THE JOURNAL OF VISUALIZATION AND COMPUTER ANIMATION, VOL. **8**: 17–32 (1997)

Priority-driven Ray Tracing

RONI YAGEL
Department of Computer and Information Science, The Ohio State University, 2015 Neil Av., Columbus, OH 43210 U.S.A.

AND

JOHN MEEKER
auto·des·sys Inc., 2011 Riverside Dr., Columbus, OH 43221, U.S.A.

SUMMARY

A typical ray tracing algorithm traces a ray through each screen-pixel and spawns secondary rays at ray–object intersection points. Unlike traditional ray tracers which follow these rays recursively, we assign a priority value to each newly spawned ray and insert it into a priority queue. The priority assigned to each ray can be based on a variety of criteria, some of which we explore here. The next ray we trace is always the one with the highest priority in the queue. Occasionally, we trigger display updates when a checkpoint or predefined threshold is reached, providing intermediate images for review and evaluation. Classical ray tracers, once given the rendering specifications, are not controllable by the user. The priority-driven ray tracing, on the other hand, provides the user with a mechanism to steer the rendering and deliver intermediate images amid processing. This paper describes the illumination model of the non-recursive priority-driven ray tracer and evaluates its memory and time requirements. We show that although worst-case memory requirements can be overwhelming, in practice, our method is both useful and feasible. © 1997 by John Wiley & Sons, Ltd.

KEY WORDS: ray tracing; adaptive rendering; incremental rendering; illumination

1. INTRODUCTION

One of the areas of intense interest in computer graphics is photorealistic image synthesis. The achievement of photorealism dictates the use of a global illumination model which incorporates the effects of indirectly reflected and transmitted light. The two global illumination models in use today are *ray tracing*[1] and *radiosity*.[2] Although both methods are computationally expensive and time consuming, they generate stunning imagery. This has made both approaches very popular, although questionable from a practicality standpoint. This has provided a strong stimulus for the investigation and development of more efficient implementations of these techniques.

Ray tracing[1,3] computes the colour of each pixel by tracing a light ray backwards from the viewer's eye through the scene while determining which object (if any) is intersected by the ray. If an object is hit, reflection and/or transmission rays may be spawned depending on the object's surface properties (i.e. reflectivity and transparency). Global illumination effects due to diffuse reflections are modelled by

CCC 1049–8907/97/010017–16 *Received August 1995*

a directionless ambient light term. This global shading model is best suited for highly reflective or specular environments and has become very popular during the past decade because of the spectacular images created with this technique. Compared to traditional scanline algorithms, however, ray tracing can take a significant amount of time to produce one image.

This has made the development of efficient solutions to the ray tracing problem an area of active research for over a decade. Many acceleration techniques have been proposed and dramatic decreases in the rendering time of images have been achieved with some of these methods. For a complete survey of those techniques the reader is referred to Reference 3. One of the disadvantages of the conventional ray tracing algorithm, which is not addressed by most solutions, is that it computes a complete solution, one pixel at a time, until the entire image is rendered. That is, ray tracing cannot support the generation of partial images that can prove useful in the scene design and modeling phase. Even when employing common acceleration techniques, ray tracing can still require a significant amount of time before a complete image is available to the viewer.

One optimization technique which has been applied to ray tracing and which does address the above problem is *progressive screen sampling*.[1,4] This method allows the user to specify a low screen sampling rate. Rays are cast a few pixels apart while missing pixel values are interpolated in screen-space. Later, in these subsampled areas, one can cast additional rays to improve image quality. This gives the user some control over the generation of the image, so that the gross features of the image may be previewed as early as possible, which allows the process to be short-circuited if the user's expectations are not met.

Another global illumination technique which has gained much popularity since the mid 1980s is *radiosity*.[5] Radiosity computes the interchange of light from diffuse surfaces using an approach based on heat transfer principles. The energy equilibrium of a system is modelled independently on the viewer's position and can be therefore computed in a preprocessing step. After the completion of this step, view-dependent visible surface determination and shading are performed. This method produces the best results with diffuse environments and suffers from the same problem as ray tracing: a very large computational cost for the generation of a single image.

Efficiency considerations for the radiosity method have also been the focus of vigorous research in recent years. These have led to the development of the *progressive refinement* approach[2] to radiosity. As the algorithm iterates toward a final solution it generates intermediate images along the way. This partially alleviates the problem of the long delay from algorithm initiation to the generation of the first image. The merit of this method is that the initial and interim images produced with progressive radiosity are still useful, even though they are incomplete. The progressive approach provides the user with interim images very quickly while it successively refines its output towards the final accurate solution.

The *importance-driven* radiosity algorithm[6] further improves the progressive radiosity solution by incorporating view-dependency in the radiosity computations. This is accomplished by assigning an importance value to the patches in the scene, which guides the iterative solution. These importance factors are updated as the algorithm iterates toward the final solution. This provides a significant speed-up because it reduces the amount of computation in areas that contribute little to the final image.

In this paper, we describe a *priority-driven ray tracing* algorithm which was developed in the same spirit as the progressive refinement solution to the radiosity algorithm: we produce intermediate but useful images while iterating towards a final solution. Different priority criteria allow the refinement of the image to progress in a variety of ways, providing adaptability to the needs of the user.

2. BASIC THEORY

The traditional approach to ray tracing is to scan the image plane pixel by pixel (e.g. from left-to-right and top-to-bottom), computing the colour of each pixel. This computation is based on casting a ray, called a *primary ray*, from the eye, through a pixel, and into the scene. At the first point where the ray intersects an object, secondary rays are spawned to simulate light reflection (with *reflected* rays) and light transmission (with *transmitted* rays) (see Figure 1). *Shadow-feeler* rays are also sent from the intersection point to all light sources, in order to determine the intensity of incident light at that point. Secondary rays are recursively traced and their colour result is used to compute the colour assigned to the primary ray. The set of rays spawned as descendants of a primary ray form what is called a *ray tree*, as shown in Figure 1. Such a tree is created at each pixel of the image and a *depth-first* traversal of the tree yields a colour for the pixel. We call the set of all ray-trees (rooted at all the screen pixels) the *ray-forest*. Since we determine the final colour of each pixel before moving on to the next pixel, we do not render a complete image until we have computed the final colour of the last pixel. If the complexity of the scene is high or if we are sampling at a high rate (e.g. anti-aliasing via supersampling), the generation of the entire image can take a significant amount of time.

