Optimized Rendering for a Three-Dimensional Videoconferencing System

Rachel Chu
Department of Structural Engineering
University of California at San Diego
La Jolla, California, USA

Daniel Tenedorio, Jürgen P. Schulze
Department of Computer Science
University of California at San Diego
La Jolla, California, USA

Susumu Date, Seiki Kuwabara,
Atsushi Nakazawa, Haruo Takemura
Cybermedia Center, Osaka University
Osaka, Japan

Fang-Pang Lin
National Center for
High-Performance Computing
Hsinchu, Taiwan

***Abstract* —** Industry widely employs the two-dimensional videoconferencing system as a long distance communication tool, but current limitations such as its tendency to misrepresent eye contact prevent it from becoming more widely adopted. We are exploring the possibility of a three-dimensional videoconferencing system for future interactive streaming of point cloud data, and present the preliminary research results in this paper. We have tested thus far with one sender and one receiver, using pre-recorded data for the sender. The sender, encircled by high-definition cameras, stands and speaks in a room. A cluster of computers reconstructs each frame of the camera images into a 3D point cloud and streams it across a high-speed, low-latency network. On the receiving end, a splat-based renderer employs a new algorithm to efficiently resample the points in real-time, maintaining a user-specified frame rate. Parallel hardware projects onto multiple screens while head tracking equipment records the viewer's movements, allowing the receiver to view a stereoscopic 3D representation of the sender from multiple angles. We can combine these visuals with appropriate use of multiple audio channels to forge an unparalleled virtual experience. This next step towards immersive 3D videoconferencing brings us closer to empowering worldwide collaboration between research departments.

1. Introduction

Standard videoconferencing currently serves government, industry, and academia as an effective communication tool for multiple parties, allowing groups to conduct meetings at their convenience without incurring long-distance travel costs. Unfortunately, collaborators may decide to forgo teleconferencing when its technological limitations make it insufficient as a substitution for in-person meetings. For example, many complain that single-camera systems give the impression that viewers avoid eye contact while speaking when they look at the display rather than the camera [1]. The stereoscopic, three-dimensional videoconferencing system that we propose does not suffer from this defect. Because it is more immersive than traditional systems, groups may decide to communicate remotely more often, saving time and money.

The system we envision works as follows, as shown in figure 1. The sender stands in a room full of cameras pointed at the center of the room. The sender speaks and moves around while the cameras take footage and record sound. Each frame, a stereo reconstruction computer uses U.C. Berkeley's CITRIS software package [10] to transform the camera images into a 3D point cloud. The system then sends the cloud across a fast network to the receiver, along with a portion of sound information from each channel. The receiving computer runs an algorithm on the new points, resampling them into a uniform cloud of as many points as it is capable of drawing within a specified framerate.

The receiving computer is now ready to send drawing commands to the rendering cluster. Each node of the cluster sends video data to one projector. The projectors are divided into pairs. Each pair superimposes two images on the same screen using orthogonally polarized filters. The pair is responsible for drawing only the subset of graphical data that lies within the bounds of its corresponding screen, allowing the rendering to run in parallel. These screens are arranged in a roughly spherical formation, facing the center. The viewer stands in the center and wears 3D glasses with the same two filters, allowing each eye to see a separate image. The brain combines the two similar images, creating the illusion of depth. The viewer also wears a visor with head tracking hardware attached; as the viewer moves, the computer recognizes his new position and updates the image accordingly. This allows the viewer to see the sender from different angles. Finally, in our system, the viewer carries a remote control, allowing him to scale and rotate the image, as well as navigate through the menu system.

We have established a one-directional working model of this ideal system at Osaka University. In section 2 we describe the research that we built upon to construct our camera capturing room and stereoscopic rendering center. We discuss these setups in sections 3 and 4, respectively. Sections 5 and 6 explain how the system streams the 3D data over the network and subsequently resamples it at the other end. In sections 7 and 8, we explain the results of a streaming experiment we conducted between the National Center for High Performance Computing in Taiwan and Osaka University in Japan.

Due to performance limitations of today's hardware, our working model currently uses a cluster of computers to perform the stereo reconstruction of the large data set, instead of a single computer. Also, assuming high network bandwidth, the receiving hardware may not be capable of rendering the data set quickly enough to maintain high framerates. The algorithm we present addresses this issue, performing a fast resampling and producing a smaller point cloud that the hardware can render in time.

