Teacher-Student VR Telepresence with Networked Depth Camera Mesh and Heterogeneous Displays

Sam Ekong, Christoph W. Borst, Jason Woodworth, Terrence L. Chambers

University of Louisiana at Lafayette

Abstract. We present a novel interface for a teacher guiding students immersed in virtual environments. Our approach uses heterogeneous displays, with a teacher using a large 2D monitor while multiple students use immersive head-mounted displays. The teacher is sensed by a depth camera (Kinect) to capture depth and color imagery, which are streamed to student stations to inject a realistic 3D mesh of the teacher into the environment. To support communication needed for an educational application, we introduce visual aids to help teachers point and to help them establish correct eye gaze for guiding students. The result allowed an expert guide in one city to guide users located in another city through a shared educational environment. We include substantial technical details on mesh streaming, rendering, and the interface, to help other researchers.

Fig. 1. Teacher guides students in an educational application. Left: teacher points in front of a TV with a Kinect depth camera. Right: Students immersed with HMDs (Oculus Rift) and tracked wands (Razer Hydra). Students also wear headsets with microphones (not pictured).

1 Introduction

Virtual reality has long been suggested as a way to enhance education [1]. Several researchers considered creating large social experiences in which students learn together. Mikropoulas [2] suggested that adding a social aspect can enable students to complete tasks more efficiently. Others such as Roussou et al. [3] created environments for students to explore together, sometimes aided by autonomous guides.

There has been less work on practical and affordable telepresence interfaces for teachers guiding students in VR. This paper presents a practical approach for teachers interacting with immersed students. The approach includes depth camera sensing and

This is an author-formatted version. It includes a minor correction to "Color set up" in Section 4.3, but it does not include the publisher's typesetting or edits.
The original publication is available at www.springerlink.com. ISVC 2016, Part II, LNCS 10073, pp. 246-258.

networking to stream a live 3D representation of the teacher, enabling teachers to guide, assist, or quiz immersed students. Such 3D mesh representations may provide richer and more immersive communication than conventional avatars [4].

While students are immersed in head-mounted displays, the teacher uses a large TV for long-term comfort and to preserve a clear view of the teacher's face for students (Fig. 1). This also allows the teacher to maintain classroom oversight when used in a local setting. The teacher's interface incorporates visual pointing aids and gaze targets to help overcome limitations of this heterogeneous display arrangement.

Our network and interaction approaches were integrated with a virtual solar energy plant to provide both virtual field trips and one-on-one activities guided by remote experts. A prototype was demonstrated connecting Lafayette, LA to Austin, TX over Internet2 (https://youtu.be/dYt01_3xo4g). Our summarized contributions are:

1. From an educational technology perspective, the work provides a novel tool for teacher-guided VR and for remote teacher-student communication.
2. Our description of the networked VR framework provides substantial practical knowledge about integration with a major VR development tool, Unity, which does not support such techniques in its standard features or plugins.
3. We present motivation, problems, and solutions for the heterogeneous display approach and especially for improving teacher pointing and gaze.

2 Application and Interaction Summary

An educational application of our networking and interaction techniques is described elsewhere [5][6]. Students explore a solar energy plant by visiting various educational stations. Students use standard ray-based pointing to trigger embedded educational elements such as animations and audio descriptions, and move between stations by selecting teleportation targets with the ray. This style of motion was chosen to minimize motion sickness risks for students seated in classrooms.

The networked system lets a teacher meet with students in various ways, for example, individual meetings to help or quiz students at device stations, and group meetings at a lookout tower for introductions and discussions. A main requirement for teacher-student interaction is for the teacher and students to point to various objects to support verbal descriptions and questions. Pointing is relatively straightforward for immersed students using rays, but pointing by the teacher viewing the TV interface presents some problems, detailed later in Section 5.

