Kevin Lesniak
Computer Science and Engineering,
The Pennsylvania State University,
University Park, PA 16802
e-mail: kal5544@psu.edu

Janis Terpenny
Industrial and Manufacturing Engineering,
The Pennsylvania State University,
University Park, PA 16802
e-mail: jpt5311@engr.psu.edu

Conrad S. Tucker
Engineering Design,
Industrial and Manufacturing Engineering,
The Pennsylvania State University,
University Park, PA 16802
e-mail: ctucker4@psu.edu

Chimay Anumba
Design, Construction and Planning,
University of Florida,
Gainesville, FL 32611
e-mail: anumba@ufl.edu

Sven G. Bilén
Engineering Design,
Electrical Engineering,
The Pennsylvania State University,
University Park, PA 16802
e-mail: sbilen@psu.edu

Immersive Distributed Design Through Real-Time Capture, Translation, and Rendering of Three-Dimensional Mesh Data[1]

With design teams becoming more distributed, the sharing and interpreting of complex data about design concepts/prototypes and environments have become increasingly challenging. The size and quality of data that can be captured and shared directly affects the ability of receivers of that data to collaborate and provide meaningful feedback. To mitigate these challenges, the authors of this work propose the real-time translation of physical objects into an immersive virtual reality environment using readily available red, green, blue, and depth (RGB-D) sensing systems and standard networking connections. The emergence of commercial, off-the-shelf RGB-D sensing systems, such as the Microsoft Kinect, has enabled the rapid 3D reconstruction of physical environments. The authors present a method that employs 3D mesh reconstruction algorithms and real-time rendering techniques to capture physical objects in the real world and represent their 3D reconstruction in an immersive virtual reality environment with which the user can then interact. Providing these features allows distributed design teams to share and interpret complex 3D data in a natural manner. The method reduces the processing requirements of the data capture system while enabling it to be portable. The method also provides an immersive environment in which designers can view and interpret the data remotely. A case study involving a commodity RGB-D sensor and multiple computers connected through standard TCP internet connections is presented to demonstrate the viability of the proposed method. [DOI: 10.1115/1.4035001]

1 Introduction

The availability of low-cost computing and networking infrastructure is enabling design teams to collaborate in a distributed manner. The value of interpreting complex data about design concepts/prototypes and environments is highly dependent on the size and quality of the data being shared. The emergence of affordable immersive virtual reality hardware, such as the Oculus Rift [1], HTC Vive [2], and the Playstation VR [3], is transforming the manner in which distributed teams are able to interact with virtual concepts/prototypes and environments. For example, a truly realistic virtual environment may be used to expedite training processes [4,5], allow immersive remote observation of job sites or educational events [6], or reduce travel costs for design reviews. Using a system that provides the above features allows design teams to work more efficiently and productively while remaining distributed [7].

The availability of off-the-shelf color and depth, RGB-D, sensing systems has opened many opportunities for technological advancement into 3D rendering and reconstruction [8]. The democratization of these technologies is enabling everyday individuals to process large amounts of data into usable 3D models and virtual representations. RGB-D sensor systems are commonly used to scan real-world objects and output a 3D model that can then be imported and viewed or edited in traditional CAD software [9–11]. RGB-D sensing systems have also been used to scan large environments and generate virtual representations in interactive 3D environments. These large scans required the original software for the sensor systems to be expanded with new algorithms that allow the sensors to move around the environment being scanned [12,13]. One of the major limitations of this kind of algorithm is the lack of interaction and visibility in real time. Whelan et al. [13] captured data sets using a Kinect sensor attached to a laptop computer. These recorded data sets were later processed by their algorithm on a separate machine. This prevents the user from interacting with, moving around in, or fully visualizing the reconstruction as it is being created.

Incorporating color data into the virtual reconstruction allows the receivers of the information to gain a deeper understanding of the environment of interest. This is due to the receivers of the information having access to a more natural representation of the space. Research has shown that having this natural representation allows the user to gather information similarly to viewing the physical environment [14]. The Kinect Fusion Explorer-WPF C# Sample, which is heavily based on the work of Newcombe et al. [15], is able to incorporate color data into a real-time reconstruction [15]. However, their method lacks the ability to share the reconstruction with distributed design teams or interact with the reconstruction. Turner et al. also incorporated color data into their algorithm. However, the resulting reconstruction did not occur at the same time the data were being captured, and also had limited detail for small objects in the environment [16].

This paper presents a method that enables the real-time creation of the virtual representation of physical environments with which the user can subsequently interact. In order to achieve this, both the depth and color information from an RGB-D sensor are dynamically rendered in a virtual environment that is remotely connected to the sensor. The proposed method enables the sensing system to be independent of the computer that is rendering the virtual environment. The proposed method provides the ability to generate realistic virtual representations of real-world objects and

[1]2016 Best Paper Award: ASME IDETC-CIE Virtual Environments and Systems (VES); Conference Version: DETC-2016-59762.

Contributed by the Computers and Information Division of ASME for publication in the JOURNAL OF COMPUTING AND INFORMATION SCIENCE IN ENGINEERING. Manuscript received September 15, 2016; final manuscript received October 10, 2016; published online xx xx, xxxx. Editor: Bahram Ravani.

locations and allows users to view this process in real time in a virtual reality environment.

The remaining sections of this paper are organized as follows: Section 2 presents literature closely related to this work. The method for dynamically creating interactive, virtual representations of physical environments is presented in Sec. 3, while Sec. 4 presents a case study that demonstrates the feasibility of the proposed method. Section 5 presents the results of the case study, and Sec. 6 concludes the paper and outlines areas for future expansion.

2 Literature Review

2.1 Digital Representation of Physical Artifacts Through 3D Scanning. Advancements in commercial, off-the-shelf technologies have enabled the digital representation of physical artifacts in a timely and efficient manner [17]. For example, Newcombe et al. [18] developed the KinectFusion library to allow commodity RGB-D sensors to construct accurate virtual representations of the real world. The group focused on scanning a fixed area, with either a static sensor or a moving sensor. The resulting system is able to produce high-fidelity virtual representations of the scanned area and incorporate color data into the representation. However, the system only integrates data into a predefined area around the position where the sensor started scanning. This means that the area to scan is predefined and limited in size to the range of the sensor. This can be seen in the Kinect Fusion Explorer-WPF C# Sample [15] that was used as a basis for a portion of the method proposed in this paper. The Kinect Fusion Explorer application has a maximum scanning limit of ∼8 m cube. Anything outside of the cube will not be included in the reconstruction. This was due to how the application handles memory and the incorporation of new data. If the reconstruction volume is made any bigger, there are problems storing and processing all of the data in the reconstruction.