An initial image can be generated, however, with a *breadth-first* traversal of the ray forest. Actually, ray tracing reduces to *ray casting* if only the primary ray for each pixel is processed. This is depicted in Figure 2(a) where the currently processed rays (in this case, primary rays) are represented by solid lines. This produces a correct image with respect to visible surface determination but an incomplete image with respect to global illumination (Figure 2(a)).

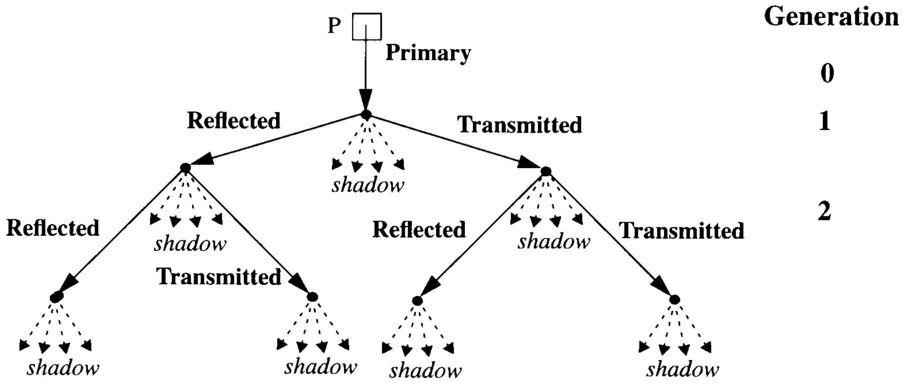

Figure 1. A three-level tree of rays rooted at a single pixel P. The first generation (labelled by 0) is the primary ray emitted at the pixel. At an intersection point (denoted by solid circle), reflected and transmitted rays are spawned as well as shadow feelers. Third generation rays (labelled by 2) use only local illumination at the intersection point

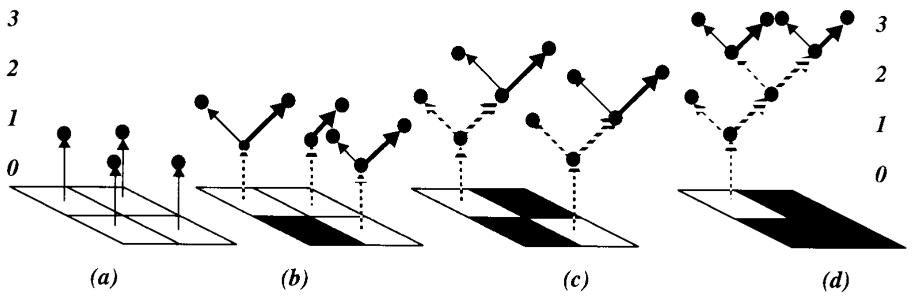

Figure 2. A forest of four ray-trees. (a) Results of primary rays are used to generate the intermediate image. (b) Rays participating in the computation of the second image (solid lines, where reflection rays are drawn as wide arrows branching to the right). (c), (d) third and fourth generation images. Grey areas denote pixels for which a final colour has been computed

After processing the primary rays, we can continue refining the image in an iterative fashion. In Figure 2(b), reflection rays (wide arrows branching right) and transmitted rays (thin arrows branching left) are spawned where appropriate (depending on the material attributes of the intersected object). The bottom left pixel does not spawn secondary rays (possibly because no intersection was found) and the pixel is coloured with the background colour. The results of these rays are combined with the values in the image (a) to produce a second image (b), using a method we describe in Section 3. By processing the third (Figure 2(c)) and fourth generation (Figure 2(d)) of rays, a sequence of images is generated that converges to the final image. The order in which these rays are processed is not important to the final image. If the ray tree level is used to establish the sequence in which the rays are handled (as depicted in Figure 2), a breadth-first traversal of the entire ray forest would be the ensuing result.

An interactive ray tracing algorithm[7] has been developed which shortens the time to generate the first image in a manner similar to that of progressive radiosity. The speed-up occurs by generating the first image based upon the results from the primary (eye) rays. More detail is added in later stages by considering the recursive reflection and/or transmission rays. The user is given interactive control over the rendering process by permitting the specification of parameters to increase the realism of the image or abort the synthesis if the image is not converging towards the desired result. The approach we propose here is much broader and unlike Reference 7 allows convergence to the final image in one of many ways.

We observe that estabishing the ray processing order is basically a prioritizing scheme that assigns a priority to each ray. Using the ray tree level to control the processing of rays is only one prioritizing method. The intensity a ray contributes to the image (also called *weight* or *contribution*), can also be used to govern the generation of the image. This scheme would allow those rays that contribute more intensity to the image to be processed first. Another possibility is prioritizing the ray type, where primary, reflection, transmission, and shadow rays receive different priorities. This provides the capability to initially generate a crude approximation to the final image via ray casting, after which precedence may be given to, for example, translucent or reflective objects before any shadow determination is performed. In a similar way, object type, object complexity, or surface properties may be used to control the order in which rays are processed. This gives the ability to selectively render certain objects or areas of the screen which may be of particular interest to the user.

Incremental screen supersampling can also be accommodated. If for example, we cast two rays per pixel, we can assign high priority to the first ray in each pixel and a lower priority to the second. Our first encounter with a low priority ray in the queue indicates that the first ray per pixel has been traced for all pixels. We can trigger image display and then go on to trace the second ray for all pixels. Finally, another priority scheme deals mainly with shadow rays. These can also be traced according to the intensity of the light source they try to reach. We can, for example, assign low priority to a shadow feeler that is aimed at a low intensity light source, since the result of tracing that ray will not greatly influence the image intensity. Alternatively, we can assign higher priority to light sources according to their colour etc. These and other schemes are possible and deserve consideration because of their potential usefulness.