3 Networking

A networking framework (Fig. 2) was developed with custom modules to integrate multiple tools into a cohesive application. Our networked system uses a star topology with a central server node at a school, one outer node as the teacher station (local or remote), and remaining nodes at student desks. The main networking components are:

FFUnity: our FFmpeg interface for sending and receiving video streams from a Unity application. This is used for streaming Kinect mesh data (Section 3.1).

Student-Teacher State Manager: stores each user's environment state (especially educational activity progress) on the server and shares this based on the teacher's controls. Custom components, rather than Unity-provided synching, were required to support per-student local environment states and sync source selection.

Voice System: Our Unity package embeds TeamSpeak (a voice-over-IP application) into the system for audio communication between students and the teacher.

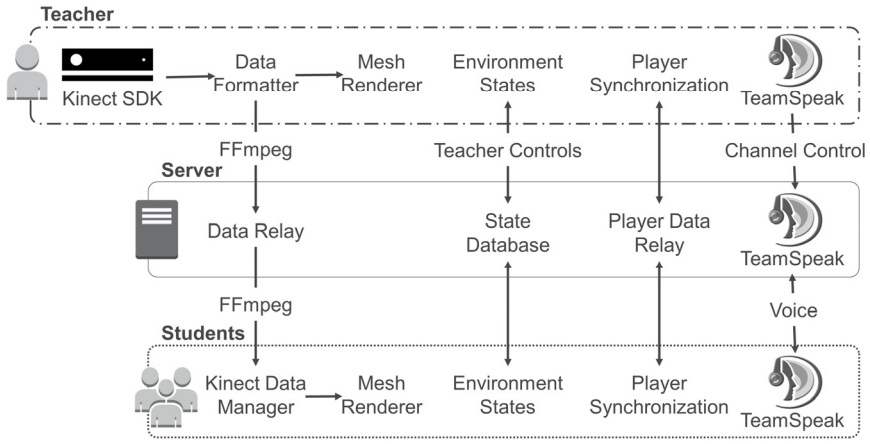

Fig. 2. Network framework and component communication.

3.1 Depth Camera Streaming in FFUnity

The depth camera, a Microsoft Kinect V2, provides the following images, per frame:

Color Frame: a conventional camera image, with 1920 x 1080 resolution. We choose its RGBA32 pixel format (alpha unused) for ease of use in Unity.

Depth Frame: a depth map from a depth camera, with 512 x 424 resolution and unsigned short (16 bit) pixels, each representing a distance from the depth camera plane in millimeters (range 500 mm to 8000 mm).

Body Index Frame: an image with 8-bit pixels related to depth frame pixels (512 x 424). Each pixel specifies which of up to 6 people is associated with the corresponding depth pixel. We use this as a mask to separate bodies from the background. Specifically, we process the depth frame to mark background pixels as invalid based on the body index frame (setting background depth values to 0).

Color and (masked) depth frames are transmitted as separate video streams through our custom C# API that exposes the encoding, decoding, and streaming capabilities of the popular FFmpeg tool to Unity applications. We chose FFmpeg for its flexibility

and networking support. We use the libx264 implementation of h264 for compressing and decompressing video streams. Libx264 was chosen for its quick encoding and decoding and a lossless encoding option. The depth stream uses lossless compression (setting 0) because standard lossy approaches do not provide the sharp and precise edges necessary for visual integrity of 3D meshes constructed from depth data. Color stream quality can be adjusted according to a desired bandwidth-quality tradeoff.

Libx264 converts a supplied pixel format to YUV before compression. For depth frames, we used YUYV as an intermediate format supplied from our application. YUYV pixels are luminance-chroma pairs with subsampled chroma (alternating chroma components). We used only the 8-bit luminance (Y) channel to store depth, splitting the upper and lower bytes of 16-bit depth pixels into two halves of a double-width YUYV image. Chroma values were constant (unused), temporarily increasing memory cost, but constant chroma is efficiently compressed, minimizing final stream impact. Alternative approaches to encoding and compressing depth data with YUV and x264, with loss tradeoffs, are studied by Liu et al. [7]. We found that lossless use of YUV's chroma channels resulted in higher bitrate than our YUYV approach.

