Virtualized Reality Using Depth Camera Point Clouds

Jordan Cazamias
Stanford University
jaycaz@stanford.edu

Abhilash Sunder Raj
Stanford University
abhisr@stanford.edu

Abstract

We explored various ways to achieve Virtualized Reality, the technique of scanning a user's real-world surroundings and reconstructing it as a virtual scene. Using the Kinect and Intel RealSense depth cameras, we attempted both real-time and offline techniques to construct virtual scenes from real-world scenes, such as KinectFusion, point cloud stitching, and raycasted point clouds. In particular, the Kinect live depth feed and RealSense live depth feed were both rather effective as prototypes. In addition, we tested point cloud reconstructions of a complex object given various capture angles and found that point clouds closest to 0 or 180 degrees tend to work best. Given the opportunity to pursue the topic of virtualized reality further, we would ideally like to create a system that can capture parts of a scene in real-time, automatically stitch their point clouds together, and pipe this reconstruction into a game engine like Unity for instant visualization and the opportunity for interaction.

1. Introduction

1.1. Motivation and Problem Definition

The promise of Virtual Reality lies mainly in its ability to transport a user into a totally different world and make them feel present and immersed within it. That generally implies a total separation from the real world, and as such, a user's real-world surroundings are rarely (if ever) used as part of the virtual world. In contrast, *Virtualized Reality* captures a user's real-world surroundings and reconstructs it in 3D to use as the user's virtual world. It differs from traditional VR in that the worlds are not entirely fictional and a user's real-world surroundings matter. Likewise, while it sounds similar to Augmented Reality, Virtualized Reality differs in that everything is reconstructed from scratch. With these distinctions, a designer of Virtualized worlds has the opportunity to create powerful, personal experiences that would be difficult to replicate in either VR or AR. As an indirect benefit, most of the technology and techniques required for Virtualized Reality would also aid in the creation of higher-fidelity VR and AR experiences.

Our ideal system would be one that can continuously gather information about a scene, in real-time, as the user navigates about it while wearing a VR head-mounted display. As they interact with the virtualized objects, they will have real-time input through a positionally accurate reconstruction of their hands. Furthermore, the reconstruction of the entire scene should also be properly registered so that the virtualized objects are in the same position as they would be in the real world.

2. Related Work

The paper most relevant to our work is the paper on KinectFusion [3]. KinectFusion is a real-time 3D reconstruction algorithm and pipeline that uses the Kinect's depth camera to create a mesh of a static scene. It will be touched on in more detail later.

In the line of Virtualized Reality-specific work, Kanade and Rander experimented with virtualization by building a dome of 51 cameras that record synchronously at 60 frames per second. Any objects within the dome can be rendered at any different viewpoint using an interpolation algorithm, and even reconstructed in 3D. However, this system does not compute a reconstruction in real-time; it only stores the video data in real-time, and even the bandwidth required for this real-time storage is rather extreme [4].

By contrast, Vitelli et. al. use a server-based computer vision AR system to estimate the layout and lighting of a room from a smartphone camera in real-time, allowing for the augmentation of floors and walls as well as placement of virtual objects. This works quite well on a phone or tablet, but without true depth information of the scene, would not translate well to a VR display since stereo rendering would be impossible [6].

3. Our Approaches

We saw several different approaches to tackling this problem, each with its pros and cons, and explored each of them in turn.

3.1. Kinect

One of the depth cameras we used was a Kinect for Windows V2. It uses an infrared time-of-flight procedure to capture a depth map of a scene (effective range 50 - 450 cm), as well as an RGB camera to capture color information. Using these two pieces of information, one could easily create a point cloud or mesh that approximates the 3D scene as seen from the Kinect. With one such device, of course, any occluded regions will not be captured, resulting in artifacting such as holes or phantom surfaces (this happens in a naive mesh reconstruction when the holes caused by occluded regions are filled in with triangles). With a head-mounted sensor, this is not as much of a problem since the regions occluded to the sensor would be roughly equivalent to the regions occluded to the eyes. However, in the case of the Kinect, it is rather impractical to make it head-mounted due to its bulky form factor.

