

Dynamic Rendering of Remote Indoor Environments Using Real-Time Point Cloud Data

Kevin Lesniak

Industrial and Manufacturing Engineering
The Pennsylvania State University
University Park, PA 16802
Email: ka15544@psu.edu

Conrad S. Tucker

Engineering Design, Industrial and Manufacturing
Engineering
The Pennsylvania State University
University Park, PA 16802
Email: ctucker4@psu.edu

ABSTRACT

Modern RGB-D sensing systems are capable of reconstructing convincing virtual representations of real world environments. These virtual reconstructions can be used as the foundation for Virtual Reality (VR) and Augmented Reality (AR) environments due to their high-quality visualizations. However, a main limitation of modern virtual reconstruction methods is the time it takes to incorporate new data and update the virtual reconstruction. This delay prevents the reconstruction from accurately rendering dynamic objects or portions of the environment (like an engineer performing an inspection of a machinery or lab space). The authors propose a multi-sensor method to dynamically capture objects in an indoor environment. The method automatically aligns the sensors using modern image homography techniques, leverages Graphics Processing Units (GPUs) to process the large number of independent RGB-D data points and renders them in real-time. Incorporating and aligning multiple sensors allows a larger area to be captured from multiple angles, providing a more complete virtual representation of the physical space. Performing processing on GPU's leverages the large number of processing cores available to minimize the delay between data capture and rendering. A case study using commodity RGB-D sensors, computing hardware, and standard TCP internet connections is presented to demonstrate the viability of the proposed method.

1 INTRODUCTION

A human's sense of sight is largely responsible for gaining the information necessary to understand the physical objects in an environment and how they relate to each other [1]. A person's sense of sight, and their ability to manipulate their perspective in an environment, provides a sense of "presence" in the environment [2]. This "presence" enables an individual to interpret the environment around them in a natural manner, utilizing their existing knowledge and past experiences to gather information and gain an understanding of the objects and interactions they are viewing [3]. Mesh surface reconstructions of physical environments allows users to view a virtual version of a physical location while using a VR system. However, challenges arise when these methods attempt to mimic the dynamic nature of real world environments. Due to the large amount of data that can be captured by commodity RGB-D sensors, reconstructing meshes from incoming frames of data can take an extended period of time. This delay between the data being captured and the reconstruction being available for rendering limits the usefulness of surface reconstructions in representing active or dynamic real-world locations. A large collection of independently colored and rendered data points, a colored point cloud, serves this purpose more effectively by limiting the amount of complex computation needed to transform the raw RGB-D data into a format that can be rendered and viewed in a virtual environment. When it comes to engineering or design environments, being able to capture and visualize the dynamic portions of environments, like people and machinery, provides crucial information to remote users.

Commodity RGB-D sensors allow large amounts of data to be collected about physical environments and displayed as virtual reconstructions. Leveraging the advancements in RGB-D sensors, researchers have been able to provide accurate virtual reconstructions of real-world environments [4]–[8]. While these reconstructions provide useful information and experiences, they are limited to static environments, ones where the objects and features of the environment have little to no change over time. An example of a static environment would be an empty workshop, as the objects in the environment will not typically move on their own. The process of converting the 2D depth image captured by the sensor into 3D points and then incorporating those points into a virtual surface reconstruction takes orders of magnitude more time than what is available between frames of data. This extended reconstruction process makes it extremely challenging to perform surface reconstructions while capturing and updating data at the rate of the sensor. More specifically, existing methods are unable to capture dynamic objects in the scene, like a person or object moving through the scene, nor are they able to stream the captured data to a remote location in real time. Such limitations make it difficult to mitigate the challenges of information loss experienced within distributed design teams when translating 3D data into video or voice communication mediums.

In addition to the need for the dynamic translation of a physical environment to its virtual representation, attempting to view the results on a standard 2-dimensional (2D) display sacrifices the ability to immersively view the environment. The advent of affordable VR systems, such as the Oculus Rift [9] and the HTC Vive [10], provides a solution to this problem in the form of an immersive display that enables the user to gain the sense of “presence” that they would have when physically present in the environment. This is accomplished by providing the user with a head-mounted display that tracks their real-world motion and renders their view of the virtual environment from the perspective of the headset [2]. The piece that is missing is the virtual environment itself. An Oculus Rift [9] or an HTC Vive [10] can be used as the display, but the creation of the objects to be rendered is the real challenge, due to the large size of the captured data and the rate at which new frames are made available from the sensor systems. The method proposed in this paper provides a process for the creation and distribution of these objects that are made accessible in immersive VR headsets.

Without a realistic and immersive representation of a distant environment, users do not have the sense of “presence” that allows them to use the information they are gathering to gain an understanding about specific objects or interactions in that environment [2]. This creates a difficult problem when working with a distributed design team, since collaboration on design requires that designers understand the environment, objects, and interactions being designed or designed for. When trying to provide this immersive experience to users who are not physically present at the environment of interest, challenges arise relating to the data necessary to construct an accurate virtual reconstruction and how to get that data to the user for viewing.

Table 1: Data Sizes

	Width	Height	Bytes/Pixel	Total Data	FPS
Depth	512	424	2	434 KB	30
Color	1920	1080	4	8.3 MB	30

Table 1 shows the sizes of the data that is available from a commodity RGB-D sensor, for example the Microsoft Kinect for Windows v2. The image data that is available is a high definition color image and a smaller resolution depth image. The color image can be as large as 1920x1080 pixels with 4 bytes of data to represent the color in each pixel. The depth image has a resolution of 512x424 pixels with only 2 bytes of data to represent the depth at each pixel location. That is ~8.3 MB for each color image and 434 KB for each depth image, ~8.8 MB total. This sensor outputs these images at 30 frames per second, meaning that there is a total of 264 MB per second worth of data being collected by the sensor. While this is only an example, similar sensors have similar volumes of data available. With an average broadband connection speed of 6.9 MB/s in the United State [11] the amount of data collected from a single sensor is too much to be transmitted on standard internet connections. This is a problem that needs to be addressed to be able to provide an immersive experience to remote individuals.

The method presented in this paper allows multiple remote users to view a realistic representation of a physical environment in real-time using an immersive VR system. The RGB-D data from multiple RGB-D sensors is automatically aligned to construct a large dataset in a universal coordinate system. The RGB-D sensors are all independent of each other, allowing for variation in the number of sensors that can be set up, based on environment size and coverage needs. Automatic sensor alignment increases the accuracy and consistency of the alignment while also reducing the need for user interaction in the alignment process. Having multiple sensors that are properly aligned allows the capturing of data about the physical environment from multiple perspectives. This helps to reduce the issues that arise when a sensor’s view becomes obstructed by a person or object moving through the scene, as well as extending the volume of space that can be reconstructed by overlapping the field of view of multiple sensors. Providing the ability to render the large dataset captured by the sensors into a VR system, gives remote users a sense of “presence” and allows them to gather information about the environment in an immersive and realistic manner. By mimicking the way in which a human views and interprets the physical world in a VR system, a realistic and immersive distributed collaboration system is created. The distributed nature of the proposed method allows the sensors, the central server, and the remote individuals to all be independent of each other, allowing for flexibility in hardware, location, and customization to meet specific needs of the environment or user. This distribution of the proposed method utilizes data processing and compression methods to minimize the amount of data that is necessary to transmit between the

various components. This enables the method to be run on standard network connections without overloading or exceeding the available bandwidth.

