Introduction

The generation of realistic computer images has many useful applications. Many of the most important applications of computer technology involve simulation of the real world. The more realistically such simulations can be presented to the user, the more meaningful and useful they will become. Graphs typically enhance our intuitive understanding of simple data sets. The capacity to display high-quality 3-dimensional images allows for visualizing much more complex relationships. Realistic rendering also has a wide range of applications in entertainment. Complex photography and animation can be replaced by computer simulation; recent Disney films, for instance, have demonstrated that computer-generated images can be combined with traditional animation to achieve impressive effects. In addition, our glowing visions of virtual reality systems depend heavily on fast and realistic rendering.

Ray tracing provides us with one method for rendering realistic images. It is useful because it is capable of producing high-quality realistic images. It is attractive because it is conceptually simple and uniform: all effects are achieved by tracing the propagation of photons along rays. Unfortunately, ray tracing can also be quite expensive in comparison to other rendering methods. The purpose of this project is to achieve reasonably fast and portable ray traced rendering through the development of parallel ray tracing algorithms. Currently, the code is written in NESL, a data-parallel language developed by Guy Blelloch. The platforms which are currently supported are Cray YMP, Connection Machine CM-2, and simulators on serial workstations.

Ray Tracing

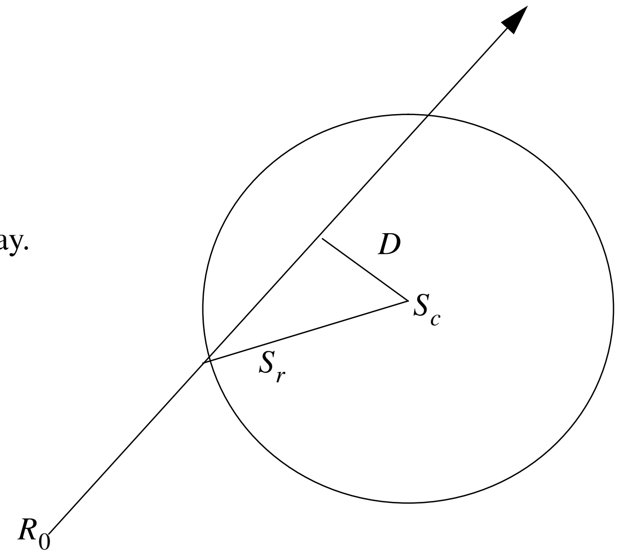
One of the methods which can be used in the generation of realistic images is ray tracing. Ray tracing is based on a simple optical model: photons propagate from light sources along rays, and those photons which strike the viewer's eye form the image of the world. This model allows us to simulate a great many optical phenomena from simple reflections to complex lens effects. However, being only an approximation of reality, some effects such as diffuse ambient reflection and interference patterns are beyond the capacity of the simple ray-based model.

Tracing all light from light sources through the world would be extraordinarily time consuming. In order to make ray tracing practical, we must make an important observation. Most photons travel off into space without ever hitting your eye; only a small number of the rays sent out from a particular light source ever reach your eye. Thus, we can trace rays backwards from your eye to the light sources, never considering the myriad other light rays in the world, and the number of rays traced will be drastically reduced. Unfortunately, even with this crucial optimization, the number of rays we must consider is still quite large.

The primary computation performed during ray tracing is that of ray-surface intersection. In order to determine what light arrives at the eye along a given ray, we must trace this ray through the world and see what object (if any) it hits. Upon striking an object, we can then determine the light being propagated from it to the viewer's eye. For general surfaces, the ray-surface intersection calculation can be very complex. This, coupled with the often large number of rays, The ray is described by the parametic equation:

$$R(t) = R_0 + tR_d$$

Where t is the distance along the ray.



1) Find distance squared from ray origin R_0 to sphere center S_c

$$OC = S_c - R_0$$
$$L_{2oc} = OC \cdot OC$$

2) Calculate ray distance closest to center:

$$T_c = OC \cdot R_d$$

3) If $T_c < 0$, then the ray points away from the sphere ... stop.
4) Find square of half-chord intersection distance:

$$T_{hc} = S_r^2 - L_{2oc} + T_c^2$$

5) If $T_{hc} < 0$, then the ray missed the sphere ... stop.
6) Calculate intersection distance:

$$L_{2oc} < S_r^2 \Rightarrow t = T_c - \sqrt{T_{hc}}$$
$$L_{2oc} > S_r^2 \Rightarrow t = T_c + \sqrt{T_{hc}}$$

Figure 1: Ray-Sphere Intersection Algorithm

accounts for the relative high cost of ray tracing.

The primary object used in my system is the sphere. An initial attempt at an intersection algorithm would probably be an algebraic one. Knowing the parametric equation for a ray and the parametric equation for a sphere, we can substitute the ray equation into the sphere equation and then solve. However, this is not a particularly efficient algorithm and we can easily do better. The algorithm presented in Figure 1 is geometrically based and is quite a bit more efficient.

