IEICE TRANS. INF. & SYST., VOL.E92–D, NO.7 JULY 2009

PAPER

Design and Implementation of a Real-Time Video-Based Rendering System Using a Network Camera Array*

Yuichi TAGUCHI[†a)], ***Student Member***, **Keita TAKAHASHI**[††], ***and*** **Takeshi NAEMURA**[†], ***Members***

SUMMARY We present a real-time video-based rendering system using a network camera array. Our system consists of 64 commodity network cameras that are connected to a single PC through a gigabit Ethernet. To render a high-quality novel view, our system estimates a view-dependent per-pixel depth map in real time by using a layered representation. The rendering algorithm is fully implemented on the GPU, which allows our system to efficiently perform capturing and rendering processes as a pipeline by using the CPU and GPU independently. Using QVGA input video resolution, our system renders a free-viewpoint video at up to 30 frames per second, depending on the output video resolution and the number of depth layers. Experimental results show high-quality images synthesized from various scenes.

key words: *real-time video-based rendering, light field, camera array, depth estimation, GPGPU*

1. Introduction

Image-based rendering (IBR) has attracted a lot of research interest, since photorealistic rendering quality is affordable by using a set of images of a 3D scene captured from multiple viewpoints. IBR is based on a framework in which visual information of a 3D scene is represented as a collection of light rays, such as the 7D plenoptic function [2] and 4D parameterizations using planes called ray space [3] and light field [4], [5]. Early IBR techniques often used static multi-view images captured by moving a single camera. More recently, many camera array systems have been developed with video-based rendering techniques to handle dynamic 3D scenes and produce interactive rendering applications, commonly called free-viewpoint video and 3D TV.

This paper presents the design and implementation of a system that renders a free-viewpoint video in real time from the live multi-view videos captured with a network camera array. As shown in Fig. 1, our system consists of 64 (8×8) commodity network cameras that are connected to a single PC through a gigabit Ethernet. The cameras are mounted on a mobile cart so that we can easily move them to capture various scenes. We present an on-the-fly depth estimation method to synthesize high-quality novel views. Using a layered representation, our method reconstructs a view-dependent per-pixel depth map based on a color consistency measure as well as a temporal smoothness constraint. The rendering algorithm is fully implemented on the GPU using GPGPU (General-Purpose computation on GPUs) techniques. This approach has the following advantages: 1) the GPU is suitable for parallel processing of the same instructions for each pixel, which accelerates our rendering algorithm; and 2) the software can use the CPU and GPU independently and in parallel for real-time processing. Using QVGA (320×240) input video resolution, our system enables a rendering rate of up to 30 frames per second (fps) depending on the output video resolution and the number of depth layers.

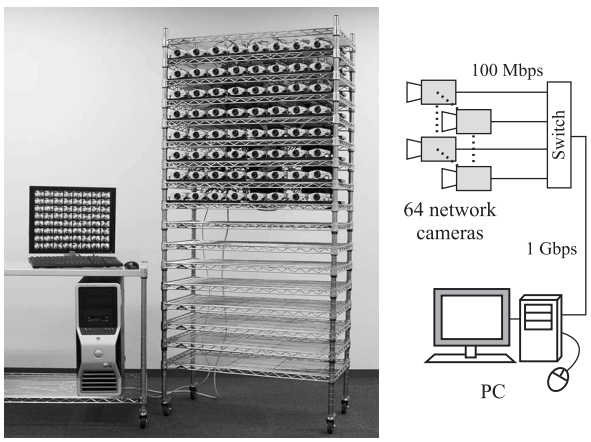


Fig. 1 Our camera array system consists of 64 (8×8) network cameras connected to a single PC through a gigabit Ethernet. The array can be easily moved on a mobile cart.

The rest of this paper is organized as follows. In Sect. 2, we review prior work and identify our contribution. Section 3 describes our rendering algorithm, and Sect. 4 presents the system implementation details. Section 5 shows rendering results obtained from various scenes as well as the performance measurement of our system, and Sect. 6 concludes the paper.

2. Related Work

This section reviews prior multi-view video capturing systems and rendering algorithms. One of the pioneering studies in this area is Kanade et al.'s Virtualized Reality project [6], in which 51 cameras were mounted on a 5-meter geodesic dome. Systems using such a sparse, circular cam-

Manuscript received October 3, 2008.
Manuscript revised March 6, 2009.

[†]The authors are with the Graduate School of Information Science and Technology, The University of Tokyo, Tokyo, 113–8656 Japan.
[††]The author is with IRT Research Initiative, The University of Tokyo, Tokyo, 113–8656 Japan.

*A preliminary version of this paper appeared in [1].

a) E-mail: yuichi@hc.ic.i.u-tokyo.ac.jp

DOI: 10.1587/transinf.E92.D.1442

era arrangement basically aim at rendering objects inside the capturing volume, and they often reconstruct voxel models by using the silhouette of the objects as well as color consistency between views [7]–[12]. Einarsson et al. [13] presented a dome-type system that captures cyclic human motion from multiple viewpoints under a sequence of controlled lighting conditions. The system enables rendering of objects under variable illumination (image-based relighting) as well as from variable viewpoints.

On the other hand, the following camera arrays (including ours) aim at capturing whole scenes by using a relatively dense, planar camera arrangement. Wilburn et al. [14], [15] developed 100 custom video cameras that have accurate timing control using a trigger signal. Using multiview videos captured by their first prototype, Goldluecke et al. [16] presented warping-based dynamic light field rendering. In [15], Wilburn et al. presented a spatiotemporal optimal sampling method and an optical flow algorithm to improve view interpolation quality. They also presented several high-performance imaging applications, such as high-speed imaging [17] and synthetic aperture photography [18]. Zitnick et al. [19] arranged eight cameras along a 1D arc and presented high-quality view interpolation using a sophisticated stereo reconstruction method followed by matting at object boundaries. Tanimoto et al. presented a large camera array system with 100 high-definition video cameras [20] and a view interpolation method based on view-dependent depth estimation [21]. Ng et al. [22] developed a hand-held plenoptic camera, in which a microlens array is placed in front of the photosensor. Using the captured static (not video) light fields, they performed refocusing and all-in-focus rendering by producing refocused images at multiple depths and gathering in-focus parts from them. All of the systems described in this paragraph need to process the captured data offline before interactive rendering, because they use complex geometry reconstruction algorithms or handle a large amount of data.

In contrast to the above offline rendering systems, the systems described in the next paragraph perform capturing and rendering all in real time. Such online rendering systems typically use simpler geometry reconstruction and rendering algorithms than the offline rendering systems to perform real-time processing. Our goal is to develop an online rendering system that renders higher-quality novel views and has higher system performance than the existing systems.