Roth and Vona [12] and Whelan et al. [13] sought to expand the range of the KinectFusion library [18] by altering the management of data and incorporation of new frames. These teams created algorithms in which the volume of space being reconstructed is moved as the RGB-D sensor is moved in the real world. This allows for space that was outside of the reconstruction volume when it was initialized to be considered for reconstruction as the sensor moves. In essence, this allows for the scanning of much larger environments, while maintaining a small working set of the reconstruction for data integration. The limitation for these systems is that prerecorded data sets are being used as the input to the developed algorithms. This prevents the real-time representation of the reconstruction to be shown in the VR environment. These systems also only incorporate the depth image data, and not the RGB images. The result is a mesh representation of the environment without any color data incorporated.

Hamzeh and Elnagar [19] used a commodity RGB-D sensor to create an algorithm that generates floor maps of the area being scanned to use with robot navigation and planning operations in environments where it is dangerous or difficult to send people in to produce a map. Hamzeh et al. did not incorporate color, or build a complete 3D representation of the environment being scanned. Turner et al. [16] built an algorithm to model and texture large scanned environments. However, their method lacks the real-time rendering and interaction component that has been shown to result in a deeper conceptual understanding of an environment [14]. The algorithm proposed by Turner et al. runs on a data set that was prerecorded and then processed by the developed algorithm. The goal was to represent architectural features by generating a floor plan from the scanned data and extruding a 3D building model from them. The result is a more structured 3D model, but lacks the features and detail of the proposed method. Also, the algorithm used the RGB images captured by the sensor to texture the resulting model, but did not incorporate the RGB data into the point cloud generated from the depth data. Incorporating the RGB data directly into the point cloud, as is proposed in this paper, gives each vertex that is being rendered a color. This allows the resulting colored mesh from the proposed method to appear accurate under various lighting schemes and from differing perspectives in the VR environment. AQ5

Some groups, like Roth and Vona [12], Whelan et al. [13], and Turner et al. [16], sought to overcome the limitation of the size of the volume being scanned by building an algorithm that allowed the volume being scanned to move while the RGB-D sensor moves. This allows an increase in size of the volume being scanned, but requires that the data capture system is portable, limiting the amount of processing power that is available to the algorithm in real time. Recorded data sets are captured using a sensor attached to a computer and later processed by the developed algorithm to produce mesh results. Using this method, the same quality of scans is achieved, but the real-time reconstruction of Newcombe et al. [18] and the Kinect Fusion Explorer example [15] is lost. Turner et al. [16] developed an algorithm that integrates the color data that are being captured by the RGB-D sensor into their textured 3D reconstruction, which was something lacking in Roth and Vona [12] algorithm and the algorithm of Whelan et al. [13]. Others, like Hamzeh and Elnagar [19], were only focused on building a floor plan rather than a full 3D reconstruction of the space. This greatly lowered the processing power requirement, but resulted in a much simpler and less accurate representation of the space. Hamzeh and Elnagar [19] did incorporate a networking component into their algorithm that allowed the constructed maps and the video feed to be sent back to a remote user who was tele-operating the robot to which their sensor was attached. While this improved the usability of the system, the developed algorithm did not send the complete RGB-D data set, and only sent a simplified reconstruction after processing.

The proposed method improves upon existing systems by providing a system that allows the receiver of information to view and interact with the reconstruction as it is being built. The color data from the sensor system are also incorporated into the reconstruction to provide a level of realism to the resulting virtual environment that is missing from existing systems [12,13,16]. The process also allows for the separation of the sensor system from the machine that processes the data, meaning that the RGB and depth data can be streamed from a remote location to the processing machine in real time, unlike the data recording process used in existing systems [12,13,16]. The proposed method also incorporates a mesh subdivision algorithm to limit the size of the virtual objects that will be rendered in the immersive environment. This subdivision keeps individual virtual objects under a specified number of vertices to meet memory and rendering requirements. The proposed method is divided into three components that are networked together to allow data to pass between them across standard TCP connections. This allows the processing to be distributed across as many as three computers, increasing efficiency and improving the flexibility of the system. The three components are separated to focus on the capturing and formatting of data, the processing and integration of data into the virtual reconstruction, and the rendering of the result of the virtual reconstruction. Having the three components share data over a network allows the RGB-D sensor and computer running the capturing component to be in a remote location, sending data back to a machine running the processing and integration component, which can then send the reconstruction result to a remote user running the rendering component to view the result.

Table 1 shows related systems and the features they support. The green entries show features that the corresponding system supports. Table 1 reveals that, while others have implemented a subset of the features we are providing, to the best of our knowledge, none has achieved them in a combined manner. The authors of this paper present a method that allows for the reconstruction of an accurate colored 3D representation of a physical space. A remote user can view and interact with this reconstruction in real

AQ6

Table 1 Literature review of supported features, compared to what is being proposed in this work (Lesniak et al. 2016)

Authors	Features: Mesh Reconstruction	Color Data Incorporation	Real-Time Reconstruction	Real-Time Rendering	Virtual Reality Hardware Support	Network-Distributed Processing and Rendering	Real-Time Interaction
Hamzeh *et al.* (2015)	Partial						
Curless *et al.* (1996)	Full						
Whelan *et al.* (2012)	Full						
Roth *et al.* (2012)	Full						
Turner *et al.* (2015)	Full	Full					
Newcombe *et al.* (2011)	Full		Full	Full			
Kinect Fusion Explorer	Full	Full	Full	Full			
Lesniak *et al.* (2016)	Full	Full	Full	Full	Full	Full	Full

Partial Feature	(grey hatched)
Full Feature	(green)

time, due to the distribution of data capture, data processing, and rendering into separate network-enabled components. The resulting reconstruction is also rendered in a modern VR environment that allows for a more complex representation, including physics, VR hardware integration, and the ability of the user to interact with the virtual reconstruction.