Each raw color frame is 66.4 Mbit in size and each depth frame is 3.5 Mbit, doubling to 6.9 Mbit in our intermediate format. Streaming uncompressed frames at 30 frames per second would require rates of 1.99 Gbps and 208 Mbps. Based on a test of a teacher walking back and forth at the typical teacher location, we measured a compressed data rate of 4.1 Mbps for color and 5.1 Mbps for depth, or 4.4 Mbps and 5.5 Mbps when adding network overhead (color using FFmpeg's "veryfast" preset).

We set up FFmpeg from our application in multiple ways. For sending, we start an FFmpeg process per stream and feed it frames by pointing its system input to the Unity instance and writing raw frame data directly to its input stream. To accept this kind of input, we start the FFmpeg process with parameters such as the following:

```
-framerate 30 -f rawvideo -vcodec rawvideo -s 1024x424 -
pix_fmt yuyv422 -i - -vcodec libx264 -preset medium -g
60 -filter:v fps=30 -crf 0 -f mpegts udp://{ipaddress}
```

In summary, this sets a raw video input (depth frames) with specified resolution and pixel format, using system input, and with specified output format and destination.

The server runs an FFmpeg executable to relay the stream without transcoding (it copies the incoming stream to a multicast address to stream to multiple students).

On student computers, to get received frame data into Unity, we use a native C++ DLL modeled after an FFmpeg-provided C example, allowing Unity to interface with FFmpeg. The DLL uses FFmpeg to open a video input and triggers a callback when a new frame is read. To avoid blocking Unity's main thread, this is done in a separate thread per stream, which communicates decoded frames to the main thread (only objects in the main thread can be passed to a shader for rendering). To sync depth and color frames, we use a 32-bit frame number encoded into a corner of depth and color frames before sending. To survive moderate compression or scaling, each bit is represented as 3x3 pixels in a channel (48x3 and 32x3 total area for the depth and color frames, respectively). We pair received frames having matching frame numbers.

3.2 Other Networking Aspects

More conventional network aspects are summarized here in reduced detail. Some standard synchronization was supported by Unity's high-level networking API; for example, sharing the pose of each user's head. However, synching student's local instances of interactable objects was done with a custom manager with low-level networking calls. The server records local states per student as they progress through educational activities, and the teacher can select which student's states to sync to, for example, to monitor an individual student's effects on the environment.

Our system supports two voice communication modes: group meetings between the teacher and students and a one-on-one mode for assisting or for assessing an individual student. This involves TeamSpeak voice channel management, which we handle by a custom Unity component that allows the teacher to select modes. To sync teacher video and audio more tightly, a short delay can be added to the teacher's mic signal.

Student computers are able to stream webcam images to the teacher for remote monitoring and for future extensions placing student images on gaze indicators (Section 5). This is done with a simplified version of techniques already described.

4 Teacher Mesh Rendering

4.1 Base Mesh

We developed a custom graphics shader (Fig. 3) for Unity to render a 3D mesh of the teacher based on Kinect data. The shader processes base meshes that are first set up as Unity objects with the following static information (values set only once):

Position, encoding MeshUV: Per-vertex position is required by Unity. However, we set actual position separately in a shader, and therefore can use position slots to store a different coordinate, which we call MeshUV. MeshUV is a 2D coordinate, per vertex, for looking up vertex attributes. It acts like a floating point index, or texture coordinate, for accessing data encoded into textures.

Index List: Specifies triangles as an ordered list of vertices to be connected.

Following the Kinect depth frame structure, the teacher mesh uses a topologically regular grid of 512 x 424 vertices, giving 511 x 423 cells, each split into 2 triangles with consistent diagonals. Unity has a 65,535 vertex mesh limit, so we split this grid into 4 Unity mesh objects, the minimum number accommodating all vertices (an alternative is to use one compute buffer for all vertices, but we wanted to maintain features of Unity's object interface). For a seamless result, neighboring submeshes are joined by a row duplicated to appear in both submeshes. The number of vertices per submesh is 65,024 for the first (upper) 3 submeshes and 23,552 for the remainder, chosen to include the largest submeshes allowed without dividing within a row.