Yang et al.'s distributed light field camera [23] performed real-time rendering at a rate of 18 fps using 64 FireWire cameras (the camera capture rate was 15 fps). Since their rendering method approximates the scene geometry as a single plane, their system produces low-quality synthesized images in which only the objects at the depth of that plane are clear (in-focus) and the objects at other depths are blurred or appear with ghosting artifacts [24], [25]. This single-plane rendering would produce enough results if we use light field data densely sampled with a small camera interval [25]. However, because the cameras cannot be arranged enough densely in practice, we need to estimate the scene geometry (e.g., depth maps) for higher-quality rendering. Schirmacher et al. [26] used an array of six FireWire cameras and generated views at 1–2 fps with dense depth maps estimated from the stereo camera pairs, but the rendering quality was limited due to wrong depth reconstruction. Zhang and Chen [27] presented a self-reconfigurable camera array using 48 network cameras, each of which can move sideways and pan using servo motors. They estimate depth values only on a multi-resolution 2D mesh for rendering novel views at 4–10 fps. Our method, by contrast, estimates per-pixel depth maps to improve rendering quality. Using commodity graphics cards, Yang et al. [28] presented an efficient GPU implementation of a depth estimation method using plane-sweeping [29], [30] and view synthesis. Their rendering rate was 15 fps using five input cameras. We use similar, but more recent GPGPU techniques to perform depth estimation and rendering fully on a GPU. Moreover, they use all input cameras evenly as reference cameras for the depth estimation and color interpolation. Their approach is reasonable since their system has only five cameras. However, such an approach is not suitable for directly applying to a larger camera array system consisting of a larger number of cameras in a larger spatial arrangement, because its computational complexity increases with the increase of input cameras and it is more likely to be subject to occlusions due to the larger baseline. Our method, by contrast, uses only neighboring input cameras of each target light ray as the reference cameras, which can keep the computational complexity constant regardless of the number of the input cameras and make the occlusion effects small.

Finally, we summarize the previous work by our research group. For real-time capturing and rendering, Naemura et al. developed an array of 16 cameras [31] with hardware specialized to estimate depth maps in real time [32]. They render novel views at 10 fps by approximating the depth maps with three layers. Our system described in this paper does not require such additional hardware. Yamamoto et al. [33], [34] developed a system called LIFLET that captures a 3D scene through a micro-lens array with an XGA video camera. The captured image, called integral photography, includes thousands of different viewpoint images. They estimate a single depth value for each viewpoint image and render novel views at 15 fps. Although their system records dense light fields thanks to the dense microlens array, the capturing volume is relatively small. Takahashi et al. [35]–[37] presented a layer-based rendering algorithm that generates images and their corresponding costs (what they call focus measure) at each depth layer, detects in-focus parts from the images by evaluating the cost, and synthesizes an all-in-focus novel view. Their rendering rate was 13.9 fps from 81 static input views without offline processing. Our rendering algorithm is based on this layer-based approach, but we use 1) a more straightforward measure (i.e. variance) for evaluating the similarity of light rays, instead of their focus measure; and 2) a temporal smoothness constraint to make the depth estimation more stable.

We also show a complete system implementation of the rendering algorithm from live videos, while they use static multi-view data sets.

3. Rendering Algorithm

Our rendering algorithm assumes that multi-view videos are captured with calibrated cameras that roughly lie on a plane and are arranged on a 2D grid, as shown in Fig. 1, and that there is no prior knowledge of the scene geometry. As described in Sect. 2, since the cameras cannot be arranged enough densely in practice, we need to estimate the scene geometry (depth), rather than approximating it as a single plane, to synthesize novel views without blur and ghosting artifacts [24], [25]. This section first describes our depth estimation method that calculates a view-dependent per-pixel depth map in real time for each time frame. Next, we present a temporal smoothness constraint for stabilizing the depth estimation and reducing flickering artifacts on synthesized videos. Because our rendering algorithm processes each synthesized pixel independently, it can be efficiently implemented on a GPU, as described in Sect. 4.4.

3.1 Depth Estimation and Color Interpolation

As shown in Fig. 2, a layered depth model, $z = \{z_n | n = 1, 2, .., N\}$, is assumed in the object space to equally divide the disparity space as

$$\frac{1}{z_n} = \frac{1}{z_{max}} + \frac{n - 1/2}{N}\left(\frac{1}{z_{min}} - \frac{1}{z_{max}}\right), \tag{1}$$

where z_{max} and z_{min} are the maximum and minimum depths of the scene, respectively. As Chai et al. [25] showed, the number of depth layers required for appropriate interpolation of light field data depends on the camera intervals and resolutions. Using more depth layers basically produces higher-quality images, but needs more computational cost. We empirically choose an appropriate number of depth layers that produces enough visual quality while keeping the computational cost low.

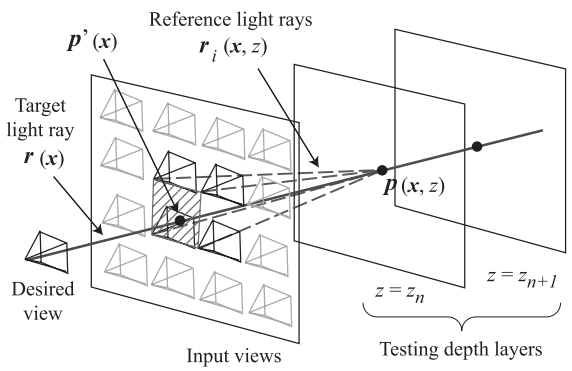


Fig. 2 Configuration for rendering a desired view. The four-nearest input cameras are used to compute the depth and color of the target light ray passing within the positions of the cameras (the region with diagonal lines).

Our method estimates the depth for each target light ray, $\boldsymbol{r}(\boldsymbol{x})$, where $\boldsymbol{x}$ represents the position of the light ray in the desired view. At the intersection of the target light ray with each of the depth layers ($\boldsymbol{p}(\boldsymbol{x}, z)$), the method evaluates the color consistency (focus measure in [37]) of the reference light rays, which correspond to the back-projections of the intersection point to the input cameras. The reference light rays are denoted by $\boldsymbol{r}_i(\boldsymbol{x}, z)$, where i is the camera index. Using a larger number of input (reference) cameras to compute the color consistency makes the evaluation stable because of a larger stereo baseline. However, it increases occlusion effects (some of the light rays may be occluded by foreground objects) and needs higher computational cost. To make the occlusion effects small and keep the computational cost low, our method performs the evaluation only using nearby reference cameras of the target light ray, as the methods described in [27], [37]. In our implementation, we used four-nearest reference cameras, as shown in Fig. 2, and the sum of variances for each RGB component of the reference light rays as the consistency measure. The color consistency cost is therefore given by

$$\begin{aligned} C(\boldsymbol{x}, z) &= consistency\left(I(\boldsymbol{r}_i(\boldsymbol{x}, z))|_{i \in V}\right) \\ &= \frac{1}{|V|} \sum_{c=r,g,b} \sum_{i \in V} \left(I_c(\boldsymbol{r}_i(\boldsymbol{x}, z)) - \bar{I}_c(\boldsymbol{r}(\boldsymbol{x}, z))\right)^2. \end{aligned} \tag{2}$$