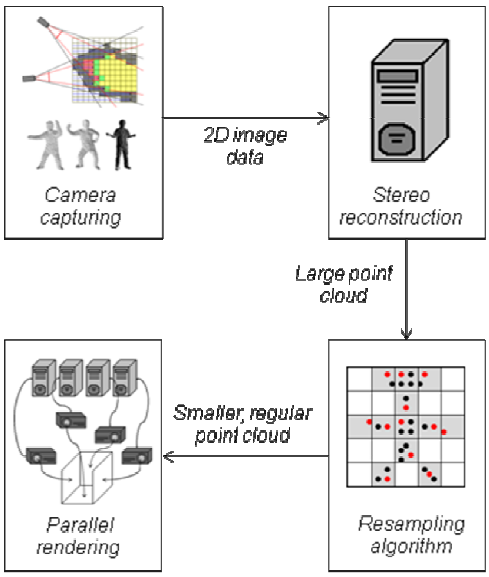

Fig. 1. *Overview of the proposed system.*

2. Previous Work

Our project builds on U.C. Berkeley's CITRIS Tele-immersion Project [10] [2]. Their capturing system, comprising 48 cameras, sends camera footage to a cluster of thirteen nodes for parallel reconstruction into 3D point clouds. Each camera in our setup sends its data via Firewire to its own reconstruction computer. However, we constructed a CAVE setup to render the data using sixteen orthogonally polarized projectors, four for each screen, while Berkeley's virtual viewing environment uses only a total of two projectors with circular polarization.

After establishing their tele-immersion labs, U.C. Berkeley conducted a streaming test with U.C. Davis [11]. Berkeley scientists used a small cluster to transfer the data over a dedicated gigabit connection to a single renderer in U.C. Davis. The test achieved framerates of ten to fifteen frames per second at two to six hundred Mbps. They were able to solve latency issues and maintain playback synchronization by attaching timestamps to the packets. In exchange for a short lag, the renderer buffered about twenty frames of data, dropping any packets that were delayed past a certain threshold. While conducting similar tests between Osaka University and the National Center for High Performance Computing in Taiwan, we were able to achieve similar framerates, but at a much lower data rate of about 1.5 Mbps.

Several schemes for efficient streaming transmission of point clouds have been proposed, including [6]. One of the most popular is the network adaptation [5] of QSplat, a software package for displaying large point-based models in real time [4]. The QSplat format restructures the point data in an optimal format for rendering on systems of various processing resources, achieving as high of a level of detail as possible under specified time constraints.

The point cloud resampling procedure by [7] uses a recursive algorithm to process a large point cloud and produce a smaller cloud of regularly spaced points. While its designers originally created the procedure to manipulate 3D geometry using conventional 2D image processing techniques, we found it to be a useful starting point for investigating different resampling techniques. The algorithm we present is superior to the prior art because it runs in O(N) time on the cloud size, quickly enough to interactively process hundreds of thousands of points on a typical desktop CPU. We describe the resampling algorithm in greater detail in its own section later in this paper.

3. Camera Capturing Setup

Our camera studio at Osaka University is simpler than its Berkeley counterpart. We installed eight cameras in an octagonal shape such that they surround an actor. The cameras use the visual hull algorithm to capture time series volume data of the human body [12]. All cameras are calibrated and synchronized and can capture XGA video sequences at 30fps. The software detects the actor's image regions using background subtraction. We use the CIELab color space at this point to remove the shadows on the floor. Then, we apply a silhouette-based visual hull to reconstruct the target volume data. The pipeline of this step is shown in figure 2.

