Relighting 3D Scenes with a Continuously Moving Camera

Soonhyun Kim, Min-Ho Kyung, and Joo-Haeng Lee

This paper proposes a novel technique for 3D scene relighting with interactive viewpoint changes. The proposed technique is based on a deep framebuffer framework for fast relighting computation which adopts image-based techniques to provide arbitrary view-changing. In the preprocessing stage, the shading parameters required for the surface shaders, such as surface color, normal, depth, ambient/diffuse/specular coefficients, and roughness, are cached into multiple deep framebuffers generated by several caching cameras which are created in an automatic manner. When the user designs the lighting setup, the relighting renderer builds a map to connect a screen pixel for the current rendering camera to the corresponding deep framebuffer pixel and then computes illumination at each pixel with the cache values taken from the deep framebuffers. All the relighting computations except the deep framebuffer pre-computation are carried out at interactive rates by the GPU.

Keywords: Relighting, rendering, image-based technique, automatic camera placement.

Manuscript received Dec. 4, 2008; revised Apr. 14, 2009; accepted June 24, 2009.

This work was supported by the New Faculty Research Fund of Ajou University in Rep. of Korea and also partly supported by the IT R&D Program of MKE/MCST/IITA, Rep. of Korea [2009-S-045-1, Development of Function Extensible High Speed Renderer.].

Soonhyun Kim (email: kkubs@ajou.ac.kr) and Min-Ho Kyung (phone: +82 31 219 1853, email: kyung@ajou.ac.kr) are with the Visual Computing Lab., Ajou University, Suwon, Rep. of Korea.

Joo-Haeng Lee (email: joohaeng@etri.re.kr) is with the S/W & Content Research Laboratory, ETRI, Daejeon, Rep. of Korea.

doi:10.4218/etrij.09.0108.0696

I. Introduction

In 3D animation productions, designing light setups for 3D scenes is a very time-consuming and labor-intensive task, mostly carried out at a late stage of production pipelines. The major difficulty of lighting design comes from a lack of rapid visual feedbacks with high fidelity. Graphics hardware is not relevant for rendering a complex 3D cinema scene because of low rendering quality and limited in-core memory. Software rendering gives very accurate feedback, but it cannot be so frequently used because of long rendering times from several minutes to hours. Therefore, faster and more reliable feedback methods have been demanded by lighting artists to improve both efficiency and quality of lighting design.

The relighting technique presented in this paper is based on deep framebuffers used as cache storages for shading parameter values pre-computed at each pixel for fast illumination computation. The cached shading parameters include ambient color, diffuse color, specular color, roughness, pixel depths, and so on. Since the deep framebuffer is made at a specified viewpoint, the deep framebuffer image should be computed again if the rendering camera is moved.

The major contribution of our work is to make it possible to freely move the rendering camera without time-consuming re-caching of the shading parameters. Previous deep framebuffer-based relighting methods allowed only a fixed viewpoint used for parameter caching. Some vertex-based indirect light transport methods can relight a 3D scene in real time, allowing the camera to move freely. However, they usually require a lot of pre-computation time and cannot capture high frequency surface shading features commonly occurring in cinematic scenes. To overcome such limitations, we adopt an image-based technique to retrieve the required parameter values from existing deep framebuffers in run time. The set of deep framebuffers are cached with automatically generated cameras. Shading parameter values are accessed through a correspondence map containing the cache location of a screen pixel for the current rendering camera. The correspondence map is rebuilt at every movement of the rendering camera.

We have tested our algorithm on various scenes with up to 1.5 million polygons shaded by Lambert, Phong, and Blinn models lit by directional, point, and spot light sources [1]. Environment lighting is not yet supported, but it can be implemented with a little effort. In all these scenes, our algorithm has achieved interactive performance with acceptable image quality for lighting design.

The remainder of this paper is organized as follows. Previous work related with relighting and image-based rendering techniques is reviewed in section II, and then the details of the proposed relighting technique are described in section III. Section IV discusses the results with two test scenes, and section V concludes the paper with summary of our work and remaining problems for future work.

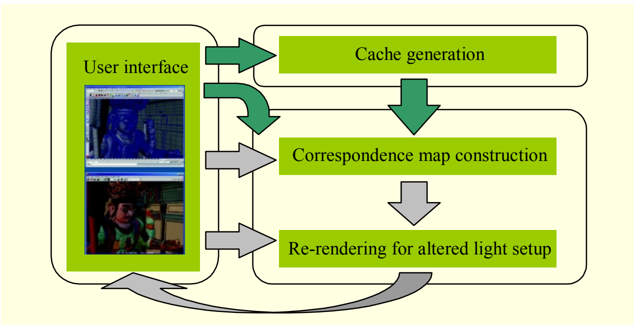
II. Related Work

Several existing techniques for re-rendering 3D scenes have been developed using deep framebuffers, mostly to cache the parameter values that are required in order to compute illumination at each pixel. The notion of deep framebuffers originated from the geometric buffer (G-buffer) [2], which stores pre-computed geometric properties such as object IDs, depths, and surface normals for fast comprehensible rendering. Séquin and Smyrl [3] proposed a similar idea of maintaining a ray tree at each pixel to avoid repeated ray intersections in re-rendering 3D scenes. As modern graphics hardware grows more powerful and versatile, relighting 3D scenes at interactive rates becomes feasible through GPU-based rendering. Gershbein and Hanrahan [4] first proposed a deep framebuffer-based relighting engine for cinematic lighting design. Ragan-Kelly [5] developed another GPU-based relighting system which used data-flow analysis to separate light-independent components for caching. Similarly, a cinematic relighting engine supporting complex RenderMan scenes, called Lpics, was developed by Pellacini and others [6]. They also extended it to indirect lighting in [7] by pre-computing multi-bounce light transports and representing them with wavelet compressed matrices.