The sequence of images shown in Plate 1 was generated using a modified tree level priority scheme where shadow feelers are assigned a lower priority than the reflection and transmission rays generated at the same level of the tree. This further accelerates the time to the first image since all shadow processing is delayed until a later stage. We later use this dataset to evaluate the performance of our approach (see Section 7).

3. ILLUMINATION MODEL

The intensity at a screen pixel (P_x, P_y) is denoted by $I_\lambda(P_x, P_y)$, where λ indicates wavelength (e.g. RGB). If the ray that passes through (P_x, P_y) does not intersect any object, $I_\lambda(P_x, P_y)$ is set to some background intensity. If that ray intersects some object O, then $I_\lambda(P_x, P_y)$ is assigned the intensity of O's surface at the intersection point. This is typically computed by some rendition of the Phong illumination equation:[8]

$$I_\lambda(O_x,O_y,O_z) = I_{a\lambda}k_aO_{d\lambda} + \sum_{i=1}^{m} S_iA_iI_{\lambda i}\,[k_dO_{d\lambda}\,(N\cdot L_i) + k_s\,(N\cdot H_i)^n] + k_sI_{r\lambda} + k_tI_{t\lambda} \tag{1}$$

where O_x,O_y,O_z are the co-ordinates of the 3D point that we illuminate, $I_{a\lambda}$ is the ambient light intensity, $O_{d\lambda}$ is the diffuse colour of the object, m is the number of light sources, k_a,k_d,k_s,k_t are the ambient, diffuse, specular, and transmission coefficients of the surface, S_i is the light-source i occlusion factor, A_i is the light-source i attenuation factor, $I_{\lambda i}$ is the light-source i intensity, N is the surface normal at O_x,O_y,O_z, L_i is the direction to light-source i from O_x,O_y,O_z, H_i is the half-way vector between L_i and the viewer, n is the specular reflection exponent, $I_{r\lambda}$ is the intensity of the reflected ray and $I_{t\lambda}$ is the intensity of the transmitted ray.

The first two terms in equation (1) compute what is commonly called *local illumination*. The last two expressions compute the *global illumination*; they model the influence of other objects in the environment. The term S_i are computed by following shadow feelers to the light sources. The values of $I_{r\lambda}$ and $I_{t\lambda}$ are computed by recursively tracing a reflection and transmission ray, respectively. As long as this recursive computation of the global illumination is not complete, no value can be assigned to the pixel at (P_x, P_y). The illumination equation used by our priority-driven ray tracer is basically a reordering of the summation in Equation (1) which allows for non-recursive handling of secondary rays. Likewise, the processing of the shadow-feelers differs from the way traditional ray tracers handle these rays since the priority-driven ray tracer might defer processing shadows feelers until a later time.

In the priority ray tracing approach, we give each ray a unique identifier number Ψ. Given a ray Ψ, its parent is denoted by $P(\Psi)$. For primary rays $P(\Psi)$ is undefined.

We associate, with each ray, a weight denoted by $\omega(\Psi)$ and defined by

$$\omega(\Psi) = \begin{cases} 1, & \text{if } \Psi \text{ is a primary ray} \\ k_s \omega_{P(\Psi)}, & \text{if } \Psi \text{ is a reflection ray} \\ k_t \omega_{P(\Psi)}, & \text{if } \Psi \text{ is a transmission ray} \end{cases}$$

We denote by $C(\Psi)$ the co-ordinate of the pixel the ray Ψ belongs to. That is, a primary ray traced through the pixel at (P_x, P_y) and all its descendants will have the same value of $C(\Psi) = (P_x, P_y)$. We denote by R^{Ψ}_{λ} the insensity contributed by the local illumination components in equation (1) at the point where a ray Ψ first intersects an object. We also assume that $S_i = 1$; that is, for the time being we do not deal with shadowing effects. We can now extract from equation (1) the following expression for R^{Ψ}_{λ}:

$$R^{\Psi}_{\lambda} = I_{\alpha\lambda} k_{\alpha} O_{d\lambda} + \sum_{i=1}^{m} A_i I_{\lambda i} \left[k_d O_{d\lambda} (N \cdot L_i) + k_s (N \cdot H_i)^n\right] \tag{3}$$

If we expand the recursive terms in equation (1), $I_{r\lambda}$ and $I_{t\lambda}$, we will eventually have a large sum solely consisting of terms similar to those in equation (3), namely, local illumination computations at the intersection points of all the rays in the ray tree. Therefore, the intensity of the pixel $I_{\lambda}(P_x, P_y)$ can be formulated as the sum of all the local illuminations, as defined in equation (3), computed by all the rays of the ray tree rooted at (P_x, P_y). That is

$$I_{\lambda}(P_x, P_y) = \sum_{\forall \Psi : C(\Psi) = (P_x, P_y)} \omega(\Psi) R^{\Psi}_{\lambda} \tag{4}$$

At run time, when a ray Ψ is popped from the queue and traced, the resulting value is added to the pixel value by

$$I_{\lambda}(C(\Psi)) \leftarrow I_{\lambda}(C(\Psi)) + \omega(\Psi) R^{\Psi}_{\lambda} \tag{5}$$

In equation (3) we assume that no occlusion exists between the point of ray–object intersection and any of the light sources i; hence the unity assumption for S_i. At a later stage, shadow-feelers are processed. At that time, if a light source i is (partially) occluded, some light intensity should be subtracted from the pixel.

