Real-Time Lighting Changes for Image-Based Rendering

Huong Quynh Dinh, Ronald Metoyer and Greg Turk
Georgia Institute of Technology
{quynh | metoyer | turk}@cc.gatech.edu

Abstract

We describe techniques for varying the surface illumination and the shadows of objects that are drawn based on a collection of pre-rendered images. To our knowledge this is the first instance of an image-based rendering approach that provides such changes in appearance due to lighting in real time. As with other image-based techniques, our methods are insensitive to object complexity. We make changes in lighting using texture hardware, and we can freely intermix image-based models with traditional polygon-based models. We accomplish lighting changes by discretizing the surface normals at each pixel and storing these normals in a texture map. We change the lighting of an object simply by changing a texture lookup table. We project shadows on to a ground plane by selecting a pre-rendered view of the object from the direction of the light source and distorting this silhouette to match the object as seen from the current viewer position. Potential applications of these techniques include vehicle simulation, building walkthroughs and video games.

Keywords: Image-based rendering, texture mapping, illumination, shadows.

Introduction

Traditional rendering techniques create images of a scene based on 3D geometric descriptions of the objects in the scene. Image-based rendering, on the other hand, produces images of a scene either directly or indirectly from one or more images of the given scene. Image-based rendering is now an active area of research in computer graphics and in computer vision. Two factors that contribute to the popularity of image-based rendering are the wide availability of texture mapping hardware and the low cost of memory. We use both of these observations as motivation for our approach to image-based rendering. Our approach stores many rendered images of a given object, and keeps additional information with each luminance sample (image pixel) in order to allow real-time lighting changes to the object.

There are many ways to store samples from images, each with their own tradeoff between simplicity and speed versus generality of viewing. Our emphasis is on speed. We store our image samples in an image fan, a collection of rendered images of an object taken from a circle of camera positions (Figure 1). This representation is not new, and can be found in systems such as Quicktime VR [1]. An object is rendered by displaying an image from the fan on a rectangle whose plane is most nearly perpendicular to the camera's line of sight (Figure 2). Using an alpha channel allows the background to show through where the object is absent. Similar alpha channel techniques are commonly used in flight simulators to render trees with textured poly-

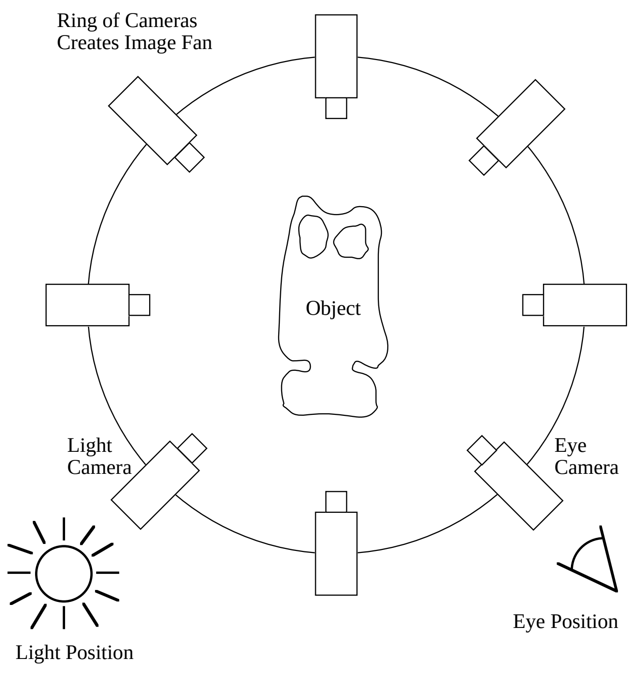

Figure 1: Ring of cameras produces an image fan.