Figure 1: Kinect for Windows v2 sensor

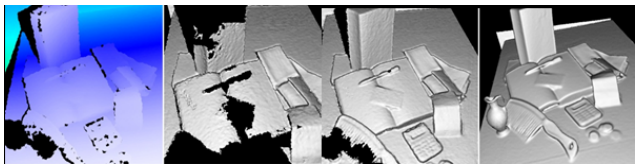
3.2. RealSense

We also experimented with an Intel RealSense SR300 depth sensor. With its short-range depth capture (effective range 20 - 150 cm) and much smaller form factor than the Kinect, the RealSense is perfect for mounting on the front of a VR headset and tracking the user's hands.

Similar to the Kinect, we used Unity to construct a dynamic mesh of the user's hands for every frame and rendered that into the virtual world. This presents the possibility for higher-fidelity inputs than controllers.

Figure 2: Intel RealSense SR300 depth camera

Figure 3: Example of KinectFusion reconstruction over time. Left: depth image from a single frame. Right: reconstruction results as more depth frames are integrated.

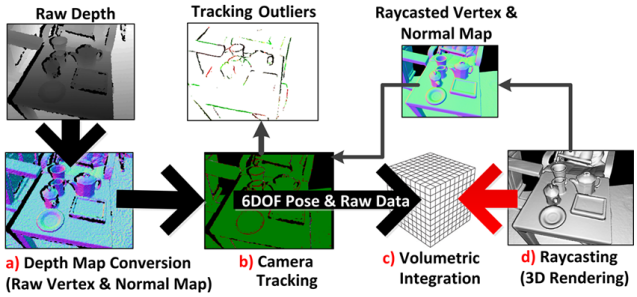


3.3. KinectFusion

As previously mentioned, the primary inspiration for this work was KinectFusion, Microsoft's 3D reconstruction / SLAM software for the Kinect. As a user moves the device around a small scene, the KinectFusion pipeline will take every depth frame captured by the Kinect and merge them into one high-fidelity 3D reconstruction, thus filling in occluded regions and eliminating most of the noise over time. At the same time, the Kinect sensor's position can be quickly and accurately tracked relative to the scene.

This lends itself quite well to virtualizing a scene in VR. Using a multi-frame reconstruction takes care of the occlusion and phantom surface problems seen in a one-frame reconstruction (as shown in Figure 7). It also allows the scene to remain fixed in virtual space as the virtual camera's position can be updated in real-time. Essentially, the Kinect serves as a "flashlight" to the real world, uncovering it for the user as it is pointed around.

Figure 4: The full KinectFusion pipeline. The two major outputs of the pipeline are a raycasted vertex and normal map, which can be used to create a 3D mesh, and the camera tracking parameters which is helpful for proper mesh registration with the real world



3.4. Point Cloud Stitching

One of the principal objectives of this project was to allow the user to scan in and reconstruct an entire room in VR. The KinectFusion[cite] system implemented on the Kinect for Windows Sensor provides real-time 3D object scanning and volume reconstruction. However, due to memory constraints, it cannot be used to reconstruct an entire room in one go. Therefore, to reconstruct a large real-world scene with high resolution, we will need to capture different parts of the scene separately and assemble them together after the fact.

We captured point cloud representations of different sections of the room (with some overlap) using KinectFusion. These point clouds were then merged using the Iterative Closest Point Algorithm.

Iterative Closest Point (or ICP) refers to a class of algorithms that try to find the transformation(i.e rotation and translation) between two point clouds. If the transformation between each point cloud and some reference point cloud is computed, all the point clouds can be transformed into the reference frame and then merged together to form a complete scene.

Here, we summarize two widely used algorithms, the standard ICP which was first described in [1] and its "point-to-plane" variant, originally introduced in [2].

