Interactable Mixed Reality Environment using a Depth Camera

Juan Diego Mendez
The University of Texas at Austin
jdmg1308@utexas.edu

Sarah Abraham
The University of Texas at Austin
theshark@cs.utexas.edu

Abstract

Current virtual reality (VR) and augmented reality (AR) technologies allow users to experience highly immersive virtual environments. However, a key issue in these technologies is the limited ability to seamlessly integrate and interact with the user's real-world physical surroundings. This research addresses these limitations by developing a system that dynamically captures the real-world environment using depth sensor data, allowing for real-time interactions between virtual objects and the user's physical space. The proposed voxelization approach divides the 3D scene into discrete voxels, efficiently representing surfaces in the physical environment while inherently handling sensor noise. Integrated with an Oculus Quest 2 headset and an Intel RealSense D435 depth camera, the system generates a voxelized reconstruction of the physical environment, enabling virtual elements to collide and interact with real-world surfaces and objects. Experiments demonstrate the system's capability to provide smooth mixed reality experiences, with processing times within acceptable bounds for real-time applications. Key challenges addressed include time efficiency, accurate representation of curved surfaces, and alignment of the virtual assets with the scanned physical environment. The research highlights the potential of mixed reality technology for enhancing human-computer interactions across various domains, paving the way for further advancements in seamlessly blending virtual and physical worlds.

1 Introduction

1.1. Motivation and Problem Definition

Virtual Reality (VR) is a digital simulation of a three-dimensional environment that can be interacted with by a person using equipment, such as a VR headset. *Augmented Reality* (AR), on the other hand, is a technology that superimposes a computer-generated image on a user's view of the real world, thus providing a composite view. *Mixed Reality* (MR) takes it a step further by combining both ideas, mixing the physical and digital worlds, where digital objects can interact with the real environment and vice versa. MR technology allows virtual imagery to be seamlessly blended with the real world in real-time.

While the concepts of VR and AR have been around for decades, mixed-reality technology is still in its early stages, with significant room for improvement. Current mixed reality systems struggle to achieve seamless integration of virtual and physical elements, often resulting in a disconnected or jarring experience for the user.

One of the key limitations of this technology is the inability to truly integrate the real, physical world into the virtual experience. During the past few years, the demand for better and more efficient mixed reality technology has grown (Medium, 2023), making these limitations something worth researching and overcoming.

An event that particularly tends to break the immersion in a mixed reality experience is the lack of interaction between the physical world and the virtual elements being manipulated by the user. Picture the following scenario, a user is playing a game with a virtual sphere, they grab it and throw it against a wall, but instead of colliding and interacting with it, the sphere goes through the wall. While this interaction makes sense within a solely VR environment, it leads to less immersive *mixed reality* experiences, and it takes away from the potential that this technology has for improving human-computer interactions. After all, the point of this technology is to seamlessly blend both the virtual and the physical environments into one, such that the user can make the most out of both. For this reason, the goal behind this research project was to build a system capable of dynamically capturing the surrounding physical environment and making it accessible for the user to interact with it in real-time. To achieve this, VR technology, specifically an Oculus Quest 2 headset was used, and the passthrough mode on this device was utilized to test the project in mixed reality.

Most modern devices utilize either computer vision algorithms or depth sensors to scan the environment around the headset. Computer vision algorithms process pixels from captured images to estimate the environment, while depth sensors generate point cloud data. Point cloud data is a 3D set of points

that represent a scene scanned by a depth sensor, which measures distances to objects by emitting signals and analyzing their return time. We'll opt for the latter approach. The Oculus Quest 2 device, unfortunately, does not allow access to the point cloud data that it already obtains with its built-in sensors, so in order to overcome this the solution was to use an external depth sensor. After looking at different depth sensors, the most efficient and affordable was the Intel RealSense D435, which enables for accurate depth perception of up to 10 meters. By converting real-time sensor point cloud data into 3D assets and using them in a VR application, virtual content can interact with real physical surfaces and objects. Going back to the previously given example, implementing this solution said the virtual sphere would appear to bounce off the real wall present in the room. This advancement could open up new possibilities for MR gaming, training simulations, assistive technologies, and more.

1.2 Background Research:

Cazamias and Sunder Raj from Stanford University investigated the feasibility of Virtualized Reality using depth cameras, such as the Kinect and Intel RealSense devices. Their research aimed to construct virtual scenes from real-world environments in both real-time and offline settings (Cazamias & Raj, n.d.). They employed techniques like KinectFusion, point cloud stitching, and raycasted point clouds.

KinectFusion, a real-time 3D reconstruction algorithm, was a focal point of their investigation. By leveraging depth frames captured by the Kinect camera, KinectFusion merges multiple depth images to create a high-fidelity 3D reconstruction of a scene. This technique addresses occlusion and noise issues over time, enabling a more accurate representation of the real world in VR. However, limitations exist in terms of capturing large scenes due to memory constraints, necessitating the use of point cloud stitching (merging of multiple point cloud datasets) to assemble different parts of a scene together.

