almost never still, parts of the video frame were blurry. Although the blur was not noticeable inside the camera's view, it was apparent when viewing the stored color outside of the camera's view (Fig. 14). A mechanism was created to prevent sampling color in areas prone to blur, although it was later deactivated because it prevented enough color data from being captured.

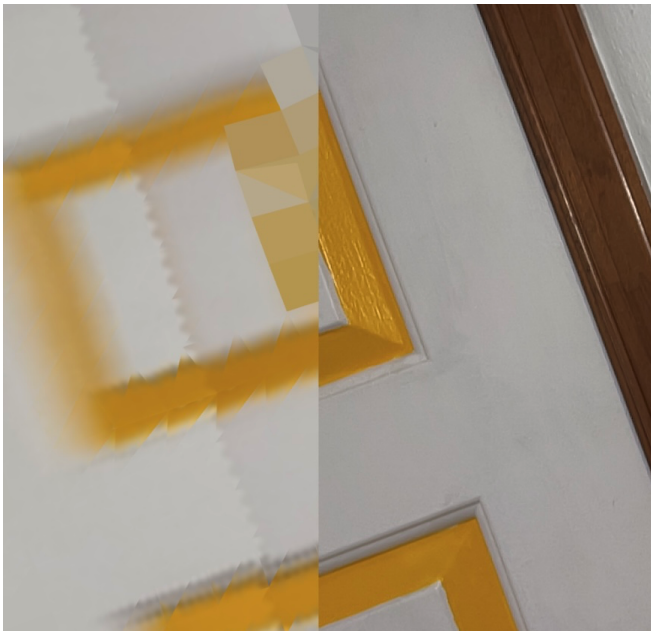

Figure 14. The left side of the image is outside of the camera's view. The upper yellow area appears blurry because the phone was moving quickly when that area's color was scanned.

The first attempt to account for motion blur was to prevent all color updates from happening when the phone was moving too quickly. However, data on the phone's motion was not consistent (Fig. 15 and 16) and did not account for the fact that blur decreased color quality more severely in areas farther from the camera.

The second approach allowed each triangle to choose whether to update its color based on how quickly its position in the video frame changed. If a triangle was moving too quickly, it was more likely to be affected by blur, so it did not update its color. This approach allowed for

more color to be sampled than the previous approach, but it did not correct all blur, and required the user to hold the phone still for short periods of time to gather color data. Since the phone was moving at least a small amount during the majority of any AR session, the lack of sampling was more detrimental than motion blur, prompting the deactivation of blur correction.

The second issue was a bug that took a week to solve and was created before this project started. During the geometry culling before each frame rendered to the phone's screen, vertices were assigned areas of memory that stored whether they were included in the user's frame of view and/or the camera's frame of view. Any vertex's inclusion data could be written to by multiple processing units at the same time.