Here, V is the set of camera indices near the target light ray and $|V| = 4$. $I(\cdot)$ and $I_c(\cdot)$ denote the color of the light ray and its color component, respectively. $\bar{I}_c(\boldsymbol{r}(\boldsymbol{x}, z))$ is the average of the colors of the reference light rays:

$$\bar{I}_c(\boldsymbol{r}(\boldsymbol{x}, z)) = \frac{1}{|V|} \sum_{i \in V} I_c(\boldsymbol{r}_i(\boldsymbol{x}, z)). \tag{3}$$

The color consistency cost is then smoothed in each depth layer in order to reduce noise effects. We average the cost over a square window

$$\bar{C}(\boldsymbol{x}, z) = \frac{1}{|W|} \sum_{\boldsymbol{x}' \in W} C(\boldsymbol{x}', z), \tag{4}$$

where W is a square window whose center is $\boldsymbol{x}$. We used a relatively large window of 11×11 pixels in the experiments, because the captured images were noisy and had large color variations. Using a large window size helps to reduce noise effects and works well for smooth regions, but produces artifacts near depth boundaries, as we discuss in Sect. 5.3. This is a fundamental limitation of window-based (local) depth estimation methods [38]. However, we use this simple smoothing method to perform real-time processing on a GPU.

Finally, the depth value that minimizes the cost is selected for each target light ray:

$$z_{opt}(\boldsymbol{x}) = \arg\min_{z} \bar{C}(\boldsymbol{x}, z). \tag{5}$$

For the color interpolation of the target light ray, using too many reference cameras would produce an unnecessarily blurred result, because the object point from which the

target light ray comes may not be completely diffusive and the projection of reference light rays may not be perfect. Therefore, we only use nearby reference light rays similarly to the depth estimation. This approach can keep the view-dependent components of the target scene and prevent the blur [39]. Using the reference light rays that correspond to the estimated depth $z_{opt}(\boldsymbol{x})$, we compute the color of the target light ray by bilinear interpolation as follows:

$$\begin{aligned} I(\boldsymbol{r}(\boldsymbol{x})) &= \sum_{i \in V} w_i(\boldsymbol{x})\, I(\boldsymbol{r}_i(\boldsymbol{x}, z_{opt}(\boldsymbol{x}))) \\ &= s\, t\, I(\boldsymbol{r}_{i_{00}}(\cdot)) + (1-s)\, t\, I(\boldsymbol{r}_{i_{01}}(\cdot)) + s\,(1-t) \\ &\quad I(\boldsymbol{r}_{i_{10}}(\cdot)) + (1-s)\,(1-t)\, I(\boldsymbol{r}_{i_{11}}(\cdot))^{\dagger}. \end{aligned} \tag{6}$$

Here, $w_i(\boldsymbol{x})$ is the bilinear weight for the i-th reference light ray $\boldsymbol{r}_i(\boldsymbol{x}, z_{opt}(\boldsymbol{x}))$. As shown in Fig. 3 (a), it takes a floating-point value between 0 and 1 depending on the positions of the reference cameras ($\boldsymbol{c}_{i_{00}}$, $\boldsymbol{c}_{i_{01}}$, $\boldsymbol{c}_{i_{10}}$, and $\boldsymbol{c}_{i_{11}}$) and the intersection of the target light ray with the input camera plane ($\boldsymbol{p}'(\boldsymbol{x})$), and $\sum_{i \in V} w_i(\boldsymbol{x}) = 1$. To efficiently calculate the weight values s and t, we prepare an image that encodes the weight values on a regular 2D grid, as shown in Fig. 3 (b), and use texture mapping to apply this image to the quadrangle whose vertices are the positions of the reference cameras.

3.2 Temporal Smoothness Constraint

If we use the above algorithm to estimate depth maps independently for each time frame, the estimate becomes unstable, especially in textureless regions, due to camera noises. This causes flickering artifacts on synthesized videos. To reduce the artifacts and make our depth estimation stable, we incorporate a temporal smoothness constraint into the cost function. Given the depth estimate at the previous time frame, $z_{opt}^{t-1}(\boldsymbol{x})$, this constraint is described as

$$\hat{C}(\boldsymbol{x}, z) = \begin{cases} \bar{C}(\boldsymbol{x}, z) & \text{if } z^t(\boldsymbol{x}) = z_{opt}^{t-1}(\boldsymbol{x}) \\ \bar{C}(\boldsymbol{x}, z) + \lambda & \text{otherwise} \end{cases}, \tag{7}$$

where λ is a small positive constant. We set it to 40 in the experiments. The minimum cost search is then applied to $\hat{C}(\boldsymbol{x}, z)$, instead of Eq. (5). This constraint encourages that the current depth is similar to the previous depth, while it allows for large depth changes (i.e. motion) because the penalty λ is equal for all depths with $z^t(\boldsymbol{x}) \neq z_{opt}^{t-1}(\boldsymbol{x})$. If the rendering viewpoint smoothly moves (as is often the case in interactive rendering applications), we can still use this constraint because the depth map should change smoothly. If the rendering viewpoint dynamically changes, we can simply ignore this constraint.

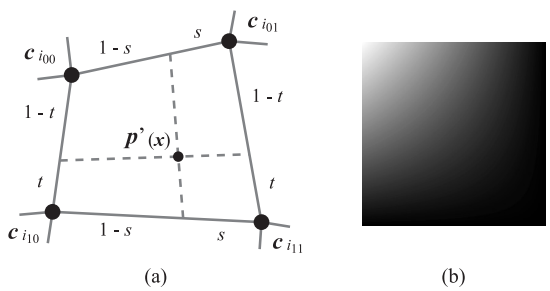


Fig. 3 Bilinear weight used for color interpolation. (a) The weight is determined by the positions of the four-nearest input cameras and the intersection of the target light ray with the input camera plane. (b) Image that encodes the weight values for the camera $\boldsymbol{c}_{i_{00}}$ on a regular 2D grid.

3.3 Discussion about the Rendering Algorithm

Since the reference camera set V depends on the position of the target light ray $\boldsymbol{x}$, the number of input cameras used for rendering the entire view depends on the desired viewpoint. Our method, however, has constant computational complexity regardless of the number of input cameras, because it calculates the color and cost for each target light ray. The computational complexity is determined by the number of target light rays (i.e. the resolution of the desired view) and the number of depth layers.

Our method may assign incorrect depth values in textureless regions, where several depth layers have similar color consistency costs. For image-based rendering, however, the depth values do not need to be correct as long as the interpolated color is visually correct. Our method interpolates such visually correct colors by selecting the reference light rays with the minimum color variance. It is also robust to the color variations of the input cameras because of the selection algorithm and bilinear color interpolation, as we show in Sect. 5.1.

The bilinear weight we use for color interpolation considers the distance penalty between the reference camera and the intersection of the target light ray with the input camera plane. As described in [39], the angular penalty between the reference light ray and the target light ray would be a more natural measure. However, we use the distance penalty because it can be efficiently implemented on a GPU by using texture mapping of the image shown in Fig. 3 (b).