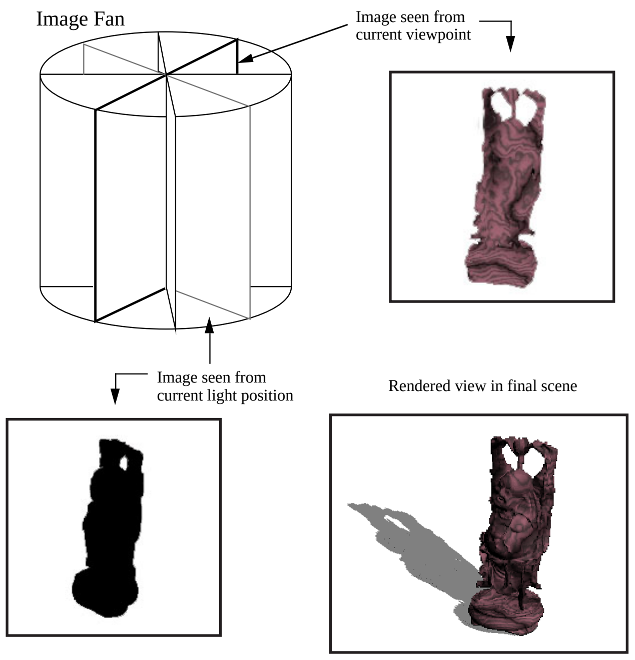

Figure 2: Two images from fan are selected to render model and its shadow.

gons so that background objects can be seen through those portions of the trees where leaves and branches are absent. In addition to storing the color at each sample, we also store the depth and the surface normal of the object at the sample. We used the public-domain ray tracing program Rayshade to create the images and to give us the depth and surface normal information [2]. The surface normal allows us to re-calculate the illumination of the surface due to one or more light sources. The depth information permits the creation of shadows.

Our image fans contains on the order of 80 to 120 images of the object that are synthetically rendered from a 360 degree sweep around the object at a given elevation. This number of images is quite sufficient to allow smooth movement of the object in a real-time application. Note that there are two assumptions being made when we represent the object by selecting one of these images as a stand-in for the object. One assumption is that our elevation is not dramatically different from the elevation of the camera that captured the image fan. We have found that people's visual systems are amazingly forgiving in this respect, and plausible views of an object can deviate by plus or minus 20 degrees from the original image fan camera. This restriction can be lifted if we are willing to store a collection of images that are created by rendering from positions on a hemisphere surrounding the object. The second assumption that we make when using an image fan is that our viewing distance from the object is not too different than the distance between the object and the camera that created the image fan. Here again we have found that human vision is forgiving, and that objects begin to look incorrect only when our view is very close to the object.

After reviewing related work, we describe fast shading of images using normal coding and then describe our shadow rendering algorithm. We conclude with integration of this method into a VR library and then discuss future directions.

Related Work

This section reviews some image-based rendering approaches that have made use of texture mapping hardware to accelerate rendering. Maciel and Shirley use automatically generated textured polygons as stand-ins for complex geometry when an object or collections of objects are far from the viewer [3]. Their images are generated in a preprocessing step. They demonstrate this approach in a city environment with many houses and buildings. A similar approach is used by Shade and his collaborators to display complex environments, but in their case the texture maps are rendered on-the-fly [4]. They demonstrate this method using a flyover of a hilly island with many trees. Debevec and co-workers create realistic looking synthetic views of buildings using texture maps created from photographs of real buildings [5]. Aliaga and Lastra accelerate building walkthroughs by only rendering the room the viewer is in with full polygon geometry, and use texture maps placed at doors to render the geometry through each door [6]. We believe that many of the above systems that use textures as stand-ins for geometry could use the approach we describe for changing the apparent lighting direction.

There are other image based rendering systems that use texture mapping hardware as a computational aid in a more indirect fashion. Levoy and Hanrahan use texture hardware to rapidly compute four-dimensional coordinates in a light field representation of an object [7]. Gortler et al. use small texture mapped triangles to create images from their 4D lumigraph representation of an object [8]. It remains to be seen if there are efficient ways in which to incorporate lighting effects into such image-based rendering systems.