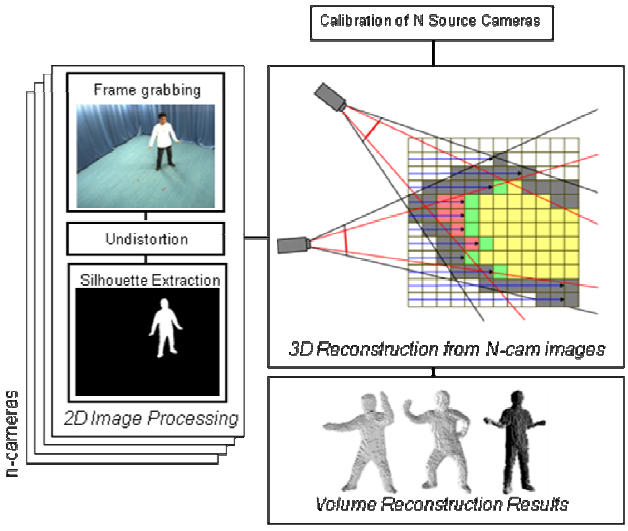

Fig. 2. *Camera capturing system.*

4. Stereoscopic Rendering

We use Osaka University's CAVE to render the point data in an immersive environment. Our CAVE, shown in figure 4, is shaped like a cube; left, front, right, and ground screens are all perpendicular to each other. The CAVE uses one master computer and sixteen worker nodes to process graphics data, as shown in figure 3. We installed Fedora Core 7 on all computers. Each group of four nodes communicates with four projectors, which displays the image through orthogonally polarized filters. When the viewer wears 3D glasses with the same filters, each eye sees images from one filter. The brain then combines the two images, producing a stereoscopic effect. All worker nodes project from behind the screens except the one responsible for the ground screen.

The tracking system of the CAVE uses an Ascension Flock of Birds system with VRCO's trackd software, and a Wanda hand remote. The tracking equipment is attached to the 3D glasses, and continually transmits the head location to the rendering system. The software then updates the virtual camera position of the viewer accordingly. With the Wanda remote, the user can scale, rotate, and move the image as he pleases.

For the CAVE software, we decided to use

COVISE, developed at the University of Stuttgart [13]. We performed the first installation onto the master node, then mounted each worker node to continue the installation. COVISE applications take the form of software library plugins which divide into several concurrent processing steps that can arbitrarily spread across heterogeneous machine platforms. For this project, we only utilized COVISE's OpenCOVER feature, its virtual reality renderer. When running OpenCOVER, each worker node only renders the portion of the image that falls within its viewing space.

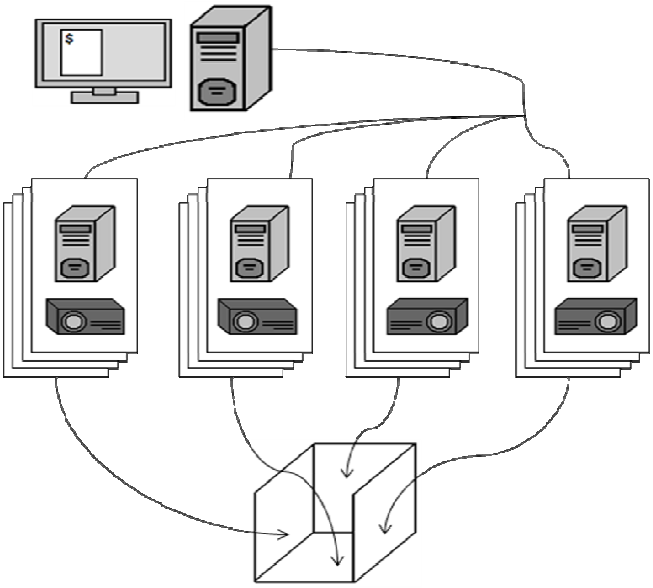

Fig. 3. *Parallel stereoscopic rendering system. One master computer sends drawing commands to sixteen worker nodes. Projectors use orthogonal filters to display images onto each of four perpendicular screens.*

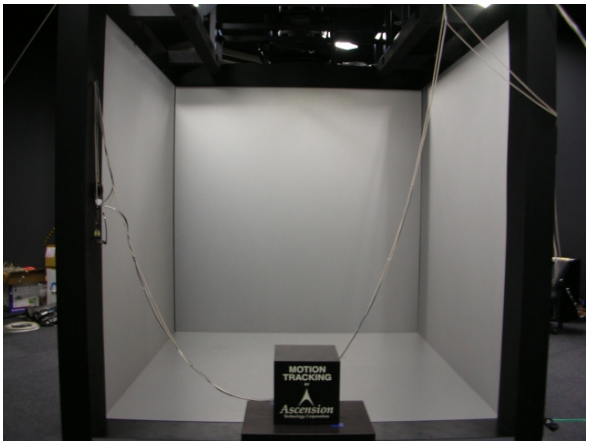

Fig. 4. *Four-wall CAVE setup at Osaka University. Left, right, front, and ground screens lie at right angles, creating a cube shape.*

5. Network Transfer

After the reconstruction equipment at the sending site generates a point cloud, it must stream the cloud over the network. We initially investigated the streaming QSplat algorithm [5] for this purpose. To gain familiarity with the QSplat software package, we implemented it as a plugin for COVISE [13]. The plugin worked well for single QSplat models, as shown in table 2 below, but we determined that a progressive, tree-based approach was infeasible for a streaming situation involving a different point cloud each frame, since tree creation could no longer occur in a preprocessing step. For example, preprocessing for the 172,974-point Stanford Armadillo model consumed an average of three hundred milliseconds.