The content of this paper is based on our previous work presented at the ASME IDETC 2017 Conference. This paper is organized as follows. This section provides background information on VR systems, RGB-D sensors, and the virtual reconstruction of real-world environments. Section 2 presents existing literature that relates to this work and discusses their capabilities and limitations. The details of the proposed method are presented in Section 3, while Section 4 describes a case study that demonstrates the viability of the method. Section 5 discusses the results of the case study with a conclusion and potential future work in Section 6.

2 LITERATURE REVIEW

2.1 Digital Reconstruction of Physical Environment

Reconstructing physical environments in a virtual space has been accomplished by groups in the past. Many of these groups focused on capturing static data about a scene to create a 3D reconstruction that can be used as the basis for a virtual application or for 3D modeling. Newcombe et al. leveraged one of the first affordable RGB-D sensors to create the KinectFusion library [6]. This library allows users to take the data from an RGB-D sensor and use it to construct a virtual representation of the physical environment that is seen by the sensor. One of the challenges that these papers do not address is capturing data for dynamic objects in the environment,

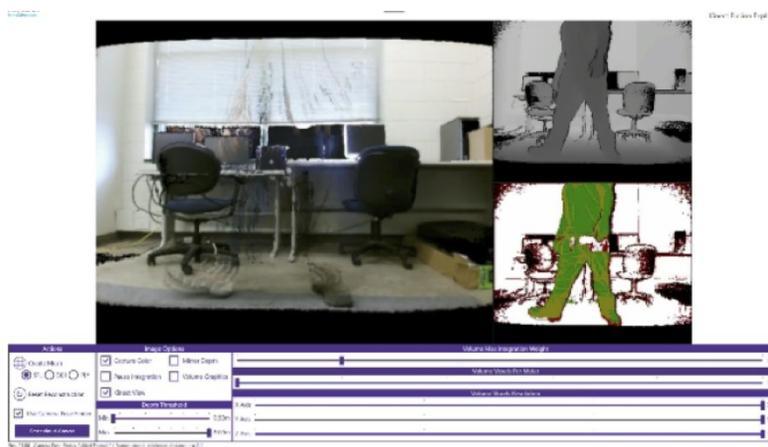


Figure 1: KinectFusionExplorer Missing Dynamic Data

and efficiently relaying that information to users. Figure 1 shows a sample of the KinectFusionExplorer [7] application scanning an environment. On the right-hand side of the figure, two images are displayed showing the data that is being collected by the sensor. In these images, a person is visible moving in front of the sensor. The large colored image on the left of the figure is a rendering of the reconstruction created by the KinectFusion library [6]. While the static objects in the background can be clearly seen in the reconstruction (left), the individual moving in the scene is only displayed as a faint outline. The lack of representation for dynamic

objects in the environment limits this library's usefulness when trying to collaborate on design problems or team-based work in real time.

Turner et al. [4] sought to expand the functionality of the KinectFusion library [6] by extending the size of the area capable of being scanned and reconstructed. Turner et al. [4] created functionality allowing for the position of the sensor to be tracked while it moves through a larger space. This extended the range of the sensor and allowed for large spaces to be recreated in virtual environments. While they were able to achieve their goal of extending the size of the reconstruction volume, they faced challenges when it came to dynamic scenes and objects. The sensor only captures data for the area it is currently viewing. If the environment changes after the sensor has moved away from an area (e.g., as in Figure 1), those changes are not recorded, and that data is lost. There are also issues that arise when an object is moving in the sensor's field of view. Since the sensor is using feature detection and changes between frames to track its movement, other dynamic objects in the environment can cause problems or failures in the tracking, resulting in the system not being able to properly integrate new data into the reconstruction.

Hamzeh et al. [5] used the Microsoft Kinect to allow a robot to navigate in indoor environments and construct a floor-plan of the area. The authors utilized the depth data being acquired by the sensor to determine what areas were traversable and the location of the robot in the environment. These areas were converted into a 2D representation and transmitted back to a user so that the user could "see" what the layout of the indoor environment was. While this process utilizes the data from an RGB-D sensor and transmits the results to a remote user, there is still data lost in the conversion from 3D space to a 2D image to be transmitted. With just the floorplan view of the environment, the remote user is not able to immersively view the results to gain information about the environment and form understandings of the objects and interactions it contains.

Lesniak et al. [8] sought to allow remote users to view an accurate virtual reconstruction of a physical environment. They achieved this by utilizing the KinectFusion library [6], which they modified to allow various components of the process to occur at different remote locations on separate hardware. This allowed for the data to be collected in one location, transmitted to a second location where it was processed into a 3D mesh, and finally transmitted to a remote user to be rendered in a VR system. While these authors achieved many of the same goals as the method proposed in this paper, their system is still limited to only capturing data about a static environment. Any dynamic objects or interactions in the scene are discarded during the process to convert the raw data into a 3D mesh. This is an inherent limitation from utilizing the KinectFusion library [6] for the mesh reconstruction step. The second limitation is that the authors were only working with one sensor's data. This limits the area that can be captured at one time to the field of view of a single sensor and leaves potential for gaps or occlusion issues in the environment being viewed by the sensor, an issue that can be reduced or eliminated when combining multiple data sets.

2.2 Multiple RGB-D Sensor Data Alignment

Yang et al. [12] sought to extend the functionality of the Microsoft Kinect in a different way by combining the data collected from multiple sensors into one large dataset. They used the original Kinect for Windows sensor and proposed a process for accurately aligning the data collected from each sensor. Their main goal was to use multiple aligned sensors to quickly and accurately capture data about a specific object. Since all the sensors are focused on a specific object or point, the physical volume being viewed by the sensors is limited. While the proposed methods are useful in constructing a system with multiple aligned sensors, other use cases need to be explored to see the full potential of this kind of system.

Asteriadis et al. [13] also worked on using multiple RGB-D sensors, but in a different scenario than Yang et al. [12]. Asteriadis et al. [13] sought to use multiple Kinect for Windows sensors to capture the motion of an individual performing an activity. One example they focused on was someone running on a treadmill. Their method focused on using multiple sensors to solve the problems of self-occlusion and partial reconstruction that are encountered when only using a single sensor. Again, this group had the sensors focused on a small space and were mostly concerned about gathering data on human motion. This method does not address the possibility of larger spaces, like a classroom or meeting room, and is limited in its ability to capture accurate 3D data about other dynamic objects in the environment. While this group was focused on tracking human motion, they were not concerned with the context of that motion or other objects that may be involved in the motion or interaction.

Automatic sensor alignment stems from Harris et al. [14] and their work on a corner and edge detection algorithm for 2D images. This work provided an algorithmic basis for comparison between images, which is necessary if one is trying to generate a function to transform points between the coordinate systems of multiple sensors or cameras. Many groups have built upon or expanded this work to achieve results tailored to specific goals. In 2009, Dubrofsky [15] provided an analysis of the RANSAC family of algorithms used for homography estimation. This analysis discusses the increased accuracy and efficiency of RANSAC algorithms and did a comparison of available software packages that provide this functionality. To achieve automatic sensor alignment in the proposed method, the Accord.Net [16] library, a C# wrapper for OpenCV [17], was used to do the RANSAC based homography estimation between images captured from each sensor in the system.