4. Implementation

4.1 System Setup

As shown in Fig. 1, our camera array consists of 64 (8×8) Axis 210 network cameras. The distance between cameras is about 100 mm both horizontally and vertically. The camera employs a built-in HTTP server, which sends motion JPEG sequences in response to HTTP requests from clients. Its capture rate is up to 30 fps with 640×480 pixels. The JPEG compression factor can be adjusted between 0 (the best quality) and 100 (the worst quality). In our experiments, we set the image resolution to 320×240 and the JPEG compression factor to 10. This compression factor was enough to prevent visible compression artifacts such as blocking. In practice, we chose the relatively high-quality

$^{\dagger}\boldsymbol{r}_i(\boldsymbol{x}, z_{opt}(\boldsymbol{x}))$ is abbreviated as $\boldsymbol{r}_i(\cdot)$.

factor because the network bandwidth was not a bottleneck in our system.

The cameras have 100 Mbps Ethernet ports. We connected them to a single PC using gigabit Ethernet switches. We used an Intel Xeon 5160 (3 GHz) dual processor machine with 3 GB main memory and an NVIDIA GeForce 8800 Ultra graphics card.

4.2 Camera Calibration

For geometric calibration of the cameras, we tried two standard methods proposed by Tsai [40] and Zhang [41]. Although Zhang's method can be easily used by capturing a checkerboard pattern at several arbitrary positions, we found that the calibration parameters computed by the method produced good rendering results only within the volume where the checkerboard was placed during the calibration, as pointed out in [18]. For our camera array system that aims to capture and render a large space, Tsai's method, which requires known 3D geometry of a calibration object, worked better. We therefore used Tsai's method by capturing a checkerboard pattern at several depth positions with known translations.

We did not use color calibration and only relied on automatic white balance and exposure control of each camera, which is a similar setting to [27]. This is because the cameras do not have the flexible control of capturing parameters. In Sect. 5.1, we show that our method synthesizes visually good images even from input images that have large variations.

4.3 Software Architecture

Figure 4 shows the software architecture and data flow of our system. The system performs network I/O (receiving 64 motion JPEG sequences from cameras) and JPEG decoding in parallel on the CPU. The decoded images are uploaded to the GPU texture memory. The following rendering process is fully implemented on the GPU, so there is no data transmission from the GPU to the CPU. This allows our system to efficiently perform the processes as a pipeline by using the CPU and GPU independently.

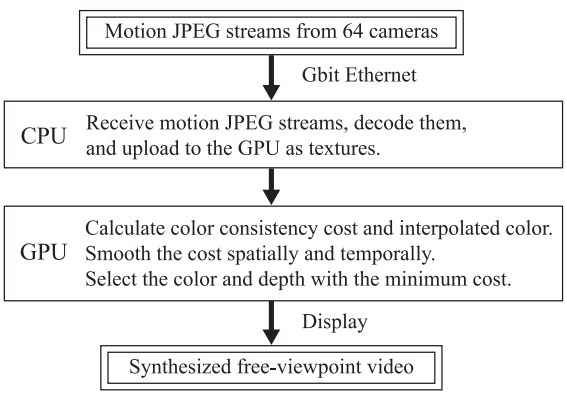


Fig. 4 Software architecture and data flow. No need for data transmission from the GPU to the CPU allows our system to efficiently perform the processes as a pipeline by using these processors independently.

The Axis 210 cameras do not have a synchronization function. Our rendering process therefore uses the most recently uploaded images. This approach mostly works well because our method interpolates the colors with minimum variance in any case and we use a relatively high frame rate.

4.4 GPU Implementation of Rendering

We implemented the rendering algorithm using OpenGL and fragment programs with Cg (C for graphics) [42]. For rendering a novel view, we first calculate the color consistency costs (Eq. (2)) and the interpolated colors for each depth layer in a single rendering pass. The interpolated colors are therefore generated for all depth values, instead of only for the optimal depth value (Eq. (6)), for computational efficiency. As shown in Fig. 2, the four-nearest input cameras are used to compute the depth and color of the target light rays that pass through the quadrangle whose vertices are the positions of the cameras. A novel image is therefore generated as a set of quadrangle regions, each of which is rendered by using the four-nearest cameras (see Fig. 7 (c), for example). To each quadrangle, we apply the currently uploaded input images with projective texture mapping, and the images that encode the bilinear weight for each input camera (Fig. 3 (b)) with normal texture mapping. In this rendering pass, we use a fragment program that calculates the interpolated colors and the costs at the same time and stores them in the RGB channel and the alpha channel of the GPU frame buffer, respectively.

The above process does not require to synthesize quadrangle regions that are out of the field of view of the rendering camera. Therefore, to accelerate the process, we compute which quadrangle regions are necessary for the current rendering viewpoint based on the positions of the input cameras on the CPU, and perform the texture mapping only for the necessary regions. Yang et al. [28] presented a similar GPU implementation of the depth estimation and color interpolation methods, but generated the color of target light rays using the average color of reference light rays. Our method, by contrast, uses bilinear interpolation of the colors of reference light rays, which provides better rendering quality. Note that too large cost values are automatically truncated when the calculated colors and costs are rendered to the GPU frame buffer, because the frame buffer limits the calculated floating-point values to a range between 0 and 1. This suppresses noise and outliers for the following smoothing operation.

In the next rendering pass, the costs in each depth layer are spatially and temporally smoothed (Eqs. (4) and (7)), and then the optimal depth for each pixel are selected by comparing the smoothed cost with the current optimal cost (Eq. (5)). To apply the same instructions to each pixel, we set the projection matrix to orthogonal and use texture mapping of the data to be processed, which is a standard GPGPU technique. We use a fragment program that aggregates neighboring al-

pha values and add λ if the current depth layer is not equal to the previous depth estimate. The current optimal depths are stored in the texture memory together with their costs and are updated by iteration over the depth layers. The optimal color for each pixel is also selected similarly to the optimal depth in another rendering pass. Consequently, our algorithm uses $3N$ rendering passes for rendering a novel view, where N is the number of depth layers.

Pseudocode of the above implementation is listed in Appendix.

5. Experiments

In this section, we first show rendering results and validate the effectiveness of the depth estimation and bilinear color interpolation. The performance of our rendering algorithm is then evaluated by changing the number of depth layers and target light rays (output resolution). The processing time was proportional to the numbers of layers and target light rays, as discussed in Sect. 3.3, and our system rendered a free-viewpoint video at up to 30 fps depending on those parameters. Throughout the experiments, we used 320×240 pixels as input image resolution and a JPEG compression factor of 10.

Note that output free-viewpoint videos, as well as comparison videos that show the effectiveness of the temporal smoothness constraint, are available at

http://www.hc.ic.i.u-tokyo.ac.jp/project/camera-array/

5.1 Rendering Results

Figure 5 shows input and output images from *Meeting room* sequence. We can see correct parallax in the synthesized images from different viewpoints. All of the objects are clearly synthesized, although some artifacts are still visible as we discuss in Sect. 5.3. Although some estimated depth values are incorrect in textureless regions, the rendered colors are visually correct.