Once the point of intersection has been found, we need a way to compute the illumination which will propagate through that point to the eye. This illumination (or shading) calculation can be very complex or fairly simple. A complex model offers the possibility of highly accurate rendering while the use of a simple model represents a willingness to sacrifice accuracy for rendering speed. Ideally, we would like a model which provides a reasonable approximation of the real world while not requiring inordinate amounts of computation or storage space. The model which I use is a standard one; it is essentially simple while yielding reasonably realistic results.

After determining that a ray has indeed hit an object, we want to compute the light intensity I_λ at the point of intersection. The subscript indicates that the intensity is wavelength dependent and hence must be computed for each primary wavelength: red, green, and blue. The light arriving at a particular point on a surface is divided into two categories: specular illumination and direct illumination. Specular illumination is the light which arrives along the reflected direction ($\bar{R}$ in Figure 2). Direct illumination is that which arrives directly from a light source at the point of intersection.

The surface that the ray has hit is characterized by three quantities: a specular reflectivity

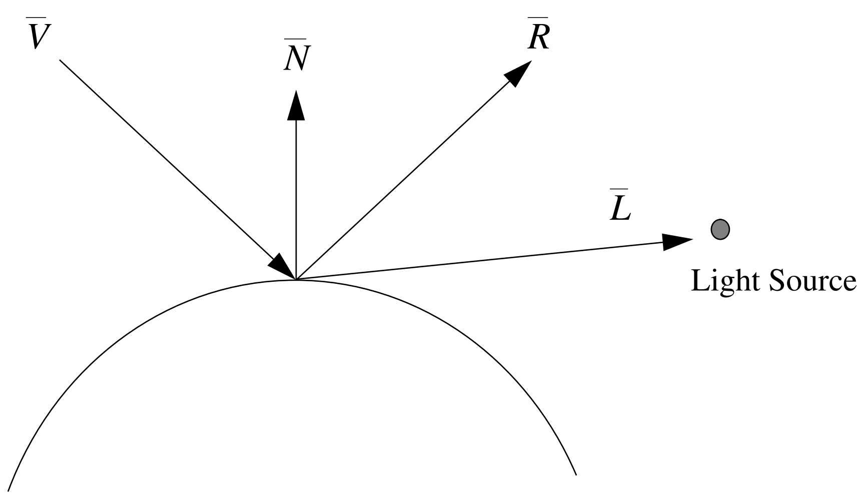


Figure 2: Light rays considered in shading

coefficient k_s, a diffuse reflectivity coefficient k_d, and an RGB color vector O_d. The two reflectivity coefficients control what percentage of the two kinds of illumination are actually reflected off the surface. For instance, a diffuse reflectivity of 0.9 would indicate that 10% of all direct illumination on that surface is absorbed and that the other 90% is transmitted along the incident ray. The specular coefficient can be understood as controlling the shininess of the object; a 0.0 would correspond to chalk and a 1.0 would correspond to chrome. The relative brightness of the object is controlled by the diffuse coefficient. Finally, the color vector describes the fundamental color of the object.

The calculation for determining the reflected light is simple. We spawn a reflected ray, compute its intensity $(I_{r\lambda})$, and multiply by the specular reflectivity coefficient. The intensity of the illumination carried by the reflected ray is computed by recursively calling the ray tracer with the newly reflected ray.

The calculation of the direct illumination is a bit more complex. We must consider each of the light sources in the world. The shadow function S_i returns 1 if light from that source is arriving at the current point and 0 if it is not. The intensity of the light source is given by $I_{p\lambda i}$. All these factors are multiplied together and are weighted by the diffuse reflectivity coefficient and the angle of incidence of light from the source. In practice, we can determine whether light from a source is arriving at a particular point by shooting a ray towards it. If no objects are hit before the ray reaches the light source, then light will arrive from that source. Otherwise, the current point will be in shadow with respect to that light source.

These two calculations demonstrate an important feature of ray tracing: almost all tasks are performed by spawning and tracing new rays. The same basic algorithms allow us to perform most of the work that is required. Consequently, the system can be fairly compact in comparison to other rendering systems and the various computations it performs are quite uniform.

Having calculated the specular and direct illumination, we can compute the total light intensity at the point of intersection. For each primary wavelength, the light intensity is given by:

$$I_\lambda = k_s I_{r\lambda} + \sum_i S_i I_{p\lambda i} k_d O_{d\lambda} (\bar{N} \cdot \bar{L}_i)$$

In this model, rays can be divided into three categories: *eye rays*, *shadow rays*, and *reflected rays*. Eye rays are those which are initially spawned from the viewer's eye through each pixel of the image plane. They compose the first "generation" of rays which we will need to trace. Shadow rays are those rays which we shoot from the surface of an object toward light sources to determine the amount of direct illumination falling upon the surface at that point. Shadow rays are second-class rays; we don't care what they hit, only whether they hit anything at all, and they never spawn children. Reflected rays are in essence the same as eye rays; the only difference between them being that reflected rays originate on the surface of an object rather than at the viewer's eye. It is the reflected rays which make up all subsequent generations of rays.

