Towards Interactive Photorealistic Rendering of Indoor Scenes: A Hybrid Approach

Tushar Udeshi and Charles D. Hansen

Dept. of Computer Science
University of Utah
{tudeshi, hansen}@cs.utah.edu

Abstract. Photorealistic rendering methods produce accurate solutions to the rendering equation but are computationally expensive and typically non-interactive. Some researchers have used graphics hardware to obtain photorealistic effects but not at interactive frame rates. We describe a technique to achieve near photorealism of simple indoor scenes at interactive rates using both CPUs and graphics hardware in parallel. This allows the user the ability to interactively move objects and lights in the scene. Our goal is to introduce as many global illumination effects as possible while maintaining a high frame rate. We describe methods to generate soft shadows, approximate one-bounce indirect lighting, and specular reflection and refraction effects.

1 Introduction and previous work

Research in photorealistic rendering has concentrated on numerically solving the rendering equation[9]. Ray tracers[22] use Monte Carlo methods while radiosity systems[6] use finite element methods. These give accurate solutions but are computationally expensive. Classic radiosity methods typically converge faster to a solution to the rendering equation than ray tracers but cannot account for specular reflection and refraction. Moreover, they are mainly used with planar geometry. Ray tracers obviate both these limitations but require path tracing[9] to get indirect illumination effects which introduces noise and is computationally expensive.

Many interactive environments such as Virtual Building Systems[1] rely on precomputation of static environments to form progressive radiosity solutions. These suffer from large computational overhead and unchangeable geometry. Even in incremental radiosity solutions[3], geometry changes require significant recomputation time. Moreover these solutions do not account for specular reflection and refraction.

Systems based on ray tracing are mostly non-interactive. Recently Parker et al.[14] developed an interactive ray tracer. However it does not account for indirect illumination. Moreover, moving objects must be spatially bounded which implies that all motion is known *a priori*.

Hardware-based solutions, which are interactive, are inherently of lower quality due to the limited feature set of the graphics accelerator. Diefenbach[5] demonstrated techniques of using graphics hardware to get specular reflection, refraction, caustics and transparency but not at interactive frame rates.

In this paper, we describe techniques of using both graphics hardware and multiple CPUs in parallel to achieve near photorealistic rendering of simple indoor scenes[1] at

[1]Up to 10,000 polygons and eight area light sources

interactive rates of one to ten frames a second. The user has the flexibility to interactively move both lights and scene geometry. We have developed our application on a SGI Reality Monster with eight Infinite Reality^{TM} graphics pipes and 64 MIPS R10000 processors. Our goal is to make use of all these resources to introduce global illumination effects while keeping the image generation as interactive as possible. The near photorealistic effects we have introduced are soft shadows, approximate one bounce indirect lighting and specular reflection and refraction effects.

1.1 Shadow generation algorithms

Shadows play a key role in the overall realism of computer-generated images because they provide important visual cues about the 3D arrangement of objects. Today's graphics hardware supports directional and point lights, Lambertian surfaces, and Gouraud or Phong shading. However it does not take visibility into account while shading; thus shadows are not an inherent rendering feature. There are three algorithms which make use of graphics hardware to calculate visibility: projective texture, shadow buffer and shadow volume. We give a brief description of these algorithms in the following paragraphs.

Projective textures. This algorithm has been used to generate soft shadows during walk-throughs[8]. To find the shadows cast on a particular polygon P, with respect to a light source L, all other polygons in the scene are projected on to P, with L as the center of projection. This can be done in hardware by fixing the view point at L and setting the view frustum so that P is tightly bounded at its base. Then P is rendered with lighting enabled and all other polygons in the scene are rendered only with ambient light. The image thus obtained is then texture mapped onto P. Hence for p polygons and l point light sources, $O(lp^2)$ polygons need to be rendered to determine visibility. This algorithm works well for walk-throughs in static scenes but is unsuitable for scenes with moving objects.

Shadow map / Shadow buffer algorithm. Shadow maps use the depth buffer and projective texture mapping to create a screen space method for shadowing objects[15, 16, 23]. The scene is rendered from a light source L and also from the eye. The depth values of pixels from the eye are transformed to the light-view coordinate system and then compared to depth values recorded from the light corresponding to the same region in object space. If the depth value from the light is less, that pixel is inferred to to be in shadow with respect to L. Hence for l point light sources and p polygons, $O(lp)$ polygons need to be rendered to determine visibility.

This technique has the advantage that occluders can have arbitrary geometry. However it suffers from aliasing problems. The number of pixels a particular object occupies when seen from the light is different from that when seen from the eye. Hence we do not have a one to one correspondence of pixels in the two images and so aliasing effects can be seen. This effect is illustrated in Figure 1. Another drawback of this algorithm is that omni-directional lights require multiple renderings of the scene in order to cover the entire scene. Also, even if one object in the scene moves, the entire shadow map needs to be recalculated. Therefore, this algorithm is not suitable for dynamic scenes.

Shadow volume algorithm. The shadow volume algorithm [4] is an object-space technique for generating shadows. A shadow volume, defined with respect to an occluder O and a point light source L, is that region in space where L is blocked by O.