Figure 6 compares images synthesized with different numbers of depth layers. The image synthesized with a single depth layer (Fig. 6 (a)) suffers from blur and ghosting artifacts except for the left person on which the depth layer is placed. The rendering quality increases with the increase of the number of depth layers. However, as we have reported in [36], [37], the rendering quality gradually reaches a ceiling; that is, the earlier depth layers have a larger impact on the rendering quality. Therefore, the image synthesized with 7 depth layers (Fig. 6 (c)) and the one synthesized with 15 depth layers (Fig. 6 (d)) have no significant visual difference.

Figure 7 compares images synthesized with different color interpolation methods. As shown in Fig. 5 (a), the captured images have large variations due to individual differences between cameras. Therefore, as shown in Fig. 7 (a), average color interpolation produces annoying color discontinuities at the reference camera boundaries shown in Fig. 7 (c). Our method using bilinear interpolation, by contrast, produces better-looking images (Fig. 7 (b)).

The videos on our website compare free-viewpoint videos and depth maps computed either with or without the temporal smoothness constraint, and show that the constraint suppresses flickering artifacts. This advantage can be seen more clearly when we render videos at a fixed viewpoint. Meanwhile, the videos rendered at a moving viewpoint confirm that the temporal smoothness constraint can be used even when the rendering viewpoint smoothly moves.

5.2 Performance Measurement

Figures 8 and 9 plot the processing time of our rendering algorithm versus the number of target light rays and depth layers, respectively. Here we used a set of static input images (i.e. no texture uploading) and measured the average processing time of 100 executions of the rendering algorithm. The rendering viewpoint was set behind the center of the input cameras such that all of the input cameras are used to render the novel view[†]. The smoothing window size was fixed to 11×11. We measured the time by rendering the results to back buffers, because the refresh rate of the display (60 Hz) limits the processing time if we draw the resulting images on the display.

The processing time was proportional to both the number of target light rays (output resolution) and that of depth layers, as discussed in Sect. 3.3. These times are fast enough for real-time interactive rendering. The difference in processing time between bilinear and average interpolations is small, which means that bilinear interpolation can be used for higher-quality rendering. The most time-consuming process is the spatial cost smoothing, which needs to aggregate all neighboring values.

The total throughput of our system, including the CPU processing and texture uploading, was 30 fps using 300×300 output resolution and 7 depth layers, and 20 fps using 500×500 output resolution and 20 depth layers, for example. At a throughput of 30 fps, the total network bandwidth from the camera array to the PC was about 270–330 Mbps. The system throughput was limited by the GPU processing (rendering and texture uploading). The network I/O and JPEG decoding were not bottlenecks even at 30 fps in our system. When we tried to use higher-resolution input images, the JPEG decoding became the bottleneck as well as the GPU processing.

5.3 Discussion

Our system renders high-quality novel views in real time

[†]The complexity of our rendering method does not depend on the number of input cameras used to render a novel view, as discussed in Sect. 3.3. In theory, therefore, the processing time does not depend on the rendering viewpoint. In practice, however, the processing time slightly increases with increasing the number of input cameras used for rendering, because the software needs to perform more texture mapping functions with a larger number of input cameras.

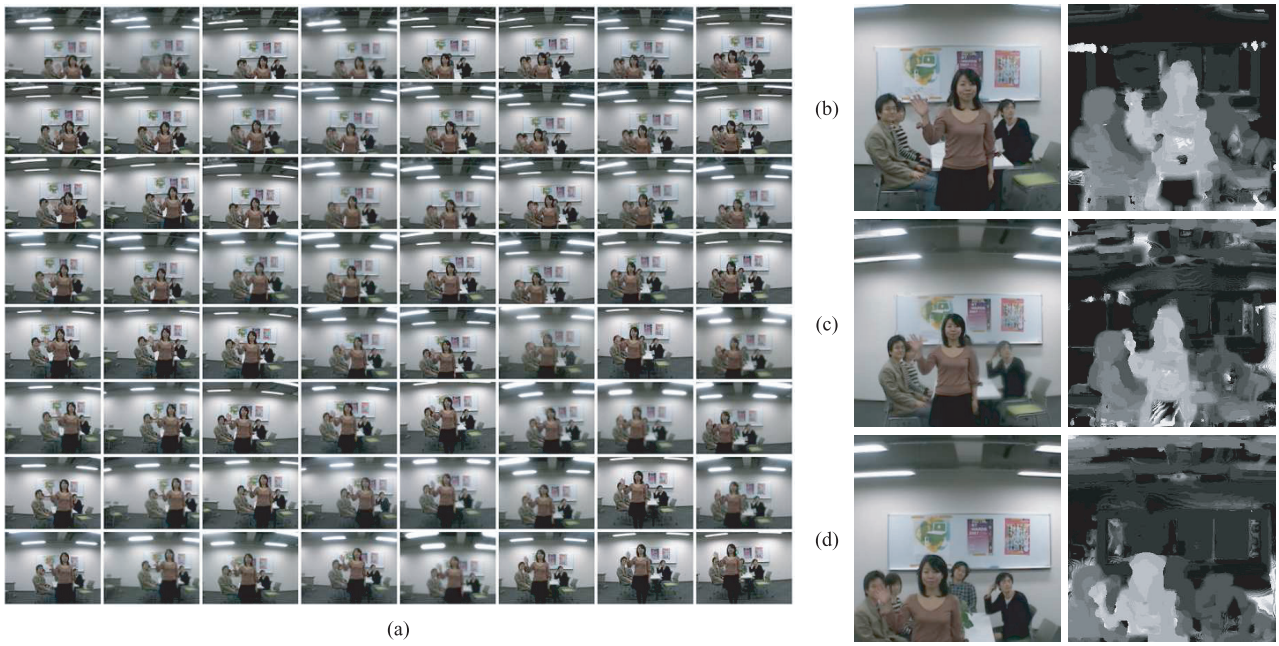


Fig. 5 Example images from *Meeting room* sequence. (a) Input 64 images. (b–d) Output synthesized images and their corresponding depth maps from various viewpoints.

Fig. 6 Synthesized images and their corresponding depth maps using (a) a single layer, (b) 3 layers, (c) 7 layers, and (d) 15 layers. Although the rendering quality increases with the increase of the number of depth layers, it gradually reaches a ceiling. Therefore, the synthesized images in (c) and (d) have no significant visual difference.

by using the view-dependent depth estimation and bilinear color interpolation methods. The interpolation method prevents color discontinuities at reference camera boundaries and generates visually natural images. We use bilinear interpolation for higher-quality rendering because the difference in processing time between average and bilinear interpolations is small. However, some artifacts are still visible in the rendered views. They are caused by the following three sources.