For practical reasons, we must impose limits on how long rays are allowed to survive. There are two separate problems which must be dealt with. First, it may be that objects have been positioned such that reflected rays might bounce back and forth between them forever (or at least longer than your average user is willing to wait). Second, at some point the weight associated with a ray may dwindle so far as to be virtually negligible. An easy way to deal with these problems is to impose two limits: a limit on the maximum number of generations and a threshold for ray weights below which rays are simply dropped. This approach yields quite acceptable results without requiring inordinate computation. A slightly more sophisticated approach is referred to as *adaptive depth control*. It represents a reluctance to place constant predetermined limits on ray generations and weights. Instead, it monitors the color of pixels in the image. When they seem to have converged (their last update did not change the color beyond a certain threshold), any further rays descended from them are killed. One problem with adaptive depth control is finding an acceptable way to determine when a pixel is converging to a particular color.

Another important observation to make is that the size of each successive generation of rays will never be larger than the size of its parent generation, although it may be smaller. It guarantees that the performance of our system will not be destroyed by rapid growth in the number of rays. Unfortunately, this property is not preserved in more complex shading models. There are several models in which we would spawn several child rays; for instance, we might spawn a spread of reflected rays rather than a single one to improve the rendering of reflections. In systems based on such models, the proliferation of rays is a major concern.

Suitability to Parallel Systems

Of all the various rendering models, ray tracing is particularly attractive from the perspective of parallel computation. Because all the various light rays which we are interested in are independent of each other, ray tracing is naturally adaptable to parallel architectures. Each ray can be traced separately without any knowledge of progress along any other ray; rays do not interact and the chances of the same ray being traced twice are minute. In addition, any given ray will contribute light to exactly one image pixel, and that pixel will be the same as that for all its ancestor and descendent rays. Therefore, we can build up individual image elements in parallel and then combine them when they are complete to form the final picture. An additional benefit is that since the operations of ray tracing are quite uniform (everything is accomplished by recursively tracing new rays), it can be implemented on SIMD architectures just as easily as on MIMD architectures.

Other rendering methods do not map onto a parallel architecture nearly as well as ray tracing does. The method used by most commercial rendering software makes use of various global data structures. For example, they perform hidden surface elimination by using a z-buffer. While ray tracing requires global information, in particular the world database, this information is never altered so it may easily be distributed. On the other hand, z-buffers are continuously updated which naturally presents difficulties for rendering various image elements in parallel.

Prior Efforts

At this time, there has not been a great deal of work done in building parallel ray tracing systems. What systems have been developed tend to be either platform-specific or rely on dedicated hardware.

Much of the work that has been done has centered around very low-level issues. For instance, Green and Paddon have written one of the few papers on parallel ray tracing. Their system was developed on a Connection Machine, and their research was very much tied to the platform they were running on. The major issue which Green and Paddon address is how to distribute the world database across the nodes of a CM-2. Their discussion reads very much like a work on virtual memory management rather than ray tracing; they use traditional memory-management terms such as “working set” to describe the operation of their ray tracer.

Like Green and Paddon, most other research has also focused on particular architectures and how best to map ray tracing algorithms onto them. In contrast, my research represents an attempt to create a highly portable ray tracing system. Portability is achieved through the use of the NESL language which was developed by Guy Blelloch. Using NESL, the same ray tracing program has been tested on a Cray YMP, a Connection Machine CM-2, and a parallel simulator running on a DECstation 5000.

NESL

NESL is a strongly-typed functional language designed specifically for writing parallel programs which can be targeted at a variety of parallel architectures. In fact, NESL is capable of generating code for both SIMD and MIMD systems. The language provides a data-parallel model of parallel computation; the basis for parallelism in NESL is the vector data type. NESL provides a variety of primitive parallel vector operations such as **min-reduce** and **permute**, and all NESL functions can be applied in parallel to each element of a vector. In addition to this, NESL allows for nested parallelism. Thus, a parallel function can itself be applied in parallel. For instance, one might apply **min-reduce** in parallel to each element of a vector of vectors. This is a powerful feature which eases the task of programming parallel systems. One need not consider at which level parallelism occurs, since it may occur at many levels. The programmer is free to consider the most natural way of writing each function with less concern for how any parallelism within that function will interact with other functions within it or which invoke it.

Before presenting any code from my system, I will present a brief and slightly simplified version of the fundamentals of NESL syntax. The language supports both a Lisp-like syntax and an ML-like syntax, but my development has been solely in the Lisp-like syntax. Therefore, it is the only one which I shall discuss.

Vectors can be written explicitly as #v(1 2 3 4 5 6).

Functions are defined in NESL by the **defop** form:

(**defop** (name arg_1 ... arg_n) exp)

Data structures are defined with **defrec**:

(**defrec** (name $type_1$ ... $type_n$))

The **with** form allows for the binding of local variables:

```
(with ((id1 exp1)
       (id2 exp2)
        ...
       (idn expn))
   exp)
```

The various identifiers will be bound sequentially to the result values of the corresponding expressions. The body of the **with** is evaluated in the resulting environment.