Table 2: Literature of supported features

		Features				
		3D Reconstruction	Real-Time Rendering	Network-Distributed Processing and Rendering	Dynamic Object Representation	Automated Multi-Sensor Alignment
Authors	Newcombe <i>et al.</i> (2011)					
	Turner <i>et al.</i> (2015)					
	KinectFusionExplorer					
	Yang <i>et al.</i> (2013)					
	Garret <i>et al.</i> (2017)					
	Lesniak <i>et al.</i> (2016)					
	Ni <i>et al.</i> (2017)					
	Su <i>et al.</i> (2017)					
	Proposed Method					

Table 2 shows the features available in existing solutions. These solutions provide the ability to reconstruct large areas from RGB-D sensor data. There are also solutions for utilizing multiple RGB-D sensors in the same environment to collect large amounts of data. What is lacking from these existing solutions is a way to capture data for highly dynamic objects and environments and display this data to remote individuals in real-time. The method proposed in this paper accomplishes the goal of providing real-time data about a dynamic environment to remote individuals for them to view in an immersive VR system, a goal that has not been achieved to the best of our knowledge.

2.3 Point Cloud Manipulation and Rendering

The popularity and accessibility of VR systems, advanced rendering hardware and GPU's has led many groups and organizations to pursue advancements in capturing and rendering multiple RGB-D datasets to use as virtual representations of real-world locations and phenomenon. Ni et al. [18] proposed a method for combining multiple RGB-D datasets as point clouds for robotic teleoperation. Their work focused on providing information on the physical artifacts around a known robotic system. The method they proposed uses a virtual model of the known robotic component and displays RGB-D point clouds relative to the location of the robotic component. This method allows for a high-quality model of the robotic component combined with real-time RGB-D data sets. This system is limited to known robotic components to fully utilize the rendering capabilities and is limited in the size of the environment of interest

to what can be aligned to the robotic component using depth-point alignment. This can lead to issues when dealing with sparse or noisy depth data.

Kim and Park [19] focused their work on developing an algorithm to fit haptic surfaces to point clouds captured with RGB-D sensors. The method proposed by them incorporates the use of a haptic device to allow the user to have a sense of interaction with the physical components of the virtual environment. Their work advances interaction methods for virtual environments, but does not work to overcome the challenges with capturing data for large environments, minimizing network resource usage when dealing with multiple streams of sensor data or automated alignment of multiple sensor systems.

Su et al. [20] proposed a method for rendering point cloud data captured from multiple RGB-D sensors on a virtually curved mirror. Their goal was to develop a system that would allow users to view 3D environments realistically on a standard 2D display when combined with eye-tracking technology. Their method aligns data sets from multiple sensors using a colored sphere to register known points across multiple sensors. Their method also triangulates surfaces from the captured RGB-D data and then uses these triangles to ray-trace reflections on the virtually curved surface. The method proposed by these authors still requires a known object to be used for sensor alignment and requires that all the sensors can see the known object when placed in the scene. The method also incorporates more complex processing on the RGB-D data, triangulation and raytracing, before rendering for the user, an added bottleneck when trying to minimize the hardware and processing requirements for the system.

Garrett et al. [21] sought to use the point cloud data captured from RGB-D data to identify and track real-world objects that are visible to the sensor. They proposed an algorithm for performing a global mesh reconstruction based on captured RGB-D data and known point clouds of objects. While this method has the potential to provide more realistic virtual reconstructions by providing a high-quality mesh representation of the environment, it is limited to only identifying objects where a point cloud representation of the object is previously known. This makes the proposed method less generalized and can lead to problems if a known object is manipulated, deformed, or altered in the real-world. This method also adds a complex processing step by using an Iterative Closest Point algorithm whose computation time scales with the size of the incoming RGB-D data set. With the inclusion of multiple sensors, and therefore multiple data sets, the hardware requirements to continue to process the incoming data sets in real time increases. This would increase the overall cost of the system, possibly prohibiting its use.

2.4 Rendering Engine and VR Systems

To be able to display the collected data, a rendering engine is necessary. For interactive 3D applications, Unity [22] and Unreal [23] are two of the most popular. Both of these engines are readily available to consumers, and support various platforms, such as Windows, OSX, and Linux. In addition, various VR system are supported as build targets such as the Oculus Rift [9] and HTC Vive [10]. Supporting these platforms and systems enables the proposed method to be customized to meet an individual user's needs. If they have a VR system or a particular platform available, components of the proposed method can be changed to support those needs. The authors chose to use Unity [22] and the HTC Vive [10] for the case study due to ease of access and support for the hardware with which the case study was performed. These components could be changed as outlined above to meet a specific team's needs.

3 METHOD

The method presented in this section outlines the steps necessary to align multiple RGB-D sensors into a universal coordinate system and create a virtual representation of the data that can then be accessed by multiple users. This method allows for communication between the physically present and remote individuals in real-time and presents the virtual environment in an immersive and realistic

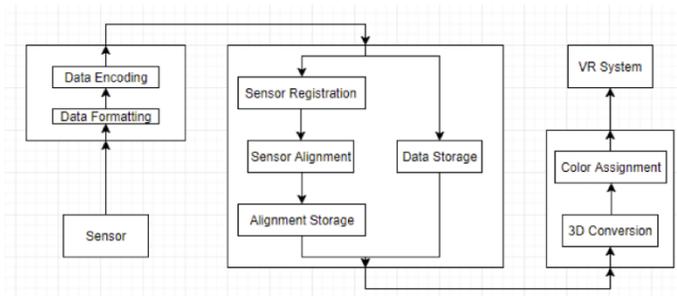


Figure 2: Proposed Method

manner. An outline of the proposed method can be found in Figure 2. Step 1 begins with the alignment of multiple RGB-D sensors and the data collected from them.

3.1 Sensor Alignment

To accurately represent the data that is being acquired from multiple RGB-D sensors, the relative positions and orientations of the sensors need to be calculated. Without correctly calculating the positions and orientation of each sensor, the data sets will not be properly aligned when rendered together. Figure 3 shows a sample of when two data sets are not properly aligned. In this figure, the



Figure 3: Two Mis-Aligned Datasets

there is a clear gap in the data along the boundary of two datasets. This is due to incorrect positioning and orienting of the data that is coming from each sensor. Correct alignment of the sensors is achieved on the server by first sending an initial image from each sensor to the server. The server relates this image to the unique identifier it assigns to each sensor when it first connected. The Accord.Net [16] library is then used to estimate the homography between each pair of images. The homography represents a transformation matrix that changes pixels in the coordinate system of one image into pixels in the coordinate system of another image. This homography is estimated by first converting the image to greyscale and then using an edge-detection algorithm to find feature points. These feature points are found by comparing the results of the edge detection algorithm for the two images to determine common edges, and a transformation matrix is constructed to represent the translation and orientation change between sensors.