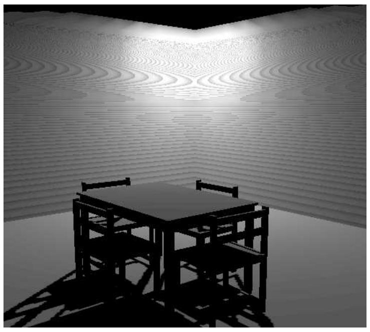


Fig. 1. “Shadow Acne”(aliasing artifacts) caused by shadow map algorithm.

Consider a ray shot from the view point into the scene. If the ray enters a shadow volume but does not leave it before intersecting with an object in the scene, the corresponding pixel can be concluded to be in shadow.

For l light sources and p polygons, the number of shadow volumes generated is $O(lp)$ in the worst case. Shadows are generated with object space precision. There are a few subtleties associated with this algorithm. A solution to these is given in Section 2.

We use multiple pipes and CPUs to implement both the shadow map and the shadow volume algorithm. The details are in Section 4.1.

1.2 Indirect lighting

As Appel[2] recognized, greater realism requires global illumination models, which account for inter-reflection of light between surfaces. Studies[19] have shown that indirect illumination also gives an important visual cue signaling contact between two surfaces.

Radiosity systems give accurate solutions for indirect lighting in diffuse scenes but not at interactive rates. Several systems use a static, precomputed radiosity solution for indirect lighting. While this is acceptable for walk-throughs and minor scene geometry changes, it is unsuitable for scenes in which light sources can move.

Keller[10] suggested a method to get fast solutions for indirect illumination by using hardware-assisted particle tracing. Particles are shot from the light in software. When a particle hits a diffuse object in the scene at a point P, the scene is rendered by the hardware with a virtual light source placed at P. With this technique, a very good approximation for indirect illumination is obtained in much shorter time compared to the physically-based ray tracing and radiosity methods. However, this technique did not run at interactive rates.

We use a similar technique for approximate one-bounce indirect illumination. We use a hardware-accelerated energy shoot approach as used in progressive radiosity. The algorithm is described in detail in Section 3.

1.3 Specular effects

In real scenes, we often encounter objects made of specular materials such as glass and metal. The specular reflection and refraction associated with these materials is view-dependent.

Diefenbach implemented reflection and refraction using graphics hardware[5]. The scene is rendered from a virtual view point to get the reflected/refracted image which is then projected onto the original image. This gives a good approximation for reflective surfaces. However for refraction, this approach is an acceptable approximation only if the rays incident on to the refractive surface are *para-axial* [7], which requires all refractive surfaces to be perpendicular to the line of sight vector. This approach cannot be used for non-planar surfaces. Moreover each reflective/refractive surface visible from the eye generates a virtual eye point, thus requiring an extra rendering of the scene. This technique is too computationally expensive for interactive applications.

Ofek et al.[13] demonstrated a technique to render curved, reflective surfaces using both the CPU and the graphics hardware. A *reflection image* is generated by creating and rendering virtual objects corresponding to reflections of scene objects. This *reflection image* is then merged into the primary image.

Ray tracing inherently accounts for reflection and refraction. We use multiple CPUs to ray trace the specular regions in the scene. Graphics hardware is used to speed it up further. This is described in detail in Section 4.3.

2 Soft shadow generation using shadow volumes

The shadow volume algorithm[4] is well suited for today's graphics hardware if the scene is composed of planar geometry. In this case, a shadow volume is a truncated semi-infinite pyramid and is bounded by what we call *shadow quads*. This is shown in Figure 2. A shadow quad is constructed from rays cast from the light source, intersecting the vertices of an edge, then continuing outside the scene.

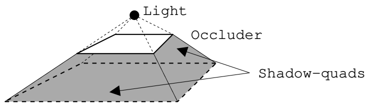


Fig. 2. The shaded area shows the shadow volume generated by an occluder due to a light source. The shadow volume is a semi-infinite pyramid bounded by quadrilaterals (shadow quads).

Consider a ray shot from the eye into the scene. An intersection with a front-facing shadow quad implies that the ray has entered a shadow volume . Similarly an intersection with a back-facing shadow quad indicates an exit from a shadow volume. Hence, for each pixel we need to determine the number of front-facing shadow quads minus the number of back-facing shadow quads which are closer to the eye than any object in the scene. We refer to this as *s-count*. If the *s-count* of a particular pixel is positive, we infer that the pixel is in shadow. Heidmann[18] used the stencil buffer to determine *s-count* for each pixel. In our implementation we use the red channel in the back buffer to maintain this count. The scene is rendered in the front buffer and the depth values copied into the back buffer. The color buffer of the back buffer is cleared. Shadow quads are assigned a color of (1, 0, 0). All front-facing shadow quads are first rendered with additive blending enabled and then all back-facing shadow quads are rendered with subtractive blending enabled. Note that these are rendered with depth-testing enabled and the depth mask set to FALSE. Consequently, the red channel of a particular pixel is modified by a shadow quad only if the shadow quad is closer to the view point than any object in the scene. The stencil buffer is used to resolve the shadow volume straddling the view plane issue as described in Section 2.1.

This is an object space algorithm and hence independent of the view point. Thus, shadow quads need not be recalculated when the view point changes. If shadow quads are constructed from each edge, the number of shadow quads generated will be very large. This number can be reduced by constructing shadow quads only from silhouette edges. However this can be done only with convex geometry as shown in Figure 3. Also face-edge data needs to be maintained which increases storage requirements.