Positions (MeshUVs) are vertex attributes and the index list is a mesh attribute. Unity passes these attributes to our shader. The MeshUV coordinates are calculated like normalized texture coordinates evenly spaced along grid vertices, but accounting

for half-pixel offsets between normalized texture rectangle boundaries and actual edge pixel centers. For example, across a row, a value increases evenly from 0.5/512 to 511.5/512. These coordinates are used to access dynamic vertex attributes stored in textures having 512 x 424 resolution, via texture sampling (tex2Dlod).

Normally, Unity can determine the bounds of its meshes to define bounding boxes for culling, but we set these manually because vertices are repositioned by our shader (otherwise, the mesh could be culled while it should be visible). We set up bounds to represent the entire teacher interaction range in front of the Kinect.

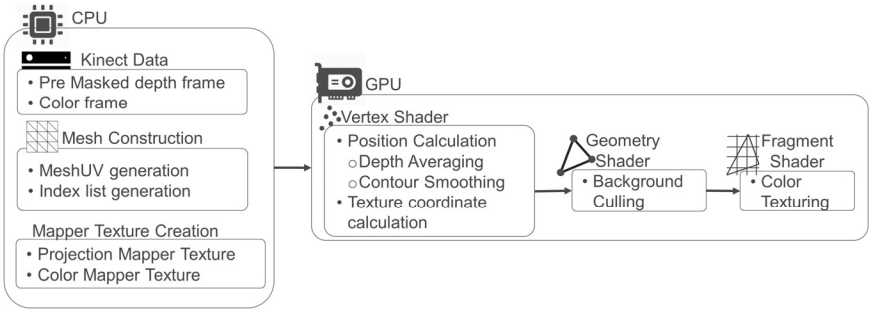

Fig. 3. The mesh rendering pipeline using Kinect data.

4.2 Texture-based Coordinate Mappings

Our vertex shader uses the MeshUV coordinates to retrieve data encoded into textures, such as depth values from the Kinect's depth frame. Other textures are created to hold pre-computed (per Kinect) data for the following mapping operations:

Projection Mapper: Used for setting 3D vertex coordinates from depth (unprojection). Given the indices and depth value of a depth frame pixel, the projection mapper provides the 3D point in camera space (Kinect's 3D base coordinate system).

Color Mapper: Provides information for texturing the color frame onto the mesh, accounting for differences between color and depth camera projections. Given the indices and depth value of a depth frame pixel, this mapper provides 2D coordinates of a corresponding color frame position (like texture space coordinates).

The Kinect SDK provides these mappings only to the directly-connected computer (the teacher station). They involve camera parameters (intrinsic and extrinsic) unique to a specific Kinect. To provide the operations on student computers and to exploit GPU parallelism, we reproduce the mappings in our vertex shader based on texture-encoded data obtained and distributed during setup.

The Kinect's projection mapping functions can provide a value, per depth pixel, that is multiplied by the depth value (Z) to compute the other two coordinates (X, Y). The entire table of all such values can be stored in 1.6 MB space and is used by our

shader to reproduce the exact results of the Kinect's mapper. For our shader, we encode this table into an RGFloat-type image having two 32-bit floats per pixel.

The Kinect's color mapper does not provide information for exact replication by custom functions, although the Kinect provides parameters for an unspecified camera model. We achieve efficient calculation and good visual results using a 3D texture as a lookup table for shader-based calculation. To generate the table, we sample the Kinect's color mapper results at (typically) 64 depth slices, each with 512 x 424 values, resulting in a 108 MB table, which is readily handled by VR-capable GPUs.