The **over** form provides a means for the parallel application of NESL expressions:

```
(over ((id1 vector1)
        ...
       (idn vectorn))
   exp)
```

This will construct a new vector by evaluating the body for each element of the listed vectors (which must all have the same length). For each parallel invocation of the body, the listed identifiers will be bound to the corresponding elements of the listed vectors. The result values of the body are used to construct the new vector.

A syntactically more convenient method for writing parallel function applications is:

(**v.**name arg_1 ... arg_n)

The arguments must be vectors (all of the same length). The named function will be applied in parallel to the various elements and the resulting values will be collected into a new vector. For example, (v.+ #v(1 2 3) #v(5 6 7)) would produce the vector #v(6 8 10).

The final basic syntactic form is **if**, which mirrors the Lisp if expression:

(**if** test true-exp false-exp)

The slots of data structures are accessed using pattern-matching. For instance, if you have declared a data type with

```
(defrec (triple a b c))
```

You would access it using the following:

```
(with (((triple a b c) some-triple))
   ...)
```

Portability is achieved through compilation into a standard intermediate code (VCODE) rather than native machine code. The intermediate code is interpreted on the target machine, thus the same code can be run on a variety of different platforms.

System Overview

The NESL compiler resides in a central Lisp process, presumably running on a Unix workstation. When the rendering system is loaded and invoked, the compiler outputs a VCODE file and starts a remote VCODE interpreter on the target machine. To support my rendering system, three separate processes are required. First, the Lisp process which hosts the NESL compiler also acts as the interface manager. It is responsible for creating whatever display windows are necessary for displaying the final image. Next is the VCODE interpreter which is spawned by the Lisp process on the target machine. Finally, a vector plotting process (Vplot) written in C is started to act as an intermediary between the VCODE interpreter and the display window(s) created by the Lisp interface manager.

The VCODE interpreter performs all the computations involved in rendering and it is the sole parallel process. It generates a stream of data describing pixels in the image which it is rendering; this data is sent to the Vplot process. Vplot is responsible for translating the raw image data into a form which can be displayed. In this case, it translates the data into calls to the X server. Vplot also provides a facility for dumping the image data directly into a 24-bit color TIFF file. This feature is particularly critical on displays which are incapable of generating all the colors

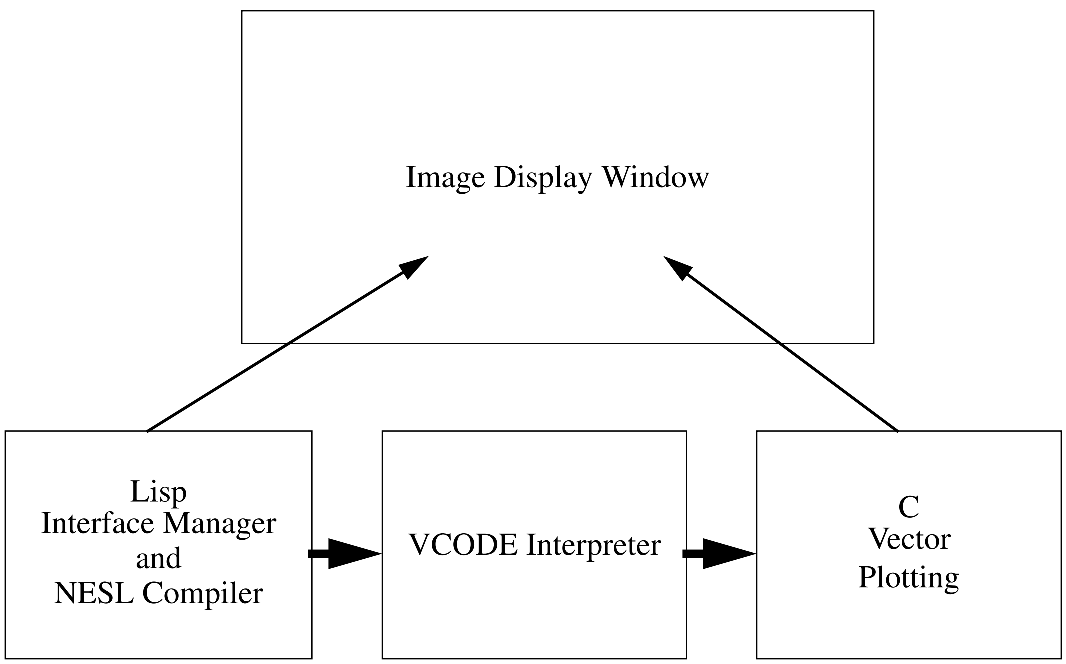


Figure 3: System Organization Overview

present in the image. The TIFF file will accurately record the entire image with little loss of fidelity while the physical display might show significant degradation.