Each shadow ray, when spawned by ray Ψ and sent to light source i, is associated with light intensity denoted by $I^{\Psi}_{\lambda i}$, which we define to be

$$I_{\lambda i}^{\Psi} = A_i I_{\lambda i} \, [k_d O_{d\lambda} \, (N \cdot L_i) + k_s \, (N \cdot H_i)^n] \tag{6}$$

that is, $I_{\lambda i}^{\Psi}$ is the intensity contributed by the light source i to ray Ψ (following the assumption $S_i = 1$). Now equation (3) becomes

$$R_{\lambda}^{\Psi} = I_{\alpha\lambda} k_{\alpha} O_{d\lambda} + \sum_{i=1}^{m} S_i I_{\lambda i}^{\Psi} \tag{7}$$

which can be rewritten as

$$R_{\lambda}^{\Psi} = I_{\alpha\lambda} k_{\alpha} O_{d\lambda} + \sum_{i=1}^{m} I_{\lambda i}^{\Psi} - \sum_{i=1}^{m} (1 - S_i) I_{\lambda i}^{\Psi} \tag{8}$$

Therefore, at run time, when a shadow-feeler, spawned by a ray Ψ towards light source i, is traced, it returns the value denoted by S_i. Then, the value of the pixel at $C(\Psi)$ is adjusted by

$$I_{\lambda}(C(\Psi)) \leftarrow I_{\lambda} \, (C(\Psi)) - \omega(\Psi) \, (1 - S_i) I_{\lambda i}^{\Psi} \tag{9}$$

4. IMPLEMENTATION

Since we need to control the processing order of rays based upon some user-specified criteria, the core of our ray tracer is a priority queue implemented using a heap data structure.[9] We obtain good performance with this approach since a heap can support any priority-queue operation on a set of size n in $O(\log_2 n)$ time. Some of the schemes which can be used to prioritize the rays are:

(a) weight (or contribution) of the ray
(b) ray type (e.g. primary, reflection, transmission, shadow)
(c) ray origin (for incremental supersampling)
(d) ray tree depth
(e) object complexity or type
(f) surface properties
(g) light source attributes (e.g., intensity, colour)
(h) a combination of some of the above schemes

Selection of a priority scheme can be easily accomplished by encoding the choice in an input configuration file. This gives the user the maximum flexibility on an image-by-image basis.

Our implementation begins by reading several input configuration files, including the priority criteria selection and the scene description file. Ray processing beings by inserting all primary rays into the priority queue in two steps. In the first step we insert the rays into the queue in an arbitrary order (regardless of priority). The initial number of rays will be at least equal to the number of pixels in the image plane, but this number could be much larger if anti-aliasing techniques such as supersampling are employed. In the second step, following completion of the ray entry step, we 'heapify' the queue according to a heap property key, which is based upon the user-specified priority scheme. We are now ready to begin ray tracing.

The data structure for a queue entry (representing a ray) has to contain data fields which enable the ray tracer to identify the affected pixel, the heap key, and any parameters needed by the shader for correct rendering of the pixel colour. We use the RayNode data structure, shown in Figure 3, in our implementation. The fields Px, Py, heap_key, and R, G, B are added for the priority-driven ray tracing while all other fields are also used by traditional recursive ray tracers.

In addition to the queue, the entire image is maintained in a two-dimensional array of floating point RGB values, each representing one pixel (or sub-pixel if anti-aliasing is being performed). These are the entities that are updated when a ray produces some update information for a pixel. The update is performed according to equation (5) for reflection and transmission rays, and according to equation (9) for shadow rays.

Compared to traditional recursive ray tracing, our algorithm requires a substantially larger amount of memory consumed by the priority queue and floating point image buffer. However, this is not a severe limitation due to the relatively low cost of memory in current workstations. An analysis of the memory needs of this algorithm can be found in Section 5, and test case results are described in Section 7.

When we remove each ray from the top of the heap (the head of the priority queue), we re-heapify it to maintain the heap property. Now we need to process the ray, but first we must decide if a display update should be performed. We decided this by inspecting the heap key of the ray. For example, if priority values are based on the ray tree level, a display update is triggered as processing of rays passes from one level to another (the heap key value makes the transition from one value to another). The incremental nature of our algorithm is visually demonstrated with these display updates (see Plate 2).

Finally, we shoot the ray based on the origin and direction values obtained from the RayNode data structure (see Figure 3). As we step through the 3D space-subdivision grid structure along the path of the ray, we perform ray–object intersection calculations with the objects within the current grid cell, as in traditional ray tracers.[10] We depart from classical recursive ray tracing when we intersect an object since we do not recursively follow reflection and/or transmission rays originating from the intersection

```
RayNode is record: -- $\psi$
    Px, Py:      integer;    -- pixel coordinates of ray-tree origin, $C(\psi)$
    heap_key:    integer;    -- heap property (priority) key
    R, G, B:     float;      -- real RGB color components, $I^{\psi}_{\lambda i}$
    ray_level:   integer;    -- ray tree level
    ray_type:    RayType;    -- primary / reflection / transmission / shadow
    weight:      float;      -- ray contribution, $\omega(\psi)$
    ray:         RayRecord;  -- origin coordinate and direction vector
    end RayNode;
```

Figure 3. The RayNode *data structures representing one ray in the queue*

point. Instead, these rays, and one shadow feeler per light source, are assigned a priority value and then inserted into the priority queue.

We compute the local illumination model in a slightly different manner than the one used by recursive ray tracers. Typically, the local shading model includes the logic for the generation of shadow feelers. These rays, which are shot from the ray–object intersection point to each light source, determine if the intersection point is in shadow (or in partial shadow if translucent objects are in the scene). The intensity of the light reaching the intersection point is appropriately attenuated if the light sources are occluded by other objects in the scene. However, if shadow feelers are placed in the priority queue with all the other rays, we have to delay this part of the local illumination model calculations until the shadow feelers are actually shot. Initially, the local shading model must assume that the full intensity of light from each source reaches the point of intersection. Later, when the shadow feelers are removed from the queue and processed, the intensity at the pixel of interest may have to be adjusted (attenuated) if the light sources are partially or fully occluded.