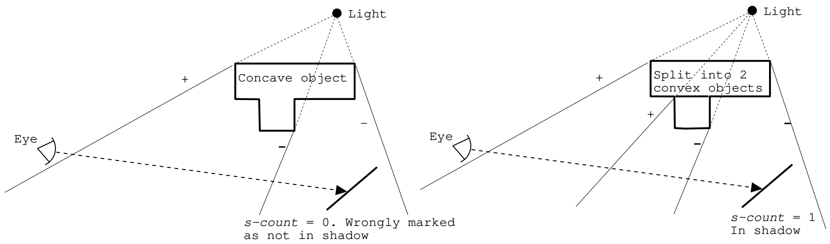


Fig. 3. Convex geometry needed for shadow volume algorithm. *s-count* is incremented when ray intersects with front-facing shadow quads (marked +) and decremented when intersected with back-facing shadow quads (marked -).

2.1 Shadow volume straddling the view plane

If the entire view plane is inside a shadow volume, then the *s-count* can be incremented by one for each pixel. We can determine the number of such shadow volumes with respect to a light source l by shooting a ray from the view point to l and keeping track of the number of light-facing objects which the ray intersects. *s-count* can then be initialized to this count for all pixels before drawing the shadow quads. However, as demonstrated in Figure 4, a shadow volume can straddle the view plane and the shadow quads are clipped at the boundaries of the view frustum. Consequently, not all pixels on the view plane are inside this shadow volume. The *s-count* of only those pixels which lie inside this clipped shadow volume should be corrected. Correcting *s-count* for all pixels will not work.

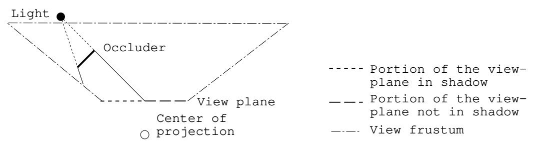


Fig. 4. A clipped shadow volume may straddle the view plane.

Consider a ray shot from the view point into the scene. The number of intersections with back-facing shadow quads minus those with front facing shadow quads is the number of shadow volumes which enclose the corresponding pixel in the view plane. Note that the ray intersection with scene geometry is ignored. We refer to this as the pixel's *g-count*. Note that both *g-count* and *s-count* are non-negatively clamped. Incrementing *s-count* by *g-count* for each pixel increments the *s-count* by the number of shadow volumes which enclose that pixel. This has the effect of registering an entry into these shadow volumes. In our implementation, we maintain *g-count* in the stencil buffer. When drawing front-facing shadow quads, the stencil test is set to decrement regardless of the outcome of the depth test. The stencil test is similarly set to increment while drawing back-facing shadow quads. Front-facing shadow quads are drawn first and then back-facing ones. This is achieved in hardware by rendering the polygons once with back-face culling enabled and once with front-face culling enabled, i.e. we need to make two passes over the polygon data, but each shadow quad is rendered only once. At the end of this process, the red channel has the *s-count* value and the stencil buffer has the *g-count* value for each pixel. The only other method we know of which addresses this issue requires the shadow quads to be drawn twice[5] (with four passes made over the polygon data). As shown in Section 5, drawing the shadow quads is the bottleneck of our implementation. Hence, drawing them twice would deteriorate performance considerably.

Capping the shadow volumes. The shadow volumes drawn are open at both ends. This may cause incorrect values of *g-count*. In Figure 5a, the ray shot from the view point towards the object O does not intersect any shadow quad and hence O is wrongly marked as not in shadow. This is because the ray passes through the open end of the shadow volume near the light. Note that this can occur only if the light is in front of the view plane. Similarly, the ray passes through the open end of the shadow volume away from the light in Figure 5b and this can occur only if the light is behind the view plane. One technique[5] suggested to resolve this issue was to extend the shadow quads to the light while determining *g-count*. However this gives incorrect results if the eye is between the light and the occluder. In this case, all visible objects are marked in shadow. In fact, even the occluder will be marked as being in its own shadow! This is illustrated in Figure 6. It also fails to cap the end of the shadow volume away form the light as shown in Figure 5b.

We cap the shadow volumes by drawing light-facing scene polygons for the case in which the light is in front of the eye, as shown in Figure 5a, and drawing a perspective projection with the light as the center, of every light-facing polygon when the light is behind the eye, as shown in Figure 5b. The rendering of these is done with stencil test enabled so that *g-count* is corrected and the color mask set to FALSE so that *s-count* is not modified.

Figure 7 shows the shadows generated under a table with and without the shadow volume correction.

Approximating the shadow volume capping. While the above solution always works, there is a performance degradation because every light-facing polygon, or its projection, needs to be drawn once for each light sample. However, note that a shadow volume needs to be capped only if it encloses at least one pixel of the view plane, i.e. only shadow volumes constructed by polygons which occlude the light sample from some region of the view plane need to be capped. We determine these occluding polygons by sampling the view plane and shooting rays from these samples towards the light source; all objects intersected are tagged. Only shadow volumes generated from silhouette edges of tagged polygons are capped. However, a few occluding objects can sometimes be missed by the ray tracing which cause incorrect shadows. We decided to include this feature because it resulted in a large performance gain. The user can toggle this approximation off if correctness is required at all times.