Normal Coding for Shading

In this section we describe our method of changing the shading of an image-based object when the light source is moved. We make the assumption that our light sources are directional, that is, that they are positioned infinitely far away. The key to our lighting-invariant image based rendering technique is the retention of surface normal vector information for the images. We can calculate the appropriate illumination of an object by relating the direction of the object's surface normals to the light source position. Hence, we can maintain correct illumination of the object as the object and light source positions change if we have the normal vector information. We store the normal vector for each pixel of each image in the fan in what is called the *normal image*, and we convert these normal vectors into 8-bit values that index into a color map table. The color map table essentially stores the luminance values to be associated with each normal vector for a given light and view position. We modify the shading of a surface by changing the entries in the color map table rather than performing per-pixel lighting calculations.

We must first discretize the normal images to reduce the number of possible normals to the number of available color map entries. We use Glassner's unit circle method [9] to discretize the normal space. This method requires placing a grid over a unit circle and precalculating a set of normal vectors corresponding to each grid center as follows:

$$x = (CircleCenterX - GridCenterX) / Radius$$
$$y = (CircleCenterY - GridCenterY) / Radius$$
$$z = \sqrt{1 - (x^2 + y^2)}$$

Placing a 17×17 grid over the unit circle results in 225 different normal vectors. By using a unit circle, we achieve a higher resolution near the normals pointing towards the viewer and less at the sides where changes in gradient are not as easily perceived. For example, a normal vector such as [1, 0, 0] that lies on the boundary of the grid and therefore cannot be perceived by the user is not represented in this scheme.

We store the discretized normal vectors corresponding to each of the 225 values in a ByteToNormal table. This table is indexed by byte values 0 to 255 and holds the discretized normal corresponding to each byte code. We use this table to encode the normal images. For each pixel's normal in the image we find the closest discretized normal and store the corresponding 8-bit code in a more compact version of the normal image (Figure 3).

At run time, we use the ByteToNormal table to update the color map entries. For each frame in which the position of a light source or view changes, we traverse the 225 values in the ByteToNormal table to obtain the discrete normal vectors. We calculate the intensity for each normal vector and store the result in the color map entry corresponding to the byte code (Figure 4). We calculate the intensity using a diffuse shading model:

$$final\_color = (N \bullet L) * surface\_color \qquad (1)$$

N is the surface normal and *L* is the direction of the light. If a given surface is uniform in color, we may use just a single texture map to render the object. If, however, the surface is textured so that the color of the object varies per-pixel, we must resort to using one texture map to alter the lighting (*N* • *L*) and also use a second texture to capture the per-pixel variation in surface color. Notice that this requires us to multiply together the pixel values of two textures, one that captures the *N* • *L* term and the other that has per-pixel surface color variation. We perform this multiplication of textures using the OpenGL procedure glBlendFunc. This function controls a blend between two source textures using an alpha map. We treat the object's per-pixel surface color as one source and a black background polygon with an alpha map as the other source. This black background polygon with corresponding alpha map is simply the silhouette of the object. The two sources are blended as follows:

$$C = Alpha * ColorTexture + (1 - Alpha) * BackPoly \qquad (2)$$

ColorTexture is the colored texture image from the image fan, and it represents the term surface_color in equation (1).

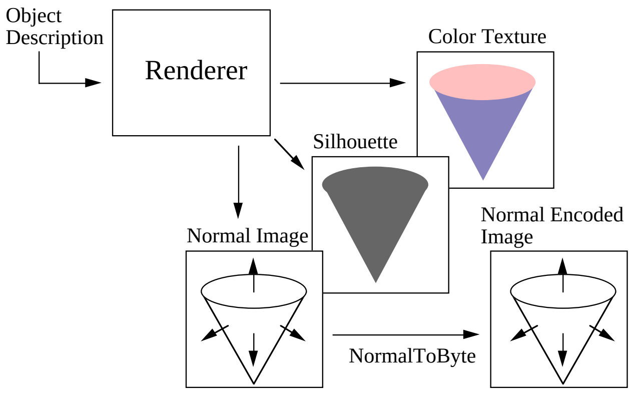

Figure 3: Creating the image fan makes color image, normal image and silhouette for each camera view.