In our current system, we decided to implement a standard TCP connection in C++ using Unix Berkeley network sockets. The software reads IP and port information from the COVISE XML configuration file in figure 5, along with a variable indicating whether the machine on which the file resides should actively request a TCP connection, or passively wait for one. For research purposes, we did not implement any kind of compression with our point data; one point comprises six floating point values, three for the 3D location and three for the color, totaling 24 bytes per point. Because the models we used ranged in size from tens of thousands to millions of points, network bandwidth emerged as a bottleneck in many test scenarios. We therefore recommend the use of a local area network for streaming large data sets.

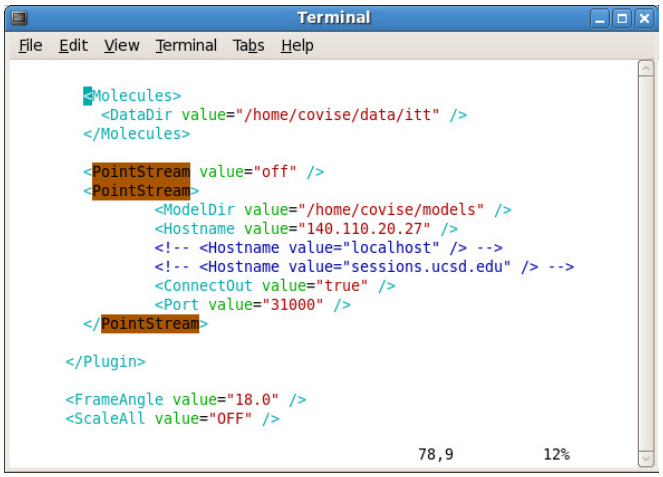

Fig. 5. *The COVISE configuration file. This is an XML-based text file where the user can set values for different plugins. Here, we set various values for network streaming.*

6. Adaptive Resampling of the Cloud

Tele-immersion requires framerates to be high enough for interactive participation, typically at least 15fps. This necessitates careful management of potential bottlenecks at all stages of the pipeline, including video capturing, network transfer, and rendering. We have already witnessed via the CITRIS research that it is possible to use a cluster to process multiple video streams in parallel and efficiently generate 3D point clouds. Bandwidth limitations sometimes caused slow rendering speeds during our tests, but streaming over a LAN mitigates this issue. As network technology improves over time, we will gradually encounter a situation in which the video capturing equipment produces very large clouds of points with no spatial locality and quickly transfers them to another workstation for rendering. The question at hand is: what happens if the workstation cannot render millions of points per frame and maintain fifteen frames per second?

A streaming point renderer clearly cannot use the same acceleration techniques as other renderers that initially load

a single model and then display it from different angles. Because a new cloud of up to millions of points arrives each frame, preprocessing becomes a luxury of the past. One tempting idea is to draw only every N point; if the cloud contains C points and the renderer is capable of drawing F points at the desired framerate, $N = C/F$. Unfortunately, the fact that the points arrive in no particular order implies that simply skipping points will cause random parts of the model to disappear each frame, resulting in a flickery rendering process. For example, if the point cloud represents a person and every third point lies on the person's arm, skipping every third point might cause the rendered frame to miss an entire arm. The solution is to resample the cloud instead, taking advantage of the ability to perform limited processing on each point in less time than it would take to send it all the way through the rendering pipeline.

Our algorithm for resampling a point cloud into a smaller, representative cloud runs as shown in figure 6 below. The idea is that we split the 3D space in which the point cloud resides into small cubes; if at least one point lies in any cube, draw only one point in that cube. By varying the size of the cubes, we decide how many points in the original cloud to render. The algorithm produces a regular point cloud in O(N) time because it processes each point exactly once; some previous approaches, such as [7], run in $O(N^2)$ time because they split the points into groups multiple times. Of course, the algorithm requires enough memory to store the state of all such cubes. We use a voxel array of $512^3 = 134{,}217{,}728$ elements, consuming 128MB of RAM.

```
ResampleCloud(points)
{
  Allocate a list v, initialized to all zeroes
  Allocate a list d of points to draw, initialized to empty
  For each point p
  {
    Determine in which voxel p lies
    If the corresponding list element v[i] = 0:
      assign v[i] = 1 and append p to d
  }
  Render all points in d
}
```