3.4.1 Standrard ICP Algorithm

The standard ICP algorithm has two main steps which are repeated until convergence:

(i) Compute correspondences between the two point clouds.

(ii) Compute a transformation which minimizes the distance between the corresponding points

This works well only if there is a complete overlap between the two point clouds. However, in our case the overlap is only partial. Therefore, the algorithm provides a matching threshold, d_{max}. We only look for a match inside a sphere of radius d_{max}. This accounts for the case in which there are no correspondences for some of the points in the reference point cloud. The algorithm can be formally summarized as follows:

Given a *reference* point cloud $A = a_i$, a *moving* point cloud $B = b_i$ with partial overlap and an initial transformation T_0, the goal is to find a transformation T which best aligns B to the reference frame of A. This is done through the following steps:

1. Initialize $T = T_0$.
2. For each $b_i \in B$, find the point $c_i \in A$ which is closest to $T.b_i$.
3. If $\|c_i - T.b_i\| < d_{max}$, set $w_i = 1$. Else, set i$w_i = 0$.
4. Set $T = argmin \sum_i w_i \|c_i - T.b_i\|^2$
5. Repeat steps 2 to 4 till convergence

3.4.2 Point-to-plane

This is a variation of the standard ICP algorithm. The only difference is in the cost function in step 4. In this algorithm, the cost function used is:

$$T = argmin \sum_i w_i \|n_i.(c_i - T.b_i)\|^2 \qquad (1)$$

where n_i is the surface normal at c_i.

3.5. KinectFusion Over Network

One potentially interesting alternative to directly piping KinectFusion reconstructions to an engine like Unity would be to send this data over a network. For a user mapping their own room, packets could be sent over localhost. The possibilities get even more interesting when considering sending reconstruction data over a remote network. In this way, a user could experience the reconstruction of someone else's environment. Furthermore, with the addition of a dynamic reconstruction like DynamicFusion [5], it would be possible to teleconference with high-fidelity reconstructions of people in VR. This would serve as an alternative to using pose estimation techniques to find the pose of a person and animating a rigged model of a person.

4. Experimental Setup and Results

4.1. Experimental Setup

For long range scene capture, we used the Kinect for Windows sensor. The Unity Plugin provided with the

Figure 5: Experimental setup: The RealSense depth camera attached to the HMD assembled in the EE267 class

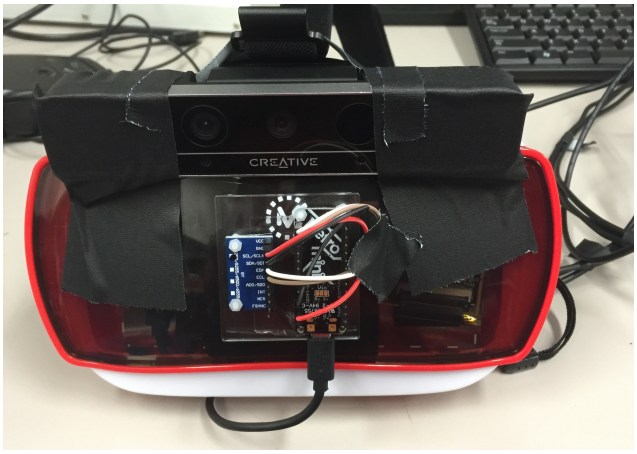


Figure 6: Experimental setup: The Foosball table used to test the point cloud stitching algorithm

Kinect SDK was used to transmit the RGB and depth images and visualize the resulting point cloud in Unity as shown in Fig 7.

For the purpose of hand visualization and tracking, we mounted the Intel RealSense depth camera on top of the HMD which we had assemebled in the EE267 class (Fig.5). The Unity Plugin provided in the RealSense SDK was used for interfacing with Unity.