The way we implement this idea is as follows: if the ray we currently process is a primary, reflection, or transmission ray, the local illumination is computed as in equation (3). This value is multiplied by weight and added to the pixel at (Px,Py), as described in equation (5). At this intersection points, if reflection or transmission rays are spawned they are assigned the same (Px,Py) value, a new weight is computed for them according to equation (2), and a heap_key is assigned according to the user-defined priority scheme. If a shadow feeler is spawned for light source i, the value of $I^{\Psi}_{\lambda i}$ is computed, as in equation (6) and stored in the R, G, and B fields. When a shadow feeler is popped from the queue, it is traced and a value for S_i is computed. Then, the colour of the pixel at (Px,Py) is adjusted as described in equation (9).

5. ANALYSIS OF MEMORY USE

The advantages of the priority-driven approach over classical ray tracing do not come for free. Compared to classical ray tracing, the priority-driven ray tracer requires significantly more memory. For example, let us consider the synthesis of an image with a resolution of $N \times M$ pixels. If we use double-precision representation for all floating point values (i.e. 8 bytes), the amount of memory needed for the floating point RGB frame buffer is $24MN$. For $N = M = 256$ we need 1·5 Mbyte, whereas a traditional ray tracer, which processes one pixel at time, does not require this storage at all.

The biggest memory consumer, however, is the priority queue. Let us consider a worst-case scenario, where we trace rays up to h generations in an environment with m light sources. Assume also that all objects are both reflective and transparent. Furthermore, the background is never hit by a ray (i.e. we terminate each ray tree by reaching the maximum recursion level). Finally, assume that shadow rays are given lower priorities than reflection and transmission rays. In this case, at some point the ray tracer will maintain a priority queue that corresponds to a maximal ray-forest. In such a ray-forest each tree is a binary tree of height h with m additional sub-nodes at each internal tree node (as shown in Figure 1). Now we carefully compute the number of rays each of these trees contains.

A pop operation of a shadow ray from the queue reduces its length by one. A pop operation of a reflection or transmission ray from the queue leads to intersection calculations. If the level (generation) of the popped ray is less than h, and the intersected object is both reflective and transmitive (as we assume), $m+2$ rays are pushed into the queue—m shadow feelers, one reflected ray, and one transmitted ray. If the level of the popped ray is equal h, only the m shadow feelers are pushed into the queue. If, as we assume, shadow rays have lower priority, we will always perform the second type of pop until we reach level h. Therefore, the worst-case queue size will occur when we have iterated to the lowest level of the ray-forest, where each ray tree has a height of h.

Examining the queue corresponding to one pixel, it will consist of all shadow rays for all intersection points (i.e. all tree nodes), and all rays of level h (i.e. leaf nodes), as shown in Figure 1. We show, by induction on h, that the queue length corresponding to the worst-case tree of level h, denoted by $l_q(h, m)$ is

$$l_q(h,m) = m(2^{h+1} - 1) \tag{10}$$

When $h = 0$, we find in the queue the shadow rays at intersection point of the primary ray and an object (represented by the only node in the tree), and $l_q(0, m) = m$. Assume now that we have the queue corresponding to the worst-case tree of level h, having $l_q(h,m)$ rays as in equation (10). To generate a tree of level $h+1$ we have to add a reflection and a transmission ray at each node. Then, when we pop each one of these rays from the queue, we replace it with m shadow feelers. A tree of level h has 2^h leaf nodes, therefore, we add 2×2^h rays and then replace them by $m2^{h+1}$ shadow feelers. As a result, the length of the worst-case queue of level $h+1$ is

$$\begin{aligned} l_q(h+1, m) &= l_q(h, m) + 2^{h+1} - 2^{h+1} + m2^{h+1} \\ &= m(2^{h+1} - 1) + m2^{h+1} = m(2^{h+2} - 1) \end{aligned} \tag{11}$$

which proves the correctness of equation (10) by induction.

For an image of $N \times M$ pixels, each of them super-sampled by r rays, the total queue length, denoted by $L_q(h, m, N, M, r)$, is

$$L(h, m, N, M, r) = r \times N \times M \times l_q(h, m) \tag{12}$$

If we assume $N = M = 256$, $h = 4$, $m = 2$, and $r = 1$, then $l_q(h, m) = 62$, and $L_q(h, m, N, M, r) \approx 3 \times 10^6$ rays. Since each RayNode structure (described in Section 4) requires approximately 100 bytes, memory requirement for the heap data structure can potentially reach, in this worst-case scenario, 387 Mbytes. For a high-resolution image of 512^2 where 9 rays are cast from each pixel (and still assuming $h = 4$ and $m = 2$), memory needs may soar to 13.6 Gigabytes! Fortunately, these conditions can only arise through contrived worst-case examples and are very unlikely to occur in practice. Moreover, many priority schemes do not require storage of the whole tree, such as in Reference 11. Memory consumption for all cases we have tested were all in the range affordable by common workstations, as described in Section 7.

6. PRIORITY SCHEMES

We have developed five priority schemes: *level*, *object*, *ray-type*, *region*, and *contribution*.

6.1. Level

In this priority scheme, rays of the *n*th generation have higher priority than rays in the $(n + 1)$th generation. In practice, this method translates to breadth-first traversal of the tree. If we trigger an image display when the priority changes, the first image generated depicts the scene rendered by primary rays (see Plate 2(a)), whereas the second image shows the contribution of first generation shadow and relection rays (Plate 2(b)), and the second generation of rays (Plate 2(c)).

6.2. Object

In this scheme, priority is assigned to objects rather than to rays. When a ray hits an object it inherits the object's priority. This leads to a priority scheme where some objects in the scene are rendered more accurately, whereas other objects are only shown rendered by primary rays. In Plate 3, we first observe the scene rendered by primary rays (Plate 3(a)). Next, the sphere-flake is rendered with secondary rays. Only then does the ray tracer render the secondary rays onto the boxes on the left (Plate 3(c)) and the floor (Plate 3(d)).