Early image-based rendering techniques were introduced by Adelson and Bergen [8]. The 7D plenoptic function defines all radiance observed at every location from every direction. McMillan [9] presented a 3D warping technique which warps a set of images given with depth information onto a new picture from a novel viewpoint. Shum and Kang [10] carried out a comprehensive review of most classic works on image-based techniques. These previous image-based techniques were focused on capturing real-world visual appearances and adapting them to fast and realistic image synthesis. Instead of capturing real world appearances, we use the image-based technique to capture surface shading parameters from different viewpoints in a 3D scene for fast relighting.

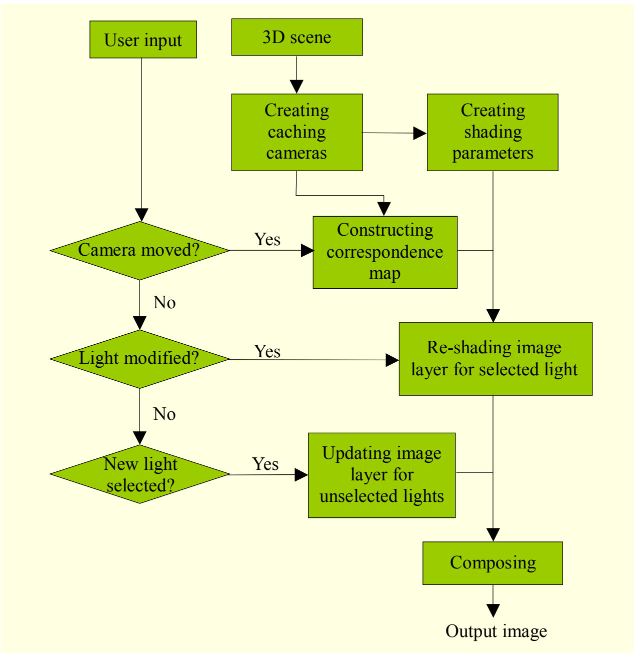
III. Image-Based Relighting Engine

Our relighting renderer is developed as a plug-in module to an animation production system, Autodesk Maya (see Fig. 1). Relighting is computed as follows. First, the relighting renderer finds a set of caching cameras used to capture shading parameter values on surfaces. Then, with each camera, it



User interface
Cache generation
Correspondence map construction
Re-rendering for altered light setup

Fig. 1. A 3D animation production system integrated with the proposed relighting renderer. The green arrows represent the pre-computation flow, and the grey arrows represent the interactive flow.



User input
3D scene
Creating caching cameras
Creating shading parameters
Camera moved?
Yes
Constructing correspondence map
No
Light modified?
Yes
Re-shading image layer for selected light
No
New light selected?
Yes
Updating image layer for unselected lights
Composing
Output image

Fig. 2. Workflow of the relighting renderer.

renders the scene into a deep framebuffer to cache shading parameters. During the lighting design, constructing a correspondence map and updating the rendered image are repeatedly executed in real time whenever required. The overall workflow is illustrated in Fig. 2.

In the following subsections, we will discuss the details of each step.

1. Caching Cameras

Deep framebuffer images are obtained by rendering the target scene with a set of caching cameras. Since the deep framebuffers have shading parameters used for re-rendering, the set of caching cameras should be carefully determined such that the deep framebuffers capture most of the important surface areas with sufficient accuracy. Otherwise, a portion of surface area not captured in the deep framebuffers will make a hole in the resulting image because the shading parameters are not available. Also, if the image resolution used for caching is too low to capture featured details, they will be blurred in the resulting image.

A brute-force method to place caching cameras uses a large number of cameras, ensuring that all surface areas are covered, and places them at arbitrary positions in the scene workspace. This is impractical in reality. As an extreme case, suppose that several hundreds cameras completely cover all the surfaces in a typical 3D scene. The deep framebuffers generated by those cameras will require several gigabytes of memory, which exceeds the on-board memory size of current commodity graphics hardware. Even worse, with so many caching cameras, updating the correspondence map at every movement of the rendering camera is computed far more slowly than at any interactive rate. This is because the computation time grows proportionally to the number of caching cameras.

Finding an optimal set of caching cameras satisfying image quality and performance requirements is a challenge. In fact, the problem is equivalent to the classic "art gallery problem" known to be NP-complete in computational geometry [11]. Thus, it is not possible to solve the problem exactly within a reasonable time. Instead, less optimal but practical solutions have been investigated in image-based modeling and rendering applications. In [12], Fleishman and others proposed an automatic camera placement method for image-based modeling. They restricted the workspace to a curved trajectory and placed capturing cameras along the trajectory. Unfortunately, this restriction does not work for lighting design because designers move the rendering camera freely in 3D space to see illuminated surfaces from various viewpoints.