2.2 Virtual Reality Environments. Two of the major Virtual Reality environments are the Unreal Engine [5] and Unity [4]. These are both video game engines that aim to allow users to design and build 3D applications. Both of these systems have been adding support for VR hardware to allow for more immersive experiences. The companies backing both engines have recently announced support for using the entirety of their editor in a VR system similar to what is shown in Fig. 1.

A VR environment is necessary to handle the rendering and interaction components of the proposed system. Once data describing the environment of interest have been captured and processed, they need to be rendered in a form that users can then view and interpret. The VR environment is also responsible for accepting inputs from the user and responding to them. These inputs can include signals from a keyboard and/or mouse, movement of a tracked device, like a controller or VR system, or even speech input. These inputs are then translated into a form of interaction with the virtual world. The combination of rendering and interaction allows the VR environment to provide a high-quality immersive experience for the user.

While both of the VR environments mentioned above have these capabilities, Unity [4] provides direct support for VR systems and also supports programming in the same language as the processing library the authors are using. Having direct support for VR systems allows the results to be easily displayed on a number of VR systems and standard 2D display system. Direct support

Fig. 1 Head mounted virtual reality display

also improves the performance of the VR environment by optimizing the rendering process for the VR system that is being used. Unity [4] also has the capability to create applications for a variety of target platforms, including Windows, OS X, and gaming consoles. This allows the method to accommodate more design teams and target platforms. This makes Unity [4] a strong choice for the VR environment used to display the resulting virtual representation. Due to the distributed nature of the proposed method, the VR environment is interchangeable to suit the needs of the user. As long as the chosen VR environment can read and process data from a TCP connection, it can be used to display the results of the authors' algorithm.

3 Method

The method presented in this paper allows for the distribution of the process to construct a 3D mesh of a physical location and visualize it in a VR system with multiple computers in different locations. This method allows for distributed teams and remote experts to collaborate in a more natural manner, increasing efficiency and the ability to share information. Figure 2 shows the three components of the method. The component to capture and format the data from the sensor is shown in the box labeled component 1. The RGB-D sensor and Algorithm 1 are connected to this component. The component to process and integrate the data is shown in the box labeled component 2. Algorithm 2 is connected to this component. The component to render the result is shown in the box labeled component 3. This is the component to which the VR system is connected. The individual steps in the method are discussed in detail in the following sections.

AQ7

Algorithm 1: Capturing and Formatting Sensor Data

Input: C-> RGB image as byte[]
D-> Depth image as ushort[]
Params-> Internal camera parameters

Output: *jA*-> Downsampled RGB image as JPG
jB-> Formatted Depth image as JPG

1. **Initialize** sensor;
2. **Initialize** output network connection, *O*;
 THREAD 1
3. **Receive C** and **D** from sensor;
 THREAD 2
4. **For** *rowD* ε **D** do
 a. **For** *pixel* ε *rowD* do
  i. **Map** *pixel* to color space using *Params*
  ii. **If** *pixel* is in color space
   1. **Add** color pixel to **A**;
  iii. **End**
  iv. **Convert** pixel to byte, *pB*;
  v. **Add** *pB* to **B**
 b. **End**
5. **End**
6. **Encode A** into JPG, *jA*;
7. **Encode B** into JPG, *jB*;
8. **Send** *jA* and *jB* over *O*;

3.1 Acquisition of 3D Mesh Data. Using RGB-D sensing systems, 3D mesh data representing a physical object can be captured and stored in digital form. RGB-D sensors are needed because both color (i.e., RGB) and depth (*D*) data are needed for the real-time reconstruction of 3D objects in an immersive, VR environment. The depth data are required to construct the 3D mesh of the environment being scanned and the RGB data are required to associate color values with each vertex of the 3D mesh.

The depth data are formatted as a grayscale image, *D*, where each pixel value, $D(i, j)$, is equal to the distance from the sensor into the environment being scanned at an angle relative to the pixel location in the image. This means that the top right pixel in the image is at the largest horizontal and vertical angle of the depth sensor's field of view.

The color data are formatted as a color image, *C*, and are captured by the sensor at the same time that the corresponding depth image, *D*, is captured. Each pixel in the color image, $C(i, j)$, represents the color of the world at an angle from the direction the sensor is facing, relative to the pixel location in the image. The top right pixel represents the color captured at the largest horizontal and vertical angle of the RGB sensor's field of view.

3.2 Formatting of RGB-D Image Data. The two main uses of the color image in the proposed method are (i) to enhance the camera tracking algorithm and (ii) to map color data into the virtual reconstruction. For the camera tracking, both the depth and color image need to have the same pixel dimensions. To achieve this, the larger of the two images needs to be down-sampled to match the dimensions of the smaller image. To map the color data into the virtual reconstruction, the pixels in the color image that match to vertices calculated from the depth image need to be extracted from the full resolution color image. To do this, the relationship between the color and depth cameras are used to determine which color pixel matches each depth pixel. This allows one pixel value to be mapped to each vertex calculated from the depth image.

Due to limitations in RGB-D sensor technology, the depth image is, in most cases, a factor smaller than the RGB image [20]. Because of this, the RGB image can be down-sampled to match the dimensions of the depth image. The algorithm for capturing and formatting the depth and RGB images can be seen in Algorithm 1. The internal parameters of the depth and color camera are used to calculate the pixels in the color image that map to pixels in the depth image. The depth data contained in *D* are then formatted so that each pixel of data fits into a single byte. This is done by limiting the range of accepted values for depth data.

The result will be a down-sampled color image, *A*, and a formatted depth image, *B*, that have the same pixel dimensions. This is necessary for the proper integration of these two data sets into the virtual reconstruction. These down-sampled images are then encoded as JPG [21] images into memory. Storing the images as JPGs [21] in memory minimizes the size of the data being sent over the network. The JPG encoding algorithm [21] is a common image format, and the EMGU wrapper for OpenCV in C# [22] allows the JPG encoding [21] to be integrated directly into the components of the method.