Alpha is actually the diffuse lighting contribution (*N* • *L*) from equation (1). It serves as a blending percentage when blending the foreground texture color image with the background object silhouette. *BackPoly* is a black silhouette image of the object. The black color of this polygon makes the part of equation (2) after the addition sign evaluate to zero, thus making equation (2) exactly the same as equation (1). The blending created by equation (2) results in an image where the luminances determine the brightness of the surface (Figure 9). This entire process requires three forms of image data from the renderer (Figure 3): a normal image (to create *Alpha*), a silhouette image (to create *BackPoly*) and a surface color image (to create *ColorTexture*).

Dithering

Because we are using 225 discrete normal values, banding is apparent in the shading of the images (Figure 5a). We solve this problem by dithering the normals. We have implemented Floyd-Steinberg dithering, which yeilds better results as shown in Figure 5b.

The Floyd-Steinberg dithering technique diffuses discretization errors to neighboring pixels. The amount of error diffused to each neighbor is dependent on the neighbor's location with respect to the pixel under consideration [10]. We define error as the difference between the actual normal value at the current pixel and the closest discretized normal value. Errors associated with the *x* and *y* components of the normal vector are calculated and diffused separately. The *x* and *y* components of the neighboring pixels' normal vectors are increased or decreased by the diffused error values. The *z* component is then calculated from *x* and *y*.

Shadows

In this section we describe rendering shadows on the ground plane using information from the pre-computed images in

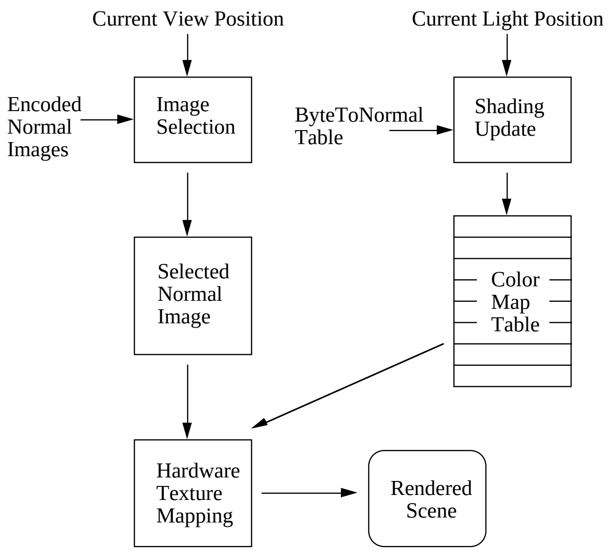

Figure 4: During real-time rendering the diffuse shading is calculated for each discrete normal vector.

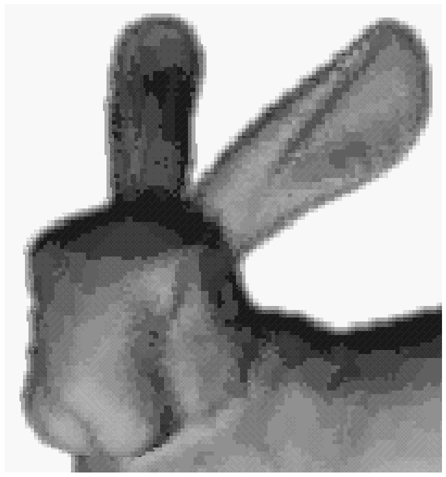

Figure 5a: Image shaded using quantized normal vector coding.