We propose a practical heuristic solution suitable for the purpose of lighting design. The heuristic algorithm may be summarized as follows.

Step 1. Find the primary camera set covering the majority of scene surfaces.

Step 2. Find the secondary camera set covering the rest of scene surfaces uncovered by the primary camera set.

Step 3. Determine the camera attributes such as a viewing direction, near and far clips, and a field of view.

Step 4. If uncovered areas remain, go to step 2.

This algorithm does not aim to find a complete set of cameras covering the entire scene including all the small details because the number of cameras can grow very rapidly. What it really aims to achieve is reasonable surface coverage with a small number of cameras allowing interactive performance. To achieve this, we intensively use a set of sample points as an approximation to the exact scene geometry. There are a few assumptions regarding the input scene. It is provided with a tight bounding box enclosing the surfaces of interest specified by the user. The "surfaces of interest" usually include the target objects of main lighting and the background objects supporting the target objects. This bounding box is used to eliminate irrelevant surface areas from consideration of camera selection. Another assumption is that the user may specify important surface areas by a weight value, which helps to find a better camera set that captures important surfaces more accurately.

A. Primary Camera Set

In our algorithm, primary caching cameras are a special type of camera with omni-directional view, that is, cameras which can see the surroundings in every direction. They are not standard cameras but can be realized as a cube map made with six perspective cameras at the center position. The challenge is to find the primary camera set that maximizes surface coverage with a reasonably small number of cameras. We conjecture that such a camera set can be mapped to the centers of inscribed

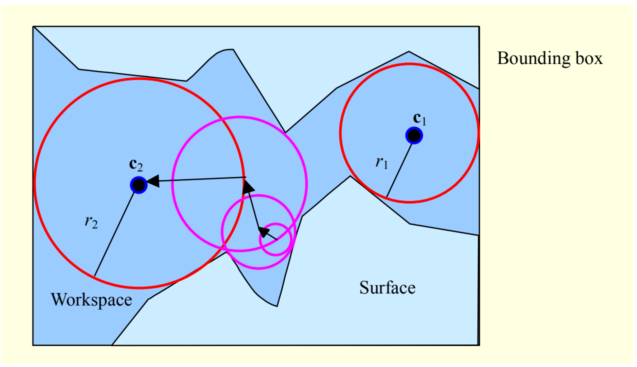


Fig. 3. Two maximal spheres marked as red circles are found at $\mathbf{c}_1$ and $\mathbf{c}_2$ with radius r_1 and r_2, respectively. The pink circles show a searching sequence of spheres starting from a seed with δ-radius and ending at $\mathbf{c}_2$.

spheres, maximally filling the scene workspace shown as the red circles in Fig. 3. This conjecture is based on the observation that, if a large sphere is tightly fit to a region of the workspace, the region usually has a large volume and is surrounded by large surface areas. Most of these surface areas are visible from the center of the sphere. Thus, a large portion of the surfaces can be covered by placing a camera at the center of this sphere.

Thus, the problem of finding primary cameras is reduced to the problem of finding maximal spheres that tightly fit the scene workspace. We developed an iterative scheme to solve this problem, and it may be summarized as follows.

Step 1. Sample a surface point randomly, and define a δ-radius seed sphere above a point in the workspace.

Step 2. Increase the sphere radius until the sphere comes into contact with surface points of the second largest pink circle.

Step 3. Move the sphere by δ in a way to disengage it from the contacting points.

Step 4. If there are no directions to move the sphere, report it and stop. Otherwise, go to step 2.

This algorithm guarantees convergence to a maximal sphere if the workspace is bounded. Otherwise, the sphere could grow to infinity without stopping. To make the workspace bounded, we enclose it with the bounding box provided with the scene. The initial position of the sphere is set to $\mathbf{p}_s + \delta \cdot \mathbf{n}_s$, where $\mathbf{p}_s$ is the sample surface point, and $\mathbf{n}_s$ is its normal vector.

In step 2, increasing the sphere radius by δ and then testing sphere/surface intersection are repeated until the test finds the sphere intersecting a surface. If intersections are found, a direction $\mathbf{d}$ satisfying the separation constraints, that is,

$$\mathbf{d} \cdot (\mathbf{r}_i - \mathbf{c}) \leq 0 \quad \text{for all intersections } i, \tag{1}$$

is chosen as a direction to move the sphere at step 3. Here, $\mathbf{c}$ and $\mathbf{r}_i$ denote the current sphere center and a surface point chosen from an intersecting area, respectively. Direction $\mathbf{d}$ can be computed by solving a simple linear problem with the above constraints. If there are no directions satisfying the constraints, the current sphere is a maximal sphere.

There may be more than one maximal sphere. To find more spheres, we run the search algorithm multiple times, starting at different sample points. Some of them may converge to the previously found spheres, and some of them may converge to new spheres. We stop repeating the sphere searching if k consecutive executions do not find new spheres. According to our experiments, k=50 was good enough to find primary cameras having large coverage.