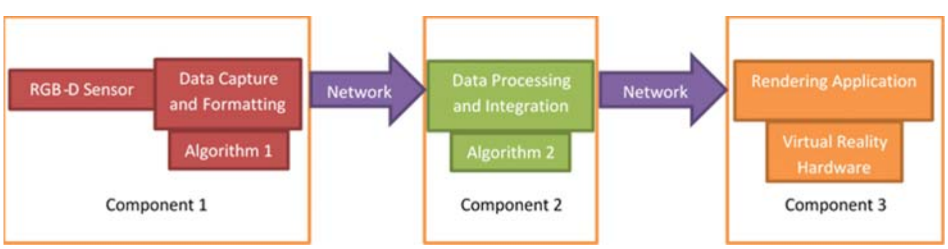

Fig. 2 Flow diagram of proposed method

Depending on the capture rate of the RGB-D sensor, there is a possibility that the sensor is capturing data faster than the available network bandwidth can send it, or the processing component can integrate it into the virtual reconstruction. To prevent unnecessary transmission of data, the data capture component waits until the processing component signals for a depth image, color image, or both. Once it receives this signal, it sends the next available frame of data, whether that is currently in memory or once it is done being formatted. Waiting for this signal means that any frames that cannot be handled by the network or the processing component are dropped before transmission. This dropping of frames prevents wasting resources on data that would otherwise not be integrated into the reconstruction. This dropping of data is discussed further in Sec. 3.4.

3.3 Integration of RGB and Depth Data. The processing component takes the formatted RGB and depth images, described above in Sec. 3.2, and integrates the images into the current virtual reconstruction. The process for this can be seen in Algorithm 2. In the processing component, one thread is responsible for reading the data received from the network and storing them in local memory. This thread reads the encoded JPG [21] images from the network and decodes the images into their raw pixel formats. Once the images are decoded, the pixel data are available to be integrated into the virtual reconstruction by another thread.

Algorithm 2: Integration of RGB and Depth Data
Input: *jA*-> RGB image as JPG
jB-> Depth image as JPG

Output: **cV**-> Vertex[] as compressed byte[]
cN-> Vertex Normal[] as compressed byte[]
cC -> Vertex Color[] as compressed byte[]

1. **Initialize** virtual reconstruction;
2. **Initialize** input network connection, *I*;
3. **Initialize** output network connection, *O*;

THREAD 1

4. **Receive** *jA* and *jB* from sensor;
5. **Decode** *jA*-> **A**, RGB image as byte[], and *jB*-> **B**, Depth image as byte[];
6. **Convert A** to int[], **B** to ushort[];

THREAD 2

7. **Track** sensor position using **B**, and **A**if necessary;
8. **Integrate A** and **B** into virtual reconstruction;
9. **Calculate** pointcloud *P* from virtual reconstruction;

THREAD 3

10. **Construct** Colored Mesh *M* from *P*;
11. **Extract V**, Vector3[] of Vertices, **N**, Vector3[] of Vertex Normals, and **C**, int[] of Vertex Colors, from *M*;
12. **Convert V**-> **bV**, Vertex[] as byte[], **N**-> **bN**, Vertex Normal[] as byte[], and **C**-> **bC**, Vertex Color[] as byte[];
13. **Compress bV** -> **cV**, Vertex[] as compressed byte[], **bN**-> **cN**, Vertex Normal[] as compressed byte[], and **bC**-> **cC**, Vertex Color[] as compressed byte[];
14. **Send cV**, **cN**, and **cC**over *O*;

A second thread is responsible for integrating the new local data into the virtual reconstruction in a stepwise fashion and calculating the sensor's movement between frames of data. First, the depth image is converted into a point cloud of vertices in 3D space. By taking the position of the sensor, the angle of each pixel, and the distance value stored in the pixel, each pixel in the depth image can be converted into a Vector3 representing a position in the virtual reconstruction. This depth image is also used to calculate the sensor's movement by aligning the 3D points the pixels in the depth image represent to the point cloud from the virtual reconstruction that already exists. This alignment provides the movement that the sensor underwent between frames relative to the reconstruction volume. This approach to tracking the movement of the sensor is beneficial because of its speed and reliance on only the depth image. If tracking with only the depth image fails, a color image is requested from the capturing component and a separate algorithm is used that combines both the depth and color image to determine the movement of the camera. While the algorithm using both depth and color data is more accurate, it is considerably slower. The benefits of faster tracking and higher frame rate are discussed further in Sec. 3.5. Once the movement of the sensor is known, the 3D points from the current depth image can be properly integrated into the point cloud of the virtual reconstruction based on the new position of the sensor. Being able to move the sensor allows for more complete scans of environments by capturing multiple angles and facets of the objects in the environment. The next step is to map the color image into the point cloud. Since the down-sampled color image contains pixels that match to pixels in the depth image, the pixel values of the color image can be assigned as the color values for the corresponding vertices that are added into the virtual reconstruction from the depth image.

A third thread is responsible for constructing the 3D colored mesh and sending the resulting data to be rendered. This mesh is constructed using a Truncated Signed Distance Function algorithm [23]. From this mesh, we can extract the vertices, normals, and vertex colors necessary for rendering. The mesh does not need to be rebuilt after each new frame of data. Since the construction of the mesh and the transmission of the extracted mesh data take time, this thread will wait until the mesh has been fully constructed and transmitted before reconstructing an updated mesh. During the time it takes for a mesh to be constructed and transmitted, the other threads in the processing application are busy receiving and integrating more data. This multithreaded approach allows the mesh to stay up to date while allowing data to be constantly integrated.

The components of the mesh that is required for rendering are the vertex array, the normal array, the vertex color array, and the triangle indices array. The vertex array is simply an array of Vector3's in which each element is a vertex in 3D space. The normal array is the normal of the surface from each vertex. The first normal in the array matches to the first vertex, the second normal to the second vertex, etc. The vertex color is an array of four-byte integers, where each byte in the integer represents an RGBA value. The first vertex color in the array is the color of the first vertex in the vertex array, the second vertex color matches the second vertex, etc. A triangle indices array is also needed to correctly render the resulting vertices in the VR environment. The triangle indices array is an array of integers that lists which sets of three vertices create a triangle in the mesh. Each integer represents an index into the vertex array. Each set of three values in the triangles array creates a triangle in the mesh. The vertex, normal, and vertex color arrays can be arranged so that the triangle indices are in sequential ascending order. This eliminates the need to send the triangle indices array over the network, reducing the bandwidth required by the algorithm.