This homography allows the authors to align the data from each sensor by transforming it into the coordinate system of an “origin” sensor. The server assigns the first sensor that connected as the “origin” sensor, with each other sensor having an offset in the physical world from the origin. These offsets are stored along with the sensor’s unique identifiers so that they can be transmitted to any clients that connect. This allows the sensor to transmit data to the server, along with the sensor’s unique identifier. The server can then correctly distribute the RGB and depth images from all sensors to any clients it has connected. This ensures that the data collected from each RGB-D camera is aligned correctly in a universal coordinate system.

3.2 Formatting Sensor Data

The RGB and depth data collected from the sensor are not necessarily the same resolution. This is due to differences in the RGB and depth camera technologies. In many cases, like the Kinect for Windows v1 and v2, the depth camera is a lower resolution than the RGB camera. This difference in resolution and the offset between the RGB and depth cameras need to be accounted for. This is done

by mapping the RGB data into the depth data's coordinate system. Using the internal offset between the depth and color cameras and intrinsic values for focal length, field of view, and distortion for each camera, a pixel location in the depth image can be matched with a pixel location in the color image. This process is outlined in lines 7-9 of Algorithm 1. It begins by receiving both the depth and color image from the sensor. Then, each color pixel is checked to see if it maps to a pixel in the depth image. If the color pixel has a corresponding depth pixel, it is then added into a new image, the down-sampled color image.

Algorithm 1: Capturing and Formatting Sensor Data	
Input:	<i>C</i> -> RGB image as byte[] <i>D</i> -> Depth image as ushort[] <i>ID</i> -> Sensor identifier
Output:	<i>ID</i> -> Sensor identifier <i>cE</i> -> Camera alignment data as compressed byte[] <i>cA</i> -> Down-sampled RGB image as compressed byte[] <i>cB</i> -> Depth image as compressed byte[] <i>cC</i> -> Color image as compressed byte[]
	<ol style="list-style-type: none"> 1. Initialize sensor; 2. Initialize output network connection, <i>O</i>; 3. Receive <i>ID</i> from <i>O</i>; 4. Receive <i>C</i> from sensor; 5. Compress <i>C</i> into byte[], <i>cC</i>; 6. Send <i>cC</i> over <i>O</i>; 7. Receive <i>C</i> and <i>D</i> from sensor; 8. For <i>pixelC</i> in <i>C</i> do <ol style="list-style-type: none"> a. If <i>D</i> matches <i>pixelC</i> <ol style="list-style-type: none"> i. Add <i>pixelC</i> to <i>A</i>; 9. End 10. Convert <i>D</i> into byte[], <i>B</i>; 11. Compress <i>A</i> into compressed byte[], <i>cA</i>; 12. Compress <i>B</i> into compressed byte[], <i>cB</i>; 13. Send <i>ID</i>, <i>cA</i> and <i>cB</i> over <i>O</i>; 14. Repeat 7-13 for new images;

Figure 4 shows the results of the down-sampling process. In the top right is a full resolution color image. In the top left is a full resolution depth image. Since the resolution of the depth image is smaller, there are a large number of unnecessary pixel values in the full resolution color image. The down-sampling process removes these unnecessary pixels by aligning each depth pixel with a matching pixel in the color image. Only the pixels from the full resolution color image that have a corresponding pixel in the depth image are saved to be transmitted and processed by the proposed method. The down-sampled color image is shown in the bottom right.

The down-sampling process achieves two things. First, it reduces the RGB data to only what is necessary to provide a value for each depth pixel. This reduces the necessary bandwidth for the networked components of the proposed method by a factor equal to the ratio of the depth and color images' resolutions. For the Microsoft Kinect, it reduces the size of the transmitted color image from



Figure 4: Image Down-sampling

8.3 MB to ~ 870 KB, before compression. Reducing the bandwidth requirements allows for a more stable connection between the networked components while running on average or below-average internet connections. Second, it brings the depth and RGB data into the same coordinate system so that they can be accurately rendered together. The RGB value for each depth point will be used in the rendering process to display the entire scene captured by the sensors in full color. This provides a more accurate representation of the space than just the object surfaces provided by the depth image. Before being transmitted to the server, the depth and down-sampled color images are compressed to further reduce the bandwidth utilization. The process for capturing and formatting data is outlined in Algorithm 1.

3.3 Data Alignment and Conversion

Once the data has been formatted properly, each sensor transmits its data to a central server. This server acts as a gateway for the remote users to connect to and receive data. The server provides a single point for all the data from the sensors to pass through. This allows the clients to only need a single connection to be able to receive all available data from multiple sensors. Whenever a new sensor is connected, the server is able to update all clients with the new configuration information. The server can also send a new client configuration information for all connected sensors to prepare it for the data it will receive. Not having a central server forces every client to know about every sensor that is outputting data, without a way to ensure that all the connections are maintained while sensors and clients are added and removed. Having a central server also provides the proposed method with more flexibility on saving or storing data, managing connections and authorization to view the data, and possible extensions or optimization to the method or the data processing.

Algorithm 2: Central Server Data Transmission

Input: *ID* -> Sensor identifier
cA -> Down-sampled RGB image as compressed byte[]
cB -> Depth image as compressed byte[]
cC -> Color image as compressed byte[]

Output: *ID* -> Sensor identifier
cE -> Camera alignment data as compressed byte[]
cA -> Down-sampled RGB image as compressed byte[]
cB -> Depth image as compressed byte[]
rP -> Render Position as compressed byte[]

1. **Listen** for input network connection;
2. **Initialize** input connection, i_1 ;
3. **Send** new *ID* over i_1 ;
4. **Repeat** 1-3 for new input connections, $i_2, i_3 \dots$;
5. **Listen** for output network connection;
6. **Initialize** output network connection, o_1 ;
7. **Send** *rP* as compressed byte[];
8. **Repeat** 5-7 for new output connections, $o_2, o_3 \dots$;
9. **Receive** *cC* from $i_1, i_2 \dots$;
10. **Align** all *cC*'s relative to *cC* from i_1 and store in *cE*'s;
11. **Send** *ID* and *cE* over o_1 ;
12. **Repeat** 11 for $o_2, o_3 \dots$;
13. **Receive** *ID*, *cA* and *cB* from i_1 ;
14. **Send** *ID*, *cA* and *cB* over o_1 ;
15. **Repeat** 14 for $o_2, o_3 \dots$;
16. **Repeat** 13-15 for $i_2, i_3 \dots$;
17. **Repeat** 13-16 while connections are active;
18. **End**;

Dynamic objects in the scene are represented by colored 3D points that are continuously updated. This representation allows for highly dynamic objects, like people moving in the environment, to be accurately represented without requiring excessive processing power to track the objects' movements. The server is responsible for making sure that the incoming data frames were captured at relatively the same time. This is to ensure that dynamic objects in the scene are displayed correctly when they appear in more than one sensors' field of view at the same time.

Algorithm 2 shows the process the data server goes through to correctly connect to multiple sensors and clients. Whenever a new sensor attempts to connect, it is given a unique identifier. This identifier is used to keep track of where each frame of data originated, as multiple streams of data pass through the server and on to the clients. This initialization step is shown in lines 1-4 of Algorithm 2. The server also receives a full resolution color image from the sensor so that the server can align the new sensor to the rest of the connected sensors in the system. This alignment contains an offset in position and rotation from a universal origin point to ensure that all incoming data can be represented in the same coordinate system. Lines 9-12 of Algorithm 2 show this step.