Unity currently lacks support for most pixel formats for 3D textures, including RGFloat. To encode the color mapper table, we used two RGBA32 3D textures: one per dimension of the mapper's 2D output. The encoding is based on multiplying a value in the range [0, 1] by increasing powers of 255 (powers 0 to 3), multiplying the fractional parts of the results by 255, and casting the products to the four bytes.

Unity requires 3D texture sizes to be powers of 2, so 512 x 424 slices were padded to 512 x 512. Based on a side-by-side visual comparison of Kinect-mapped results to GPU-mapped results, 64 depth slices, evenly spaced between 500 mm and 8000 mm, produced consistently good results in the teacher interaction area. The mapping could be optimized for quality per memory footprint using uneven slice spacing and possibly by reduced vertical/horizontal resolution. The horizontal separation of color and depth cameras results in nonlinearities being a greater concern horizontally than vertically, and nonlinearities have a greater effect at close range.

4.3 Shader Operation

Using the encoded data, the main operations of the vertex shader are as follows:

Depth: A depth is retrieved by texture sampling the depth frame with the MeshUV coordinates of the vertex (depth is the only channel of an RFloat texture).

Position: A projection mapper value is retrieved by texture sampling the mapper texture with MeshUV. Red and green channel values are scaled by depth to compute X and Y coordinates (in Kinect camera space), while the depth value gives Z.

Color set up: A texture coordinate is computed to allow a subsequent fragment shader to texture the video frame onto the mesh via standard texture mapping:

1. A slice index is calculated by mapping depth from a [500, 8000] range to [0, 63].
2. An interpolation factor is calculated as the fractional part of the slice index.
3. MeshUV's vertical component is adjusted to account for 3D texture padding.
4. Color mapping values are sampled, decoded, and interpolated from slices. The sampler uses the adjusted MeshUV and the nearest two slice indices, remapped to normalized range [0.5/64, 63.5/64], to get values for linear interpolation by the interpolation factor.

The resulting vertices, with position and texture coordinates, are passed to a geometry shader that culls any triangle having any vertex z value of 0 (depth of 0 represents an invalid depth or masked-out pixel). Valid triangles proceed to the fragment shader to be rendered with standard texture mapping, and without lighting.

Contour and Depth Smoothing. Depth camera meshes have spiky edges and surface noise due to their grid-like nature and depth sensor limitations. So, in addition to the above shader operations, we refine the mesh with depth and contour smoothing.

The vertex shader is able to access neighbor vertex information, stored in textures, using MeshUVs with offsets. Specifically, a one-pixel MeshUV offset is 1/512 horizontally and 1/424 vertically. A static offset list was defined for accessing neighbors, in clockwise order, starting at a top left neighbor, as numbered in Fig. 4. A neighbor is visited by adding the stored offsets to MeshUV.

We apply a moderate amount of mesh depth smoothing using a 3 x 3 or 5 x 5 box blur kernel centered on the current vertex. The blurred depth replaces the depth lookup step listed earlier. The kernel is adjusted to only consider valid vertices, for mesh edge integrity. This reduces surface noise and provides a foundation for larger filter kernels and more complex filters, if desired.

If the current vertex is on a mesh border, we apply contour smoothing. The vertex is identified as a border if it is valid but at least one of its 6 edge-connected neighbors is invalid, based on depth frame values. These connected neighbors have offset list indices {0, 1, 3, 4, 5, 7}. Starting from the identified invalid neighbor (example: vertex 0 in Fig. 4), the shader searches the connected neighbors for two valid neighbors: one from clockwise search (3 in Fig. 4), and one from anticlockwise order (7 in Fig. 4). These neighbors are on a common boundary with the current vertex. The search resembles Moore Neighborhood contour extraction [8] besides omitting two Moore neighbors. Finally, the current vertex's position is adjusted to an average of the two contour neighbors, reducing narrow spikes. The texture coordinate is also adjusted via averaging to consider the position change. Longer contours could be considered.