The study is relevant since it delved into the potential of Intel RealSense depth sensors to scan the physical environment. Many of the challenges faced in this research, such as memory issues, can be addressed by implementing a different methodology that trades accuracy for time efficiency.

2 Methodology

2.1 General methodology

The following is and has been, the overall idea for the methodology of this project through the entire process of development.

Step 1: 3D Data Capture

The first step would be to scan data from the environment, for this project, it seems the best tool for this is the Intel RealSense D435 lidar sensor, as it allows point cloud data capture in real-time. Using the Unity wrapper for RealSense SDK 2.0 streams of data from the RealSense device can be processed in Unity.

Step 2: Point Cloud Processing

The next step would ideally involve filtering excess data, and outliers from the point cloud, while also trying to align the data with the surrounding physical environment being scanned to then start segmenting the point cloud data into objects or surfaces.

Step 3: Generate 3D mesh/sets of meshes

For the next and most important step in the pipeline, the processed data would be used to generate 3D meshes or sets of meshes representing the physical environment within the VR/MR environment.

Step 4: Dynamic Environment

These meshes/objects that are generated in real-time are dynamically updated based on new point cloud data received on each frame.

Finally, while not a necessary step for this project, virtual objects with colliders were added to the final scene to facilitate testing, and study interactions between the dynamically generated objects and the preloaded ones.

2.2 Initial Work

The development process began by exploring different techniques to make *step 3* of the general methodology (generation of meshes) work. The exploration began through basic Python scripts that could turn point cloud data into a coherent mesh. The algorithms that were tested during this stage included ball pivoting, Poisson surface reconstruction, and alpha shape. While these scripts served as a foundation for understanding the fundamental concepts of point cloud manipulation, the amount of time they took to return a result from a relatively small point cloud file was too long.

The next approach involved using the Unity tool, as this one would most certainly be used for the final version of the project. At this stage of the development, there was no access to a depth sensor of any kind, so the focus

shifted towards experimentation with point cloud data and Unity colliders, specifically generating a convex hull around a set of randomized points. This approach aimed to efficiently create a basic mesh representation from a constantly updating set of random 3d points to study how *steps 2*, *3,* and *4* would work together.

The script generates random point cloud data and calculates the convex hull. The convex hull is the smallest convex shape that encloses or contains all the points within the cloud, and it provides a simplified representation of the point cloud's shape. To perform this operation CGALDotNet was used, which is a C# wrapper for the Computational Geometry Algorithms Library (CGAL). CGAL is an efficient open-source software library that provides algorithms for computational geometry tasks, including convex hull computation.

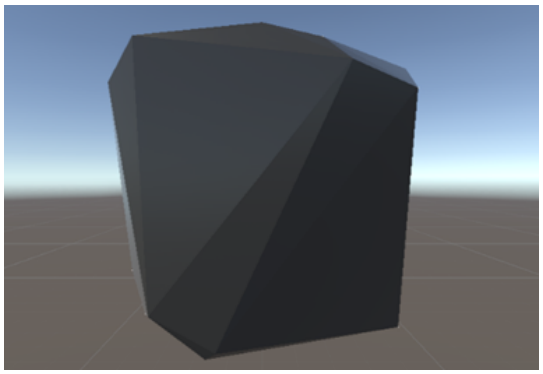

Figure 1: mesh obtained by the simple convex hull script using the CGALDotNet wrapper

2.2 Failed Approach

After experimenting with the Intel RealSense D435, it became clear that the methodology needed to be adjusted to handle the real-time point cloud data effectively. Unlike the randomly generated point cloud data used in the initial experiments, the sensor's output required additional processing to filter out noise and irrelevant information.

Upon receiving a new set of points from the sensor, the points are processed iteratively. For each point, several checks are performed to ensure the data's validity and relevance. First, the script filters out excess data and outliers by setting a range for valid points. Points too close to the sensor or too far away are discarded, as they may introduce inaccuracies or unnecessary computational overhead.

This filtering step is crucial because the raw point cloud data from the depth sensor tends to contain a significant amount of noise and irrelevant information, such as points representing objects outside the area of interest, or points that do not accurately represent the physical environment, either due to sensor limitations or environmental factors (Interference, reflective surfaces, hardware defects, etc),

For the algorithm, simple boxes where the volume is zero are created at the start, initially there are no objects in the scene that points can be compared to. As the algorithm progresses each point is compared with this one, and other generated boxes, to make them expand and accommodate to fit the 3d objects and shapes that are being scanned.

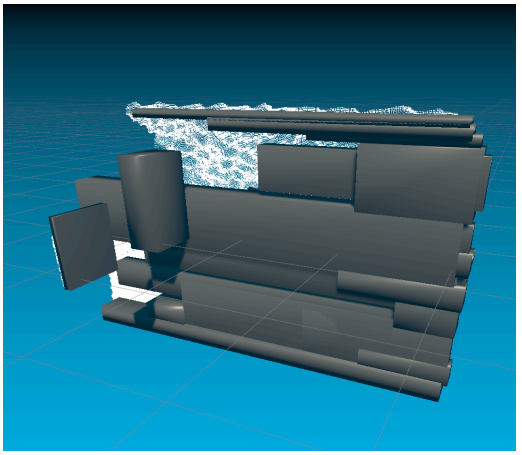

Figure 2: failed approach algorithm where mesh geometry grows (multiple meshes).