This information is used to render the mesh in the VR environment by using these data to build objects that the VR environment knows how to render. The triangle array is used to assign vertices, normals, and vertex colors to objects in the VR environment to represent the physical artifacts that were scanned. Any limit imposed by the virtual environment on the size or format of the objects being rendered is taken into consideration in this step to ensure a complete render of the scanning data.

Figure 3 shows a sample result of the mesh reconstruction. The sensor is on the left, with each green line representing a depth point that was captured by the sensor, converted into a point in the point cloud, and output as a vertex of the mesh. The blue triangles represent triangles in the output mesh. Since RGB-D sensors can capture large amounts of data, the resulting meshes contain a large number of vertices. Due to rendering requirements in VR

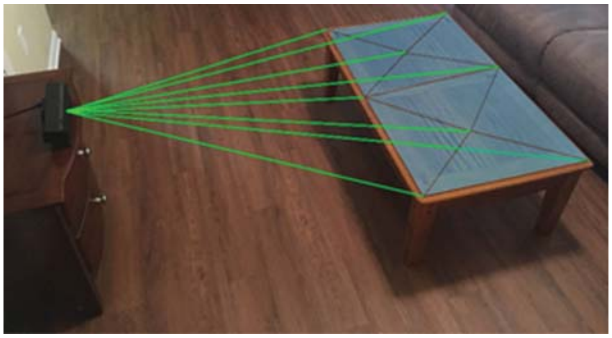

Fig. 3 Mesh constructed from sensor data

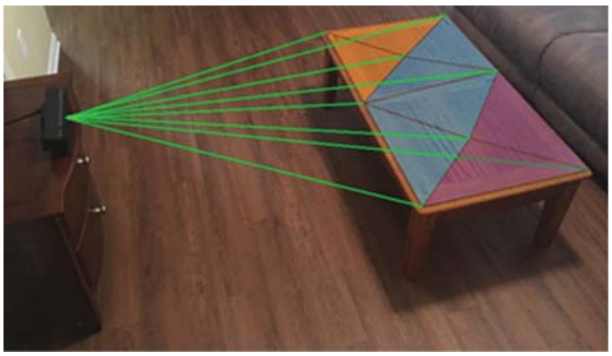

Fig. 4 Subdivided meshes from reconstructed mesh

environments, these large meshes need to be subdivided into a series of smaller meshes that can be rendered. The vertices, normal, vertex colors, and triangles are subdivided into smaller groups representing a series of meshes that, when rendered together, show the entirety of the colored mesh that was reconstructed. Figure 4 presents an example of how the constructed mesh is subdivided for rendering in the VR environment. Data are the same as that from Fig. 3, but the single mesh from Fig. 3 has been divided into three separate meshes to accommodate rendering in the VR environment. The mesh separation allows the large output mesh to be broken down so that the VR environment can handle rendering each subdivided mesh without running into any kind of constraint. If the mesh is left as one large series of triangles, the VR environment cannot handle the rendering calculations necessary to properly display the mesh. This is a limitation derived from both memory capacity and performance requirements of the rendering software. The threshold for subdividing the mesh into smaller pieces can be changed to match the limitation of the VR environment that is being used. The subdivided meshes are then transmitted individually over the network to the rendering component. This minimizes the size of each object being sent over the network.

3.4 Real-Time Rendering of Scanned Data in the VR Environment. In order for 3D mesh data to be rendered in a VR environment in real time, a multithreaded approach is needed. Since each set of subdivided meshes is approximately 2.2 MB in size, trying to read that data within the same thread that is performing the rendering of the VR environment would cause the rendering to slow down and/or freeze until all of the data have been received. The first thread is responsible for receiving subdivided meshes from the network. Once a subdivided mesh has been received, it is placed into the virtual space to align with the other subdivided meshes from the virtual reconstruction. A second thread is responsible for rendering the results for the user to visualize. This allows the user to have a fluid, uninterrupted experience in the VR environment while new data are being added.

Using this multithreaded, multicomputer approach, the rending of the virtual environment happens in real time with the data capture. This allows the user to be in the VR environment while the data are being captured, processed, and rendered. The user will be able to see new data as they are being processed and rendered in the VR environment, and be able to move around and interact with the reconstruction. Figure 5 shows an example of how the proposed method could be used distributed across the globe. The sensor could be in one location, illustrated by the photo in the top left of the figure, while a powerful processing machine, like the one shown in the lower left of the figure, could be in a separate location, and send the results to a third location where a user could see the visualization shown in the top right of the figure, but through virtual reality hardware. This system promotes collaboration and globalization while maintaining a high level of quality for information and feedback.

3.5 Quantify Frame Rate Importance in Data Processing. Two key factors in the proposed method are the frame rate at which the data images for depth and color are received and the