B. Secondary Camera Set

A secondary camera is an ordinary perspective camera that is used to cover the surface areas that are not completely covered by the primary camera set. Because exact computation of the uncovered area is time-consuming and is not easy to implement robustly, we use a set of point samples on surfaces to identify uncovered areas approximately.

There are two different ways to sample points on surfaces, namely, uniform sampling and importance-based sampling [13], [14]. Uniform sampling is easy to implement, but sometimes it misleads camera selection by regarding even unimportant samples equally. Importance sampling can avoid such undesired camera selection with user-assigned weights. Therefore, we use an importance-based sampling method with a probability distribution function defined as

$$f(i) = \frac{w_i A_i}{\sum_{k=1}^{n} w_k A_k}, \tag{2}$$

where w_i and A_i are a weight in [0, 1] and an area of polygon i, respectively. Multiplying $f(i)$ by the total number of samples will give the number of samples on polygon i. Then, the interior of each polygon is uniformly sampled.

The sample points are classified into two groups, namely, a covered set and an uncovered set, by a visibility test with the current cameras including the primary cameras. The visibility test is accelerated with a kd tree containing the surface polygons. A ray from a sample point to a camera is tested for occlusion by other surfaces. If the ray is not occluded to the camera, the sample point is already covered by it. If all the rays from a sample point are occluded, the sample point is not yet covered, and thus it will be regarded as uncovered and to be covered by the next cameras.

If there are more uncovered points than a threshold m, we need additional cameras to cover them. The intuition to find the best location having the best view of uncovered sample points is that such a location will be where the most rays from uncovered points intersects. The procedure to find this location is carried out as follows.

First a set of n rays is sampled on the unit hemisphere and transformed to the tangent plane of each uncovered point. Then, the workspace is searched for the largest ray intersection point. Finding the intersection point, however, is not trivial because the rays are discretely sampled, so they do not exactly intersect in $\mathbf{R}^3$. Therefore, instead of exact intersection, we consider proximity of rays as an approximation of ray intersection. A 3D grid discretizing the workspace is used to record the number of rays passing closely in a distance δ. Figure 4 illustrates how to find a secondary camera. A 3D grid aligned with the scene bounding box is constructed with cell size $\delta = \max(width_{\text{bbox}}, height_{\text{bbox}})/n$. We used n=128 for our test scenes. Rays are shot from the sample points on uncovered areas.

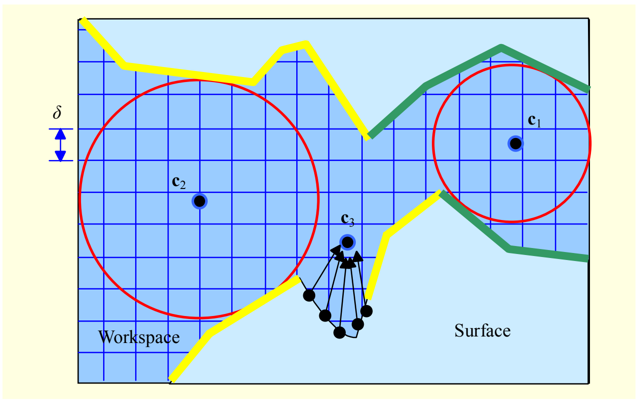


Fig. 4. The cameras at $\mathbf{c}_1$ and $\mathbf{c}_2$ cover most surface areas (yellow and green lines) except the areas with the black dots. The uncovered points are covered by the secondary camera $\mathbf{c}_3$.

While a ray is traversing the grid, the cell counters on the ray path are incremented. Finally, the counters record the number of rays passing through the cells. To reflect surface importance, we increment the counter by $\alpha \cdot w_i$, where the weight w_i is from the surface polygon at which the point was sampled, and α is a constant factor. A new camera is placed at the center of the cell whose counter is the largest and then added to the secondary camera set. Uncovered points visible from the new camera are also considered covered. If there are still more uncovered points than m, the program runs another iteration.

C. Camera Attributes

Once the locations of the caching cameras have been obtained, it is time to determine the other camera attributes including view direction, field-of-view (FOV) angle, near and far clips, and image resolution. For a primary camera, it is easy to determine these attributes. Since it is an omni-directional view camera, it has to be decomposed into six perspective cameras with view directions $-x$, $+x$, $-y$, $+y$, $-z$, and $+z$, respectively. Note that all of them have an FOV of 90°.

For a secondary camera c at position $\mathbf{c}$, we first compute the center of the sample points $\mathbf{p}_i^c$ covered by camera c as

$$\mathbf{p}_c = \frac{\sum_{i=1}^{n} w_i^c \mathbf{p}_i^c}{\sum_{i=1}^{n} w_i^c}, \tag{3}$$

where n is the number of covered sample points, and w_i^c's are the sample weights. Then, the FOV angle θ is computed as

$$\theta = \min\left(\max_{i=1,\cdots,n} \cos^{-1}\left(\frac{(\mathbf{p}_c - \mathbf{c})\cdot(\mathbf{p}_i^c - \mathbf{c})}{\|\mathbf{p}_c - \mathbf{c}\| \cdot \|\mathbf{p}_i^c - \mathbf{c}\|} \right), 120^\circ \right). \tag{4}$$

The reason to restrict the FOV angle within 120° is that cache images made with a wider FOV angle have a serious accuracy problem due to under-sampling errors. The covered sample points out of this bound will be marked again as uncovered and be taken back to the previous step to find an additional secondary camera.