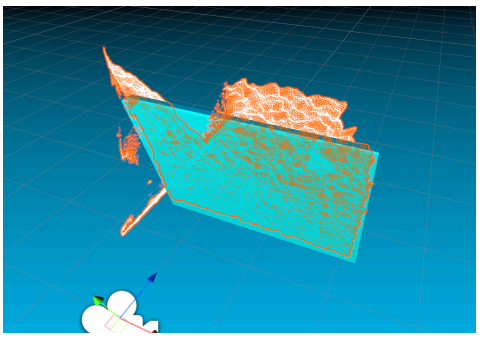

Figure 3: failed approach algorithm where mesh geometry grows (one mesh).

For each of these objects, their GameObject reference and their vertices are stored and associated with a key value for easy access. Additionally, a Vector3 value is stored to represent the normal of the plane that this box accommodates the most, which is important later when updating the structure.

The core algorithm related to box generation works as follows:

During the algorithm, each specific point is compared with every current object, all of which consist of bounding boxes, a set of 8 points in 3D space. When a point is compared, it goes through two checks:

1. If the point is inside an existing box.
2. The distance from the point to the box, in order to find the closest box.

If the first check results in true, the algorithm stops for that specific point, as it is irrelevant. If the first check always returns false, it means this point is not within the bounds of any of the boxes, and it must therefore be considered further.

If a point needs to be considered further, the next step uses the distance value and the closest box found in the previous step, and it again makes two checks. The first one is whether the point is close enough to be added to the box.

The second check is more complex. If the point is close enough to the box, the distance between the point and the plane associated with the box is checked. This step is crucial to prevent edge cases, such as a point belonging to one wall from being added to a box representing a different wall. By considering the plane normal, the algorithm can differentiate between surfaces and avoid merging distinct planes or surfaces into a single box. This is particularly important in everyday environments, where most surfaces tend to be flat or planar.

If the point passes all checks, it is added to the corresponding box it is closest to. This alters the mesh representing the box, and the collider for that box is updated as well. This could also change the plane associated with the object, which would rotate the game object to align with the updated plane.

This algorithm implements *steps 2*, *3,* and *4* of the general methodology, but it did not end up being the final version for this project, as there were several issues with it, these are explained in 3 *Experiments and Results* and 4 *Analysis*.

This is the pseudocode for the algorithm.

```
For every new set of points scanned
          Iterate through each vertex:
                    if it meets the base requirements
                              Call CheckVertex()
                              If it returns -1
                                        Create a new box
                              Else if not -2
                                        Update box

CheckVertex()

          Return -2 false if within area of one of the
          already existing boxes

          Return -1 if the minimum distance to the box
          and to its plane is not met

          Otherwise return the key for the closest box
```

2.3 Final Approach

Using a voxelization approach turned out to be more efficient and accurate than previously thought so it ended up being used as the final approach. It offers several advantages over traditional mesh reconstruction techniques. By dividing the 3D space into discrete voxels, the algorithm can efficiently represent complex geometries without the need for intricate surface reconstruction algorithms, which was the previous approach. This simplicity translates into improved computational performance, making it suitable for real-time use.

Moreover, the voxelization technique inherently handles occlusions and sensor noise, as it relies on the density of points within each voxel rather than attempting to reconstruct surfaces from sparse or incomplete data. This robustness can be particularly beneficial in scenarios where the depth sensor may encounter challenging conditions, such as reflective surfaces or obstructions.

An important challenge to overcome was aligning reality with the data being obtained from the source. Given that the Intel RealSense D435 does not have any orientation sensors, using the Unity transform for the VR headset (Oculus Quest 2) ended up being the ideal solution, the only requirement for this to work is to attach the depth sensor to the headset making sure the sensor points at the same direction the headset is facing.

This fix works because, with it, each point processed from the depth sensor can be rotated and translated by these values to “align” it with the actual physical space being scanned. Having a more accurate representation of the physical space surrounding the user, the points can be used for an accurate voxelation of said space.

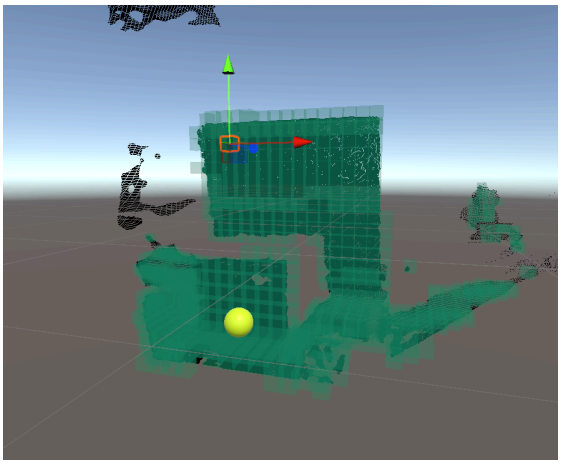

Figure 4: Initial attempt at the voxelization approach.