Whenever a new client connects it is given a position in the virtual environment to render from. It is also sent the offset data and corresponding unique identifiers for each sensor that has connected to the server. This offset data is stored as alignment data for each sensor that is connected to the server. A starting position in the virtual environment for the client to begin rendering from is also

sent. This render position allows the clients to be placed in specific locations, if that is desired, or to spread out multiple clients so that they can see how each client is moving through and viewing the environment.

As data is received by the server from the connected sensors, the server forwards this data out to each client that is connected. The data that is received contains a depth image, a color image, and the unique identifier of the sensor where the data originated. This data is stored on the server until the next frame of data is received from the corresponding sensor. Lines 13-17 of Algorithm 2 outline the process for receiving data from the sensor applications and forwarding them to connected clients. Using these components, the rendering application can correctly place and render the data for the remote user to immersively view in the connected VR system.

3.4 Real-Time Rendering

Each remote user connects to the central server to receive the data that is being collected by the RGB-D sensors. The clients only need to maintain a single connection, to the central server, where they will be receiving all the incoming sensor data. The remote users are able to view the data received from the central server using a VR system. This allows the data to be presented in a 3D virtual environment and viewed using an immersive VR system, providing a more realistic medium for the user to view the environment and gain knowledge and understanding of the objects and interactions being represented. The remote user is then able to manipulate their viewpoint and perspective in the environment as it is rendering real-time data. To maintain a comfortable viewing experience for the user in the VR system, a high framerate, minimum of 30 frames per second with a recommended 60 frames per second, needs to be maintained for the duration of the experience [24].

The process that runs on the client is outlined in Algorithm 3. When a remote user first connects to the server, they receive a reference position from the server. This is used as the position in the virtual environment from which the incoming data will be rendered. Lines 1-3 of Algorithm 3 show the initialization. These reference positions can be determined by the developers based on the goal of the application. The user will start their experience in this position, but can maneuver themselves throughout the environment after the application begins. When multiple users are connecting to the server, this allows their starting locations to vary in the virtual environment and see the rendered data from different positions. This also allows multiple users to see the objects that represent each other. This is done by assigning each new client that connects to the server their own reference position for rendering the scene. This reference position is used as the origin of the coordinate system of the user, and as they move their headset and controllers around, these devices will be moved relative to the reference position they were given.

Algorithm 3: Real-Time Rendering Application

Input: *ID* -> Sensor identifier
cE -> Camera alignment as compressed byte[]
cA -> Downsampled RGB image as compressed byte[]
cB -> Depth image as compressed byte[]
rP -> Render Position as compressed byte[]

1. **Initialize** input connection, *I*;
2. **Receive** *rP* from *I*;
3. **Place** virtual camera at *rP*;
4. **Receive** *ID*, *cE* from *I*;
5. **For** *num* = 0 to *dataSize/vertLimit*
 - a. **Create** virtual object *obj_ID₀*;
6. **End**
7. **Repeat** 3-5 for all unique *ID*;
8. **Receive** *ID*, *cA* and *cB* from *I*;
9. **Decompress** *cA* to *A* and *cB* to *B*;
10. **Match** *ID* with virtual objects, *obj_ID₀*, *obj_ID₁*,...;
11. **Convert** *A* to 3D points;
12. **Place** *vertLimit* points from *A* into *obj_ID_x*;
13. **Repeat** 12 for *obj_ID_{x+1}* until *A* is empty;
14. **Place** *vertLimit* colors from *B* into *obj_ID_x*;
15. **Repeat** 14 for *obj_ID_{x+1}* until *B* is empty;
16. **Repeat** 8-15 while connections are active;
17. **End**;

After the reference position has been received, the client will receive the sensor alignment and unique identifiers for each sensor connected to the server as compressed arrays. After this data is decompressed, the offsets in position and rotation will be used to create virtual objects to contain the data received from the sensors. The number of objects necessary for each sensor is determined by taking the maximum number of points for each sensor and dividing that by the vertex limit of the rendering engine. This ensures that every data point captured by the sensor can be placed in a virtual object to be rendered. Lines 4-7 of Algorithm 3 show the process for receiving unique identifiers and creating virtual objects to store and render the incoming data.

Taking advantage of advanced Graphics Processing Units (GPUs), the large datasets received from the central server can be processed more efficiently than they would be on the Central Processing Unit. This is due to GPU's having large numbers of processing cores designed to handle the computations involved in rendering highly complex virtual environments. While any GPU can be utilized by this process, the efficiency and effectiveness of the algorithms scale with the number of cores, the amount of on-board memory, and the clock speed of the GPU. Complicated algorithms, or processing substantial amounts of data, can also be offloaded to the GPU. This is accomplished by coding the algorithm that processes the individual 3D point data being received from the server to run on the large number of processing cores available on the computer's GPU using shaders. A compute shader is used to dispatch a large number of threads to process the incoming data in parallel, rather than sequentially processing the pixels in each image. Each data point that is captured by one of the sensors connected to the server is rendered independently in the virtual environment.

When a frame of data is received from the central server, it is first decompressed into the separate depth data, color data, and unique identifier for the sensor, as shown in lines 8-9 of Algorithm 3. The unique identifier is used to match the data with the corresponding virtual objects that contain the offset for the respective sensor. The depth and color data are then split to fit the object vertex limit of the rendering engine. A number of points are selected from the depth and color data that stays under the object vertex limit. The depth values are placed into the vertices of the object while the color data is placed into the vertex colors of the object. This process of splitting the incoming data is shown in lines 10-15 of Algorithm 3. Instead of treating the object as a mesh with render-able triangles, the object is set to only render the vertices as colored points. This greatly reduces the processing time required to render the virtual environment, since the computations for processing and rendering mesh surfaces are skipped. The processing and assigning of vertices and vertex colors is also optimized to run on the large number of cores of modern GPUs. This allows for large datasets to be processed and rendered in real-time at a comfortable framerate for viewing in a VR system. The ability to view the resulting rendered data in an immersive VR system provides the user with the sense of “presence” that allows for a more realistic interpretation and understanding of the environment in which the RGB-D sensors are capturing data [2].

4 APPLICATION OF PROPOSED METHOD

This section outlines an example use case of the proposed method. The case study used a series of Kinect for Windows v2 sensors attached to Windows Surface tablets, with Intel 4th Generation i5 Processors, 4GB of RAM, and Windows 8.1. The server application is run on an Amazon Web Services server [25]. The rendering applications are run on desktop computers containing 8GB of RAM, Intel Core i5 processors, and NVIDIA GeForce GTX 750 Ti graphics cards with an HTC Vive [10] connected to each one.