Near and far clips are set as min(d_{near}, n_c) and max (d_{far}, f_c), where d_{near} and d_{far} are the nearest and farthest distances of the samples covered by each camera. Here, n_c and f_c denote the clipping limits determined with respect to the dimension of the scene bounding box, and they are usually set to 0.01 and 1000, respectively.

The rendering resolution is set to one of 2,048×2,048, 1,024×1,024, 512×512, and 256×256 with respect to the number of covered samples.

2. Shading Parameter Caching

Shading parameters to be cached are selected according to the illumination models used by the surface shaders in the scene. A variety of reflection models have been developed to simulate natural and artistic appearances of illuminated surfaces. Among them, we support the Lambert, Blinn, and Phone models in the current implementation, which are simple and are the most commonly used models in production studios. Other reflection models presented in [13] and [15] could be supported in the future by extending the caching parameter set.

We chose a set of eight parameters as in Lpics [6], including ambient color, shader ID, surface color, depth, diffuse color, surface normal, specular attributes, and specular color. Figure 5 shows the shading parameters used in our test scene. Ambient occlusion and other parameters which are not included in the current parameter set can be additionally added if required.

The deep framebuffer is iteratively constructed for each caching camera by using an ordinary rendering technique. All the shading parameters except pixel depths are cached with a

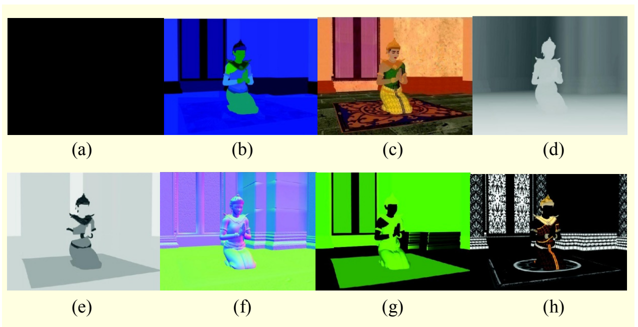


Fig. 5. Cache images of eight shading parameters for the scene in Fig. 10: (a) ambient color, (b) shader ID, (c) surface color, (d) depth, (e) diffuse color, (f) surface normal, (g) specular attributes, and (h) specular color.

standard software renderer. In our work, the Maya renderer is used with rendering options tailored to output the shading parameters into color channels. The depth image is rendered separately by OpenGL with a graphic hardware for speedup.

3. Correspondence Map Construction

The correspondence map has the cache location of the shading parameter values for screen pixels viewed from the current rendering camera (see Fig. 6). The cache location is represented by two values: the deep framebuffer ID and the (u, v) coordinates on the associated deep framebuffer.

The first step of construction is to find the visible surface samples from the current rendering camera. A visible surface sample $\mathbf{p}_w$ in the scene space is obtained by inversely projecting a screen pixel back to the scene space. Note that a depth map is rendered with the current rendering camera to provide the screen pixel depth before inverse projection is applied.

Then, for the visible surface sample $\mathbf{p}_w$, we find the best caching camera k with regard to three factors: visibility, surface normal, and distance. The computation is formulated as

$$k = \arg\max_i \left(\frac{v_i(\mathbf{p}_w) \cdot (\mathbf{n}(\mathbf{p}_w) \cdot \mathbf{u}_i)}{d_i^2} \right), \qquad (5)$$

where $v_i(\mathbf{p})$ is a visibility function giving 1 if a point $\mathbf{p}$ is visible from camera i and 0 otherwise, $\mathbf{n}(\mathbf{p}_w)$ is the surface normal at $\mathbf{p}_w$, $\mathbf{u}_i$ is the view direction (to the caching camera), and d_i is distance to $\mathbf{p}_w$ from the view point. The camera ID k and the projection coordinates of $\mathbf{p}_w$ onto the deep framebuffer are recorded on the correspondence map.

There may be surface samples that are not visible from any caching cameras, and this makes holes in the resulting image. If a hole is one- or two-pixel sized, the hole is simply filled by interpolating adjacent pixels. If a hole is bigger, a small temporary scene that includes the polygons that contain holes is constructed and rendered with the software renderer in run time to cache the shading parameters. Since the temporary scene is so small as to be rendered in under a second, the performance is only slightly slowed down.

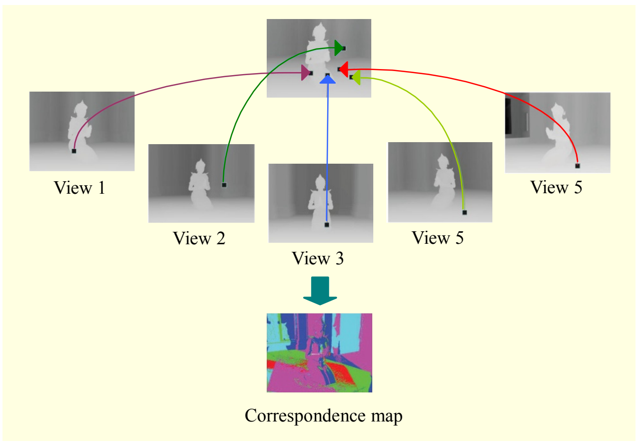


Fig. 6. Correspondence map constructed with a set of five caching cameras. The image of view i shows the depth map of the i-th caching camera. The colors in the correspondence map image indicate which caching camera is associated with each pixel.