3 Approximate indirect lighting

On the SGI Infinite Reality^{TM} graphics system, with p graphics pipes, we can have as many as $8p$ light sources. These can be used to simulate one-bounce reflected light. In

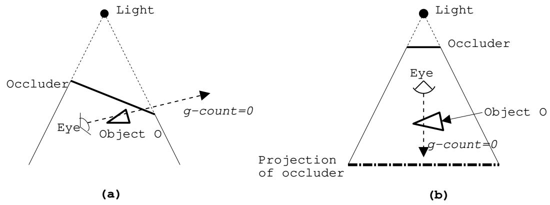


Fig. 5. If shadow volume is not capped Object O is wrongly considered to be in shadow(a) Light is in front of the eye: Occluder can cap the shadow volume (b) Light is behind the eye: projected occluder will cap shadow volume.

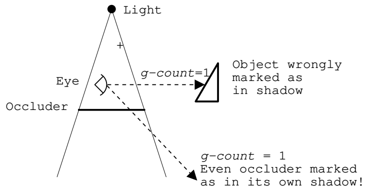
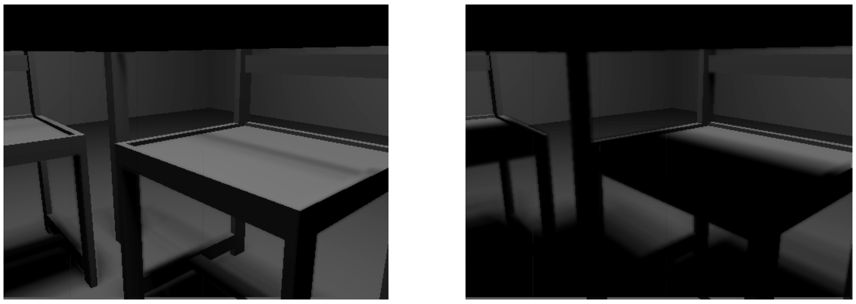


Fig. 6. Problem of extending the shadow volumes to the light for capping the shadow volumes.

our implementation, we tessellate polygons in the scene into elements of equal area for the indirect lighting pass. We approximate the one-bounce reflected light from those polygon elements which reflect maximum energy. The polygon elements are sorted, by multiple CPUs in parallel, in descending order of the energy they reflect. The energy an element P receives directly from a differential area light source dl is directly proportional to the form-factor from dl to P. Therefore, this form-factor multiplied by the reflectivity of P and the emissivity of dl is a measure of the energy which is received from dl and reflected by P.

We use a *single-plane* algorithm[17] to calculate the form-factors. Each polygon is assigned a weight initialized to zero. A wide-perspective *item buffer*[21] is rendered with dl as the center of projection. The form-factor from dl to each pixel in the *item buffer* is precomputed in a map with the help of Nusselt Analog. The weight of a polygon P is the the sum of form-factors of pixels that display P in the item buffer multiplied by the reflectivity of P and the emissivity of dl. Multiple CPUs are used to sort the polygon elements based on their assigned weight. We note that a full sort is not required. We only require the $8p$ polygon elements having the highest weight. Each pipe approximates the one bounce reflected light of 8 polygon elements by rendering the scene with light sources placed at the centroids of these elements. The light source direction is set to that of the normal of the polygon and the intensity is set proportional to the assigned weight. The scene rendered with these virtual light sources is blended into the final display image to add in the indirect illumination effect.

We note that OpenGL lighting does not take visibility into account, so the virtual light sources may light regions in the scene which should not be lit. For example in Figure 8, object O should not be lit by the virtual light source. However the intensities of these virtual light sources are low. Thus, the incorrect illumination is not noticeable for most scenes.



(a) (b)

Fig. 7. Shadow under a table (a) No shadow volume correction (b) With shadow volume correction

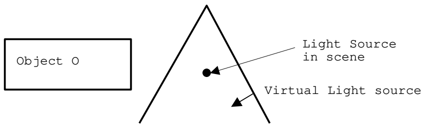


Fig. 8. Object O wrongly lit by virtual light source

4 Implementation details

We have developed our application on an SGI Reality Monster^{TM} which has 64 CPUs and 8 Infinite Reality graphics pipes. The user can select the number of pipes and CPUs to use with the restriction that the number of pipes should be greater than or equal to the number of area light sources in the scene. The user can also choose between the shadow buffer and shadow volume algorithm. The latter is generally faster and gives more accurate shadows but the former should be used if the model is not convex. If the user chooses the shadow volume algorithm, the faster approximate shadow volume capping as described in Section 2.1, can also be selected. The accuracy of soft shadows depends on how finely the light is sampled. More samples give more accurate shadows but result in a performance degradation. This can also be specified by the user. The user also has the option of selecting which global illumination effects are desired among soft shadows, indirect lighting and reflection/refraction.

Each pipe generates shadows from a few light samples and also contributes to the indirect lighting with 8 virtual light sources. Figure 10 (see color plate) shows the different stages in the rendering pipeline and Figure 11 (see color plate) shows the images generated by the different pipes before compositing. The following sections give implementation details.