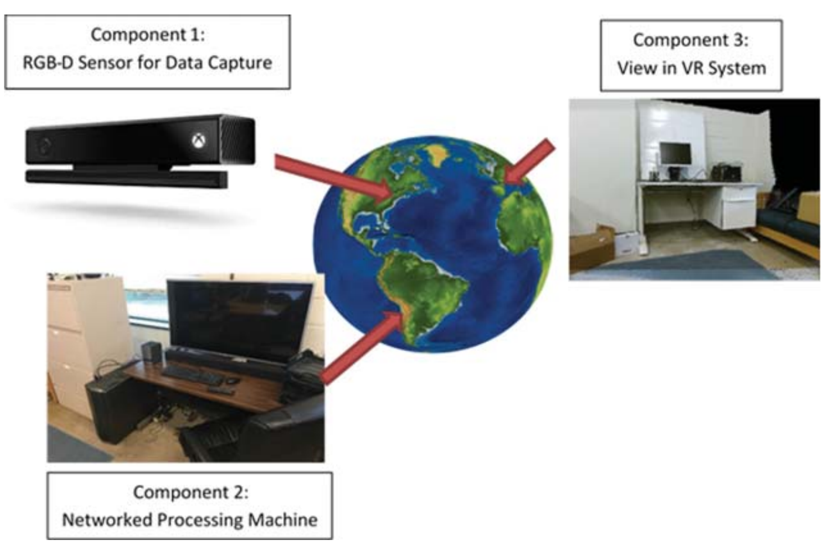

Fig. 5 Distributed components of method

frame rate at which they are integrated into the virtual reconstruction. These frame rates are important to be able to maintain tracking of the sensor while it is moving to capture as much of the real-world environment as possible. The main method for tracking the sensor's movements uses the depth image independently. Maximizing the frame rate at which the depth images are received and integrated minimizes the movement of the sensor between frames of data. This, in turn, allows the sensor to be moved faster by the user while still being able to accurately calculate the position of the sensor relative to the reconstruction and correctly integrate the data that are received.

If the amount of data being captured by the sensor is greater than what the bandwidth of the network connection can transmit, there will be a delay between when the data are captured and when they are processed. This kind of delay prevents the reconstructed meshes from containing the most recent data, and therefore prevents the VR environment from displaying the most recent data to the remote user. If the hardware where the processing component is running cannot keep up with the amount of data it is receiving from the network, frames of data will be discarded while the application waits for the reconstruction to be ready for the next frame of data to be integrated.

Two of the obstacles to maintaining a high frame rate are the amount of data being sent over the network and the speed of integrating the data into the reconstruction. The size of the images being sent is minimized by using a JPG encoding algorithm [21] to encode them before transmission and decode them afterward. This encoding algorithm minimizes the space that is used in memory and on the network by the images without sacrificing quality or data integrity. To help reduce transmitting unused data, the proposed method contains two-way communication between the capturing component and the processing component. The processing component will signal the capturing component when it is ready for a depth image, a color image, or both. Based on the signal that is received in the capturing component, it will transmit the appropriate images. If there are frames that are captured by the sensor in the capturing component before the next signal is received from the processing component, these frames are discarded before they are transmitted over the network. This prevents data that will not be integrated into the reconstruction from taking up valuable network resources. By using a faster camera tracking algorithm whenever possible, the proposed method aims to maximize the speed at which new data are integrated into the reconstruction. The overall structure of the proposed method also helps to maximize the speed of data integration by allowing each of the three components to focus on a single step in the process. Each component can then take full advantage of the resources available on their respective computers to maximize the efficiency of each step.

4 Application of Proposed Method

This section describes our process for capturing the RGB and depth data, processing the data into a reconstruction, and outputting the resulting mesh into a 3D environment. The case study utilizes the Kinect hardware [20], coupled with the KinectFusionExplorer-WPF C# sample provided by Microsoft [15]. This sample is based in part on the KinectFusion algorithm developed by Newcombe et al. [18]. The hardware our process uses consists of an unmodified Kinect for Windows v2 sensor [20], a tablet computer running Windows 10 as the Capture Machine, a desktop computer running Windows 10 as the Processing Machine, and a desktop computer running Windows 10 as the Rendering Machine.

4.1 Acquisition of 3D Mesh Data. The first component is the RGB-D sensor. An unmodified Kinect for Windows v2 sensor [20] is used for capturing RGB and depth data. The authors split the KinectFusion algorithm [18] into two components. The first component runs on the Capture Machine that is hardwired to the Kinect v2 [20] sensor that captures RGB and depth images. This component captures the RGB and depth images from the sensor, formats them to be transmitted, and waits for a signal from the Processing Machine specifying which images are needed.

4.2 Formatting of RGB-D Image Data. Both images are formatted to the required size, 512×424 pixels, by down-sampling them if they are too large. For the Kinect for Windows v2 sensor [20], the color image is down-sampled from 1920×1080 to 512×424 pixels. This is done by mapping each point of the depth image into the color image and placing the corresponding pixel into a down-sampled color image. The depth data contained in the depth image have possible values of 0–4096 and are stored in 12-bits of an ushort. These data are formatted to values between 1024 and 3064. They are then reduced by 1024, to use a zero base, and then divided by 8 to store the data in a single byte. This reduces the size of the depth data by one half while maintaining an accuracy of 8 mm in the depth data. The resulting images are then converted into arrays of bytes. These byte arrays are then encoded into JPG [21] images using the Emgu C# wrapper of OpenCV [22]. These JPG [21] images are then ready to be transmitted over the TCP connection established between the Capture Machine and the Processing Machine. Once the signal has been received from the processing component requesting certain frames, the requested compressed arrays are sent over the network. If a frame has already been compressed and is prepped to send but another frame of data arrives from the sensor before the signal from the processing component, the prepped frame is discarded so that it does not unnecessarily use up network resources. This ensures that the most current data are always transmitted over the network when they are requested by the processing component.

4.3 Integration of RGB and Depth Data. The second piece of the KinectFusion algorithm runs on the Processing Machine. This program acts as the hub for the data and handles the processing of the RGB and depth data. This program receives the data, integrates them into the existing reconstruction, constructs a colored mesh from the reconstruction, and then transmits the colored mesh.

The RGB and depth data are received on the Processing Machine from the network over a TCP connection. The resulting JPG images are then decompressed using the Emgu library in C# [22]. Byte arrays can then be read from the JPG images and parsed back into the raw RGB and depth images. These images are then used to determine the movement of the sensor since the last frame of data. Once the sensor's position is known, the position is used to integrate the new data into the reconstruction using the KinectFusion [18] library. After a series of new frames of data have been integrated, a colored mesh is built from the reconstruction using the KinectFusion library. Instead of trying to construct a mesh after every new frame of data, the mesh is constructed and output in a separate thread. As soon as the mesh data are done being sent, the thread starts building a new mesh with all the new data that have been integrated while it was creating the previous mesh. This ensures that each new mesh contains as much new data as possible, while updating the resulting mesh as often as possible. This allows for each new mesh construction to incorporate a noticeable amount of new data. This mesh reconstruction process reduces the processing that is done on the Processing Machine, while allowing the user in the VR environment to see the data appear in sections as it is processed.