- *Incorrect depth estimation at depth boundaries*: Since we use a normal square window for smoothing the color consistency cost, the regions near depth boundaries tend to prefer the foreground depth value (the "foreground fattening" effect [38]). This incorrect

(a) (b) (c)

Fig. 7 Comparison of the color interpolation methods. Synthesized images with (a) average interpolation and (b) bilinear interpolation. (c) The camera grid superimposed on the synthesized image, where the intersections of horizontal and vertical lines correspond to the positions of input cameras. To render a quadrangle region of the grid, four cameras at its corners are used as reference cameras. Average interpolation causes annoying color discontinuities at camera boundaries, while bilinear interpolation generates better-looking images.

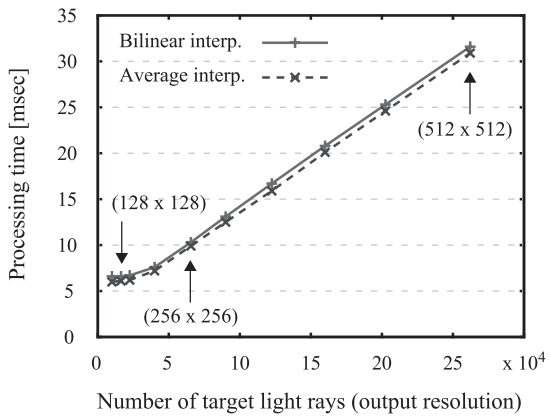


Fig. 8 Processing time of the rendering algorithm for different numbers of target light rays (output resolution). The number of depth layers was fixed to 15.

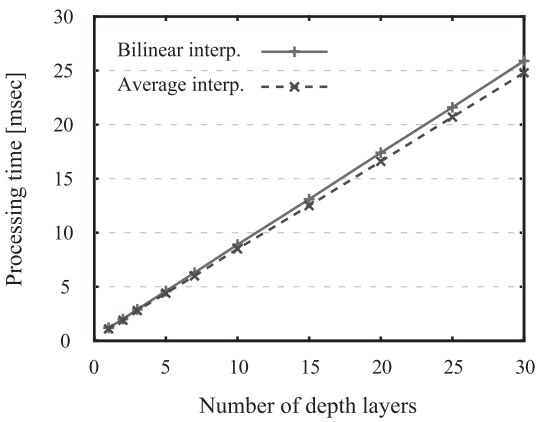
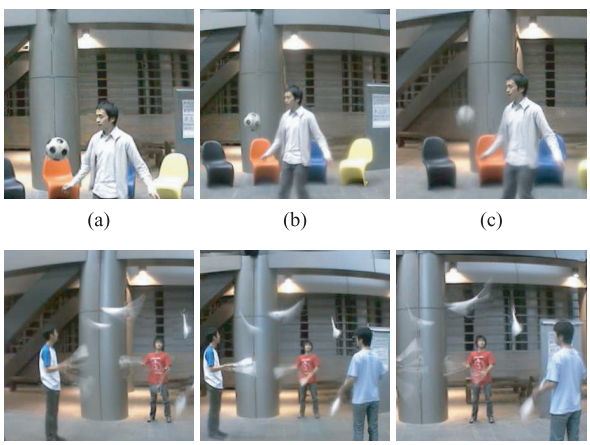


Fig. 9 Processing time of the rendering algorithm for different numbers of depth layers. The output resolution was fixed to 300 × 300.

depth estimation produces halo artifacts near objects boundaries (e.g., around the head of the foreground person in Fig. 5). A simple method for preventing the artifacts is using a shiftable window [38], which can be also implemented on a GPU efficiently (Gong [43], for example, showed a GPU implementation of the shiftable window for trinocular stereo sequences). In our system, however, the shiftable window often produced worse-looking images than the normal square window, because the outliers in the color consistency cost have a larger influence to the shiftable window. We therefore used the normal square window in our implementation.



(a) (b) (c)

(d) (e) (f)

Fig. 10 Synthesized images from (a–c) *Soccer* and (d–f) *Juggling* sequences. Images in each row are rendered from different viewpoints at the same time frame. Although these images have motion blur artifacts due to the fast motions, the output videos have visually acceptable quality.

- *Unsynchronized input videos*: Figure 10 shows images synthesized from two sequences that have fast motions. Since our camera array has no synchronization function, the frames used in the rendering process may be captured at different times. This causes motion blur artifacts in the fast moving parts (e.g., the soccer ball and the juggling clubs in Fig. 10). Even in such parts,

the bilinear interpolation produces better-looking results than the average interpolation, because it blends the misaligned parts more naturally. Moreover, because our system can run at a relatively high frame rate, the output videos are not so disturbing.

- *Blur in the input images*: Network cameras are suitable for building a compact camera array system that can run with only a single PC, because they can reduce the amount of data sent to the PC to a reasonable level by compressing the captured images. Unfortunately, however, the image quality of network cameras is generally not very high. Some of the input images of our system, in fact, suffer from blur and look unclean, as it can be seen in Fig. 5 (a). They produce low-quality regions in the final output images (e.g., the right person and chair in Fig. 5 (c) and the chairs in Fig. 10 (c)). We found that the degradation of the camera sensor causes the inevitable blur, and the only way to prevent this problem was replacing the low-quality camera with a better one. Such input images also increase outliers in the color consistency cost. We therefore used a relatively larger window of 11×11 pixels for smoothing the cost.

The processing time of our rendering method is proportional to the output resolution and the number of depth layers, as described in Sect. 5.2. However, note that synthesizing a higher-resolution image requires more depth layers [25] as well as a larger smoothing window. Therefore, the total processing time required for higher-resolution rendering is typically not proportional. Moreover, although the complexity of our rendering method is independent of the number of input cameras, a smaller number of input cameras (i.e. larger camera intervals) requires more depth layers, resulting in longer processing time.

Since synthesizing a novel view requires only parts of segments in the input images, several systems [23], [26], [27] use the region of interest (ROI) approach to reduce the amount of processed data. We could, for example, partially decode the received JPEG images and only upload the decoded segments to the GPU memory by using the ROI approach. However, we did not use such an approach because the current viewpoint is needed to determine the ROI. In this case, the rendering result reflects the current viewpoint after the CPU processing (i.e. JPEG decoding and texture uploading), which causes less interactivity. Meanwhile, our implementation performs the JPEG decoding, texture uploading, and novel view rendering independently. Because the rendering process only requires the current viewpoint, our approach has less delay than the ROI approach. Moreover, our system has a sufficient rendering rate without reducing the data amount thanks to the advancement of hardware and GPU functions.

6. Conclusions

We have presented a real-time video-based rendering system using an array of 64 commodity network cameras. Our system renders high-quality novel views from live multiview videos by using an on-the-fly per-pixel depth estimation method. The rendering algorithm is fully implemented on the GPU, which allows our system to efficiently perform the capturing and rendering processes as a pipeline and to render novel views at up to 30 fps depending on the rendering parameters. Since our system setup is simple, we can capture various scenes by moving the camera array. The rendering results show that our method produces visually natural images even from the frames that have large variations and are captured at slightly different timings.