4.1 Shadow generation

The scene is rendered into the front buffer, with one point light source which is a jittered random sample of the area light source. OpenGL supports only per-vertex lighting calculation. The polygons in the scene are uniformly tessellated with a threshold on their area only while doing the direct lighting shading.

Soft shadows are generated by sampling the light source into point light samples.

The light samples are uniformly distributed between all pipes. A binary image is generated for each light sample, i.e. the image is divided into regions lit by the point light sample and those that are not. These binary images are accumulated in the accumulation buffer to generate a gray-level image which is blended onto the front buffer to give shadows. Typically we sample the light source using 9 point light sources which produces only ten levels of gray. When the eye zooms into a shadow, discontinuities and Mach bands can be observed . To alleviate this effect, we convolve the grey level image with a 7×7 square average filter before blending into the front buffer. Convolution is supported by the graphics hardware and therefore does not introduce noticeable overhead.

Shadow map. The shadow map algorithm has been implemented as an OpenGL extension on SGI machines and is called the SGIX_SHADOW extension. The coordinate transformation of depth values from eye-view to light-view coordinates, comparison of the two corresponding depths and filtering are all implemented in hardware.

Theoretically, one rendering of the scene from the eye and one from each light sample should be enough to generate the shadow map for multiple light samples. However, the shadow map algorithm implemented by the SGIX_SHADOW extension, uses 3D texture mapping. A texture can be mapped only if geometry is drawn and so the scene needs to be rendered once from the eye and once from the light for each point light source sample.

We tried to avoid this cost by implementing the algorithm ourselves. A pure software-based implementation proved to be too slow. So we attempted to perform the coordinate transformation in hardware. We first render the scene so that the color at each pixel represented the object space coordinates at that point. This is done using 3D texture mapping. Using automatic texture coordinate generation, the texture coordinates are set proportional to the corresponding object space coordinate and scaled from 0 to 1. The texture map is then set so that the red channel is proportional to the x coordinate in object space, the green to y and the blue to the z coordinate. We call the image rendered using this texture mapping a *coordinate buffer*.

The scene is then rendered from the point light sample and the depth buffer read into a memory buffer MD. The color matrix is loaded with the product of the light-view perspective matrix and the model-view matrix as well as the scaling and biasing required to convert back to object coordinates and then the *coordinate buffer* is read into another memory buffer MB. In this pixel transfer process, we thus perform the coordinate transformation required. Now the red, green and blue color channels in MB represent the x, y and z normalized device coordinates respectively. The red and green channels of each pixel are converted to window coordinates (a scale followed by a bias) and the green channel is compared with the corresponding value in MD. However, this technique gave incorrect shadows especially at points of contact. This is because we only had 10 bits per color channel, which was not enough precision to represent the object space coordinates accurately in the *coordinate buffer* and so transformed depth values had significant errors.

Shadow volume. The CPUs extract the silhouette edges of the scene with respect to the light samples. This needs to be done only if a light has moved or the object to which the edge belongs has moved since the last frame. Let $\overline{n_1}$ and $\overline{n_2}$ be the normals to the two faces bounded by an edge. Let $\overline{v}$ be a vector from the light source to any point on the edge. Then the edge is a silhouette if $(\overline{n_1}.\overline{v}) * (\overline{n_2}.\overline{v})$ is negative. The shadow quads are calculated for the silhouette edges and stored with the edge data.

Each pipe is assigned a few samples of one area light source. If approximate shadow volume capping is enabled, jittered samples from the view plane are determined and rays are shot from these to the light source. These rays determine which objects need to be drawn to cap the shadow volumes. Else, all shadow volumes are capped.

The depth buffer is copied from the front to the back buffer and all shadow volumes generated from the assigned light samples are drawn and capped as described in Section 2. The stencil buffer, which stores *g-count*, is read into memory and then additively blended into the red channel of the back buffer, which stores *s-count*. A positive red channel value for a pixel implies that the pixel is in shadow. This binary image is accumulated in the accumulation buffer. The above process is repeated for all light samples. The red channel of the accumulation buffer then varies from 0 to the number of samples assigned to that pipe, say s, with 0 representing totally unoccluded and s representing totally occluded. We extract a gray-scale image from the accumulation buffer using the color matrix and then blend onto the front buffer.

The advantage of this algorithm over the shadow map algorithm is that there is load sharing between CPUs and graphics pipes: CPUs are used for silhouette edge detection and shadow quad construction while the graphics pipes are used for drawing shadow quads and determining which areas in the scene are in shadow. Also, shadows are generated with object space precision.

4.2 Indirect lighting

If there are l lights in the scene, the first l pipes render an item buffer from the point of view of the centroid of the l^{th} light. These are read into shared memory and the polygon elements sorted by multiple CPUs as described in Section 3. Then, the graphics hardware is used to render the scene with virtual light sources to approximate one-bounce indirect illumination.

4.3 Ray tracing for specular surfaces

The techniques suggested in Section 1.3 are mainly implemented in graphics hardware. In our implementation the graphics hardware is heavily used for shadow generation and indirect lighting effects. Therefore, these methods are not suitable. We use hardware-assisted ray tracing performed by multiple CPUs to get these effects.