The following is the step-by-step for the final algorithm. First of all, in a similar manner to the failed approach, for each frame scanned by the sensor, the code iterates over the points in the point cloud data received. Each one of these points is rotated and translated based on the headset transform, and then the corresponding voxel position (3D grid coordinate) is calculated by dividing the point's coordinates by a predefined voxel size and flooring the result. After this, the algorithm updates a dictionary

(`voxelCounts`) that stores the count of points in each voxel position. If the voxel position is not in the dictionary, it initializes the count to 1. Otherwise, it increments the existing count.

Next, the algorithm iterates over the `voxelCounts` dictionary keys. For each voxel position, it checks if the point count exceeds a minimum threshold.

If the count is above the threshold and there is no existing game object for that voxel position, it creates a game object with a box mesh and the corresponding collider for it. The created game object is stored in another dictionary (`voxelRef`) with the voxel position as the key.

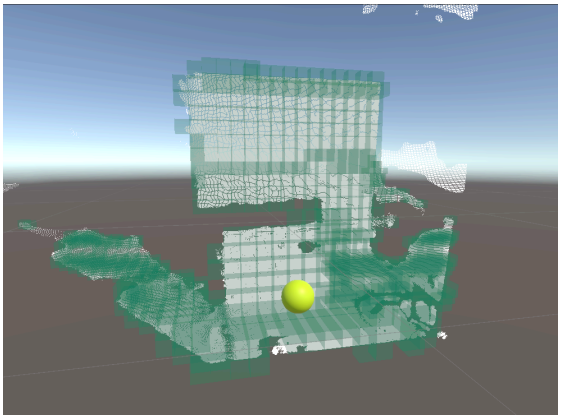

Figure 5: Initial attempts at making the voxelization approach interactable.

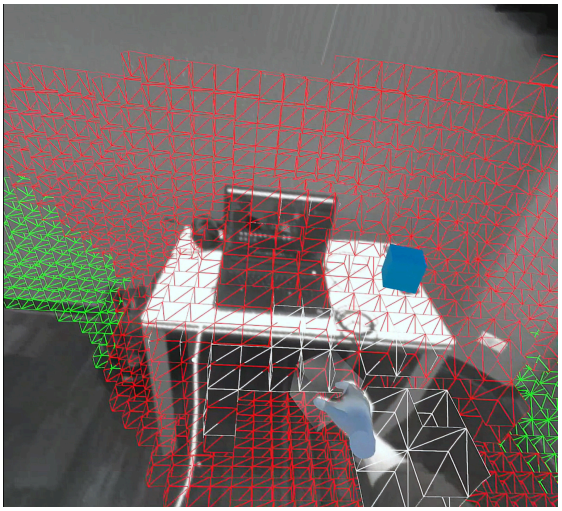

Figure 6: Interaction between the physical environment (table) and a virtual element (cube).

On the other hand, if the point count for a voxel position is below the threshold and there is an existing game object for that voxel position, the algorithm destroys the existing game object and removes the voxel position from the `voxelRef` dictionary.

Finally, regardless of the number of points on the voxel positions, the algorithm reduces the point count for each of them by a fraction (1/10th) of the point count for that voxel. This step helps to gradually remove game objects from voxels that have a decreasing number of points over time, which ends up taking out objects that have been removed from the scene, or reconstructing a new scene whenever the user moves/rotates the point of view.

The result of this algorithm is a set of game objects in the scene that represent the voxelized environment, with each voxel containing a sufficient number of points represented by a cube with a mesh collider. Using the passthrough capabilities from the Oculus Quest 2 a user is capable of seeing real space and interacting with it by real means (such as making a virtual sphere bounce off a real wall). This approach allows for collisions and interaction between virtual objects and the voxelized physical environment.

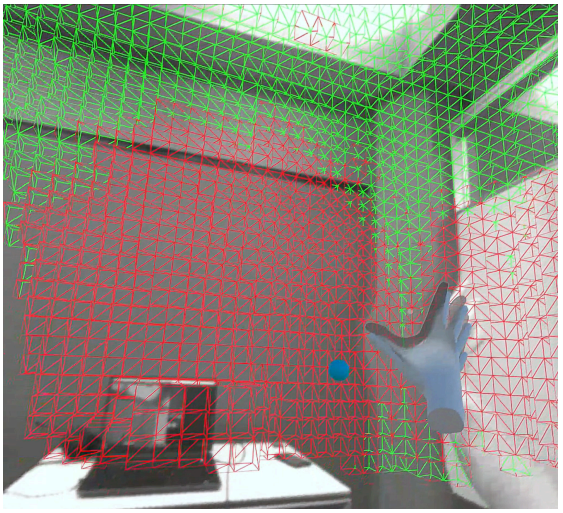

Figure 7: Interaction between the physical environment (wall) and a virtual element (sphere).