Fig. 6. *The resampling algorithm. We implement this in C++ using byte arrays for lists. The word "voxel" means a subsection of the 3D space enclosing the point model. Each element of the list* v *corresponds to one such subsection.*

We illustrate a 2D simplification of the algorithm in figure 7. First, we divide the 2D space into 25 squares. Then, we process each point, noting in which square the point lies. If the square contains no previous points, i.e. is "empty," we mark it as "full" by shading it gray. We then color the point red to indicate it is to be drawn later. Otherwise, if the square is already taken, we leave the point black, excluding it from the draw list.

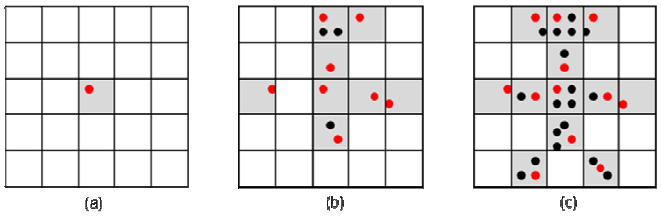

Fig. 7. *A 2D simplification of the real-time point cloud resampling algorithm. (a) Allocate a 25-element array to draw onto. (b) Shaded squares are full. The draw list comprises all red points. (c) Black points are not drawn because they lie within full squares.*

We can use this algorithm to manipulate the number of points drawn. If we divide the x, y, and z coordinates of each point by a constant number before processing, it becomes more likely that multiple points will lie within each voxel. We restricted this constant, termed the *reduce parameter*, to a power of two, allowing us to perform the division using fast integer bit shifting. If each voxel contains a large number of points, it may be possible to occasionally skip points while processing by varying a *point increment parameter*, because the skipped points are highly likely to reside in already-full voxels anyway.

The renderer can use hard-coded parameters for the above algorithm, or it can adaptively select and adjust them. In our implementation, the software runs initial tests using several possible values, then selects those that result in the most points rendered within the target framerate.

7. Results

To gauge whether the new algorithm could efficiently work with our test system, we devised a timing test of network performance. The idea was to implement the algorithm as a COVISE plugin, then use the algorithm to stream 3D point data between two geographically distant sites, measuring rendering times for different point cloud sizes. We simulated dynamic point cloud data by repeatedly sending one point model over the network connection.

We executed our streaming network test between the National Center for High-Performance Computing in Hsinchu, Taiwan, and Osaka University in Japan. We focused our testing on four point models: the Stanford Bunny (35,947 points), the Stanford Armadillo (172,974 points), and the Dawn and Lucy angel statue scans from the Digital Michelangelo Project Archives (~3 million and 14,027,872 points, respectively) [3]. Network tests revealed a sustained transfer rate of 1.5 MB/s; this constraint limited our ability to send the large models quickly, but the software was able to adapt to the size of the data sets, resampling each to a reasonable size of no more than 200,000 points and rendering within the 50 milliseconds maximum.

We noticed emerging trends in the data as we varied the model parameters. Figure 8 illustrates how increasing the reduce parameter quickly cut down the number of points sent to the GPU. However, it failed to cut down on rendering times after exceeding a certain threshold, as figure 9 shows. One explanation for this limitation is that as the algorithm sent fewer and fewer points to the GPU, the CPU processing time (constant over the number of points) remained constant.

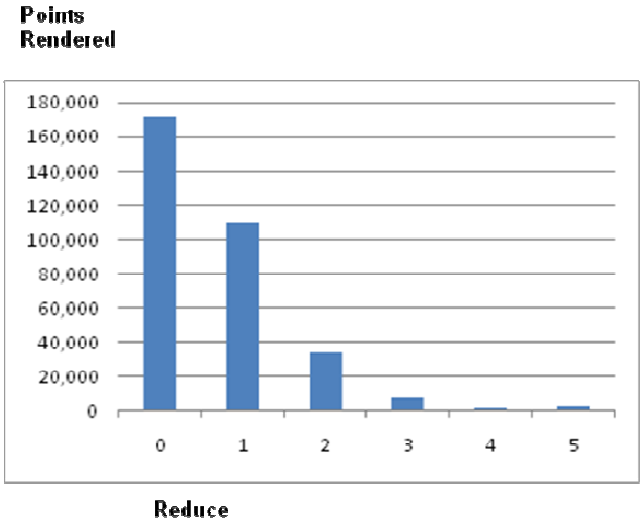

Fig. 8. *Points rendered for different values of the reduce parameter. A reduce value of 0 means the algorithm processes the original point coordinates; 1, each coordinate is divided by 2; etc.*