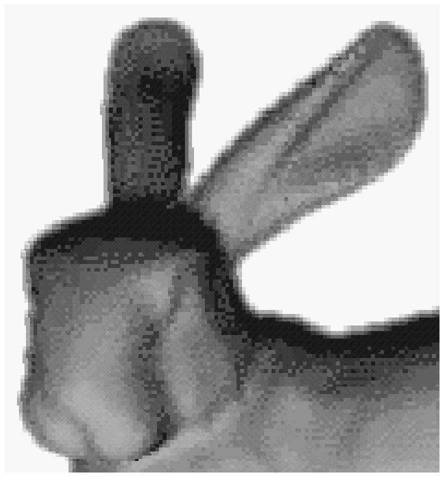

Figure 5b: Same image as Figure 5a, but created using dithered normal vectors.

the image fan. In essence, the image fan stores silhouettes of an object from a circle of camera positions. One can think of a shadow as a silhouette from the viewpoint of the light source that is then projected onto the ground plane [11]. We will refer to a particular shadow image from the image fan, composed of black (shadow) and white pixels, as a *silhouette image*. We make a shadow by creating a collection of polygon strips based on the positions of columns of black pixels in a silhouette image. To produce a shadow, these polygons are rendered translucently on the ground plane so as to darken the ground. Notice that we do not care where a black pixel in a silhouette image would project to on the ground plane unless it is either at the top or bottom of a column of black pixels. In this shadow-creation framework there are two tasks to creating shadows. First, we need to break up the silhouette image into abutting polygon strips based on the black pixels in adjacent columns. Second, we need to determine where the vertices of these polygon strips will project to on the ground.

We first address the issue of creating strips of polygons that represent the shadow region defined by the black pixels in the silhouette image. Because we know the model space coordinates of the object's silhouette at each pixel's *center*, we will create vertices of the polygon strips at the centers of the pixels. (This results in the shadow being smaller by half a pixel width, an unnoticeable amount.) Figure 6a shows a collection of black pixels from a simple silhouette image. For clarity, the pixels are represented as open squares. Figure 6b shows the vertical quadrilateral strips that we construct from the pixels of Figure 6a. The polygon strips in Figure 6b have been drawn slightly separate from one another for clarity, but actual shadows are created using quadrilateral strips that abut exactly. The strip creation process is accomplished by scanning down adjacent pairs of columns and examining blocks of four pixels. Any block of four pixels that does *not* match one of the patterns shown in Figure 7 is an indication that we are at the top or bottom of a strip, and thus a vertex is created at the center of a pixel. The final polygon strips are created by joining together vertices from adjacent pixel columns. This method not only creates vertices at the top and bottom of pixel columns, but it also correctly accounts for any holes in the shadow.

Now we will examine the task of projecting silhouette image pixels onto the ground plane. Figure 8 illustrates the steps we use, and the circled positions 1 through 5 in this figure represent the different locations in various coordinate systems that we use to perform this calculation. As described earlier, we retain depth information for images in a fan at each silhouette pixel. These depth values, taken together with the silhouette pixel's x and y positions, completely define a position on the object in model space (circled position 1 in Figure 8). We could project these positions

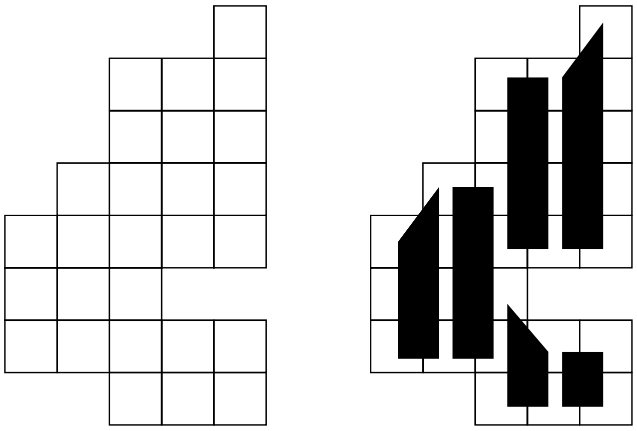

Figure 6: (a) Black silhouette pixels. (b) The quadrilateral strips created from this pattern of pixels.