The pseudocode for the algorithm is the following:

```
Initialize voxelCounts and voxelRef dictionaries
For each set of points received from the depth sensor
    for each point in the point cloud:
      -    rotate and translate the point to align with
           the headset's transform
      -    compute the voxel position of the rotated and
           translated point
      -    Increment or create the voxel count for that
           voxel position

    for each voxel position in voxelCounts:
          if pointCount >= MIN_POINTS_FOR_SHAPE:
              if voxelRef does not contain the voxel
              position:
                -    compute the minimum and maximum
                     coordinates of the voxel
                -    create a new box mesh with these
                     coordinates
                -    create a new GameObject for the box
                     mesh and add it to voxelRef
          else:
              if voxelRef contains the voxelposition:
                -    destroy the GameObject associated
                     with the voxel position
```

```
        -   remove the voxel position from
            voxelRef

decrease the voxel count for the voxel position by a
fraction of its pointCount
```

3 Experiments and results

3.1 Experiments and results for initial work

To measure the efficiency of the initial work, timings were taken and recorded, as it was important to study how *steps 2*, *3,* and *4* would work together in a time-efficient manner. For this, the average time was calculated multiple times, each time using a different number of vertices, which were randomly generated.

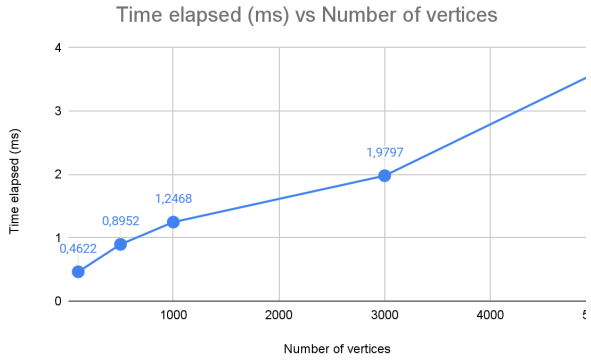

Figure 8: graph (time vs n of vertices) for initial approach.

The graph above presents the average number of milliseconds required to calculate the convex hull for different numbers of vertices in the initial randomized point cloud data. As observed, the time required for computation generally increases with an increase in the number of vertices.

The Intel RealSense D435 returns exactly 307200 points on every frame where it reads the depth values in the environment, given the above graph a fair estimate of time for processing would be around 30 to 50 ms multiplied by the number of boxes.

This processing time, however, is not fast enough for real-time mixed reality applications, which typically require frame rates of at least 30 frames per second (FPS) for a smooth experience. If we consider an average room size with a moderate number of boxes, the continuous cost of computing the convex hull and generating meshes for each frame would be too high, leading to very poor performance and potential lagging or stuttering in the mixed reality experience. This is something that became more evident in the failed approach, as this tries to construct meshes in a similar manner to the initial work (both try to construct accurate mesh/meshes based on points that constantly update), with even more overhead since more data structures and computations are being made.

3.2 Experiments and results for failed approach

To evaluate the performance of the algorithm for the failed approach an experiment was conducted and data was obtained. The program was run with the Intel RealSense D435 LiDAR sensor pointing to increasingly complex environments. The environments ranged from a simple room with minimal furniture to more cluttered spaces with various objects and surfaces.

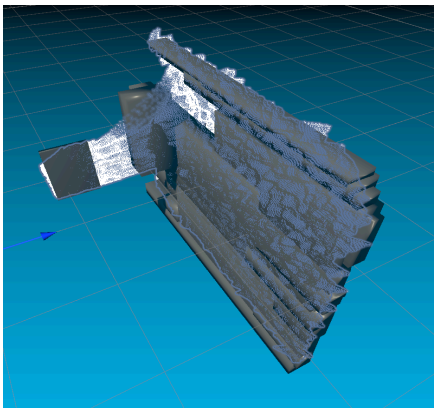

Figure 9: failed approach algorithm where mesh geometry grows (multiple meshes).

In each experiment, the control variable was the complexity of the environment, measured by the number of distinct surfaces and objects present. This allowed for a systematic evaluation of the algorithm's performance as the number of points and potential boxes increased.

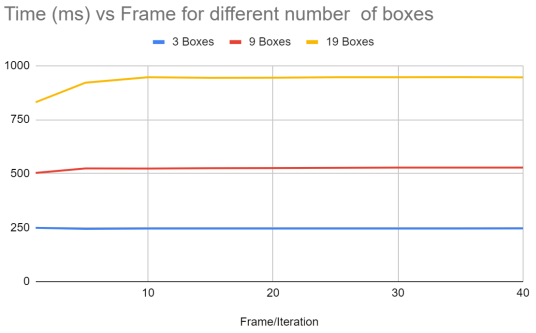

Figure 10: graph (time in different frames for different n of boxes) for failed approach.