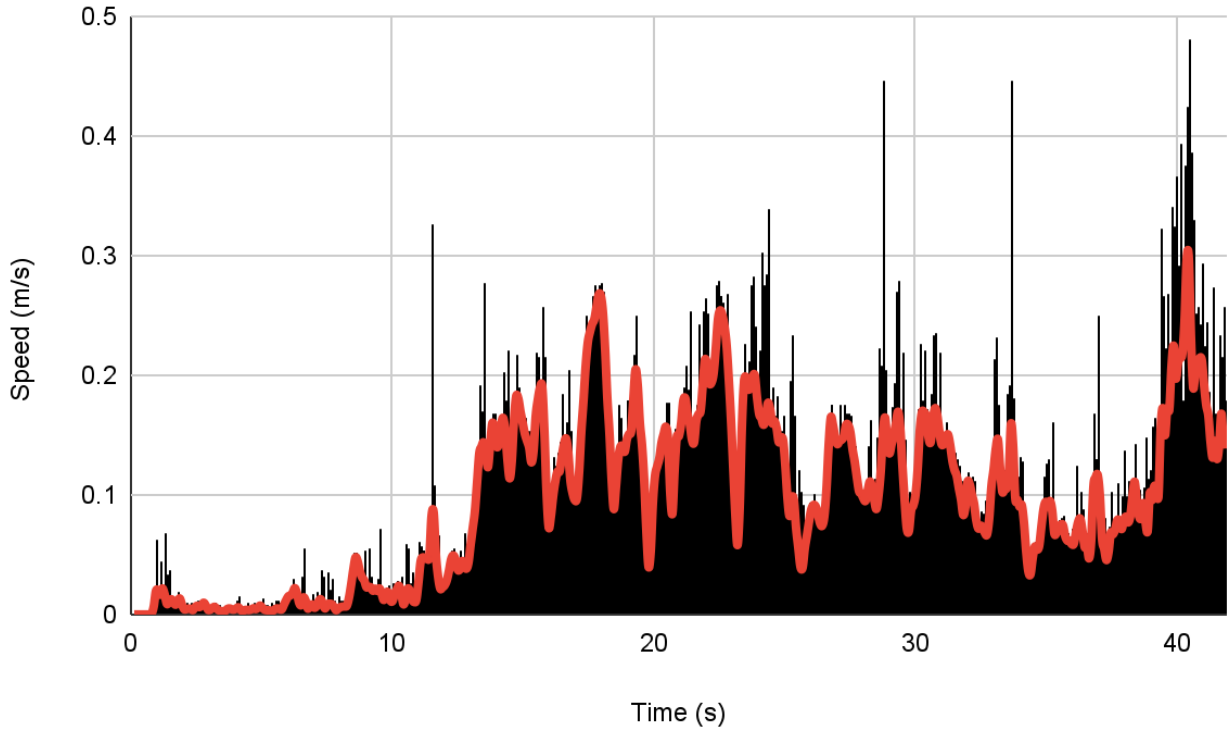

Figure 15. Approximate translational speed of the camera at various times in an AR session, calculated from the phone's position in adjacent frames at 60 fps. The red path is a 10-wide, centered moving average.