6.3. Ray-type

In this scheme, priority is assigned to the type of the ray. There are actually four levels of priority: primary, reflection, transmission and shadows. Primary rays are set to have the highest priority, whereas the priorities of the others are set by the user. Plate 1 is an example where all shadow rays are computed after all reflection and transmission rays were processed.

6.4. Region

In this scheme, the user prioritizes several rectangular disjoint screen regions. The rays are assigned a priority according to the priority assigned to the region they belong to. The outcome is that regions with a higher priority are rendered before regions with a lower priority. Plate 4 shows an example of ray tracing three chess pieces with the region priority scheme.

6.5. Contribution

In this scheme, a priority is assigned to a ray according to the contribution it can potentially make to the final image (see Plate 5). The priority is a function of the ray weight, as shown in equation (2); it is stored in the ray data structure (see Figure 3). As we show in the next section, this priority scheme coverges very quickly to the final image; the user can see most of the image early in the rendering process. In addition, as we also show in the next section, this scheme requires the least amount of memory for the ray queue.

7. RESULTS

To explore the relative advantages and disadvantages of the priority schemes we implemented, we chose one scene (the sphere-flake in the mirror room, shown in Plate 1) and applied our schemes while measuring the number and type of rays rendered, and the queue size at each moment of the rendering process. For each intermediate image, we measured the difference between that image and the final image to determine how fast a priority scheme converges to the final image. All schemes assumed rendering three ray generations ($h = 3$) and one light source ($m = 1$). Since the objects in this scene are completely opaque, no transmission rays were traced. Nevertheless, one should realize that this test case is an extreme example since we trace rays in a closed environment (room) where no ray hits the background and all objects are perfect reflectors (mirrors).

Figure 4 shows the behaviour of our ray tracer when driven by the *ray type* priority scheme (Section 6.2). The graph shows the number of rays rendered as a function of time. In the first 20 seconds only primary rays are processed. When those are exhausted, an image display is triggered (denoted by the vertical spike), and then the ray tracer processes reflective rays (see Plate 1(b)), and then shadow rays (Plate 1(c)). As a comparison, we show the behaviour of the ray tracer under the *contribution* scheme (Figure 5). One can see that secondary rays that significantly contribute to the image (e.g. shadow rays) are processed at the same time as the primary rays.

Figure 6 shows the size of the queue during the rendering of the sphere-flake in a room of mirrors (Plate 1), for different priority schemes. A traditional ray tracer needs space for only one ray at any time. Although in the worst case scenario, this 256^2 image would require a queue of $L_q(3, 1, 256, 256, 1) \approx 0{\cdot}98 \times 10^6$ rays, for this example, the queue reaches only 6–20 per cent of that size. We observe three groups of priority schemes. In the first group *ray-type*; then *level* and *object*; then *contribution* and *region*. In the *ray-type* scheme the user chose to delay the tracing of shadow rays, consequently requiring a queue that is almost 20 per cent of the worst-case queue length. If shadow

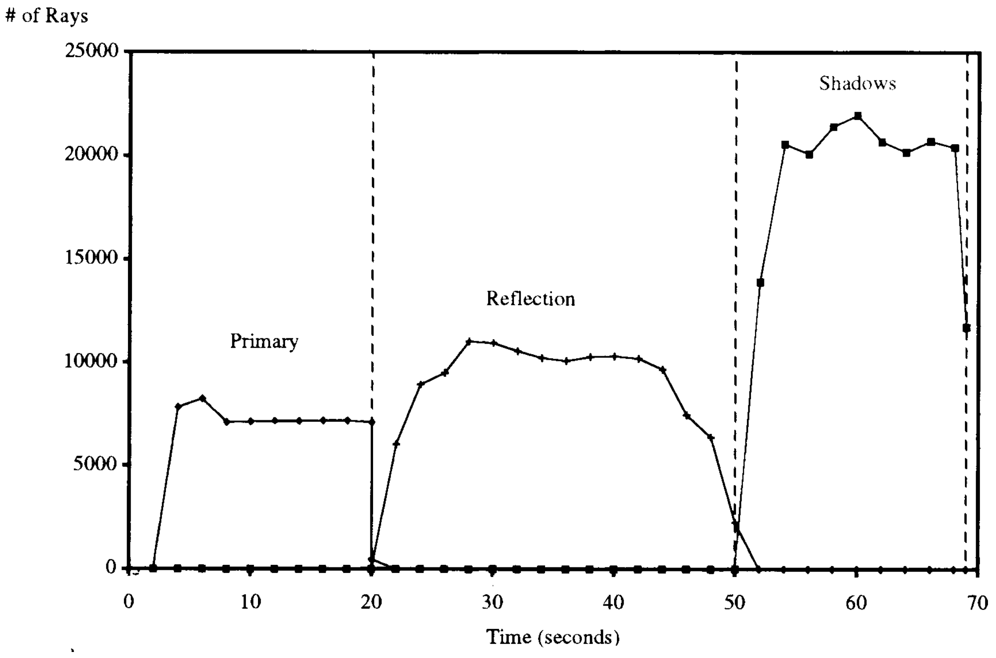

Figure 4. Number of rays rendered when rendering with the ray-type priority scheme. Vertical lines denote image display