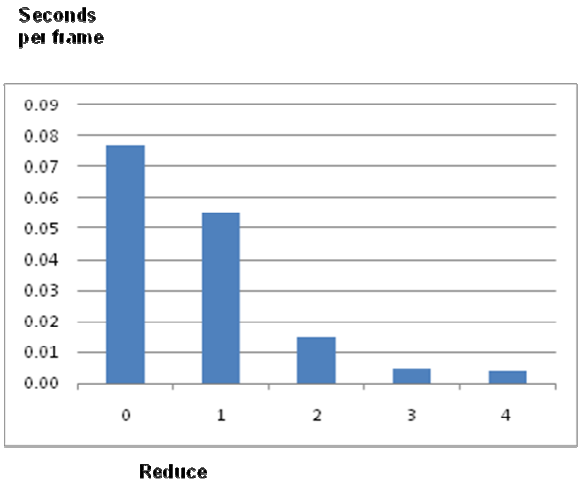

Fig. 9. *Rendering times for different values of reduce.*

On the other hand, keeping the reduce parameter constant but increasing the point increment parameter quickly decreased rendering times, but did not significantly decrease the number of points sent to the GPU. For example, resampling the Lucy model with reduce = 0, point increment = 1 yielded 349,585 points drawn, but setting reduce = 0, point increment = 11 yielded 298,157 points. Figure 10 illustrates similar results for the Stanford Armadillo model. The fact that we processed one tenth of the original data set but ended up drawing almost the same number of points suggests that the original resampling was rejecting a lot of points that happened to fall in full voxels. This means that we can process fewer points and still end up with a regular, non-flickery representation of the original model.

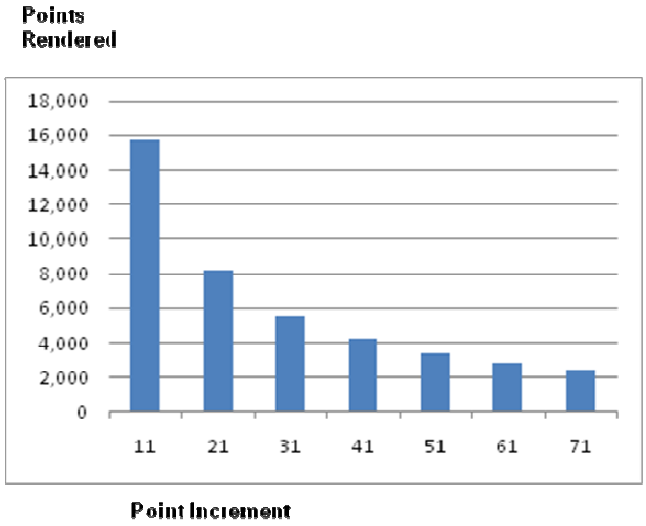

Fig. 10. *Points rendered for different values of the point increment parameter. A* point increment *value of 1 means the algorithm processes every point; 2, every other point, etc.*

Table 4 below shows comprehensive timing results for streaming the Stanford Armadillo model from Taiwan to Japan, using different values of the *reduce* and *point increment* algorithm parameters. Each table element contains a rendering time and the number of points drawn. Interestingly, two sets of parameters fit within a target framerate of 20 fps (50 milliseconds per frame) equally well. If we set *reduce* = 0 and *point increment* = 11, we can render 15,716 points in about ten milliseconds. However, if we set *reduce* = 2, *point increment* = 1, we can render 34,032 points in about twenty milliseconds. Clearly, the second pairing is preferable because more points can be rendered within the time allotted.

8. Conclusion and Discussion

Our goal was to render large, unorganized point clouds arriving over a network in real time. Table 1 illustrates how real-time triangle meshing approaches were difficult to implement; drawing the armadillo took over ten times longer as a triangle mesh vs. a point cloud. QSPLAT was not viable due to lengthy tree construction times.