From the colored mesh, three arrays are extracted. These arrays represent the vertices, normal, and vertex colors for the colored mesh. Since the colored mesh created from the reconstruction can be very large, these arrays are subdivided to create a series of meshes that the rendering application can handle. Since Unity has a limit of 65,534 vertices per mesh object, the parsed arrays are divided into multiple meshes, each containing fewer vertices than

J_ID: JCIS DOI: 10.1115/1.4035001 Date: 20-October-16 Stage: Page: 8 Total Pages: 10

PROOF COPY [JCISE-16-2084]

the limit. The vertices (Vector3 array), normals (Vector3 array), and vertex colors (integer array) for each subdivided mesh are converted into byte arrays. The byte arrays are sent to a Unity [24] application on the Rendering Machine using a TCP connection. Having a machine solely for rendering allows all available resources on the machine to focus on achieving the desired frame rate for the VR environment. This will help reduce any negative side effects from using the VR environment.

While the authors use the KinectFusion [18] library to handle some of the data integration and processing, the networking and real-time mesh reconstruction are novel processes. The ability to integrate new data from a remote source and construct mesh objects containing data as they are scanned extends the capabilities of existing systems. The separation of the resource-intensive processing from the data capturing and rendering allows for systems to be specialized for each portion of the proposed algorithm.

4.4 Real-Time Rendering of Scanned Data in the VR Environment. The final component in the proposed method is the Unity application that runs on the Processing Machine. This application is used to parse the mesh data from the KinectFusion algorithm and display them in an immersive, interactive 3D environment. The Unity application receives the vertices, normals, and vertex colors from a TCP connection as byte arrays. Each set of vertices, normal, and vertex colors represents a subdivided piece of the entire color mesh from the reconstruction.

The Unity game engine then handles the rendering of the mesh objects. Unity provides interfaces for some of the more common VR systems that are available, namely the Oculus Rift [1]. This allows for the rendered data to be easily viewed and interacted with from within a VR system. The user is also able to move around in the Unity application to better immerse themselves in the virtual representation created by the proposed method. This provides another level of immersion for remote viewing over a simple video conference or static prerendered environment.

5 Results and Discussion

Figure 6 shows a rendering of the resulting 3D mesh from the proposed method in the VR environment. This shows the quality of the mesh and the information that can be gathered from viewing the resulting mesh. Using a VR system to view the results in an immersive manner, provides the user with a more natural method for collecting information. This allows the user to gain a better understanding of the physical world without having to be physically present in it.

The proposed method was only run for 1 min for the scan that was used to collect the data for Table 2. Table 2 shows statistics

Fig. 6 Real time mesh reconstruction in the Unity VR environment

Table 2 Network and resource usage statistics for 60 s run of proposed method

	Depth	Color	Mesh
Total frames	1320	300	5
Frames/second	22	5	N/A
Data/frame (MB)	0.05	0.15	2.2
Total data (MB)	66	45	11

of network and resource usage from the proposed method. The metrics that the authors tracked are the total number of frames of data that were processed, the average frames per second (FPS) being processed, the size of each frame of data in megabytes, and the total amount of data in megabytes. FPS was not used as a metric for the mesh column because the colored mesh is not being reconstructed after every frame of data. Also, the rendering component receives a single subdivided mesh at a time and adds it into the VR environment to be rendered. This allows the subdivided meshes to be received over a period of time without affecting the frame rate of any of the components. These metrics provide valuable information about the amount of network bandwidth and processing resources required to run the proposed method and achieve similar results.

Table 2 shows that the network bandwidth required for running the proposed method is approximately 2 MB/s. This means that it is entirely feasible to run the algorithm on commodity networking hardware, without the need for specialized connection to facilitate data transmission. Table 2 also shows that processing requirements for constructing the mesh are not a limiting factor for the algorithm being run. The data for Table 2 were collected from the proposed method running on an AMD Radeon R9 270× graphics card [25]. This card is considered to be a midlevel graphics card for individuals looking for affordable performance in gaming and other 3D applications. Between the network bandwidth that is used and the ability to run the algorithm on commodity hardware, the proposed method does not limit itself to being run in specialized environments [26,27]. AQ10

6 Conclusion

The proposed method has been shown to provide a believable virtual representation of a physical space in real time. The system shown in Sec. 4 uses a commodity RGB-D sensor to provide the required data to construct this virtual representation. This system intends to improve the immersive experience of remote viewing and interacting to reduce costs and increase the awareness and familiarity of the user with the space.

By expanding upon existing systems, namely Newcombe et al. [18] and the Kinect Fusion Explorer [15], the authors are able to provide a new method for the incorporation of real-time RGB-D scanning data into a VR environment. Section 4 presented a use case in which the method was shown to provide convincing results while using readily available commodity sensors and environments.

The method proposed by the authors leaves room for expansion and extension:

- Optimizations in the (un)packing of data for transmission could further decrease the bandwidth requirements and increase the amount of data incorporated by the method.
- Improvements could be made to the down-sampling algorithms to make them faster, allowing for a higher frame rate for capturing and sending the RGB and depth images.
- Algorithms similar to Roth and Vona [12] and Whelan et al. [13] could be incorporated into the method presented in this paper. This would allow for larger areas to be scanned to provide a more complete virtual representation in the VR environment.

ID: asme3b2server Time: 16:40 I Path: //chenas03/Cenpro/ApplicationFiles/Journals/ASME/JCIS/Vol00000/160062/Comp/APPFile/AS-JCIS160062

- The mesh subdivision algorithm could be extended by attempting to identify and separate objects that are being scanned into individual meshes.
- The resulting scan could be physics-enabled in the VR environment to allow the user more possibilities for interaction.
- A more powerful Graphics Processing Unit could be used to process the RGB-D data being captured, allowing for more frequent updates to the reconstructed mesh.