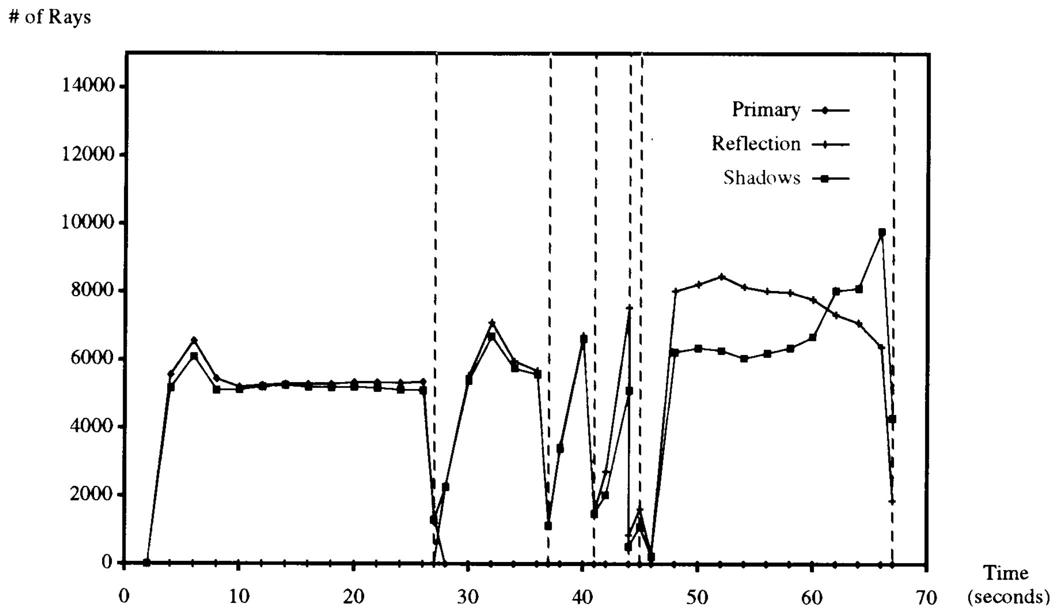

Figure 5. Rays rendered when rendering with the contribution priority scheme

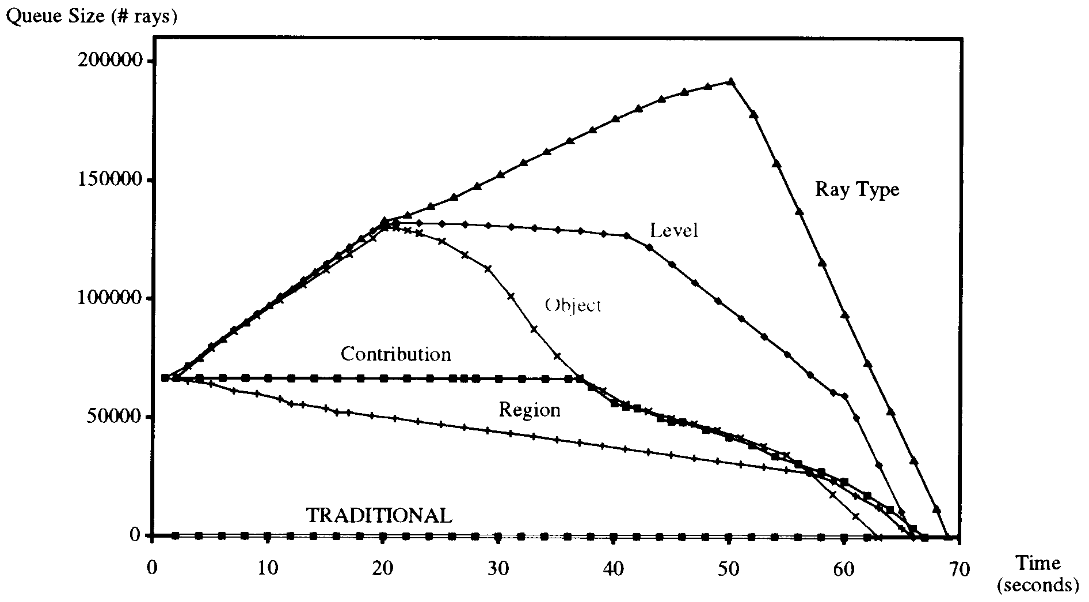

Figure 6. Queue length for the various priority schemes

rays are given higher priority than reflection rays, it requires space similar to that of the second group. The second group (*level* and *object*) needs a maximum of 13 per cent of the worst-case queue length. In the last group, the *region* priority scheme can be made to require small amount of memory if the regions the user defines are small. Also in the third group is the most useful scheme in this, *contribution*, which require up to 6 per cent of the worst-case queue length.

In terms of time performance, we see that there is no significant penalty for using priority-driven ray tracing over the traditional method. Whereas the latter takes 62 seconds, all the priority-driven methods take 63–66 seconds. Although we show here only one example, it reflects our experience when rendering other scenes.

We have also measured the speed of convergence to the final image. The metric we used is based on average pixel-wise absolute colour difference. The results are shown in Figure 7. The traditional method advances linearly, whereas all the priority schemes (except *region*) coverge faster to the final image; that is, the intermediate images generated while rendering are closer to the final image. Although all methods converge to the same final image, methods such as *ray-type* and *contribution* provide the user with rapid feedback by generating rather complete images at the early stages of rendering.

If we do not plan to use priority by light source, all shadow rays at an intersection point will receive the same priority and will also be popped from the queue and rendered at the same time. Therefore, one can push to the queue only one shadow ray with the assigned priority, but without information in the ray-direction fields (see Figure 3). When this record is popped from the priority queue, we spawn m shadow feelers, trace them, and update the pixel value. This approach conserves large amounts of memory since it is equivalent to always assigning $m = 1$ in equation (10).