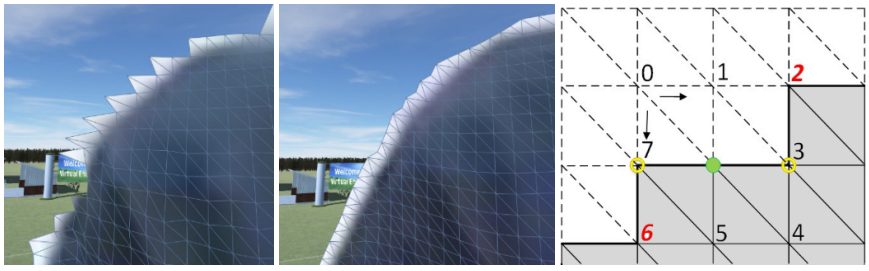

Fig. 4. Mesh edge before (left) and after (middle) refinements. Right: mesh vertices numbered by indices into a MeshUV offset list for accessing them from the central vertex.

5 Teacher Interface Approach

The interface for a teacher guiding students was developed over several iterations of observations and feedback at informal lab tests, public demonstrations, and formative tests of preliminary versions in high schools. A first version simply had the teacher use a similar interface to the students (HMD and wand), with the networked teacher represented by a floating head and wand. This is similar to early shared VR, such as

DIVE [9]. However, we wanted to improve ease and comfort for long-term teacher use over multiple sessions, and also to maximize students' sense of live teacher presence. This led to the development of the TV-based teacher interface and to the depth mesh teacher representation. For the best appearance of this representation, it is preferable to avoid worn devices that would occlude the teacher's facial expressions.

Pointing is valuable for guiding others in VR [10]. In the virtual energy center, the teacher frequently points to various features of solar power plant devices. Because the environment models a first-person view of a large real environment, and we would like the teacher to appear facing students, pointing is mainly to objects behind or to the side of teachers, rather than between teachers and students. A conventional VR-like view for TVs, corresponding to the teacher looking at students, did not support this pointing. This led to the consideration of viewpoints to better support pointing.

Based on our experiences, a view resembling the student's view was best for teacher pointing in such an environment. Per educational activity, students already teleported to landing poses with their initial view facing the relevant objects. So, we pose the camera that renders the teacher's view at the student landing pose, and render the scene with a wide field of view. This results in the teacher seeing the teacher's own virtual representation, along with relevant devices, from a student-like view.

To make the interaction with this view more natural for the teacher, the scene is mirrored by a left-right flip, which involves negating components of a projection matrix and inverting back-face culling settings in a camera script. Somewhat like the mirrored view available in videoconferencing or webcam software, this mirroring provides a perceived match between arm motion and displayed results.

However, two major problems must be overcome with this view. First, it prevents a teacher from establishing or understanding eye gaze toward a student. We consider gaze to be important, as the student experience is enhanced when the teacher looks at students being addressed. To address this, we overlay gaze targets representing students, drawn according to perspective considering teacher head tracking and TV geometry, such that the teacher can look at targets to establish proper gaze.

Second, the teacher cannot adequately control pointing depth based on the 2D image. A teacher demonstrating the system did not even realize he was pointing incorrectly, and this resulted in communication problems. The demonstrator temporarily resolved this with trial-and-error placement of paper pointing targets in the real environment. To address this, we introduce visual pointing aids, visible only to the teacher, to help the teacher control and understand pointing movements, especially depth.

An additional problem is that when multiple people enter the Kinect's range, its software does not provide a consistent teacher identification, creating problems for visual aids hinging on teacher position. We track the teacher ID based on inter-frame position similarity and allow users to take over the teacher role with a gesture.

5.1 Student Indicators (Gaze Targets)

Placing a gaze target involves the following geometric information:

Teacher Head Position: The Kinect SDK can report a position for all major joints, in Kinect camera space. We get the head position and transform it to world space.

Student Head Position: Student head position is tracked by the HMD tracker and reported over the network, in world space.