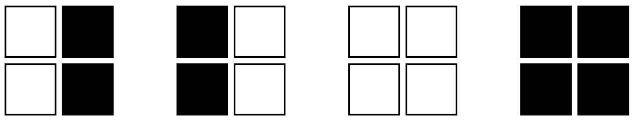

Figure 7: The patterns of four pixels that do **not** cause a vertex to be created on a quadrilateral shadow strip.

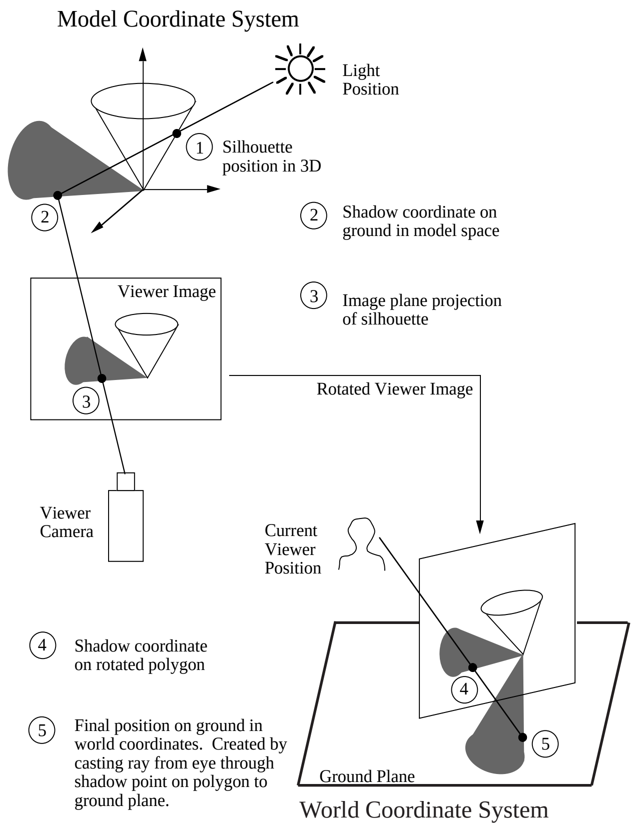
Model Coordinate System
Light Position
1 Silhouette position in 3D
2 Shadow coordinate on ground in model space
3 Image plane projection of silhouette
Viewer Image
Rotated Viewer Image
Viewer Camera
Current Viewer Position
4 Shadow coordinate on rotated polygon
5 Final position on ground in world coordinates. Created by casting ray from eye through shadow point on polygon to ground plane.
Ground Plane
World Coordinate System

Figure 8: Steps for transforming silhouette points from model coordinates into world coordinates.

onto the ground plane to create a shadow (position 2), but unfortunately a shadow created in this manner does not correctly line up with the image-based object. Recall that, in general, the position of the camera when an image in the image fan is created will not be exactly the same as the position of a viewer when the image is being rendered in the final scene. To choose which image to use to represent our object, we select the nearest image-fan camera position to our current view. We use the terms *viewer camera* to refer to the position of the camera nearest to the current viewer position, and the term *viewer image* to refer to the image that was rendered from this viewer camera. Since the viewer camera and the actual viewer position are not always the same, we must adjust for this discrepancy when we create shadows. Our solution is to transform the shadow position into the same coordinate system as that of the viewer camera.

We first project silhouette locations in model space onto the ground plane (position 2), and we then project them onto the plane of the viewer image (position 3). Now the shadow locations are in the same plane as the object as shown in the viewer image. Recall that the viewer image is in fact a texture on a polygon that has been rotated to the appropriate position for the current viewer position in world coordinates. Our final step is to project the shadow vertex locations from the properly rotated viewer image onto the actual ground plane. This projection is accomplished by extending a ray from the current viewer position through the shadow point on the rotated viewer image (position 4) and find where this ray intersects the ground plane (position 5). This position is the correct location (in the world space coordinate system) of a vertex from a shadow strip polygon.