Ray Tracing in Parallel

Having seen the general overview of the system, let us delve into the rendering engine itself. The general model which we are operating with is that a series of rays are initially spawned through the image plane. These eye rays are traced to find any possible intersections. From these intersections we shoot rays at all the light sources to determine direct illumination and we spawn reflected rays to compute specular illumination. It is the reflected rays which survive to become the next generation of rays since they are the rays which we need to recurse upon. Shadow rays are traced during the current generation and then discarded.

The most fundamental question which must be asked is what will be the basis for parallelism in the ray tracing system. Recall that the fundamental operation of the ray tracing system is the ray-surface intersection algorithm. With this in mind, there are two options available: we can distribute the objects in the world in parallel and ask each to determine what rays hit it or we can distribute the rays and ask each to determine what object they hit. The approach of distributing the rays rather than the objects seems to be the superior strategy for a couple of reasons.

We must consider that the information which we ultimately want to have is which object (if any) a given ray has hit. From this data, we perform the various calculations necessary for determining the way the world is illuminated. The basic operation available to us is to take a single ray and a single surface and determine whether the ray hit the surface, and if so, where along the ray the intersection occurred. If we parallelize over a vector of rays, for each ray we will have to look for possible intersections with each object and then find the one which occurs first along the length of the ray. This will quickly give us the information that we require. On the other hand, if we decided to begin parallelization with the objects, each object would have to consider each ray and determine whether that ray hit the surface. This is closely analogous to the computation performed in the other case. However, to proceed from this stage, we must gather the “hit” vectors for each object, collate them so that data for each ray is extracted from every vector in which it occurs, and find the intersection that occurs at the least distance from the ray origin. This task of collating the various “hit” lists makes this approach far more difficult than its counterpart.

Generally speaking, the number of rays exceeds the number of objects in the world. This assumption holds except in cases where the world is highly complex or the system has recursed through several reflected generations and only a few rays are still alive. However, these cases occur far less than the variety of others in which a large number of rays survive and there are comparatively few objects in the world. By distributing the rays rather than the objects, we gain significantly more flexibility. We are more likely to have a task to assign to any available processor and memory consumption will be decreased. Since the collection of all objects in the world is likely to be smaller than the collection of all rays in the world, distributing the collection of objects will consume less memory and entail less inter-processor communication.

In reality, this decision is not of vast significance. Since NESL allows for nested parallelism, we can parallelize over a vector of rays and then each instance of the applied function can subsequently parallelize over a vector of objects. The opposite arrangement could also be implemented. However, by using the rays rather than the objects as the basis for parallelization, we reduce the amount of work necessary (both in development and in computation), and the program is more natural and simpler to understand.

Rendering Process

In broad outline, the process of rendering involves creating the initial generation of rays, tracing them to their intersection points, performing any necessary shading calculations, and then spawning the next generation of rays and recursing. However, because of the amount of memory consumed during rendering, the image is divided into vertical slabs which are rendered sequentially. Presently, I use slabs which are 10 pixels wide. A slab size of 32 (or even higher) works well with simple worlds with few objects and few light sources. The choice of 10 for the slab width accommodates larger worlds at the cost of reducing parallelism. Ideally, the slab width could be computed based on some metric of “world complexity”, but I have not implemented any such mechanism. The decomposition of the image into slabs also allows quicker visual feedback to the user. Rather than having to wait for the whole image to be rendered before seeing any part