Step 1: Sensor Alignment

Unmodified Kinect for Windows v2 sensors are used to capture the RGB-D data. The Kinect SDK v2 provided by Microsoft is used to interface with the sensors and retrieve the data. The Windows Surface tablets are used to allow the sensor placement to be customized to the environment where the data is being captured. The position of each sensor is determined relative to the other sensor’s using the Accord.Net library [16] to perform a homography estimation for each pair of images from unique sensors. A depiction of this process is shown in Figure 5. The green points represent features that are found using a Harris Corner Detector [14] on an image received from one sensor. The blue points are similar features found in the image from a different sensor. The red points show features that are contained in the images from both sensors. These red points can be used to estimate a homography, that can then be used to calculate



Figure 5: Two-Sensor Alignment

the position and rotation offsets between the two sensors using the known camera parameters of the sensors. Once the position of each sensor is known, the position and rotation offset of the sensor is stored on the server along with the sensor's unique ID. This allows the data to be positioned correctly in the virtual reconstruction.

Step 2: Formatting Sensor Data

Once the color and depth data are received from the sensor, they are formatted to the same dimensions, 512x424 pixels. This is done using the Kinect SDK v2 to match the pixel locations in the depth image to a corresponding pixel in the color image. The color pixels that have a matching depth pixel are placed into a down-sampled color image with the same resolution as the depth image. The resulting 512x424 pixel images are compressed using the GZipStream class in C# [26]. The compressed images are then sent over a standard TCP network connection to the server.

Step 3: Data Server

For the case study, the server application is run on an Amazon Web Services server [25]. The server receives incoming data from each of the sensors and outputs the data to any clients that are connected. Whenever a sensor first connects to the server, the server saves the offset for the sensor that was found in Step 1. When clients connect to the server, the server sends out data for the number of sensors and their offsets. The server also maintains all the connections to sensors and clients and makes sure that the data received from sensors is correctly assigned to a sensor ID so it is rendered correctly in the virtual environment. The server is also responsible

for handling both clients and sensors connecting and disconnecting. If a new client connects to the server, it performs the steps outlined in Algorithm 2 for a new output connection. These steps ensure that the client has all data necessary to correctly render the data from the sensors. If a new sensor connects to the server, it receives a color image from the sensor which is used to calculate the offset of the sensor relative to the rest of the system. This offset is then distributed to all connected clients. Maintaining these connections enables the clients to correctly receive all new data being collected by the sensors.

Step 4: Real-Time Rendering

The client application runs on the computers that perform the rendering. These applications support rendering for VR systems, if they are available. When the client first connects to the server, it receives an offset for each sensor that the server has connected to. This offset contains a translation and orientation that represents the sensor's position relative to the universal coordinate system. In the rendering engine, an object is placed at the location and rotation for each sensor offset. These objects will store the depth and color data from the sensors and are the objects used in the rendering process. When a new frame of RGB-D data is received from the server, it is decompressed using the GZipStream class in C# [26]. Once the data frame is decompressed, the separate depth and color images can be extracted, along with a unique identifier for which sensor the RGB-D data came from. The depth and color images are then split into smaller pieces based on the vertex limit of the rendering engine. Once the RGB-D data is decompressed the depth data is translated into 3D points using the field of view and intrinsic values of the depth camera, and these 3D points are assigned as vertices to the offset objects. The color data is then placed as vertex colors in the same objects. This keeps the connection between the color data and depth data so that each 3D point will be correctly colored to represent the real-world location. The process of translating the depth data into 3D points and assigning vertex and vertex color values is run as a GPU-based algorithm, leveraging the large number of cores on a GPU to process the data quickly and efficiently. This also allows the processing and rendering to occur at separate times, allowing a higher framerate to be maintained when rendering for a VR headset.

5 RESULTS AND DISCUSSION

Figure 1, which was discussed in Section 2.1, shows the results of an individual walking in front of a sensor running the KinectFusionExplorer [7]. This figure shows how dynamic objects are not captured by applications that focus on static environments. The proposed method allows for data about dynamic objects in the scene to be captured, distributed, and rendered in real-time. This can be seen in Figure 6, where the individual moving through the environment can be clearly seen, while the static objects in the environment are also represented. This allows for all kinds of dynamic objects, including people, animals, and inanimate objects, to be represented in the virtual environment for the user to view in the VR system. Enabling the user to view this type of dynamic environment through a VR system increases the sense of “presence” they have while viewing the captured data and allows them to naturally gather information and form an understanding of the distant environment.



Figure 6: Rendering Results

The automatic sensor alignment process outlined in this paper achieves visually similar alignment results in the rendered environment while adding flexibility to the setup process of the system. No manipulation of the physical environment is required for the sensors to calculate the offsets between each other, and no pre-trained marker must be visible. This allows the system to function in environments that may not be conducive to marker placement or viewing from all sensors due to lighting, obstruction, lack of flat surfaces for marker placement, or other environmental factors that could affect the system’s ability to determine offset from the marker.

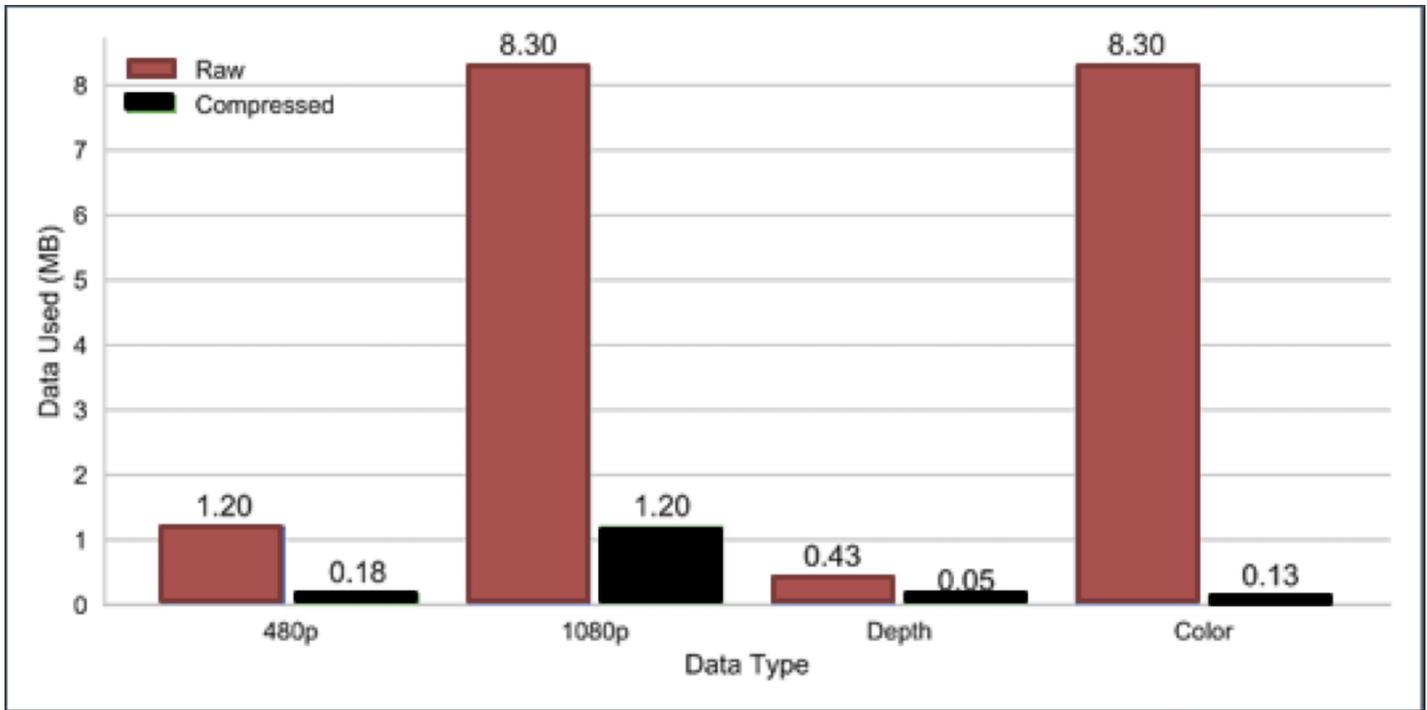


Figure 7: Resource Usage of Depth and Color Data Compared Against 480p and 1080p Images