Shadows created by this method exactly match the positions of those parts of an image-based object that touch the ground. Figure 9 demonstrate the results of this approach. Because the number of polygon strips used to represent a shadow is proportional to the number of columns of black pixels in the silhouette image, this method is inexpensive in terms of polygon count.

Incorporation into a Virtual Reality Toolkit

One advantage of using texture-mapped polygons as the basis for our image-based rendering approach is that we can easily incorporate the method into existing systems. As an example of this, we have added our image-based objects into the SVE (Simple Virtual Environment) toolkit, a system created at the Georgia Institute of Technology as a platform for VR applications [12]. Figure 10 shows several image-based objects in a scene that also contains polygonal

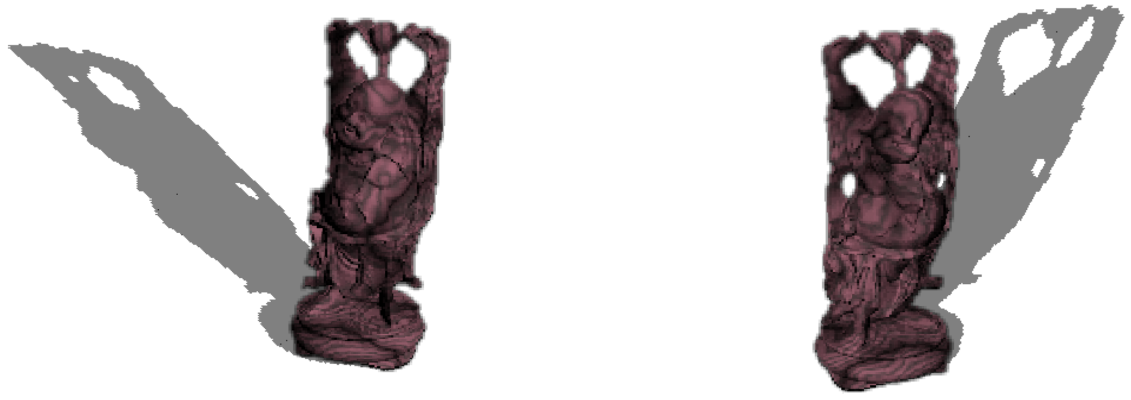

Figure 9: Images of a Buddha model (1,087,716 triangles) rendered using our shading and shadow-creation techniques.