An item buffer[21] from the point of view of the eye is rendered by the last pipe with the specular objects tagged and read into shared memory. The master thread is then signaled to begin ray tracing. We use a standard master-slave implementation. The master assigns rows of the item buffer to idle CPUs. Each CPU scans through the row it is assigned to and spawns rays only for those pixels which are tagged. The first object that a ray intersects need not be determined since that information is determined from the item buffer.

If there are not enough CPUs available this ray tracing could become a bottleneck. In this case, when the hardware has finished the task of drawing shadows and indirect illumination, the image is read back into a memory buffer MB. Now whenever a ray hits a point P on a diffuse surface which is visible to the eye, the radiance at P can be retrieved from MB. Therefore, shadow rays need not be spawned which results in a performance gain.

The efficiency structure used is a uniform grid. This allows fast update in the event of geometry changes. After all rows in the image have been processed, the ray traced image is blended onto the display pipe.

4.4 Compositing

Each pipe generates a partial image, contributing to both indirect illumination and soft shadows. When all pipes finish their tasks, their partial images are composited onto the display pipe. If the ray tracing has not finished before the compositing of these images is completed, the composited image, which has the direct and indirect illumination components, is loaded into shared memory to assist the ray tracing, as discussed in Section 4.3. When ray tracing of specular surfaces is completed, the ray traced image is blended onto the display pipe.

The partial images rendered in the pipes are read into shared memory. The composition of these involves just an additive blending. We implemented this in two ways: hierarchical binary swap[11, 20] which used graphics hardware and software compositing. When enough CPUs are available, the latter results in a higher frame rate. This is because a few CPUs can be assigned to the task of compositing. If the time taken by these CPUs for compositing is less than the time taken to render a frame, the compositing time is almost completely hidden. However there is one frame of latency[2].

4.5 Usage of resources

We summarize the utilization of resources for each rendering stage in Table 1. Observe that a combination of CPUs and graphics pipes are used for each rendering stage. Traditional rendering is done only on graphics hardware or is solely software-based. Hence, our approach is a better utilization of available resources.

Task	CPUs utilized for	Graphics pipes utilized for
Direct lighting without visibility	None	Scene rendered from the eye with a single light.
Shadow volumes	Determining silhouette edges. Calculating shadow volume boundaries	Drawing the shadow quads to generate gray-level shadow-image
Shadow maps	None	Rendering the scene twice for each light sample.
Indirect lighting effect	Scanning item buffer. Assigning brightness to each polygon and sorting them.	Rendering item buffer from light. Rendering scene with 8 virtual light sources to simulate indirect bounce.
Specular surface rendering	Ray trace specular portions of the image.	Render item buffer from eye

Table 1. Distribution of tasks between CPUs and graphics pipes

5 Results

We present results of our implementation on three example scenes. The first is an office scene consisting of 4,083 polygons. The scene has 2 area light sources which were each sampled into 9 point lights. There were 52,335 shadow quads generated. This scene was

[2] In our implementation we found that 5 CPUs perform the compositing of a 680×480 image in less than .01 seconds.

SCI office, 512x512 image size, 18 light samples							
Pipes	Shadow (%)	Indirect Light(%)	Direct Light(%)	Specular (%)	Compos-ite(%)	HC (secs)	SC (secs)
2	65.38	20.01	6.306	2.692	4.173	0.4061	0.4041
4	46.65	26.63	9.883	3.136	11.34	0.2961	0.2625
8	28.22	28.00	11.39	4.141	25.79	0.2565	0.2093
Classroom, 512x384 image size, 36 light samples							
4	80.16	6.336	2.604	2.440	7.447	0.4816	0.4372
8	61.16	10.63	4.490	3.967	18.10	0.2937	0.2490
Cornell Box with glass sphere, 512x512 image size, 25 light samples							
1	73.10	17.49	2.945	6.299	0.02277	0.293	0.3175
2	50.98	24.11	4.173	5.731	14.73	0.2093	0.1981
4	29.22	31.17	4.780	6.830	27.73	0.1751	0.1444
8	14.98	31.56	5.158	6.683	41.27	0.1650	0.1322

Table 2. Rendering times for 3 scenes. HC: Total time using hardware-based binary swap. SC:Total time using software compositing.

rendered at over 4 frames a second using 8 graphics pipes and software composition. Timings are given in Table 2 and a snapshot shown in Figure 12 (see color plate).

The second is a classroom scene composed of 3,135 rectangles. There are 4 area light sources each of which were sampled into 9 light samples. The top of the table is reflective. This scene is the worst case scenario for the shadow volume algorithm as almost all edges are silhouette edges. There were totally 112,326 shadow quads generated. Even when using 8 pipes 61% of the time is used for shadow quad rendering! Results are summarized in Table 2 and the image is shown in Figure 9.

The third scene the Cornell Box. A glass sphere is added into the scene and the top of the small box is made reflective. The sphere is tessellated into 200 polygons for the purpose of generating shadow volumes. Note that the sphere is used directly for the ray tracing pass. The scene has 217 polygons and one area light source sampled into 25 point lights, generating 1,134 shadow quads. Timings are given in Table 2. The image can be seen in Figure 10 (see color plate).

For all three scenes, the graphics hardware proved to be the bottleneck i.e. the CPUs would finish their task before the hardware was ready to use it. However, if the scene is composed of a large number of specular objects, ray tracing could become the bottleneck. We observe that shadow generation time dominates. Fortunately, this time scales almost linearly with the number of pipes used. Rendering times for all other effects are independent of the number of pipes used.