Figure 7 shows the network usage of the authors' method when compared to streaming video at both 480p and 1080p resolution. The size of a compressed 1080p PNG image is approximately 1.2 MB. For the proposed methodology, a data set includes both a RGB and depth image the size of which when compressed is approximately .18 MB. This is considerably smaller than a compressed 1080p image (1.2 MB) and closer in size to a compressed 480p image (0.18 MB). In ideal conditions where every frame of data is processed and transmitted as it is captured by the sensor, 30 data sets will be transmitted from each sensor every second. This is due to the capture rate of the Kinect for Windows v2 sensor that was used in the case study. This results in a total bandwidth usage for the proposed method of approximately 5.4 MB/s.

Considering that the average broadband internet speed in the United States is 6.9 MB/s [11], the proposed method could be run on a below average internet connection and still be able to maintain maximum data transmission from the sensors. Since each sensor can have its own independent internet connection increasing the number of sensors does not change the bandwidth requirements for an individual sensor, only for the client that needs to receive all data from all sensors. Using a powerful processing machine on the same local network as the clients can alleviate this problem since local network connections are faster and more stable than connections to external networks and resources. As the number of sensors increases the bandwidth usage increases linearly, i.e. one sensor requires the client to have a minimum bandwidth of approximately 5.4 MB/s to receive all data, two sensors require approximately 10.8 MB/s, etc. This is all assuming ideal conditions and maximum data transmission, so if fewer frames are being transmitted due to processing or network bottlenecks then the actual amount of data being transmitted and received will vary. In the situation where the network

bandwidth cannot handle transmitting all the incoming data, the transmitted framerate will become lower. This can occur on the connection from the sensor to the server or from the server to the client, the second of which is the more likely case. This lowering of framerate will keep the data coming from all sensors aligned based on time, meaning that a time-aligned frame from each sensor will be transmitted before the following frame from any sensor. This reduces the risk of the frames from the sensors becoming mis-aligned and offset in time, which can lead to a confusing and inaccurate visualization.

Table 3: Case Study Performance Metrics

	Network Type	Sensor 1 Average FPS	Sensor 2 Average FPS	Average Rendering FPS
Case Study 1	External	23	11	84
Case Study 2	Local	24	26	100

To test the proposed method and collect performance data, two case studies were conducted. Table 3 outlines the performance of the proposed methodology during each of the case studies. The first case study consisted of two Kinect for Windows v2 sensors paired with Windows Surface 3 tablets at a remote location, one hardwired to the network and the other using a WiFi connection. These two sensors streamed data to a processing machine acting as the server located on the same local network as the rendering station for the client application. A laptop with an nVidia GTX 1060 GPU was connected to an HTC Vive to run the client application and connected to the local network via WiFi. In this case study, two individuals were at the remote location in view of the sensors. They contributed to a remote discussion that involved the location they were in and moved around the space as needed to highlight areas of interest. Over the course of this case study performance metrics were gathered for both the sensors and the rendering application. Sensor 1 consistently output data with an average of 23 FPS. This is lower than the maximum FPS of the sensor, 30 FPS, and may be due to the processing hardware available in the Surface tablets and the speed of the network at the remote location. Sensor 2, connected via WiFi, had its bandwidth throttled by the automatic network manager at the remote location. After investigating it was discovered that this was due to the new WiFi connected device being treated as a potential security risk in the remote location, and its bandwidth was reduced until its connection was reviewed. This is a factor that must be considered when bringing hardware and software associated with the proposed method into remote locations. The rendering application during this case study averaged 84 FPS while rendering the received data to be viewed in the head mounted display.

During the second case study, the client application was tested using a laptop with an nVidia GTX 1060 GPU connected to an HTC Vive. The client application was connected to the central data server over a Wi-Fi connection on a local network. Both sensors were again connected to Windows Surface 3 tablets and both streamed data via WiFi to the local network. During the case study a

student was asked to perform some simple tasks in view of the two sensors. These tasks included carrying and placing a computer monitor and connecting the appropriate cables. Data from Sensor 1 was being received in the client application at an average rate of 24 frames of data per second. Sensor 2 was outputting at an average of 26 FPS during the second case study. The client application was able to process and render the received data at a higher framerate, ~ 100 frames per second. This means that multiple frames were rendered in between each frame of data that was received. This is important to consider when using VR headsets, as a high framerate is necessary to have a smooth and comfortable view for the user in the headset. Maintaining a high framerate lessens the amount of movement that occurs in the user's view between each frame, and a general guideline is to stay above 60 frames per second, preferably staying in the 90-120 frames per second range for rendering.

6 POTENTIAL USE CASES

The proposed method can be applied to any activity where being physically present is important to the quality of the experience and the authors see tremendous potential in improving the collaborative abilities of distributed design teams. Being able to share ideas and view objects and spaces in a natural and immersive manner allows these geographically dispersed teams to work together to solve design problems in a manner that more closely mimics the real world. Teams using the proposed method will be able to gain the sense of “presence” that is important to gathering information and forming understandings of environments and the objects and interactions contained within. Being able to collaborate with remote team members without anyone having to travel long distances decreases the cost of collaboration for these teams while also decreasing the time it takes to begin the collaboration process. In the simplest terms, users of the proposed method would be able to connect from any point on the globe having an internet connection and contribute to their team in real-time. The true value of these advantages over more traditional means of remote collaboration needs to be explored in greater detail and will provide numerous opportunities for future research and development.

In addition to design, there are potential use cases in a variety of fields. For example, industry could benefit through remote meetings, distributed teamwork, and training applications while the military and civil services could utilize the proposed method for training, inspection, maintenance operations, and data analysis. In the field of medicine, the proposed method could be used for expert consults without the need for travel thus increasing the availability of high-quality diagnosis and expert opinions while reducing the cost to the patient. The education sector could use the proposed method for student recruitment and engagement activities, online learning courses, campus and facility tours, and guest speakers and presentations. These are just a few examples of the broad applicability of the proposed method to potential use cases spanning numerous fields. The modularity and flexibility of the proposed

method allows it to be adapted to a wide range of applications in a large variety of fields, making it a highly useful tool for linking remote users and teams.

7 CONCLUSION

The method proposed by the authors allows for remote individuals to view a virtual representation of a dynamic environment in real time. The remote users are able to gain a sense of “presence” by viewing the virtual representation using commodity VR systems. This allows for them to have a similar experience to that of a physically present individual, enhancing their ability to understand the environment and collaborate with individuals who are geographically disperse.