Rendering Method	Msec./Frame
Direct OpenGL point drawing	3.15468
Direct OpenGL triangle drawing	42.8694
Qsorting all points by X coordinate	33.1708
Splitting points into fixed-size groups	8.57614
Creating QSplat tree data structure	312.7368

Table 1: *Rendering speeds of a PLY reconstruction of the Stanford Armadillo (172,974 points, 345,944 faces).*

However, we eventually achieved our goal of writing our renderer. The software efficiently resamples the clouds, improving the videoconferencing system in the absence of other performance bottlenecks. After running timing tests, we realized that this may not be a safe assumption to make.

Network bandwidth was often prohibitively constrictive, especially in the case of large wide-area networks. Therefore, we recommend the use of intelligent point cloud compression algorithms, e.g. [8], to increase throughput.

The way the algorithm linearly processes the points suggests that one might exploit concurrency to improve performance by directing multiple threads to process the points in parallel. We discovered that this tempting approach was difficult to implement; only one thread may hold the OpenGL rendering context at any one time, and mutual exclusion issues exist with regard to the voxels array. However, once the list of points to draw has been populated, COVISE parallelizes the rendering by making each node responsible for drawing only the points that fall within its corresponding screen space.

It is possible to adjust the rendering parameters to draw the largest amount of points within the target framerate. For a setup with stronger graphics processing capabilities, e.g. a CAVE, set the point increment parameter to a high value, keeping reduce low. This will avoid wasting time waiting for the relatively weak CPU to finish preprocessing the point cloud, and the slave nodes can render the points in parallel anyway. Care must be taken not to skip too many points, though, lest the rendered frames become too flickery. For a setup with a weak GPU, e.g. a laptop computer with onboard graphics, increase the reduce parameter and process more of the points. This will cause the algorithm to restrict the number of points sent to the GPU each frame, but ensure that the points sent accurately represent the original cloud.

9. Future Work

To further improve performance, we suggest research and testing of LAN streaming, as well as some kind of compression on the point data, such as [8]. Being able to render more points per frame profoundly improves the realism of the 3D representation and makes the videoconferencing system more immersive. Another idea is to implement UDP streaming along with some kind of mechanism for discarding duplicate or out-of-order packets. We found that the system bottleneck became network bandwidth; future tests with dedicated 10 Gbps connections should mitigate this issue.

Eventually, the ultimate goal is to establish bi-directional streaming, allowing multiple parties to send and receive data. Installing a camera setup inside a CAVE could work, although the viewers would be able to see the cameras.

The CITRIS stereo reconstruction process produces colored points, but no normal vectors. This is typically not a problem because the sender stands in a lighted area; the receiving system can simply redraw the points with the same colors to achieve realistic lighting. However, it may be possible to calculate a normal vector for each point by using the coordinates of its closest neighbors. This would allow the receiver to render using specular lighting effects, or even an arbitrary BRDF [9]. This approach is potentially computationally expensive, but would allow the system to use existing BRDF research to achieve heightened realism.

Acknowledgments

All PLY models were obtained from the Stanford 3D Scanning Repository at http://graphics.stanford.edu/data/3Dscanrep. QSPLAT models were also obtained from the sample models section of the QSPLAT main site at http://graphics.stanford.edu/software/qsplat/models, and the Digital Michelangelo Project Archive at http://www-graphics.stanford.edu/dmich-archive, with permission from Marc Levoy.

We would also like to recognize the Pacific Rim Experiences for Undergraduates (PRIME) program, supported by NSF INT 0407508 and NSF OISE 0710726 and the California Institute for Telecommunication and Information Technology (Calit2). Finally, we would like to thank Gabriele Wienhausen, Peter Arzberger, and Teri Simas for helping maintain the PRIME program.

Model Name	Max Depth	# Splats Drawn	# Vertices in Model	Rendering Speed (msec/frame)
Dragon	8	210,222	1,279,481	27.625
Lucy	8	243,944	10,072,906	31.587
Happy Buddha	6	62,774	1,060,220	8.129

Table 2: *QSPLAT for COVISE rendering speeds. All tests run on an HP Compaq dx7200 with a 3.40 GHz Pentium D CPU, 1.5GB RAM, nVidia 7900GS with 256MB GPU. Each point stored at six floats = 24 bytes.*