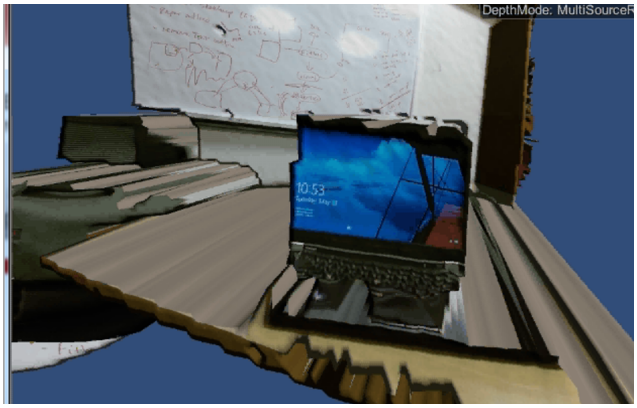
In order to test the point cloud stitching, we used the Foosball table (Fig.6) in Gates fifth floor lounge as a test subject. Using the KinectFusion system, we captured point clouds of the table from multiple orientations. Going around in a circle around the table, we captured 8 point clouds at equal angular intervals of 45 degrees.

4.2. Results

4.2.1 Real-time Scene Capture using Kinect

Given its effective range (50 cm - 450 cm), the Kinect sensor is ideal for scanning the user's surroundings into VR.

Figure 7: Live feed of Kinect depth & RGB values, reconstructed into a dynamic mesh in Unity. Note the phantom surface artifacting at the sides of objects where the Kinect could not capture depth information.



We used a plugin provided with the Kinect SDK to interface with Unity. This allowed us to transmit the live RGB and depth images captured by the Kinect sensor to Unity and reconstruct a dynamic mesh in real time. Since we only used the live RGB and depth information, the rendered mesh was only accurate in the absence of occlusions. In the presence of occlusions, Kinect cannot capture all the depth information which results in phantom surface artifacts at the edges of objects as seen in Fig. 7.
This problem can be rectified with the integration of KinectFusion with Unity, which is currently not possible due to software limitations.

4.2.2 Hand Visualization and Tracking Using RealSense

In contrast to the Kinect, the Intel RealSense is a short range depth sensor. This is ideal for VR applications like hand visualization, hand tracking and gesture detection. We used the Unity Plugin provided in the RealSense SDK to stream the live RGB and depth images from the sensor to Unity. The depth image was used to reconstruct a real-time dynamic mesh in Unity as seen in Fig. 8. It must be noted that the RealSense also supplies RGB data but this was purposely omitted due to the unsightly artifacts it produced at the edges of the hands. Also, this mesh is not only limited to hands but can capture and portray any real world object within 1m of the camera.

In addition to hand visualization, we implemented real time hand tracking (Fig. 9) using the hand tracking module provided in the Unity Plugin. This falls under the category of egocentric hand tracking, for which the state of the art algorithm uses CNNs. Here, neural nets were not used

Figure 8: Live feed of the RealSense depth values, reconstructed into a dynamic mesh in Unity. The RealSense also has RGB color information but we omitted it for the sake of this demo due to unsightly artifacts around the edges of the hands.