Virtual TV Representation: We measure the TV and its offsets from the Kinect to get a representation of the TV in Kinect space, which we transform to world space.

We define a ray between the teacher and student heads and find its intersection point with the TV representation. A TV-aligned indicator is rendered at that point in an overlay view. If the ray does not intersect within the TV screen bounds, the indicator is positioned based on the nearest TV point to roughly reveal student position.

The overlay (Fig 5) must be rendered with a different projection geometry than the main scene view. This projection corresponds to the TV acting like a window into the virtual world, the conventional behavior for VR displays. We use a Unity camera that only sees the indicators. Its position is the tracked teacher eyepoint, and it faces the TV plane (camera forward direction perpendicular to the TV plane). Its projection shape (view frustum) is set by setting its projection matrix. This is done through a standard 6-parameter model that describes the TV extents with respect to the eye.

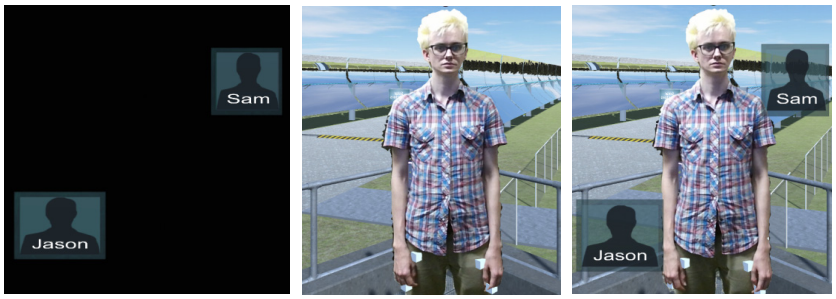

Fig. 5. Gaze targets. Left: overlay rendering. Middle: main scene. Right: composed result. The teacher looks at an indicator to make eye contact with that student.

5.2 Pointing Aids

To aid pointing, we first detect when the teacher is pointing. We define this as when the teacher has a hand raised above the waist and at a threshold distance from the spine. During pointing, we estimate the pointing direction as the vector between two specified arm joints, typical being from shoulder to wrist or elbow to hand.

We consider two different pointing cases. First, a small set of likely pointing targets may be known or estimated from context (e.g., there are some instructional targets that the students can interact with). Two approaches for this case are:

Extended Rods from Targets: During pointing, we extend translucent rods from interactable targets to the virtual shoulder (Fig. 6). We project targets, the virtual shoulder, and the hand used for pointing onto a horizontal plane to find the angle

between them. From this angle, we change the color of the rods to reflect how close the teacher points, depthwise. We also determine if the hand is in front of or behind the target-to-shoulder line and adjust the rod's mesh accordingly, expanding the end at the target when the hand moves behind and shrinking the end at the shoulder when the hand moves in front. When the teacher points directly at a target, the arm will be eclipsed by the cylinder. This provides a sense of target depth.

Top-Down Camera ("Minimap"): We placed a camera high above the teacher's virtual head, pointing down, tracking the head as a central point. Indicators for all nearby targets are drawn in a ring around the head, and a rod is drawn in the teacher's pointing direction. The camera view is rendered onto a quad near the teacher's shoulder, only showing relevant indicators. This helps illustrate pointing geometry.

Fig. 6. Indicator rod from a likely pointing target to the shoulder. Color and shape change continuously according to pointing depth. Left: The teacher points behind the target. Right: The teacher points in front of the target. Center: The teacher points directly at the target. The images also show a ray (thin cylinder) from the hand that reveals detected pointing direction.

In another case, the teacher may want to point at unforeseen items. To aid this, we developed approaches that attach something to a teacher's joint. Approaches include:

Extended Pointing Rod: We mount a thin pointing rod on the teacher's virtual hand and use a raycast to extend it out to the nearest (first hit) object. This allows the teacher to see immediately what object the rod is intersecting.