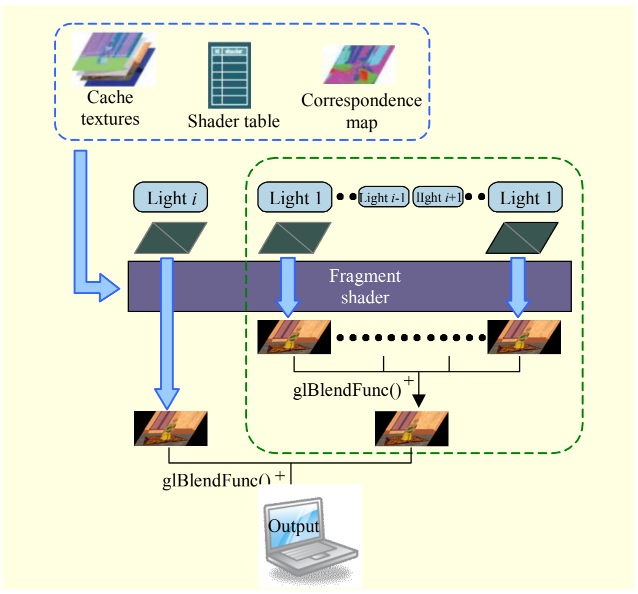


Fig. 7. Re-rendering process: image layers lit by each light are computed by the fragment shader and integrated into the final image. The layers in the green dotted box are computed for the unselected light, so they are computed once and reused until new selection occurs.

4. Re-rendering

In re-rendering, an image layer lit by each light source is computed, and all such layers are integrated into a final image (see Fig. 7). For each light source, we draw a rectangle filling the whole image space to trigger fragment-based shading at screen pixels. The fragment shading program computes illumination at every pixel with the light source and the cached shading parameter values. The cache location is read from the correspondence map with the pixel coordinates. The result of every rectangle is simply accumulated onto the framebuffer to produce a final image. If there are unmodified light sources for a while, their illumination results are stored in an extra buffer and reused until any changes are made on them (see the green dotted box in Fig. 7).

IV. Results

We tested the relighting renderer on a 3.0 GHz dual-core

Pentinum with 2 GB of RAM and an NVidia QuadroFX 5500 graphics card with 1 GB of video RAM. The first test scene has a flower vase modeled with 184,143 triangles and lit by three points and one directional light source. Refreshing the rendered image with a resolution of 640×480 took 0.03 s for correspondence map construction (Fig. 8(a)) and 0.02 s for re-rendering (Fig. 9). Five caching cameras were used for this example. Four were placed around the vase, and one was placed above the vase. They were manually placed because unfortunately our camera placement algorithm generated too many cameras to cover the leaves and the flower petals highly occluding each other. The relighted image shows little visual difference in comparison with the software-rendered image except small artifacts due to sampling errors around object boundaries.

The second example is a temple scene provided by an animation studio, which was made with 1,404,052 polygons and texture images of 473 MB. Nine caching cameras were used: one primary camera and eight secondary cameras. The computation time to find them was 354 s for the primary camera and 580 s for the secondary cameras. Refreshing the rendered image took 0.0425 s for correspondence map construction and 0.02 s for re-rendering (Fig. 10). The correspondence map used for this image is shown in Fig. 8(b), where the high frequency noises appearing on the carpet floor are due to bump-mapped surface normals. The refreshing time increased only by 25%, whereas the scene complexity increased by 662% from the first example. The small increase in the refreshing time demonstrates that our relighting renderer is nearly independent of the scene complexity except for depth-renderings which are to construct the correspondence map and shadow maps.

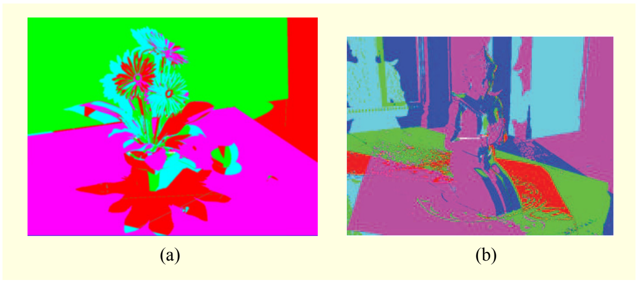


Fig. 8. Correspondence maps (a) for the vase scene in Fig. 9 and (b) for the temple scene in Fig. 10.