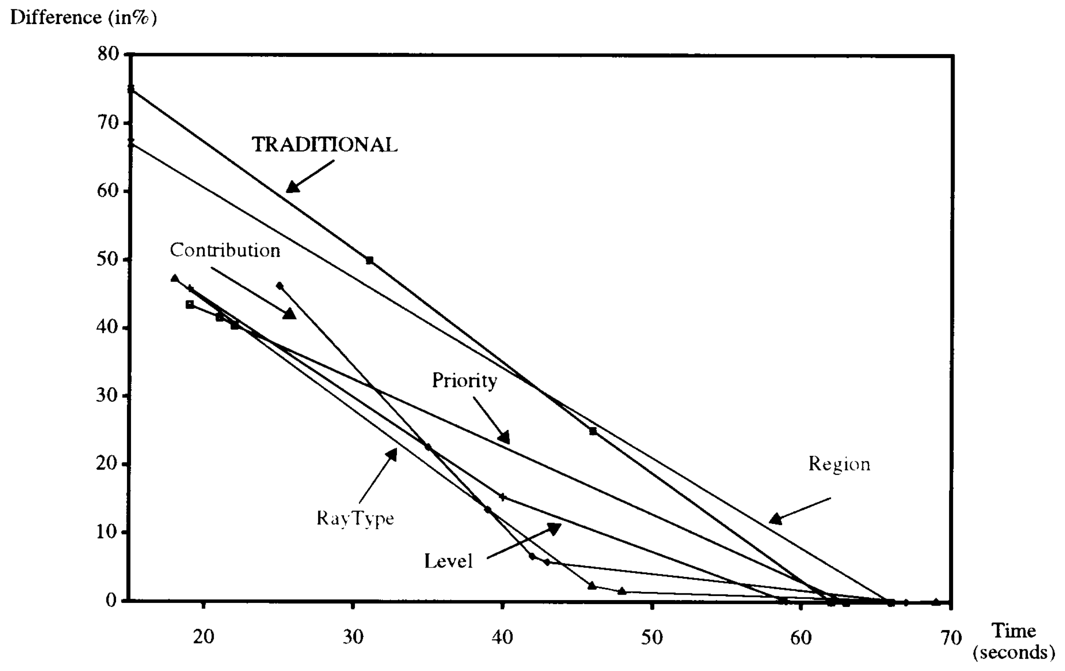

Figure 7. Convergence of the various priority schemes to the final image

The result of this strategy is mostly beneficial for the *ray-type* scheme. In the example shown in Figure 6, in which we had two lights, the queue length for this scheme will peak at 127,000 queued rays rather than 192,000, a reduction of 34 per cent. The image in Plate 6, which was rendered with three light sources, required only half the queue size when we applied this optimization.

We also observe that if one does not plan to employ a screen-based criterion (such as the *region* scheme), primary rays will always be traced before any secondary ray or shadow feelers are. Therefore, there is no need to push all primary rays to the queue. Rather, primary rays are traced as in traditional ray tracers with the exception that secondary rays and shadow feelers, instead of being recursively processed, are pushed into the queue. This optimization will be manifested in the graph of Figure 6 by causing all methods (except *region*) to start at time zero with empty queue rather than a queue containing *NMr* rays (65,536 in Figure 6 and 262,144 in Plate 6).

8. CONCLUSIONS

Existing ray tracers, once given the rendering specifications, are not controllable by the user. The priority-driven ray tracing provides a mechanism to steer rendering and deliver intermediate images amid processing. We have demonstrated the utility of five priority schemes and shown that they require reasonable amounts of memory and consume negligible amounts of overhead time. Priority-driven ray tracing can be extended to support other priority schemes, and mixtures of several priority criteria. In our current implementation, once a priority has been assigned to a ray it cannot be changed. The ability to demote or promote queued rays will provide us the support needed for dynamic steering. In addition, various other priority schemes can be explored. Examples include priority assigned to rays based on cache performance optimization[12] or priority assigned to shadow feelers according to light source intensity or distance. Finally, we have used numerical methods to show the faster convergence of the priority based method. One could employ some perceptual metric[13] to better control our priority ray tracer to acheive even faster effective image comprehension.

ACKNOWLEDGEMENTS

This work has been partially supported by the U.S. National Science Foundation under grant CCR-9211288. We thank Yair Kurzion, Raghu Machiraju, Klaus Mueller, and especially Ed Swan for reviewing the manuscript.

REFERENCES

1. T. Whitted, ‘An improved illumination model for shaded display’, *Communications of the ACM*, **23**, (6), 343–349 (1980).
2. M. F. Cohen, S. E. Chen, J. R. Wallace and D. P. Greenberg, ‘A progressive refinement approach to fast radiosity image generation’, *Computer Graphics*, **22**, (4), 75–84, (1988).
3. A. S. Glassner, *An Introduction to Ray Tracing*, Academic Press, 1989.
4. J. Painter and K. Sloan, ‘Antialiased ray tracing by adaptive progressive refinement’, *Computer Graphics (SIGGRAPH ’89 Proceedings)*, **23**, July 1989, pp. 281–288.
5. C. Goral, K. E. Torrance and D. P. Greenberg, ‘Modelling the interaction of light between diffuse surfaces’, *Computer Graphics*, **18**, (3), 212–222 (1984).
6. B. Smits, J. R. Arvo and D. H. Salesin, ‘An importance-driven radiosity algorithm’, *Computer Graphics*, **26**, (3), 273–282 (1992).
7. A. A. Sousa, A. M. C. Costa and F. N. Ferreira, ‘Interactive ray-tracing for image production with increasing realism’, *Proceedings of Eurographics ’90*, pp. 449–457.

8. J. D. Foley, A. van Dam, S. K. Feiner and J. F. Hughes, *Computer Graphics, Principles and Practice*, second edition, Addison Wesley, 1992.
9. T. H. Cormen, C. E. Leiserson and R. L. Rivet, *Introduction to Algorithms*, MIT Press, 1989, pp. 140–152.
10. A. S. Glassner, 'Space subdivision for fast ray tracing', *IEEE Computer Graphics and Applications*, **4**, (10), 15–22 (1984).
11. C. Sequin and E. K. Smyrl, 'Parameterized ray tracing', *Computer Graphics (SIGGRAPH '89 Proceedings)*, **23**, July 1989, pp. 307–314.
12. S. A. Green and D. J. Paddon, 'Exploiting coherence for multiprocessor ray tracing', *IEEE Computer Graphics and Applications*, **9**, (6), 12–26 (1989).
13. H. Rushmeier, G. Ward, C. Piatko, P. Sanders and B. Rust, 'Comparing real and synthetic images: some ideas about metrics', *6th Eurographics Workshop on Rendering*, Dublin, Ireland, June 1995, pp. 213–222.