Inspections are common in many production and maintenance environments. This kind of inspection normally consists of an expert reviewing a product or location to determine if there are any issues that need to be addressed or to determine the best course of action to fix a problem. This can become exceptionally difficult if there are only a limited number of experts for a particular task or if the expert is located far away from the product or location of interest. The proposed method provides a solution for this kind of situation by allowing the expert to view the product or location of interest remotely. An individual can scan the object or location of interest, stream the data to a dedicated processing machine, and the results can be viewed by the expert remotely, in real time. This allows the expert to communicate with the individual performing the scan or others involved with the inspection process in real time, promoting collaboration and the sharing of information during the inspection process.

Acknowledgment

This research is funded in part by NSF DUE #1449650 and Penn State's Center for Online Innovation in Learning (COIL). Any opinions, findings, or conclusions found in this paper are those of the authors and do not necessarily reflect the views of the sponsors.

References

AQ11 [1] Oculus, ■, "■," ■, ■, accessed Jan. 09, 2016, https://www.oculus.com/en-us/

AQ12 [2] SteamVR, ■, "■," ■, ■, accessed Jan. 09, 2016, http://store.steampowered.com/universe/vr

AQ13 [3] PlayStation VR, ■, "Playstation," ■, ■, accessed Jan. 09, 2016, https://www.playstation.com/en-au/explore/ps4/features/playstation-vr/

[4] Rudarakanchana, N., Van Herzeele, I., Bicknell, C. D., Riga, C. V., Rolls, A., Cheshire, N. J., and Hamady, M. S., 2014, "Endovascular Repair of Ruptured Abdominal Aortic Aneurysm: Technical and Team Training in an Immersive Virtual Reality Environment," Cardiovasc. Interventional Radiol., **37**(4), pp. 920–927.

[5] Sacks, R., Perlman, A., and Barak, R., 2013, "Construction Safety Training Using Immersive Virtual Reality," Constr. Manage. Econ., **31**(9), pp. 1005–1017.

[6] Bednarz, T., James, C., Widzyk-Capehart, E., Caris, C., and Alem, L., 2015, "Distributed Collaborative Immersive Virtual Reality Framework for the Mining Industry," *Machine Vision and Mechatronics in Practice*, Springer, ■, pp. 39–48. AQ14

[7] Larsson, A., 2003, "Making Sense of Collaboration: The Challenge of Thinking Together in Global Design Teams," 2003 International ACM SIGGROUP Conference on Supporting Group Work, pp. 153–160.

[8] Nguyen, C. V., Izadi, S., and Lovell, D., 2012, "Modeling Kinect Sensor Noise for Improved 3d Reconstruction and Tracking," 2012 Second International Conference on 3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT), pp. 524–530.

[9] Tucker, C. S., John, D. B. S., Behoora, I., and Marcireau, A., 2014, "Open Source 3D Scanning and Printing for Design Capture and Realization," ASME Paper No. ■. AQ15

[10] Azuma, R., Baillot, Y., Behringer, R., Feiner, S., Julier, S., and MacIntyre, B., 2001, "Recent Advances in Augmented Reality," IEEE Comput. Graphics Appl., **21**(6), pp. 34–47.

[11] Fernando, R., and Kilgard, M. J., 2003, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley Longman Publishing, ■.

[12] Roth, H., and Vona, M., 2012, "Moving Volume KinectFusion," BMVC, pp. 1–11.

[13] Whelan, T., Kaess, M., Fallon, M., Johannsson, H., Leonard, J., and McDonald, J., 2012, "Kintinuous: Spatially Extended Kinectfusion," Report No. ■. AQ16

[14] Cutting, J. E., 1997, "How the Eye Measures Reality and Virtual Reality," Behav. Res. Methods, Instrum., Comput., **29**(1), pp. 27–36.

[15] ■, ■, "Kinect Fusion Explorer-WPF C# Sample," ■, ■, accessed Sept. 14, 2016, https://msdn.microsoft.com/en-us/library/dn193975.aspx AQ17

[16] Turner, E., Cheng, P., and Zakhor, A., 2015, "Fast, Automated, Scalable Generation of Textured 3d Models of Indoor Environments," IEEE J. Sel. Top. Signal Process., **9**(3), pp. 409–421.

[17] Vasudevan, N., and Tucker, C. S., 2013, "Digital Representation of Physical Artifacts: The Effect of Low Cost, High Accuracy 3D Scanning Technologies on Engineering Education, Student Learning and Design Evaluation," ASME Paper No. ■.

[18] Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A. J., Kohi, P., Shotton, J., Hodges, S., and Fitzgibbon, A., 2011, "KinectFusion: Real-Time Dense Surface Mapping and Tracking," 2011 10th IEEE International Symposium on Mixed and Augmented Reality (ISMAR), pp. 127–136.

[19] Hamzeh, O., and Elnagar, A., 2015, "A Kinect-Based Indoor Mobile Robot Localization," 2015 10th International Symposium on Mechatronics and Its Applications (ISMA), pp. 1–6.

[20] ■, ■, "Kinect Hardware," ■, ■, accessed Sept. 14, 2016, https://developer.microsoft.com/en-us/windows/kinect/hardware

[21] ■, ■, "JPEG—JPEG," ■, ■, accessed Sept. 14, 2016, https://jpeg.org/jpeg/index.html

[22] ■, ■, "Emgu CV: OpenCV in.NET (C#, VB, C++ and More)," ■, ■, accessed Sept. 14, 2016, http://www.emgu.com/wiki/index.php/Main_Page

[23] Curless, B., and Levoy, M., 1996, "A Volumetric Method for Building Complex Models From Range Images," 23rd Annual Conference on Computer Graphics and Interactive Techniques, pp. 303–312.

[24] ■, ■, "Unity—Game Engine," ■, ■, accessed Sept. 14, 2016, https://unity3d.com/

[25] ■, ■, "Radeon™ R9 Series Graphics Cards|AMD," ■, ■, accessed Sept. 14, 2016, http://www.amd.com/en-us/products/graphics/desktop/r9#

[26] Blackman, S., 2013, *Beginning 3D Game Development With Unity 4: All-In-One, Multi-Platform Game Development*, Apress, ■.

[27] ■, ■, "What is Unreal Engine 4," ■, ■, accessed Sept. 14, 2016, https://www.unrealengine.com/what-is-unreal-engine-4