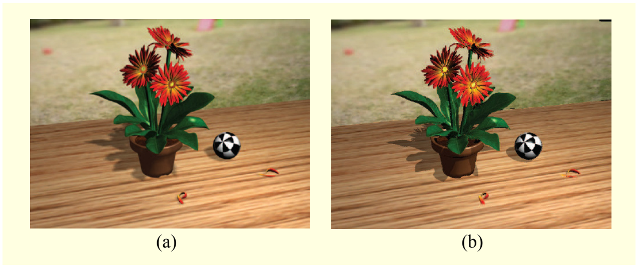


Fig. 9. Rendered images for the flower vase scene with 184,143 triangles: (a) software-rendered image and (b) relighted image.

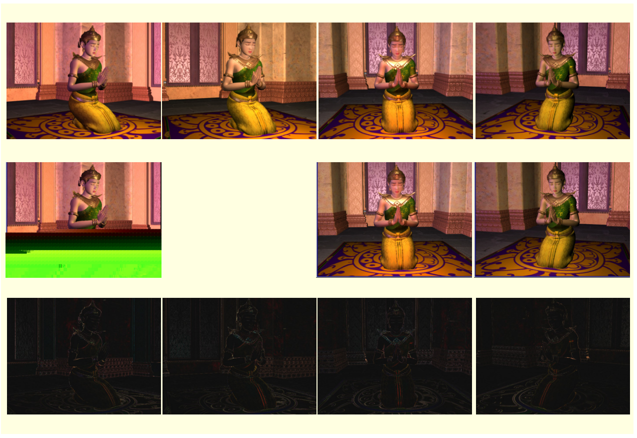


Fig. 10. Temple scene modeled with 1,404,052 polygons rendered from four different viewpoints: software-rendered images (top), relighted images (center), and difference images obtained by subtracting the center images from the top images (bottom). Nine cameras were used to cache the shading parameters of the scene.

The last example is a scene with one of four heavenly kings captured from a real wooden statue in a Korean temple (see Fig. 11). The scene shows only the face of the model because the full body is too heavy to work with for a real-time task. The image was made with 577,746 polygons, and 10 caching cameras were used: two primary cameras and eight secondary cameras. The computation time to find them was 578 s for the primary cameras and 680 s for the secondary cameras. Refreshing the rendered image took 0.043 s for correspondence map construction and 0.021 s for re-rendering. The performance slowdown is due to using more cameras than were used in the temple scene.

To verify relighting image quality, we computed residual images by subtracting the relighting images from the final-quality images rendered by the Maya software render. The residual images are shown in the third row of Figs. 10 and 11. The errors are barely noticeable over most image areas, though the surface boundaries show high errors. The high boundary errors are due to aliasing artifacts on the relighting images rendered with one sample per pixel. Since our relighting renderer is targeted to previewing for lighting design, the boundary accuracy is actually not the main concern. However, if higher quality without aliasing artifacts is required, we can use a super-sampling technique such that the renderer synthesizes an image four times bigger and reduce it to the original size. We also computed the peak signal-to-noise ratios

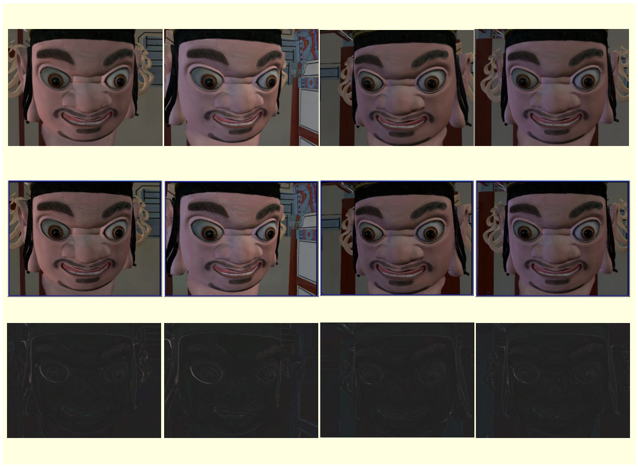


Fig. 11. Heavenly king scene modeled with 577,746 polygons rendered from four different viewpoints: software-rendered images (top), relighted images (center), and difference images obtained by subtracting the center images from the top images (bottom). Ten cameras were used to cache the shading parameters of the scene.

Table 1. PSNRs (dB) for the images in Figs. 10 and 11. The PSNRs are computed with YUV images converted from the RGB difference images.

	Frame 1	Frame 2	Frame 3	Frame 4
Temple scene	33.2	33.3	33.2	34.7
King scene	36.6	36.3	36.1	35.0

(PSNRs) of the relighting images over the final quality images, which are summarized in Table 1. The PSNRs are between 33.2 dB to 36.6 dB, which is comparable to moderate quality lossy compressed images. One major factor pulling down the PSNRs is the high peak errors on the surface boundaries; thus, if surface boundaries are disregarded, we can expect the PSNRs to be much higher. Also, the errors on small surface details also reduce the PSNRs, but such errors can be ignored for the main lighting setup over the whole scene. Therefore, we can conclude that the relighting images are visually equivalent to the software-rendered images for the purpose of lighting design.