Acknowledgments

We would like to thank Prof. Hiroshi Harashima for valuable discussions, Jinge Wang for his contribution to the camera array development, and the anonymous reviewers for their helpful comments and suggestions. We also thank Hideki Hashimoto, Fumiya Ichino, and Takumi Kida for their juggling performance.

References

[1] Y. Taguchi, K. Takahashi, and T. Naemura, "Real-time all-in-focus video-based rendering using a network camera array," Proc. 3DTV-Conference 2008, pp.241–244, May 2008.
[2] E.H. Adelson and J.R. Bergen, "The plenoptic function and the elements of early vision," in Computational Models of Visual Processing, ed. M. Landy and J.A. Movshon, pp.3–20, MIT Press, 1991.
[3] T. Fujii, A Basic Study on the Integrated 3D Visual Communication, Ph.D. thesis, Department of Electrical Engineering, School of Engineering, The University of Tokyo, 1994.
[4] M. Levoy and P. Hanrahan, "Light field rendering," Proc. ACM SIGGRAPH 96, pp.31–42, Aug. 1996.
[5] S.J. Gortler, R. Grzeszczuk, R. Szeliski, and M.F. Cohen, "The lumigraph," Proc. ACM SIGGRAPH 96, pp.43–54, Aug. 1996.
[6] T. Kanade, P. Rander, and P.J. Narayanan, "Virtualized reality: Constructing virtual worlds from real scenes," IEEE Multimedia, vol.4, no.1, pp.34–47, Jan. 1997.
[7] A. Laurentini, "The visual hull concept for silhouette-based image understanding," IEEE Trans. Pattern Anal. Mach. Intell., vol.16, no.2, pp.150–162, Feb. 1994.
[8] K.N. Kutulakos and S.M. Seitz, "A theory of shape by space carving," Int. J. Comput. Vis., vol.38, no.3, pp.199–218, July 2000.
[9] W. Matusik, C. Buehler, R. Raskar, S.J. Gortler, and L. McMillan, "Image-based visual hulls," Proc. ACM SIGGRAPH 2000, pp.369–374, July 2000.
[10] S. Vedula, S. Baker, S. Seitz, and T. Kanade, "Shape and motion carving in 6D," Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR 2000), vol.2, pp.592–598, June 2000.
[11] B. Goldluecke and M. Magnor, "Real-time microfacet billboarding for free-viewpoint video rendering," Proc. IEEE Int. Conf. Image Processing (ICIP 2003), vol.3, pp.713–716, Sept. 2003.
[12] B. Goldluecke and M. Magnor, "Space-time isosurface evolution for temporally coherent 3D reconstruction," Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR 2004), vol.1, pp.350–355, June 2004.
[13] P. Einarsson, C.F. Chabert, A. Jones, W.C. Ma, B. Lamond, T. Hawkins, M. Bolas, S. Sylwan, and P. Debevec, "Relighting human locomotion with flowed reflectance fields," Proc. 17th Eurographics Symposium on Rendering, June 2006.
[14] B. Wilburn, M. Smulski, H.H.K. Lee, and M.A. Horowitz, "The light field video camera," Proc. SPIE Media Processors 2002, vol.4674,

pp.29–36, Jan. 2002.
[15] B. Wilburn, N. Joshi, V. Vaish, E.V. Talvala, E. Antunez, A. Barth, A. Adams, M. Horowitz, and M. Levoy, "High performance imaging using large camera arrays," Proc. ACM SIGGRAPH 2005, pp.765–776, July 2005.
[16] B. Goldluecke, M. Magnor, and B. Wilburn, "Hardware-accelerated dynamic light field rendering," Proc. Vision, Modeling, and Visualization (VMV 2002), pp.455–462, Nov. 2002.
[17] B. Wilburn, N. Joshi, V. Vaish, M. Levoy, and M. Horowitz, "High-speed videography using a dense camera array," Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR 2004), vol.2, pp.294–301, July 2004.
[18] V. Vaish, B. Wilburn, N. Joshi, and M. Levoy, "Using plane + parallax for calibrating dense camera arrays," Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR 2004), vol.1, pp.2–9, June 2004.
[19] C.L. Zitnick, S.B. Kang, M. Uyttendaele, S. Winder, and R. Szeliski, "High-quality video view interpolation using a layered representation," Proc. ACM SIGGRAPH 2004, pp.600–608, Aug. 2004.
[20] M. Tanimoto, "FTV (free viewpoint television) creating ray-based image engineering," Proc. IEEE Int. Conf. Image Processing (ICIP 2005), vol.II, pp.25–28, Oct. 2005.
[21] N. Fukushima, T. Yendo, T. Fujii, and M. Tanimoto, "Free viewpoint image generation using multi-pass dynamic programming," Proc. SPIE Stereoscopic Displays and Virtual Reality Systems XIV, vol.6490, pp.460–470, March 2007.
[22] R. Ng, M. Levoy, M. Brédif, G. Duval, M. Horowitz, and P. Hanrahan, "Light field photography with a hand-held plenoptic camera," Tech. Rep. CSTR 2005-02, Stanford University Computer Science, April 2005.
[23] J.C. Yang, M. Everett, C. Buehler, and L. McMillan, "A real-time distributed light field camera," Proc. 13th Eurographics Workshop on Rendering, pp.77–85, June 2002.
[24] A. Isaksen, L. McMillan, and S.J. Gortler, "Dynamically reparameterized light fields," Proc. ACM SIGGRAPH 2000, pp.297–306, July 2000.
[25] J.X. Chai, X. Tong, S.C. Chan, and H.Y. Shum, "Plenoptic sampling," Proc. ACM SIGGRAPH 2000, pp.307–318, July 2000.
[26] H. Schirmacher, M. Li, and H.P. Seidel, "On-the-fly processing of generalized lumigraphs," Proc. Eurographics 2001, vol.20, pp.165–173, Sept. 2001.
[27] C. Zhang and T. Chen, "A self-reconfigurable camera array," Proc. 15th Eurographics Symposium on Rendering, pp.243–254, June 2004.
[28] R. Yang, G. Welch, and G. Bishop, "Real-time consensus-based scene reconstruction using commodity graphics hardware," Proc. Pacific Graphics 2002, pp.225–235, Oct. 2002.
[29] R.T. Collins, "A space-sweep approach to true multi-image matching," Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR '96), pp.358–363, June 1996.
[30] S.M. Seitz and C.R. Dyer, "Photorealistic scene reconstruction by voxel coloring," Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR '97), pp.1067–1073, June 1997.
[31] T. Naemura and H. Harashima, "Real-time video-based rendering for augmented spatial communication," Proc. SPIE Visual Commun. and Image Process. (VCIP '99), vol.3653, pp.620–631, Jan. 1999.
[32] T. Naemura, J. Tago, and H. Harashima, "Real-time video-based modeling and rendering of 3D scenes," IEEE Comput. Graph. Appl., vol.22, no.2, pp.66–73, March 2002.
[33] T. Yamamoto and T. Naemura, "Real-time capturing and interactive synthesis of 3D scenes using integral photography," Proc. SPIE Stereoscopic Displays and Virtual Reality Systems XI, vol.5291, pp.155–166, Jan. 2004.
[34] T. Yamamoto, M. Kojima, and T. Naemura, "LIFLET: Light field live with thousands of lenslets," ACM SIGGRAPH 2004 Emerging Technologies, Aug. 2004.
[35] K. Takahashi, A. Kubota, and T. Naemura, "All in-focus view synthesis from under-sampled light fields," Proc. Int. Conf. Artificial Reality and Telexistence (ICAT 2003), pp.249–256, Dec. 2003.
[36] K. Takahashi, A. Kubota, and T. Naemura, "A focus measure for light field rendering," Proc. IEEE Int. Conf. Image Processing (ICIP 2004), vol.4, pp.2475–2478, Oct. 2004.
[37] K. Takahashi and T. Naemura, "Layered light-field rendering with focus measurement," EURASIP Signal Processing: Image Commun., vol.21, no.6, pp.519–530, July 2006.
[38] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," Int. J. Comput. Vis., vol.47, no.1–3, pp.7–42, April 2002.
[39] C. Buehler, M. Bosse, L. McMillan, S.J. Gortler, and M.F. Cohen, "Unstructured lumigraph rendering," Proc. ACM SIGGRAPH 2001, pp.425–432, Aug. 2001.
[40] R.Y. Tsai, "A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses," IEEE J. Robot. Autom., vol.3, no.4, pp.323–344, Aug. 1987.
[41] Z. Zhang, "A flexible new technique for camera calibration," IEEE Trans. Pattern Anal. Mach. Intell., vol.22, no.11, pp.1330–1334, Nov. 2000.
[42] http://developer.nvidia.com/page/cg_main.html
[43] M. Gong, "A GPU-based algorithm for estimating 3D geometry and motion in near real-time," Proc. 3rd Canadian Conf. Computer and Robot Vision, June 2006.
[44] http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt