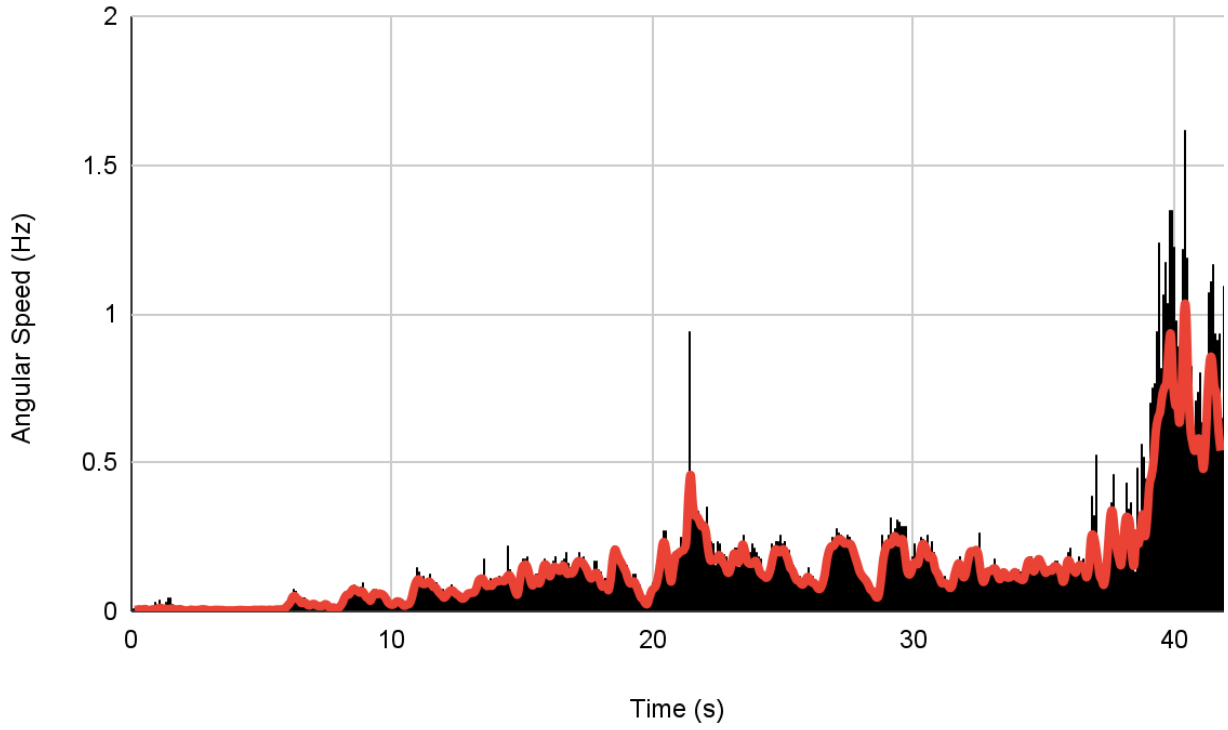

Figure 16. Approximate angular speed of the camera at various times in an AR session (in revolutions per second), calculated from the phone's orientation in adjacent frames at 60 fps. The red path is a 10-wide, centered moving average. At the end of the session, the phone was shaken rapidly, resulting in an increase in angular speed.

Each vertex's inclusion data was 16 bits wide and stored two 8-bit components. When only one component had to be modified, the GPU incorrectly read both components and wrote back both. As a result, whenever two processing units tried to write to separate components of a vertex's inclusion data at the same time, the data was corrupted.

The final issue was that a significant amount of triangles did not match to any other triangles across mesh updates, so triangles often lost all color soon after they left the camera's view. The solution was to assign color to 3.125 cm volume sectors, giving low-resolution color to triangles that would otherwise lose all color data.

Color was calculated by finding all triangles in the old mesh that were close to a sector, and calculating the sector's color from those triangles. Color data was accumulated by adding numbers through atomic operations, which means multiple compute units could modify a sector's color data at the same time without producing an undefined result. These numbers were the sums of the luma and chroma components of the color of each nearby triangle's center, and a counter that indicated how many triangles had contributed color. The sums were divided by this counter to find the average color. When a triangle in the new mesh did not have a match, it could search for nearby sectors and see if any had an average color.

Discussion

After finishing the color tracking algorithm, it was tested out in VR using the Google Cardboard headset. Since Apple's 3D scene reconstruction algorithm did not perfectly reconstruct the shape of objects, color at the boundaries of objects was often projected onto incorrect areas of the scene. However, the quality of the reconstructed color was sufficient for simulating an AR headset.

Performance

When iOS devices overheat, the system takes several measures to reduce power consumption, including reducing the display's refresh rate from 60 fps to 30 fps. The decrease in frame rate occurred 10-20 minutes into each session using scene reconstruction with color tracking. Any frame rate below 60 fps is not acceptable for VR, so the software created during this research was optimized to maximize the length of time before the system throttled the display's refresh rate.

The greatest factor contributing to power consumption was the rendering that happened every frame. In VR, headsets must pass light through a lens before it reaches the user's eye. Since the screen is too close for the user to naturally focus on it, the lens bends the light in a way that focuses the light. This bending of light warps the screen's image and produces a slightly different image for red, green, and blue light. So, software transforms the rendered image in a way that counteracts these distortions.

Most VR software renders an image to intermediate texture, which it transforms into the final output after rendering. The intermediate texture has higher resolution than the device's screen, so rendering consumes a large amount of energy. Variable rasterization rate (VRR) mitigates this problem by rendering some areas of the image at a lower resolution than others.