By utilizing commodity RGB-D sensors like the Microsoft Kinect for Windows v2 and publicly available rendering engines like Unity [22], the authors are able to create a system that allows for remote individuals to have a sense of “presence” in a virtual reconstruction of a real-world location. Section 4 shows an application of the proposed method running on readily available hardware and software platforms while Section 5 discusses the feasibility and usability of the system when compared to other networked activities.

The authors acknowledge the potential for improvements and optimizations to the proposed method, and list below some possible extensions of this work:

- Optimizations in image processing and transmission to reduce bandwidth and memory usage.
- Processing to eliminate redundant data from multiple sensors.
- Improvements to the RGB-D sensor calibration and alignment by using a more robust calibration process.
- RANSAC algorithms could be used to detect “static” objects in the scene and store them in memory. This would remove the need for them to be transmitted, further reducing the bandwidth requirements of the system.
- Methods to reduce IR interference between multiple sensors could be implemented to improve the accuracy of the depth data.
- Improvements to the client-server setup to make connections more stable and robust.
- Adding the ability to manage multiple environments, giving users the ability to choose different environments to experience.
- Human subjects trials to determine the believability and usefulness of the created Virtual Environment.

ACKNOWLEDGMENT

The authors of this paper would like to acknowledge the following for their contributions and support of this project:

- 1) 2016 IBM Faculty Award
- 2) NSF NRI #1527148
- 3) Penn State D.A.T.A. Lab for support and resources
- 4) Alexandar Devic, Kosho Hoshitsuki, and Gerard Pugliese for assistance in designing, coding and testing the proposed method.

REFERENCES

- [1] J. E. Cutting, "How the eye measures reality and virtual reality," *Behavior Research Methods, Instruments, & Computers*, vol. 29, no. 1, pp. 27–36, 1997.
- [2] D. A. Bowman and R. P. McMahan, "Virtual Reality: How Much Immersion Is Enough?," *Computer*, vol. 40, no. 7, pp. 36–43, Jul. 2007.
- [3] K. M. Lee, "Why presence occurs: Evolutionary psychology, media equation, and presence," *Presence: Teleoperators and Virtual Environments*, vol. 13, no. 4, pp. 494–505, 2004.
- [4] E. Turner, P. Cheng, and A. Zakhor, "Fast, automated, scalable generation of textured 3d models of indoor environments," *IEEE Journal of Selected Topics in Signal Processing*, vol. 9, no. 3, pp. 409–421, 2015.
- [5] O. Hamzeh and A. Elnagar, "A Kinect-based indoor mobile robot localization," presented at the Mechatronics and its Applications (ISMA), 2015 10th International Symposium on, 2015, pp. 1–6.
- [6] R. A. Newcombe *et al.*, "KinectFusion: Real-time dense surface mapping and tracking," presented at the Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on, 2011, pp. 127–136.
- [7] "Kinect Fusion Explorer-WPF C# Sample." [Online]. Available: <https://msdn.microsoft.com/en-us/library/dn193975.aspx>. [Accessed: 16-Feb-2017].
- [8] K. Lesniak, J. Terpenney, C. S. Tucker, C. Anumba, and S. G. Bilén, "Immersive Distributed Design through Real-Time Capture, Translation and Rendering of 3D Mesh Data 1," *Journal of Computing and Information Science in Engineering*, 2016.
- [9] "Oculus Rift | Oculus." [Online]. Available: <https://www.oculus.com/rift>. [Accessed: 16-Feb-2017].
- [10] "VIVE™ | Discover Virtual Reality Beyond Imagination." [Online]. Available: <https://www.vive.com/us/>. [Accessed: 16-Feb-2017].
- [11] "United States Speedtest Market Report." [Online]. Available: <http://www.speedtest.net/reports/united-states/>. [Accessed: 10-Feb-2017].
- [12] R. S. Yang *et al.*, "Multi-Kinect scene reconstruction: Calibration and depth inconsistencies," presented at the Image and Vision Computing New Zealand (IVCNZ), 2013 28th International Conference of, 2013, pp. 47–52.
- [13] S. Asteriadis, A. Chatzitofis, D. Zarpalas, D. S. Alexiadis, and P. Daras, "Estimating human motion from multiple kinect sensors," presented at the Proceedings of the 6th international conference on computer vision/computer graphics collaboration techniques and applications, 2013, p. 3.
- [14] C. Harris and M. Stephens, "A combined corner and edge detector.," presented at the Alvey vision conference, 1988, vol. 15, pp. 10–5244.
- [15] E. Dubrofsky, "Homography estimation," *Diplomová práce. Vancouver: Univerzita Britské Kolumbie*, 2009.
- [16] "Accord.NET Machine Learning Framework." [Online]. Available: <http://accord-framework.net/>. [Accessed: 03-Oct-2017].
- [17] "OpenCV library." [Online]. Available: <http://opencv.org/>. [Accessed: 03-Oct-2017].
- [18] D. Ni, A. Song, X. Xu, H. Li, C. Zhu, and H. Zeng, "3D-point-cloud registration and real-world dynamic modelling-based virtual environment building method for teleoperation," *Robotica*, vol. 35, no. 10, pp. 1958–1974, 2017.

- [19] S. Kim and J. Park, “Robust haptic exploration of remote environments represented by streamed point cloud data,” presented at the World Haptics Conference (WHC), 2017 IEEE, 2017, pp. 358–363.
- [20] P.-C. Su, W. Xu, J. Shen, and S. S. Cheung, “Real-time rendering of physical scene on virtual curved mirror with RGB-D camera networks,” presented at the Multimedia & Expo Workshops (ICMEW), 2017 IEEE International Conference on, 2017, pp. 79–84.
- [21] T. Garrett, S. Debernardis, J. Oliver, and R. Radkowski, “Poisson Mesh Reconstruction for Accurate Object Tracking With Low-Fidelity Point Clouds,” *Journal of Computing and Information Science in Engineering*, vol. 17, no. 1, p. 011003, 2017.
- [22] “Unity - Game Engine,” *Unity*. [Online]. Available: <https://unity3d.com>. [Accessed: 16-Feb-2017].
- [23] “What is Unreal Engine 4.” [Online]. Available: <https://www.unrealengine.com/what-is-unreal-engine-4>. [Accessed: 16-Feb-2017].
- [24] S. Sharples, S. Cobb, A. Moody, and J. R. Wilson, “Virtual reality induced symptoms and effects (VRISE): Comparison of head mounted display (HMD), desktop and projection display systems,” *Displays*, vol. 29, no. 2, pp. 58–69, 2008.
- [25] “Amazon Web Services (AWS) - Cloud Computing Services,” *Amazon Web Services, Inc.* [Online]. Available: <https://aws.amazon.com/>. [Accessed: 16-Feb-2017].
- [26] “GZipStream Class (System.IO.Compression).” [Online]. Available: [https://msdn.microsoft.com/en-us/library/system.io.compression.gzipstream\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.io.compression.gzipstream(v=vs.110).aspx). [Accessed: 16-Feb-2017].