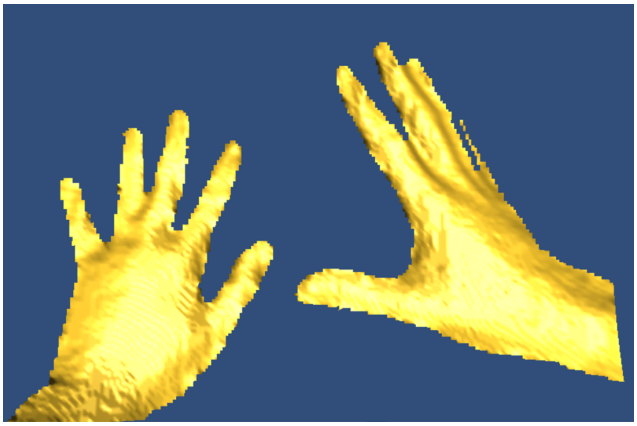


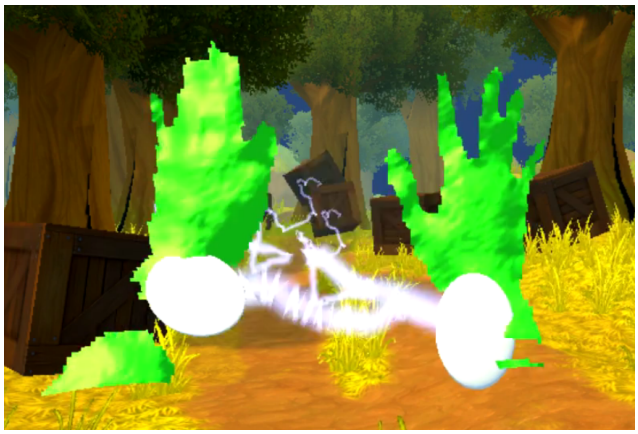
Figure 9: Real-time hand tracking in VR. The white discs attached to the hand are an estimate of the hands' centers. This is a screenshot of a live scene rendered in Unity

since we needed a real-time system. Even so, the hand recognition and tracking was very robust when only one hand was present in front of the camera. With both hands, the algorithm sometimes lost track of the second hand.

On top of this, we also implemented a simple gesture recognition system for the demo. The hand tracking module keeps track of the (x,y,z) coordinates of the hands. We used the Euclidean distance between the (x,y) coordinates of the hands as an action trigger in the scene. Whenever the

Figure 10: Real-time gesture recognition in VR. Our system uses the proximity of the hands as an action activation trigger in Unity. The green color of the point clouds indicates that it has been activated

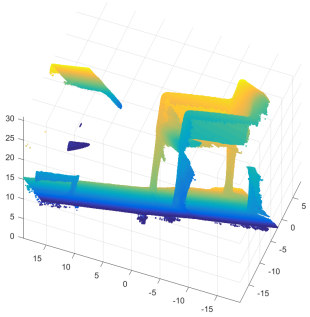
Figure 11: Interaction with the virtual environment. After activation, when the user's hands cross a depth threshold, they fire lightning at rigid bodies in the scene



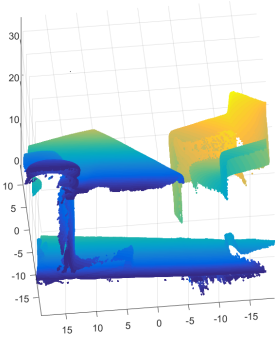
distance fell below the threshold, the point clouds turned green (indicating activation) and lightning was generated between the hands (Fig. 10). In addition, after activation, whenever the hands crossed a certain depth (z-coordinate) threshold, the user could shoot lightning into the scene and interact with it.

To achieve interaction with the virtual scene, we included a script which scans the view frustum of the user and finds all the rigid bodies within it. When such bodies

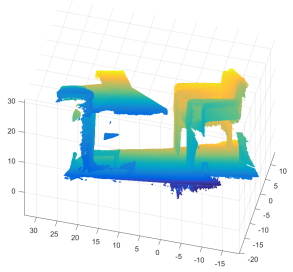
Figure 12: Example of Point Cloud Stitching with two partially overlapping point clouds



(a) Point Cloud 1



(b) Point cloud 2



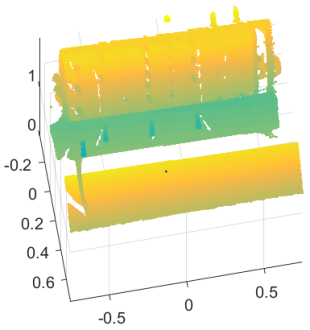
(c) Merged Point cloud

are detected, a lightning bolt is fired from the hands to the rigid body. We used the Unity Physics engine to apply a force on the rigid bodies whenever lightning struck them (Fig. 11).