The graph above shows the amount of time it takes on (different frames) to process the points based on the number of boxes generated. It can be said that on average each frame the depth sensor collects takes around 60 ms to be processed per box. This number is too high, which makes this implementation not fast enough for real-time mixed-reality applications.

Many things could be done to achieve real-time performance while keeping this algorithm the same. One potential optimization could involve parallelizing the algorithm or utilizing hardware acceleration, such as GPU computing, to take advantage of modern hardware capabilities and distribute the computational workload more efficiently.

However, it was decided that a new algorithm was required, as optimizations for this one could lead to limited progress, and without it being really worth it, as the point of the project is interaction with real spaces instead of accurate reconstruction of these. For this reason, the project ended up using voxelation as the final approach.

3.3 Experiments and Results for final approach

3.3.1 Performance Evaluation

To evaluate the performance of the voxelization algorithm, experiments were conducted and measurements were taken. Specifically, the sensor was set at different distances from a flat wall so the algorithm would generate a different number of voxels each time. For each of those distances, time measurements were taken. It is worth mentioning that these measurements were taken twice, once without headset movement, and once with headset movement, as it was noticed that this affected the results quite a lot.

The graph below illustrates the relationship between the number of voxels generated and the processing time required per frame when there is no headset movement.

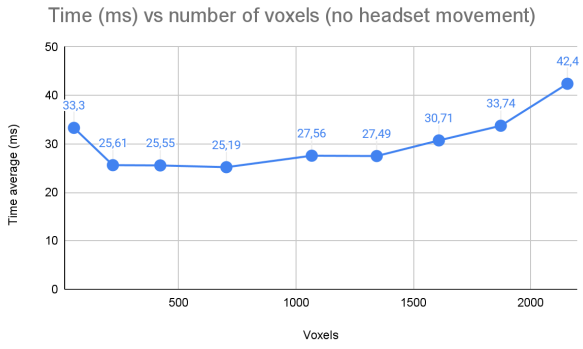

Figure 11: graph (time vs n of voxels) for final approach with no headset movement.

The processing time increases linearly with the number of voxels, which is directly proportional to how close the D435 is to the wall. However, the algorithm's performance remains within acceptable bounds for real-time applications. The graph shows a relatively stable trend in processing time as the number of voxels increases. Initially, there is a slight decrease in processing time from 33.3 ms for 53 voxels to 25.19 ms for 703 voxels. This could be because of the density of points being calculated, or perhaps it could have been a systematic error. Furthermore, there seems to be a spike in time after 1000 voxels. Overall, the average processing time remains within a relatively narrow range, varying from around 25 ms to 34 ms. This all changes however when headset movement is applied, which is the case for the following graph.

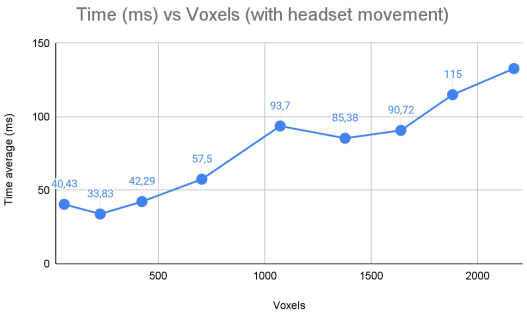

Figure 12: graph (time vs n of voxels) for final approach with headset movement.

This one exhibits a more erratic pattern in processing time with increasing voxel counts. While there is a gradual increase in processing time from 40.43 ms for 59 voxels to 115 ms for 1883 voxels, the trend is disrupted by significant spikes in processing time after 1000 voxels, the same phenomenon as the graph without headset movement. Regardless, the main thing to notice about the data is how the increase in time is way higher when the headset is moved. The following graph exposes this relationship between data better.

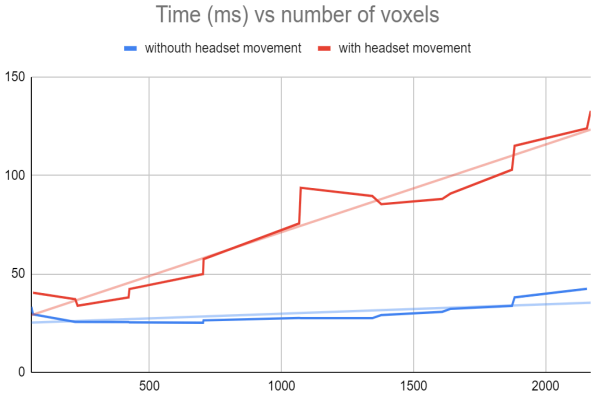

Figure 13: graph (time vs n of voxels) for final approach comparing headset movement vs no headset movement.

When observing the behavior of the program in these graphs it is evident that headset movement increases the time that processing one frame of data takes, one possible reason for this is that whenever the headset rotates it has to process completely new voxelated spaces, and add them to voxelCounts to be processed. These voxels if not rendered still remain stored in

voxelCounts, which could be slowing down the process. Below is a graph that shows the active voxels vs the ones stored in voxelCounts.