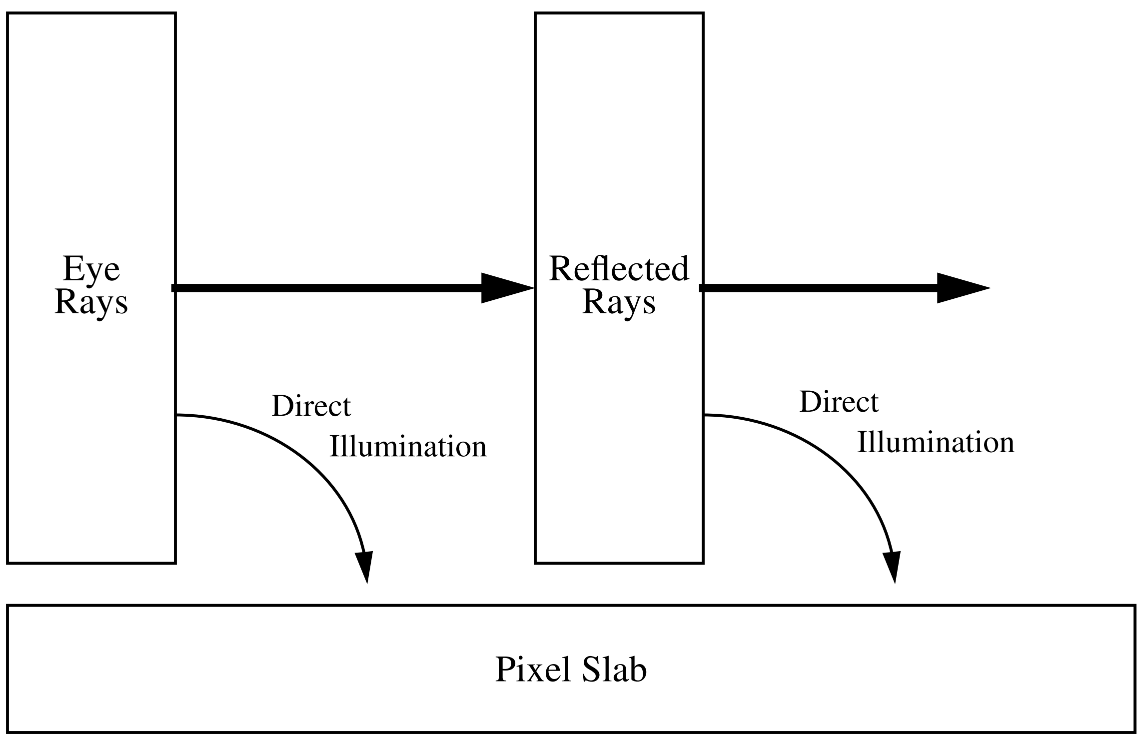


Figure 4: Processing of Successive Ray Generations

of it, the user need only wait for the first slab to be rendered before display can begin. For moderately sized worlds, the rendering process can keep the display process (Vplot) supplied with image data for consecutive slabs with little delay in between. To some extent, this is the result of high levels of traffic with the X server which can be quite time consuming.

The initial step in the rendering of a slab is the generation of a vector of "pixel descriptions". These structures store the illumination which has been accumulated into specific pixels at the current point during rendering. Each ray, in addition to having an origin and a direction, is assigned a weight and a pixel reference. The weight determines what fraction of the light that ray encounters will actually be propagated back to the image; the pixel reference records what image element that particular ray will contribute to. The process is begun by spawning all the initial eye rays through all the pixels in the current slab. Each eye ray holds a reference to the pixel which it was aimed through. In the basic system, each pixel has exactly one eye ray and the weight of that ray is 1.0, since all light arriving at that pixel must arrive through its eye ray. Anti-aliasing is supported by firing multiple rays through a single pixel and assigning each a fractional weight. For instance, one might aim 4 rays through a single pixel and assign each a weight of 0.25. As each generation of rays is processed, any direct illumination is added directly into the pixel slab and the next generation is created. When there are no longer any live rays, the pixel descriptions will hold the total illumination which will arrive at the various pixels in the image slab. The rendering of that slab is therefore completed and the descriptions can be dumped to the Vplot process for display.

The processing of the current generation of rays consists of finding every ray that hits some object and then for each of them:

- Shooting rays at the light sources to determine which are occluded and which are visible
- Adding a reflected ray to the next generation

Rays always inherit their pixel reference directly from their parent; the pixel reference can never change. The ray's weight is computed by the product of the parent's weight and the specular reflectivity coefficient of the surface that its parent struck. Any rays with weights below the threshold will be killed off, as will any rays of a generation beyond the predetermined limit.

Because direct illumination is gradually accumulated into the pixel descriptions during each generation, we do not need to maintain information on any past generations. This fact is very important, since it allows us to save considerable space. Combining this with the knowledge that the size of ray generations will never increase, we can place definite bounds on the amount of space which will be consumed during rendering.

The Ray Tracing System

Previous sections have discussed the general structure of the ray tracer and the basic algorithms which it employs. Here, I will present a detailed description of the code along with the actual source for key routines.

Before proceeding to discuss functions provided in the system, we must know what the fundamental data objects are which the system manipulates. These are detailed below.

Table 1: Data Types

Data Type	Components	Description
point	(float float float)	This is the fundamental object for describing points in space. The fields can be interpreted as (x y z) for real-space coordinates and (r g b) for color-space coordinates..
output-point	(int int int int int)	This is the record used for communicating image data to the Vplot process. The fields are considered to be (x y r g b)
ray	((point) (point) float int)	This describes a ray. Its components are origin, direction, weight, and pixel reference.
light	((point) (point))	Describes a light source with triples for position and color.
sphere	((point) float (point) float float)	Describes a sphere by origin, radius, color, specular, and diffuse reflectivity coefficients.
pixel	(int int (point))	Record used to store pixel descriptions during rendering. Its fields are (x y color).

For purposes of data abstraction, the various components of the system are provided with primitive operations for manipulating the various data types. These are detailed below.

Table 2: Primitive Operations

Function	Description
vector-x, vector-y, vector-z	Functions for accessing the slots of points by name rather than through explicit pattern-matching.
color-red, color-green, color-blue	Functions for accessing the slots of colors.
vector-scale, vector*, vector+, vector-, vector-length, vector-dot*, unit-vector	Functions for mathematical manipulation of vectors.
make-ray, ray-factor, ray-direction, ...	Functions for constructing and accessing rays.

Table 2: Primitive Operations

Function	Description
make-eye-ray	Takes the screen coordinates of a pixel and generates an eye ray passing through it.
sphere-center, sphere-radius, ...	Functions for accessing the slots of a sphere object.