V. Summary and Discussion

We presented a relighting technique which supports a moving camera in a deep framebuffer framework. Our relighting renderer caches the shading parameters into multiple deep framebuffers with automatically placed caching cameras. Because deep framebuffers provide the shading parameters required for illumination computation, time-consuming surface sampling, surface texturing, and complex shading computations are avoided in run-time, and this enables interactive performance. To allow relighting at an arbitrary viewpoint, we employed a correspondence map containing the cache locations for screen pixels viewed from the current rendering camera. The correspondence map is refreshed every time the rendering camera moves. We tested our relighting engine on several 3D scenes of high complexity to demonstrate its applicability to real 3D animation productions.

The relighting results with a moving camera were found to be of slightly lower quality than those rendered by a production quality renderer. Although the relatively low quality was still acceptable for lighting design, it raises further research problems which we will address in the future. Most artifacts affecting image quality are mainly due to intrinsic limitations of image-based approaches such as aliasing and blurring. To alleviate aliasing artifacts, we can use standard advanced anti-aliasing techniques such as mipmapping and anisotropic filtering. Ragan-Kelley and others [5] proposed a more efficient anti-aliasing technique using an indirect framebuffer with subpixel information for boundary pixels. The indirect framebuffer technique can also be adopted into our framework. Blurring is due to lack of required details in existing deep framebuffers; thus, we can reduce it by capturing further details with more caching cameras and by increasing the deep framebuffer resolution. However, using many caching cameras raises a question of how to efficiently determine the best camera in run time. Some heuristics such as nearest camera selection and camera clustering may be partial answers to this question.

References

[1] R. Barzel, "Lighting Controls for Computer Cinematography," *J. of Graphics Tools*, vol. 2, no. 1, 1997, pp. 1-20.

[2] T. Saito and T. Takahashi, "Comprehensible Rendering of 3-D Shapes," *Proc. Int. Conf. Computer Graphics Interactive Tech.*, 1990, pp. 197-206.

[3] C.H. Séquin and E.K. Smyrl, "Parameterized Ray Tracing," *Proc. Int. Conf. Computer Graphics Interactive Tech.*, 1989, pp. 307-314.

[4] R. Gershbein and P.M. Hanrahan, "A Fast Relighting Engine for Interactive Cinematic Lighting Design," *Proc. Int. Conf. Computer Graphics Interactive Tech.,* 2000, pp. 353-358.

[5] J. Ragan-Kelley et al., "The Lightspeed Automatic Interactive Lighting Preview System," *ACM Trans. Graphics*, vol. 26, no. 3, 2007, Article 25.

[6] F. Pellacini et al., "Lpics: A Hybrid Hardware-Accelerated Re-lighting Engine for Computer Cinematography," *ACM Trans. Graphics*, vol. 24, no. 3, 2005, pp. 464-470.

[7] M. Hasăn, F. Pellacini, and K. Bala, "Direct-to-Indirect Transfer

for Cinematic Relighting," *ACM Trans. Graphics*, vol. 25, no. 3, 2006, pp. 1089-1097.

[8] E.H. Adelson and J. Bergen, "The Plenoptic Function and the Elements of Early Vision," *Computational Models of Visual Process.*, MIT Press, 1991, pp. 3-20.

[9] L. McMillan, *An Image-Based Approach to Three-Dimensional Computer Graphics*, PhD thesis, University of North Carolina, Computer Science TR97-013, 1997.

[10] H.Y. Shum and S.B. Kang, "A Review of Image-Based Rendering Techniques," *Proc. IEEE/SPIE Visual Commun. Image Process. (VCIP)*, 2000, pp. 2-13.

[11] A. Aggarwal, *The Art Gallery Problem: Its Variations, Applications, and Algorithmic Aspects*, PhD thesis, Johns Hopkins University, 1984.

[12] S. Fleishman, D. Cohen-Or, and D. Lischinski, "Automatic Camera Placement for Image-Based Modeling," *Computer Graphics Forum*, vol. 19, no. 2, 2000, pp. 101-110.

[13] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, Morgan Kaufmann, 2004.

[14] S. Agarwal et al., "Structured Importance Sampling of Environment Maps," *ACM Trans. Graphics*, vol. 22, no. 3, July 2003, pp. 605-612.

[15] J. Dorsey, H. Rushmeier, and F. Sillion, *Digital Modeling of Material Appearance*, Morgan Kaufmann Series in Computer Graphics, 2007.

Soonhyun Kim received the BS and MS degrees in media from Ajou University in 2005 and 2007, respectively. Currently, he is a PhD candidate with the Graduate School of Information and Communication, Ajou University. His major research topic is real-time image synthesis for 3D animation production.

Min-Ho Kyung is an associate professor of the Department of Digital Media at Ajou University. He received the BS and MS degrees from Pohang University of Science and Technology in 1993 and 1995, respectively. He continued his graduate study at Purdue University, where he received the PhD degree in computer science in 2001. His research focuses on real-time image synthesis for 3D graphics, robot motion planning, and robust geometric algorithms.

Joo-Haeng Lee received his BS, MS, and PhD degrees in computer science from POSTECH, Korea, in 1994, 1996, and 1999, respectively. He joined ETRI, Korea in 1999 and is a senior research scientist with the Rendering Team of the Digital Contents Division. His research interests include geometric modeling and processing algorithms for computer graphics, CAD, and robotics. He is also interested in embedded intelligence for computer graphics, CAD, and robotics applications.