6 Conclusion and future work

We have demonstrated techniques for performing near photorealistic rendering of simple dynamic indoor scenes at interactive rates. To our knowledge, no other system achieves these rendering rates for comparable dynamic scenes and with comparable image quality. We have also identified and given efficient solutions for various subtleties associated with the shadow volume algorithm. We observe that soft shadow generation has the maximum computational cost among all the photo-realistic effects

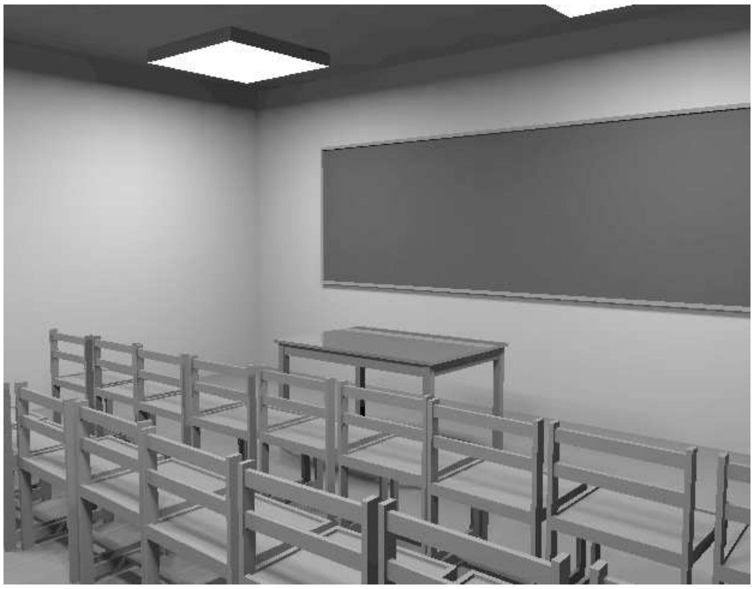


Fig. 9. Classroom scene rendered at 4 frames a second using 8 pipes and software compositing.

we have introduced. This is because area light sources need to be sampled into several point lights, which generates a large number of shadow volumes. Future work includes addressing this by using shadow volumes to divide the scene into three regions: totally occluded, partially occluded and totally unoccluded. This can be accomplished by rendering those shadow volumes which form umbral or penumbral boundaries, as described in [12]. The number of shadow volumes would be significantly reduced. However a fast way to render the partially occluded regions of the scene is still an open question.

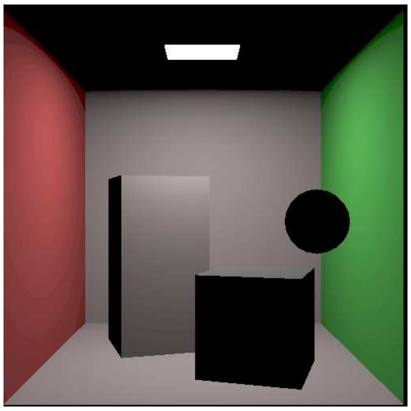
Acknowledgments

This work was supported by the SGI Visual Super computing Center at the University of Utah, NSF grant 97-20192 and the Center for Simulation of Accidental Fires and Explosions, a DOE ASCI Level 1 Alliance Center. We would like to thank Peter Shirley, Brian Smits, Peter-Pike Sloan and Steven Parker for their helpful ideas.

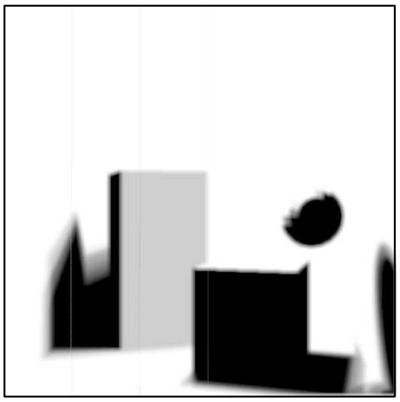
References

1. John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. volume 24, pages 41–50, March 1990.
2. Arthur Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, volume 32, pages 37–45, 1968.
3. Shenchang Eric Chen. Incremental radiosity: An extension of progressive radiosity to an interactive image synthesis system. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 135–144, August 1990.