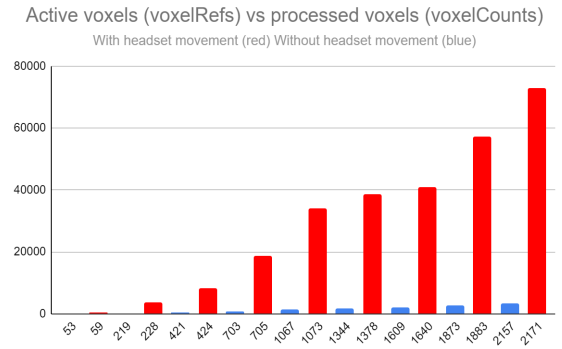

Figure 14: graph (active voxels vs computed voxels) comparing headset movement vs no headset movement.

Whenever there's no headset movement, the voxel counts remain relatively low, ranging from 53 to 2141 voxels. When there is headset movement, the voxel counts increase significantly, ranging from 502 to 72983 voxels. This makes complete sense as moving the headset automatically requires the algorithm to check completely new voxelated spaces.

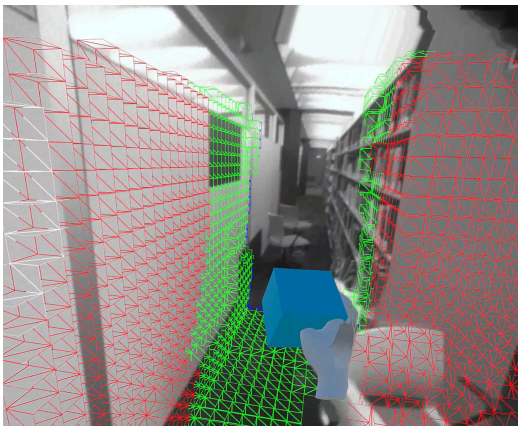

Figure 15: User before interacting with the environment

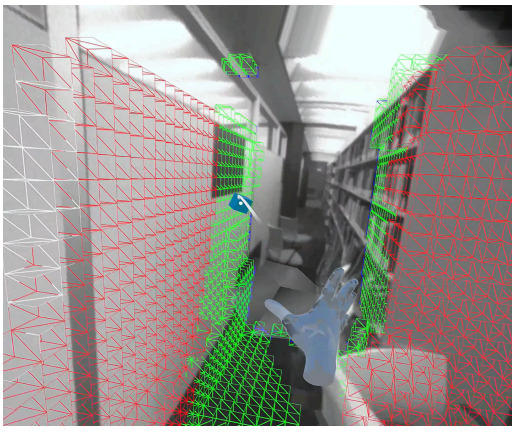

Figure 16: User after interacting with the environment (cube colliding with wall)

Finally, it is worth mentioning that to assess the effectiveness of the program from a user perspective. various virtual objects, such as spheres and cubes, were introduced into the voxelized environment, to test how they can interact and collide with the reconstruction that was generated.

4 Analysis

The voxelization approach proved to be an effective solution for real-time mixed reality applications, offering a fair balance between performance and accuracy. The algorithm's ability to handle sensor noise, combined with its computational efficiency, makes it well-suited for integrating virtual and physical environments seamlessly.

One of the key advantages of this approach is its simplicity and scalability. By dividing the 3D space into discrete voxels, the algorithm can efficiently represent complex geometries without the need for intricate surface reconstruction algorithms. This simplicity translates into improved computational performance, enabling real-time execution even on modest hardware configurations.

The initial work with randomly generated point clouds and convex hull calculation provided insights into the computational demands of accurate mesh reconstruction, highlighting the need for a more efficient approach. While valuable for understanding point cloud manipulation, the lengthy processing times made it unsuitable for real-time applications.

The failed “box generation” approach attempted to construct accurate meshes from the sensor data, similar to the initial work. However, the experiments revealed that this approach was not fast enough for real-time mixed reality, with processing times averaging around 60ms per box. Optimizations like parallelization or hardware acceleration could have improved performance, but the fundamental trade-off between accuracy and speed remained a limitation.

The experiments conducted with the Intel RealSense D435 sensor and the Oculus Quest 2 headset demonstrated the voxelization algorithm's capability to generate voxelized representations of the physical environment in real time. The processing time remained within acceptable bounds for smooth mixed reality experiences, with an average processing time ranging from 25ms to 34ms when the headset was stationary, and up to 115ms when the headset was in motion.

However, it is important to note that headset movement had a significant impact on processing time. This can be

attributed to the need for the algorithm to process and store voxel data for new areas as the user's perspective changes. As observed in the experiments, the number of voxels stored in voxelCounts increased substantially with headset movement, leading to higher processing times. Optimizations to manage and cull irrelevant voxel data more efficiently could potentially mitigate this performance impact.