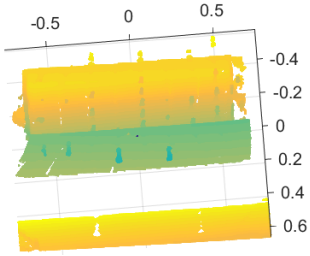
4.2.3 Point Cloud Stitching

The ICP algorithm with point-to-plane metric was implemented in MATLAB. For initial testing, we captured point clouds of different parts of the room using KinectFusion and then stitched them together offline. Currently, this step of the process is not real-time. Our initial foray into point

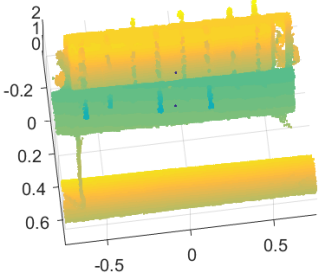
Figure 13: Stitching point clouds captured at an orientation of 180 degrees with one another. The two point clouds were captured from opposite sides of the table.



(a) Point Cloud 1



(b) Point cloud 2



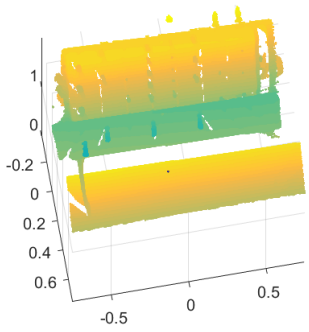
(c) Merged Point cloud

cloud stitching is presented in Fig. 12.

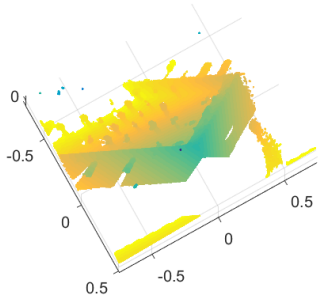
From our initial testing, we noticed that the ICP algorithm failed to compute the right transforms if the overlap between the point clouds was too small. From our observations, an overlap of 50% or above was required for the algorithm to merge the two point clouds correctly.

For further testing, we captured 8 point clouds of the Foosball table shown in Fig. 6 from multiple views. We went around the table in a circle and captured point clouds at an angular displacement of 45 degrees from one another. We made a few interesting observations while trying to

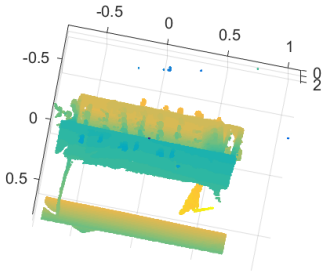
Figure 14: Stitching point clouds captured at an orientation of 45 degrees with one another. The two point clouds were captured while moving around the table, in a circle



(a) Point Cloud 1

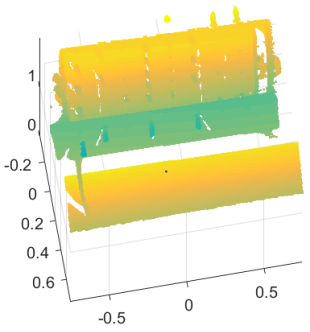


(b) Point cloud 2

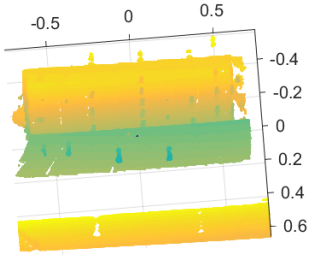


(c) Merged Point cloud

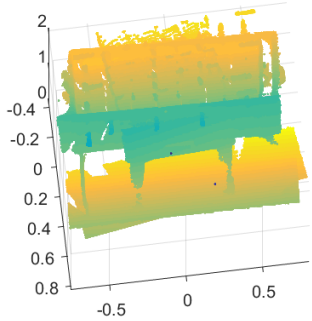
Figure 15: Stitching point clouds captured at an orientation of 90 degrees with one another. The two point clouds were captured from opposite sides of the table.



(a) Point Cloud 1



(b) Point cloud 3



(c) Merged Point cloud

fuse the point clouds together.