As I have stated earlier, the most common and critical calculation performed during ray tracing is the ray-surface intersection algorithm. The following is the code implementing the ray-sphere intersection algorithm described in Figure 1.

```
;;; sphere-intersect --
;;; This takes a ray and a sphere and determines where along
;;; the length of the ray it hits the sphere. A result of 0.0
;;; means that no intersection occurred.
(defop (sphere-intersect ray sphere)
 (with (((sphere center radius color specular diffuse) sphere)
          ((ray origin dir factor pixel) ray)
          (oc (vector- center origin)))
    ;;
    ;; After many reflections, we start to experience round-off error.
    ;; Because of this we check the difference of |OC|
    ;; and the radius against an error threshold rather then
    ;; comparing them for equality.
    (if (<= (abs (- (vector-length oc) radius)) error-threshold)
        ;;
        ;; This ray lies on the surface of the sphere
        0.0
        (with ((l2oc (vector-dot* oc oc))
                 (tca (vector-dot* oc dir)))
          (if (minusp tca)
              ;;
              ;; This ray points away from the sphere
              0.0
              (with ((r2 (* radius radius))
                       (t2hc (+ (- r2 l2oc) (* tca tca))))
                (if (minusp t2hc)
                    ;;
                    ;; Ray misses the sphere
                    0.0
                    (with ((offset (sqrt t2hc)))
                      (select (< l2oc r2)
                          ;;
```

```
                                ;; Ray origin inside sphere
                                (+ tca offset)
                                ;;
                                ;; Ray origin outside sphere
                                (- tca offset))))))))))
```

The higher level rendering functions are particularly interested in the first object which any given ray intersects with. Therefore, the following function is provided for determining that information.

```
;;; nearest-sphere-intersection --
;;; This takes a ray and a vector of spheres. It determines what
;;; sphere (if any) is the first one that the ray intersects with.
;;; The function ray-hit-p should be used in conjunction with
;;; the result of this function to determine whether a valid
;;; intersection was found.
(defop (nearest-sphere-intersection ray spheres)
  ;;
  ;; First, we intersect the ray with all the spheres.
  ;; Then, we pack out all the ones that didn't hit.
  (with ((places (v.sphere-intersect v.ray spheres))
         (flags (v.not (v.zerop places)))
         (places (pack (zip places flags))))
    (if (zerop (length places))
        ;;
        ;; No spheres were hit
        (pair 0.0 null-sphere)
        ;;
        ;; Some spheres were hit. We need to find the
        ;; first point of intersection.
        (with ((spheres (pack (zip spheres flags)))
               (mindex (min-index places))
               (min-place (elt places mindex))
               (min-sphere (elt spheres mindex)))
          (pair min-place min-sphere)))))
```

These two functions provide the real core of the ray tracing system. The rest of the program is concerned with rendering. In other words, how to take the provided algorithm for tracing rays and use that to generate realistic images. The system actually provides two different options for rendering: quick rendering and full rendering. In quick rendering, initial eye rays are generated and traced but no shading calculations are performed. The color returned is the color of the struck surface. This allows the user to get relatively fast feedback and the general spatial layout of the scene in question. It is meant to be used as a sort of preprocessing stage where the user can verify that the world is roughly arranged in the intended manner. In contrast, full rendering does perform all shading calculations; it is the real implementation of the ray tracer, the one which produces realistic images. The following are the two top-most functions which implement the full rendering system.

```
;;; render-world --
;;; This function takes a command-line string which will be
;;; used to invoke the display process, it spawns the display
;;; process and begins the rendering process.
(defop (render-world cmdline)
    (with (((pair (pair instr (pair outstr errstr))
                  (pair message flag))
              (spawn cmdline nullstr stderr stderr))
             (filep (file_pointer instr "")))
        (if flag
            ;;
            ;; Calling preprocess-world allows us to determine
            ;; which slabs are of interest and which are empty. Using
            ;; it is essentially optional.
            (render-slab 0 (preprocess-world world) filep)
            ;;
            ;; Spawning of the display process failed.
            (with ((result (printl message))) flag))))

;;; render-slab --
;;; This function takes a column number describing the start
;;; of a slab, a structure describing what sections of the image are
;;; "interesting", and a file descriptor to send the output to. It
;;; then renders the current slab and tail recurses to render
;;; the rest. All the real work of rendering is actually done in
;;; sub-render-slab. This is just a wrapper for controlling which
;;; slabs are rendered.
(defop (render-slab start intervals filep)
    (with ((end (+ start slab-width))
             (result (if (is-interesting-slab start end intervals)
                         (with (((pair pixels rays)
                                   (generate-slab start end)))
                             (write_object
                               (sub-render-slab rays pixels 0) filep))
                         ;; Bogus thing for type consistency
                         #v((output-point 0 0 0 0 0)))))

        (if (< end screen-width)
            (render-slab end intervals filep)
            t)))
```