The voxel size can be adjusted to achieve higher or lower resolutions, but smaller voxel sizes will result in increased computational demands. Finding the optimal balance between resolution and performance was crucial for delivering a seamless mixed reality experience. In recent years, when trying to use a voxelation approach for this problem, a large amount of memory is required for reconstructing a large-scale environment. (Chen et al., 2020)

Currently, when a scene is too large time efficiency goes down a lot, and when trying to fix this by increasing the size of the voxels to create less of them scenes tend to become inaccurate and sometimes can obstruct the user. After experimenting with the program it was concluded that the key element to make it accurate and efficient was to find the perfect balance for an average user, which meant finding the best relationship between voxel *size* and the point *threshold* required to render a voxel, while also keeping into consideration the value by which voxelCounts are reduced to delete unused voxels.

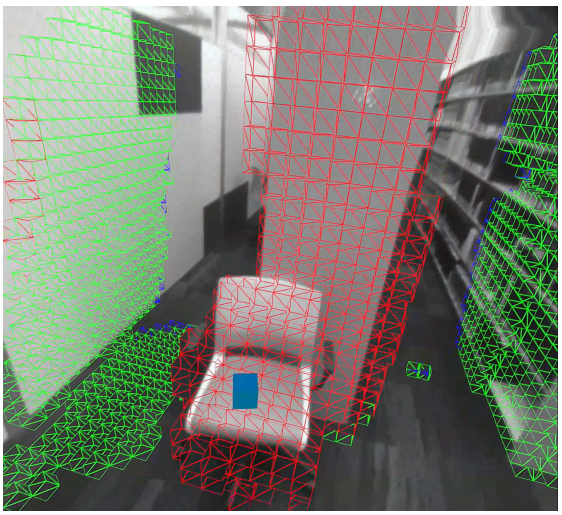

Figure 17: Interaction between the physical environment (chair) and a virtual element (cube).

5 Conclusion and future work

The research project successfully developed a system capable of dynamically capturing and integrating the physical environment into a mixed-reality experience. The voxelization algorithm, combined with the Intel RealSense D435 depth sensor and the Oculus Quest 2 headset, demonstrated the potential for seamless interaction between virtual and physical elements.

The voxelization approach proved to be an efficient and practical solution for real-time mixed-reality applications, showing that a balance between performance and accuracy is possible. By dividing the 3D space into discrete voxels, the algorithm can efficiently represent surfaces in environments, while handling occlusions and sensor noise, enabling real-time execution on modest hardware configurations.

While the current implementation shows promising results, there is still room for improvement and further exploration. One potential area for future work is optimizing the management and culling of irrelevant voxel data, particularly when the user's perspective changes and the headset moves. This could help mitigate the performance impact observed during headset movement and improve overall efficiency. Additionally, investigating techniques for dynamic adjustment of voxel size based on the complexity of the environment or user requirements could lead to more optimal trade-offs between resolution and performance.

Finally, two possible things that could enhance the overall quality of the mixed reality experience would be a better passthrough mode, which exists in newer headset models, such as the Oculus Quest 3 or the Apple Vision Pro. These devices offer more advanced passthrough capabilities, providing a more seamless integration of virtual and physical elements. Another potential improvement would be the ability to directly access and utilize the point cloud data collected by the headset's built-in sensors, instead of relying on an external depth sensor like the Intel RealSense D435. This would eliminate the need for additional hardware and could potentially lead to better performance and accuracy by using the specialized sensors and processing capabilities of the headset itself.

Acknowledgements

Thanks to Etienne Vouga for providing the equipment for this project!

References

[1] Cazamias, J., & Raj, A. S. (n.d.). Virtualized Reality Using Depth Camera Point Clouds - Jordan Cazamias Stanford University. Stanford University. https://web.stanford.edu/class/cs231a/prev_projects_2016/cs231a-project-report.pdf

[2] Chen, Y., Zhang, B., Zhou, J., & Wang, K. (2020). Real-time 3D unstructured environment reconstruction utilizing VR and Kinect-based immersive teleoperation for agricultural field robots. Computers and Electronics in Agriculture. https://www.sciencedirect.com/science/article/pii/S0168169920311479
[3] StellarX Blog. (2023, August 15). The slow rise of Mixed Reality. Enabling organic interactions between… | by StellarX | StellarX Blog. Medium. Retrieved April 26, 2024, from https://medium.com/stellarx-en/the-slow-rise-of-mixed-reality-c6e3ef5ad3dd
[4] J. P. Collomosse, "REAL-TIME ENVIRONMENT MAPPING FOR STYLISED AUGMENTED REALITY," The 3rd European Conference on Visual Media Production (CVMP 2006) - Part of the 2nd Multimedia Conference 2006, London, 2006, pp. 184-184, doi: 10.1049/cp:20061950. https://ieeexplore.ieee.org/abstract/document/4156036
[5] Chaudhary, Purushottam. “Point Cloud to Mesh Conversion - A Rookie Way.” Linkedin, https://www.linkedin.com/pulse/point-cloud-mesh-conversion-rookie-way-puru-chaudhary/.