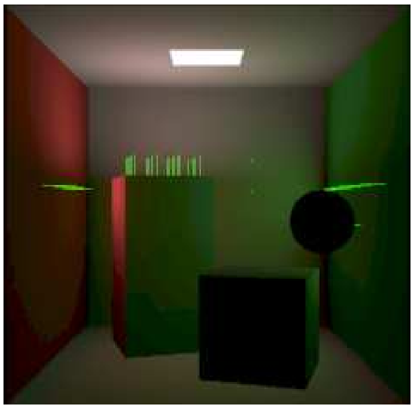
4. Franklin C. Crow. Shadow algorithms for computer graphics. volume 11, pages 242–248, July 1977.
5. Paul J Diefenbach. *Pipeline rendering: Interaction and realism through hardware-based multi-pass rendering*. PhD thesis, University of Pennnsylvania, 1996.
6. Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modelling the interaction of light between diffuse surfaces. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 212–22, July 1984.
7. Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 119–127, July 1984.
8. Michael Herf and Paul S. Heckbert. Fast soft shadows. In *Visual Proceedings, SIGGRAPH 96*, page 145, Aug. 1996.
9. James T. Kajiya. The rendering equation. In David C. Evans and Russell J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 143–150, August 1986.
10. Alexander Keller. Instant radiosity. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 49–56. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.
11. Kwan-Lui Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. Parallel volume rendering using binary-swap image composition. *IEEE Computer Graphics and Applications*, 14(4), July 1994.
12. T. Nishita and E. Nakamae. Half-tone representation of 3-D objects illuminated by area sources or polyhedron sources. *Proc. COMPSAC 83: The IEEE Computer Society's Seventh Internat. Computer Software and Applications Conf.*, pages 237–242, November 1983.
13. Eyal Ofek and Ari Rappoport. Interactive reflections on curved objects. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 333–342. ACM SIGGRAPH, Addison Wesley, July 1998. ISBN 0-89791-999-8.
14. Steven Parker, William Martin, Peter-Pike Sloan, Peter Shirley, Brian Smits, and Chuck Hansen. Interactive ray tracing. In *Symposium on interactive 3D graphics*, April 1999. Accepted for publication.
15. William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 283–291, July 1987.
16. Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul E. Haeberli. Fast shadows and lighting effects using texture mapping. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 249–252, July 1992.
17. Francois X. Sillion and Claude Puech. A general two-pass method integrating specular and diffuse reflection. In Jeffrey Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 335–344, July 1989.
18. Heidmann T. Real shadows - real time. In *IRIS Universe*, December 1991.
19. William B. Thompson, Peter Shirley, Brian Smits, Daniel J. Kersten, and Cindee Madison. Visual glue. Technical Report UUCS-98-007, Computer Science Department, University of Utah, March 1998. http://www2.cs.utah.edu/vissim/papers/glue/glue.html.
20. Tushar Udeshi and Charles D. Hansen. Parallel multipipe rendering for very large isosurface visualization. In *Joint EUROGRAPHICS - IEEE TCCG Symposium on Visualization*, May 1999.
21. Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984.
22. Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.
23. Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 270–274, August 1978.



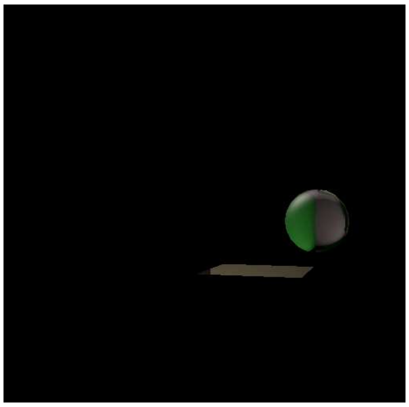
(a) Direct lighting



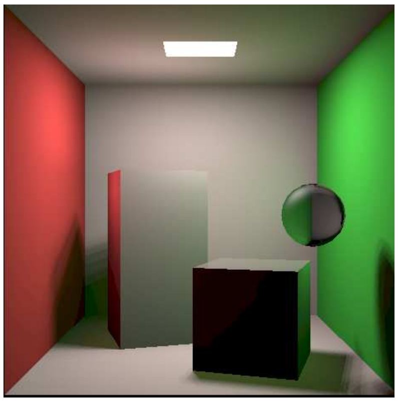
(b) Gray-level shadow(25 light samples)



(c) Indirect lighting (brightened for display. 32 virtual light sources shown in green)



(d) Raytraced portions



(e) Final blended mage

Fig. 10. Images showing different stages of rendering. These images were generated using four pipes.

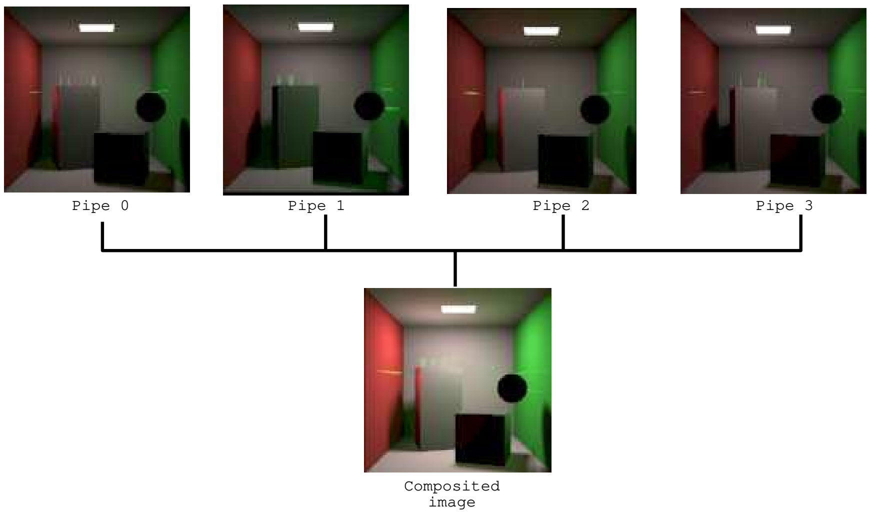


Fig. 11. Images rendered by different pipes. Each pipe contributes to both the shadow and indirect lighting generation.

Fig. 12. Office scene rendered at over four frames a second using 8 pipes and software compositing.