Most of the work of rendering is done by **sub-render-slab**. It is called by **render-slab** to begin processing of a slab, and it recursively calls itself until processing is complete.

```
;;; sub-render-slab --
;;; This function takes the current generation of rays, the pixel
;;; descriptions for the current slab, and the current generation count.
;;; Given this information, it performs all necessary shading calculations.
(defop (sub-render-slab rays pixels level)
    (if (or (> level max-level) (= 0 (length rays)))
        ;;
        ;; We're done, make the output points and return
        (v.make-output-point pixels)
        ;;
        ;; We still have rays to trace
        (with ((hits (v.nearest-sphere-intersection rays v.world))
                  (hit-flags (v.ray-hit-p hits))
                  (rays (pack (zip rays hit-flags)))
                  (hits (pack (zip hits hit-flags)))
                  (hit-points (v.find-ray-point rays (v.first hits)))
                  (hit-spheres (v.second hits))
                  ;;
                  ;; After computing all the hits, we spawn off
                  ;; whatever child rays are necessary.
                  (child-rays
                      (spawn-child-rays rays hit-points hit-spheres)))
            ;;
            ;; Now, we begin processing the next generation by
            ;; making a recursive call.
            (sub-render-slab
                child-rays
                ;;
                ;; Direct illumination is calculated and added here
                (add-illumination
                    pixels
                    (compute-direct-illumination
                        rays hit-points hit-spheres))
                (+ level 1)))))
```

I will not present detailed listings of the rest of the system. The two most important functions beyond the ones given above are **spawn-child-rays** and **compute-direct-illumination**. These functions implement the two halves of the shading calculation: computing specular illumination and direct illumination. They each create new rays; **spawn-child-rays** simply creates a reflection of each ray that hit and **compute-direct-illumination** shoots rays at light sources to determine if they are visible from the various points of intersection.

The functions given implement slightly simplified versions of the ones actually used in my system. For instance, they do not perform anti-aliasing, although that is easily added. However, they do convey the general structure of the program and demonstrate how the tracing of rays is parallelized.

Results

As a demonstration that the ray tracing system can actually generate images, I have included a sample collage of four images. The following is a brief summary of the image characteristics of each:

Table 3: Sample Image Parameters

Description	Number of Spheres	Number of Light Sources	Time to Render on CM-2 (sec)
Single gray sphere	1	2	28.93
Red and gray spheres	2	3	113.65
4 spheres in a square	4	2	142.70
Big sphere behind 3	4	2	83.33

These results demonstrate that ray tracing can produce images in fairly reasonable periods of time. It also becomes clear that the real expense of ray tracing comes from recursively tracing reflections. In the single sphere example, where no reflections will hit, the image is completed rather quickly. The second and third images are slightly more complex but show marked increases in rendering time. In both of these images, there are fairly large reflections which are often reflected in other surfaces as well. The fourth picture, which is of essentially the same complexity as the third is rendered in a shorter time; the three smaller spheres in front are non-reflective and the reflections that are visible are fairly small and not compounded in other surfaces.

Conclusion

We have seen that ray tracing is a powerful technique for rendering. It provides a uniform model for calculating optical effects and it can simulate effects, such as lenses, which are beyond the scope of traditional methods. There are some optical phenomena that it does not model at all, such as diffuse ambient reflection, but recent research seems to indicate that this problem might be solved by combining ray tracing systems with radiosity methods. Unfortunately, ray tracing can also be quite expensive, and a natural reaction is to consider implementing parallel ray tracing systems. I have discussed why ray tracing is particularly amenable to parallelization and the methods for doing so in a portable fashion. As the performance figures for the sample images demonstrate, ray tracing can still be expensive, although it need not be unreasonably so. Since it is potentially more powerful and accurate than methods such as environment-mapping, a certain level of added expense is acceptable.

The system as it stands does not possess the full power that one might reasonably desire. Its greatest short-coming is the lack of support for primitives other than spheres. This lack is largely due to unexpectedly slow development. Although NESL is a useful language, it possesses very little in the way of programming tools. In particular, there are no debugging facilities beyond simple output primitives. I feel that this has impeded progress toward including certain desirable features such as extended primitive support and constructive solid geometry. Supporting multiple primitive types is also made more difficult by the lack of higher-order functions and dynamic type dispatching. The rendering system is also in need of a front end which would allow a user to interactively construct a world rather than describing it directly to the system. However, despite these lacking features, it does produce quality images in reasonably short periods of time, and it meets the goal of implementing portable parallel ray tracing algorithms.

References

Defend, Daniel Evan, Parallel Ray Tracing in vector-multiprocessor environments. Technical Report 677, University of Illinois, 1987.

Delaney, Hubert C., Ray Tracing on a Connection Machine. Technical Report TMC-181, Thinking Machines Corp., 1988.

Foley, James D., Andries van Dam, Steven Feiner, John Hughes, *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, Mass., 1990.

Glassner, Andrew S. ed., *An Introduction to Ray Tracing.* Academic Press, London, 1989.

Green, S. A. and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, v9 p12-27, 1989.