	0		1		2		3		4	
1	227.194	349.585	206.252	90.186	200.595	22.628	199.328	5414	203.047	1208
11	92.769	298.157	75.2105	83.089	70.728	21.414	69.3535	5241	69.031	1192
21	58.315	263.666	44.711	79.812	38.95	21.020	37.611	5181	37.294	1179
31	43.439	231.286	31.6695	76.885	27.246	20.689	26.089	5129	25.765	1167
41	35.509	203.302	25.638	74.466	21.48	20.383	20.1635	5099	20.056	1171
51	33.126	179.706	21.7455	72.059	17.6855	20.127	16.7965	5061	16.4065	1160
61	29.073	160.397	19.0445	69.704	15.239	19.837	14.0115	5019	13.7425	1155

Table 3: *Streaming Stanford's Lucy angel (14,027,872 points) to another computer in the same room over a short LAN. Across:* reduce *parameter. Down:* point increment *parameter. Data: render times (milliseconds/frame) and number of points rendered.*

	0		**1**		**2**		**3**		**4**	
1	77.798	171349	55.8245	109568	15.9705	34032	5.306	7904	4.3705	2294
11	9.2665	15716	56.955	15150	5.6775	12630	2.9305	6679	0.586	2074
21	3.07	8235	29.54	8109	2.753	7390	0.6255	5032	0.407	1920
31	0.8415	5579	29.695	5517	2.409	5186	2.539	4042	0.5325	1779
41	0.3955	4219	0.507	4184	0.4455	3988	0.7165	3287	5.3355	1669
51	2.0385	2291	0.302	3367	0.3675	3249	0.3105	2788	1.253	1553
61	0.406	2836	2.471	2825	0.274	2750	0.251	2404	0.1665	1443
71	0.1925	2437	1.9995	2426	0.236	2376	0.2335	2125	0.155	1362

Table 4*: Streaming the Stanford Armadillo (172,974 points) to the newly established CAVE at Osaka University. Across:* reduce *parameter. Down:* point increment *parameter. Data: render times (milliseconds/frame) and number of points rendered.*

References

[1] R. Vertagal: Explaining Effects of Eye Gaze on Mediated Group Conversations: Amount or Synchronization? *ACM Conference on Computer Supported Cooperative Work*, 2002.

[2] W. Wu, Z. Yang, K. Nahrstedt, G. Kurillo, & R. Bajcsy: Towards Multi-Site Collaboration in Tele-Immersive Environments. *Proceedings of the 15th international conference on Multimedia*, 2007.

[3] M. Levoy, et al.: The Digital Michelangelo project: 3D Scanning of Large Statues. In *Proc. SIGGRAPH*, 2000, pp. 131–144.

[4] S. Rusinkiewicz, M. Levoy: QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proc. SIGGRAPH*, 2000.

[5] S. Rusinkiewicz, M. Levoy: Streaming QSlpat: A Viewer for Networked Visualization of Large, Dense Models. In *Proc. SIGGRAPH*, 2001.

[6] Kimura Y., Mashita T., Nakazawa A., Machida T., Kiyokawa K., Takemura H.: A Hierarchical 3D Data Rendering System Synchronized with HTML. *The International Journal of Virtual Reality*, Vol. 5, No. 2, 2006, pp. 67-72.

[7] Sim J.-Y., Lee S.-U., Kim C.-S.: Construction of Regular 3D Point Clouds Using Octree Partitioning and Resampling. In *IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS*, 2005, pp. 956-959.

[8] Schnabel R., Klein R.: Octree-based point-cloud compression. In *Symposium on point-based graphics,* 2006

[9] Schaaf C., Martonchik J., Pinty B., Govaerts Y., Gao F., Lattanzio A., et al: Retrieval of Surface Albedo from Satellite Sensors. In *Advances in Land Remote Sensing: System, Modeling, Inversion and Application*, 2008, pp. 219-243.

[10] Jung S. *An Overview of the TI Project* [PowerPoint slides] Retrieved from http://tele-immersion.citris-uc.org/files/teleimmersion/Teleimmersion.ppt.

[11] Kurillo G. *Performance Analysis of the Tele-immersion system* [PDF document] Retrieved from http://tele-immersion.citris-uc.org/files/teleimmersion/CV_Class_presentation.pdf.

[12] A. Laurentini: The visual hull concept for silhouette-based image understanding. In *IEEE Trans. on PAMI*, Vol. 16, No. 2, pp. 150–162, 1994.

[13] D. Rantzau, K. Frank, U. Lang, D. Rainer, U. Wössner, COVISE in the CUBE: An Environment for Analyzing Large and Complex Simulation Data. In *Proc. 2nd Workshop on Immersive Projection Technology*, 1998