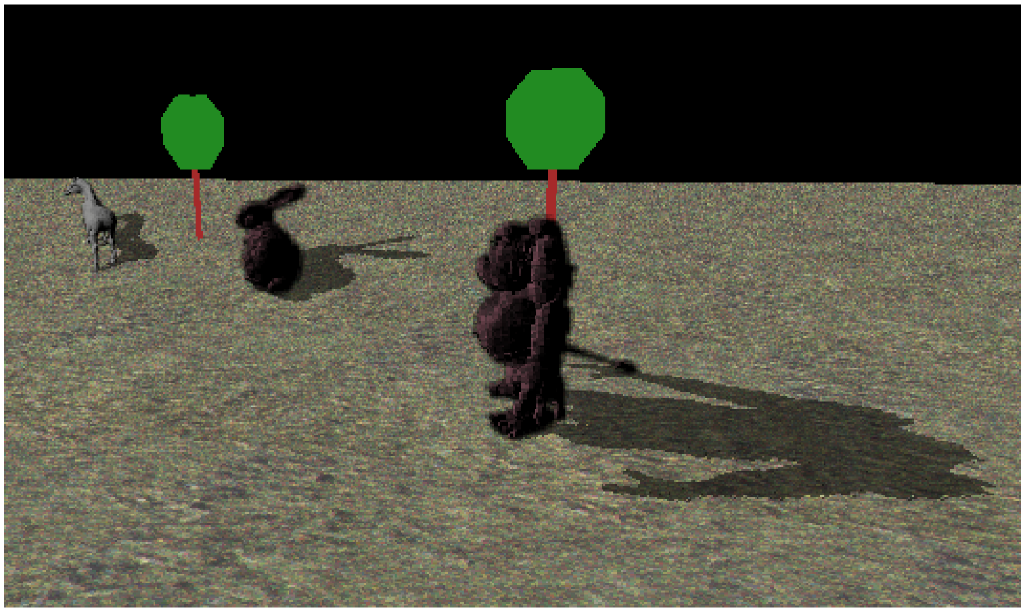

Figure 10: Scene with three image-based objects that was created using the SVE System.

trees and a textured ground plane. The average frame rate for this scene was 15 Hz on an SGI Infinite Reality.

Future Work

We have demonstrated changing the illumination of a given object using image-based rendering. One direction for future research is to determine whether similar techniques can be used to change the illumination of an entire scene, perhaps using a representation similar to environment maps. Another possibility is to extend our method so that it may cast shadows on arbitrary objects, not just on a ground plane. Yet another avenue for exploration is self-shadowing. We speculate that an image-based approach to self-shadowing will require a representation of an object that stores multiple samples along a given ray, such as the approach of [13]. Even the highly flexible (and storage-intensive) light field and lumigraph approaches [7] [8] that use 4D object descriptions do not seem to contain enough information for self-shadowing.

Bibliography

[1] Chen, Shenchang Eric, "QuickTime VR - An Image-Based Approach to Virtual Environment Navigation," Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH 95), August 1995, pp. 29-38.

[2] Kolb, Craig, The Rayshade Homepage,

http://www-graphics.stanford.edu/~cek/rayshade/info.html.

[3] Maciel, Paulo and Peter Shirley, "Visual Navigation of Large Environments Using Texture Clusters," Symposium on Interactive 3D Graphics, Monterey, California, April 9-12, 1995, pp. 95-102.

[4] Shade, Jonathan, Dani Lischinski, David H. Salesin, Tony DeRose and John Snyder, "Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments," Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH 96), August 1996, pp. 75-82.

[5] Debevec, Paul E., Camillo J. Taylor and Jitendra Malik, "Modeling and Rendering Architecture from Photographs: A Hybrid Geometry- and Image-based Approach," Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH 96), August 1996, pp. 11-20.

[6] Aliaga, Daniel G. and Anselmo A. Lastra, "Architecural Walkthroughs Using Portal Textures," Proceedings of Visualization 97, Phoenix, Arizona, October 19-24, 1997, pp. 355-362.

[7] Levoy, Marc and Pat Hanrahan, "Light Field Rendering," Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH 96), August 1996, pp. 31-42.

[8] Gortler, Steven J., Radek Grzeszczuk, Richard Szeliski and Michael F. Cohen, "The Lumigraph," Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH 96), August 1996, pp. 43-54.

[9] Glassner, Andrew, "Normal Coding," in Graphics Gems, Edited by Andrew Glassner, Academic Press, 1990, pp. 257-264.

[10] Floyd, R. and L. Steinberg, "An Adaptive Algorithm for Spatial Gray Scale," SID Symposium, 1975, pp. 36-37.

[11] Blinn, James F., "Me and My (Fake) Shadow," IEEE Computer Graphics and Applications, Vol. 8, No. 1, January 1988, pp. 82-86.

[12] Kessler, Drew, Simple Virtual Environment Toolkit, http://www.cc.gatech.edu/gvu/virtual/SVE.

[13] Max, Nelson and K. Ohsaki, "Rendering Trees from Precomputed Z-buffer Views," Proceedings of the Sixth Eurographics Workshop on Rendering, Trinity College, Dublin, Ireland, June 12-14, 1995.