Firstly, the stitching algorithm worked very well when the inputs were point clouds captured from exactly opposite orientations (i.e at an orientation of 180 degrees). An example is shown in Fig. 13. The two points clouds were captured while looking in from the two long edges of the table. The algorithm seamlessly fuses the two to create a single point cloud containing features from both the inputs. Perhaps this effectiveness comes from the symmetric nature of the capture object; further study with more irregular objects could help to determine whether this is the case.

Fig. 14 shows our attempt to fuse point clouds captured at an orientation of 45 degrees to one another. While the algorithm did fuse the two correctly, the result is not as qualitatively good as in the previous case.
Finally, we tried fusing point clouds which were captured at orthogonal orientations, i.e one was captured looking in from the longer edge and the other from the shorter edge. As seen in Fig. 15, the output is completely skewed. We can clearly make out the two input point clouds which have been fused at an orientation of 90 degrees.
From this experiment, we see that the ICP algorithm is very sensitive to the initial orientation of the two point clouds.

When the input point clouds are at an orientation close to that in the real world (in our case, close to 0 degrees or 180 degrees), the algorithm works really well. However, when this orientation comes close to being orthogonal, the algorithm fails. This is because, in this case, it tends to converge and stagnate at a local minimum. Therefore, we can conclude that in addition to reasonable overlap, if the initial transformation T_0 is too far off from the ground truth, the algorithm ends up converging to a local minimum most of the time and the stitching fails.

5. Conclusion and Future Work

Ultimately, our various explorations of Virtualized Reality are first steps. Our ideal, real-time room-scale virtualization system is simply out of the scope of a class project, but we believe that with a follow-up to our work, it is certainly feasible.

Our goals for future development in this area would likely include the following:

- Real-time point cloud stitching: Since KinectFusion can only reconstruct a small volume, stitching together multiple reconstructions will still be an important part of scene capture. Ideally, we would like for users to be able to capture an entire scene in one take for maximum convenience. Performing the stitching in real-time also offers extra information that would be helpful for the ICP algorithm; for instance, if the current tracked position of the Kinect sensor were used to place the two point clouds initially, then these conditions would help the algorithm converge faster.

- Automatic point cloud registration: Once a full scene is virtualized, it still needs to be registered to (at least approximately) line up with the real-world scene. Currently, we have to manually set the transform for the scene. This task is especially laborious for the KinectFusion reconstruction, as the coordinate system is reset every time the reconstruction is reset. However, there is no one obvious solution for this, and performing this task alone would likely merit its own research paper.

- Improved integration with a game engine such as Unity: As it currently stands, the Kinect SDK is compatible with Unity but KinectFusion is not. Finding a way to port the KinectFusion capability, either through an open-source alternative like PCL or waiting for Unity to upgrade its .NET capability, would add an important layer onto reconstruction projects such as ours. This has the capability of becoming an important plugin into Unity and making virtualization far more accessible for developers.

- Experimenting with different depth cameras: For example, using both the short-range and long-range Intel RealSense cameras could help solve many of the problems with using KinectFusion and even help tackle the registration problem (since the depth cameras would be in a locked position relative to the headset).

Acknowledgements

Thanks to Gordon Wetzstein and Robert Konrad for providing the equipment for this project!

References

[1] P. J. Besl and N. D. McKay. Method for registration of 3-d shapes. In *Robotics-DL tentative*, pages 586–606. International Society for Optics and Photonics, 1992.

[2] Y. Chen and G. Medioni. Object modelling by registration of multiple range images. *Image and vision computing*, 10(3):145–155, 1992.

[3] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, et al. Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 559–568. ACM, 2011.

[4] T. Kanade, P. Rander, and P. Narayanan. Virtualized reality: Constructing virtual worlds from real scenes. *IEEE multimedia*, (1):34–47, 1997.

[5] R. A. Newcombe, D. Fox, and S. M. Seitz. Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 343–352, June 2015.

[6] M. Vitelli, S. Dasgupta, and A. M. Khwaja. Synthesizing the physical world with the virtual world: Computer vision for augmented reality applications.