Hand-Attached Light/Projector: We place a projector or two (concentric) spotlights with a small angle on the teacher's virtual hand. Either will project a target reticle ("bullseye") or selected pattern onto whatever the teacher is pointing at.

Hand-Attached Camera: We mount a camera slightly above the teacher's virtual hand, and display its rendering at a viewport placed above the shoulder. This mainly helps with making fine adjustments in complex scenes.

The Fig. 1 teacher has a pointing rod and projected bullseye. Anecdotally, the visual aids reduce the pointing problem, but formal evaluation is needed to identify the best combination and quantify any tradeoffs. Stereoscopic TV viewing was also considered and should be compared as a baseline, but visual aids may remain useful with stereo. We prefer not to obscure facial expressions with 3D glasses. Autostereo displays may help resolve this tradeoff, but affect image quality and cost.

6 Conclusion

We presented a networked educational VR approach that streams a live teacher mesh into a virtual environment explored by students. Considering the requirements of an application and a heterogeneous display configuration, we designed a practical teacher interface including approaches to correct teacher eye gaze and to improve pointing. The next step is to formally test the various approaches and compare them against baselines to find the best combinations and to produce guidelines for use.

Our prototypes are being shown in schools to test feasibility and to provide virtual field trips with expert guides [6]. Given the affordability of new VR devices and the emerging high-performance networks, their combination can overcome geographic and scheduling constraints to let more students receive expert instruction.

This work was supported by the National Science Foundation under Grant Number 1451833 and by the Louisiana Board of Regents Support Fund under contract LEQSF(2015-16)-ENH-TR-30. We thank Kenneth A. Ritter for prior work.

References

1. Youngblut, C.: Educational Uses of Virtual Reality Technology. Institute for Defence Analyses, Alexandria, VA. (1998)
2. Mikropoulos, T.A.: Presence: A Unique Characteristic in Educational Virtual Environments. Virtual Reality. **10** (2006) 197-206
3. Roussou, M., Oliver, M., Slater, M.: The Virtual Playground: An Educational Virtual Reality Environment for Evaluating Interactivity and Conceptual Learning. Virtual Reality. **10** (2006) 227-240
4. Beck, S., Kunert, A., Kulik, A., Froehlich, B.: Immersive Group-To-Group Telepresence. IEEE Transactions on Visualization and Computer Graphics. **19** (2013) 616-625
5. Borst, C. W., Ritter III, K. A., Chambers, T. L.: Virtual Energy Center for Teaching Alternative Energy Technologies. In: Proceedings of 2016 IEEE VR, Greenville, SC, IEEE (2016) 157-158
6. Ritter III, K. A., Chambers, T. L., Borst, C.W.: Work in Progress: Networked Virtual Reality Environment for Teaching Concentrating Solar Power Technology. In: Proceedings of the 2016 ASEE Annual Conference, New Orleans, LA, ASEE (2016)
7. Liu, Y., Beck, S., Wang, R., Li, J., Xu, H., Yao, S., Tong, X., Froehlich B.: Hybrid Lossless-Lossy Compression for Real-Time Depth-Sensor Streams in 3D Telepresence Applications. In: Ho, Y.-S., Sang, J., Ro, Y.M., Kim, J., Wu, F., (eds): Proceedings of the 16th PCM, Gwangju, South Korea, Springer (2015) 442-452
8. Pradhan, R., Kumar, S., Agarwal, R., Pradhan, M.P., Ghose, M.K.: Contour Line Tracing Algorithm for Digital Topographic Maps. International Journal of Image Processing (IJIP) **4** (2010) 156-163
9. Carlsson, C., Hagsand, O.: DIVE: A Multi-User Virtual Reality System. In: Proceedings of the 1993 IEEE VRAIS, Seattle, Washington, USA, IEEE (1993) 394-400
10. Nguyen, T.T.H., Duval, T.: A Survey of Communication and Awareness in Collaborative Virtual Environments. In: Proceedings of the 1st IEEE VR CVDE, Minneapolis, MN, IEEE (2014) 1-8