Appendix: Pseudocode

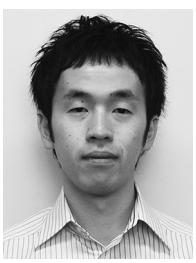
Algorithm 1 is pseudocode of OpenGL commands for our rendering algorithm, and Algorithm 2 shows the details of the fragment programs. Texture names in Algorithm 2 are defined in Algorithm 1. The textures are passed to the fragment programs by texture mapping functions, and the fragment programs load the value corresponding to each pixel on the rendered image from the textures. We could use an OpenGL extension EXT_framebuffer_object [44] for outputting the processed data directly to the textures, instead of copying the output data from the rendered frame buffer to the textures in Algorithm 1. In the experiments in Sect. 5.2, however, we did not use it because the processing times were similar.

Algorithm 1 Rendering algorithm

```
//Allocate texture memory on the GPU
Texture tInputImage [64]
Texture tBilinearWeight //Initialized with the image in Fig. 3 (b)
Texture tLayer, tCurOptImage, tCurOptDepth, tPrevDepth

procedure RENDERING
  for all camera idx do
    UploadTexture (tInputImage [camera idx])
  end for
  //The view synthesis procedure is invoked
  //after each uploading of 64 input views
  VIEWSYNTHESIS(current viewpoint)
end procedure

procedure VIEWSYNTHESIS(viewpoint)
  for all depth layer do
    SetFragmentProgram (CALCCOSTINTERPCOLOR)
    //The sets of 4-nearest input cameras used for rendering
    //the current view are computed from the camera positions
    for all 4-nearest cameras contributing to the view do
      for idx ← 1, 4 do
        cIdx ← FindCameraIdx (target 4 cameras, idx)
        //Set matrices based on the current viewpoint
        //and the calibration parameters of the camera
        SetMatrices (viewpoint, CalibParams [cIdx])
        ProjectiveTextureMapping (tInputImage [cIdx])
        //The weight texture is mapped with different
        //directions for each camera
        SetMatrices (viewpoint, CalibParams [cIdx], idx)
        TextureMapping (tBilinearWeight)
      end for
    end for
    CopyFrameToTexture (tLayer)

    //Set matrices to orthogonal for applying the same
    //instructions to each pixel in the textures
    SetMatrices (Orthogonal 2D)

    //Smooth the cost and select the depth with the
    //minimum cost
    SetFragmentProgram (SMOOTHCOSTRECONDEPTH)
    TextureMapping (tLayer, tCurOptDepth, tPrevDepth)
    CopyFrameToTexture (tCurOptDepth)
    //Store the estimated depth map for the next iteration
    if last depth layer then
      CopyFrameToTexture (tPrevDepth)
    end if

    //Select the color with the minimum cost
    SetFragmentProgram (COMPOSITELAYERCOLOR)
    TextureMapping (tLayer, tCurOptDepth, tCurOptImage)
    CopyFrameToTexture (tCurOptImage)
  end for
end procedure
```

Algorithm 2 Fragment programs

```
procedure CALCCOSTINTERPCOLOR
  //Calculate color consistency cost
  cost ← Sum_{c←r,g,b} (Variance_i (tInputImage[i].c))
  //Interpolate color
  color ← Sum_i (tInputImage[i].rgb · tBilinearWeight[i])

  //Output the calculated values to (RGB, A) channels
  //of the frame buffer
  Output (color, cost)
end procedure

procedure SMOOTHCOSTRECONDEPTH
  //Smooth cost in the neighborhood window W
  smoothCost ← Average_W (tLayer.a)

  //Temporal depth smoothing
  if current depth value ≠ tPrevDepth then
    smoothCost ← smoothCost + λ
  end if

  //Update the current depth map
  if smoothCost < tCurOptDepth.a then
    Output (current depth value, smoothCost)
  else
    Output tCurOptDepth
  end if
end procedure

procedure COMPOSITELAYERCOLOR
  //Update the current color image
  //Smoothed cost is stored in tCurOptDepth.a
  if tCurOptDepth.a < tCurOptImage.a then
    Output (tLayer.rgb, tCurOptDepth.a)
  else
    Output tCurOptImage
  end if
end procedure
```



Yuichi Taguchi received the B.E. and M.E. degrees in information and communication engineering from The University of Tokyo, Japan, in 2004 and 2006, respectively. He is currently working toward the Ph.D. degree in Graduate School of Information Science and Technology, the University of Tokyo, as a Research Fellow of the Japan Society for the Promotion of Science. His research interests include image-based rendering and 3D image processing.

Keita Takahashi received the B.E., Master, and Ph.D. degrees in information and communication engineering from The University of Tokyo, Japan, in 2001, 2003, and 2006, respectively. He is currently a Project Assistant Professor of IRT Research Initiative, the University of Tokyo. His research interests include 3D image production, object recognition, and video segmentation.

Takeshi Naemura received his B.E., M.E., and Ph.D. in Electronic Engineering from The University of Tokyo in 1992, 1994 and 1997, respectively. He is currently an Associate Professor in Information and Communication Engineering at the University of Tokyo. His research interests include image-based rendering, 3D image processing, human interface, mixed reality and computational photography.