For example, areas that fall into the user's peripheral vision do not need as much detail as the center, so they are rendered at a lower resolution.

Most graphics APIs accomplish VRR by rendering to several textures, each at a different resolution. Tiles (small square chunks) of the final image are each assigned rasterization rates, and tiles with the same rasterization rate render to the same intermediate texture. After rendering, the intermediate textures are combined. Although the process of merging textures consumes a large amount of bandwidth, energy can be saved if the cost of producing a pixel during rendering is high.

Apple implements VRR in a vastly different way in Metal. This API renders all pixels to the same intermediate texture, minimizing bandwidth consumption. It accomplishes this by splitting the intermediate texture into tiles, instead of the final texture, and mapping tiles in the intermediate texture to areas in the final texture. However, this method prevents each tile from having a truly unique rasterization rate. For example, if one tile needs a high resolution, every tile in the row and column of tiles that contains it will also have that high resolution. This issue prevents the developer from having fine control over every tile's resolution, resulting in slightly longer rendering time than other APIs' implementations.

However, for this research, Apple's implementation was beneficial. Since VR uses an intermediate texture, the intermediate texture from VRR can be substituted as the intermediate texture for VR. Then, the mapping between the intermediate and final resolutions becomes another part of distortion correction.

Limitations

Most iPhones do not have a LiDAR scanner and therefore cannot do 3D scene reconstruction. However, these devices can pass a 2D video of the user's surroundings into VR,

like the "Passthrough" feature on Oculus VR headsets. Although this technique does not let the user view their surroundings through their peripheral vision or perceive depth, it still allows the user to interact with virtual objects and simulates the experience of using an AR headset.

References

Apple. (2020, June 24). *What's new in RealityKit.* [Video]. https://developer.apple.com/videos/play/wwdc2020/10612

Bohn, D. (2019, February 24). *Microsoft's Hololens 2: A $3,500 mixed reality headset for the factory, not the living room.* The Verge. https://www.theverge.com/2019/2/24/18235460/microsoft-hololens-2-price-specs-mixed-reality-ar-vr-business-work-features-mwc-2019

Diaz, J. (2019, September 11). *Project StarBoard: New evidence of Apple AR glasses emerges.* Tom's Guide. https://www.tomsguide.com/news/project-starboard-new-evidence-of-apple-ar-glasses-emerges

Harris, M. (2010) *State of the Art in GPU Data-Parallel Algorithm Primitives.* NVIDIA. https://www.nvidia.com/content/GTC-2010/pdfs/2084_GTC2010.pdf

Hawthorne, M. (2021, February 22). *VR Oculus Quest 2: What is Passthrough?* Technipages. https://www.technipages.com/vr-oculus-quest-2-what-is-passthrough

Malyshau, D. (2020, April 23). *A taste of WebGPU in Firefox.* Mozilla Hacks. https://hacks.mozilla.org/2020/04/experimental-webgpu-in-firefox/

Polok, L. and Smrz, P. (2016, April 3). Increasing double precision throughput on NVIDIA Maxwell GPUs. *HPC '16: Proceedings of the 24th High Performance Computing Symposium.* https://doi.org/10.22360/SpringSim.2016.HPC.032

Thomas (2010, June 21). *Anisotropic filtering* [Online Image]. Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Anisotropic_filtering_en.png

Wilson, M. (2020, October 13). *What is a LiDAR scanner, the iPhone 12 Pro's camera upgrade, anyway?* TechRadar. https://www.techradar.com/news/what-is-a-lidar-scanner-the-iphone-12-pros-rumored-camera-upgrade-anyway

Zykure (2016, September 1). *Interpolation-bicubic* [Online Image]. Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Interpolation-bicubic.svg

Zykure (2016, September 1). *Interpolation-bilinear* [Online Image]. Wikimedia Commons. https://commons.wikimedia.org/wiki/File:Interpolation-bilinear.svg