The Holodeck Interactive Ray Cache

Gregory Ward Larson *Maryann Simmons*
Silicon Graphics, Inc. *University of California, Berkeley*

Abstract

We present a new method for rendering complex environments using interactive, progressive, view-independent, parallel ray tracing. A four-dimensional *holodeck* data structure serves as a rendering target and caching mechanism for interactive walk-throughs of non-diffuse environments with full global illumination. Ray sample density varies locally according to need, and on-demand ray computation is supported in a parallel implementation. The holodeck file is stored on disk and cached in memory by a server using an LRU beam-replacement strategy. The holodeck server coordinates separate ray evaluation and display processes, optimizing disk and memory usage. Different display systems are supported by specialized drivers, which handle display rendering, user interaction, and input. The display driver creates an image from ray samples sent by the server, and permits the manipulation of local objects, which are rendered dynamically using approximate lighting computed from holodeck samples. The overall method overcomes many of the conventional limits of interactive rendering in scenes with complex surface geometry and reflectance properties, through an effective combination of ray tracing, caching, and hardware rendering.

Keywords: ray tracing, illumination, image reconstruction, rendering systems, virtual reality, mesh generation.

1. Introduction

A rendering system with full global illumination can be a powerful tool for designers who wish to visualize a proposed complex object or space. The feedback provided by such a simulation aids the designer in making intelligent choices regarding materials, geometry, and lighting. To get a true feel for exactly how a candidate building or car design will look, the user must be able to walk around and view their design from every angle, checking out highlights, shadows, and reflections as they move. (Image 17 shows a space where reflected glare is particularly important.) Existing rendering systems do not provide this functionality. Conventional ray tracing cannot support a mobile viewer at interactive rates, because its view-dependent calculation must start over at each new eye point. A radiosity solution allows free movement, but only in diffuse environments. For accurate visualizations of realistic environments, what is needed is a system that is both interactive and progressive, exploiting all of the rendering and computing power available to provide the fastest feedback possible to the designer.

Realistic, interactive rendering is one of the long-standing goals in computer graphics. Advances in specialized and general-purpose hardware have brought us to where we can visualize complex environments in real-time; however, shading must either be precomputed or approximated using local lighting models. Even when global illumination is precomputed over surfaces, a diffuse reflection assumption is usually necessary to apply the results in a walk-through or fly-through scenario [Chen88, Sillion94]. Other techniques offer approximate solutions to global illumination for specularly reflecting and refracting surfaces in simple environments at interactive rates [Diefenbach96, Ofek98]. Walter et al [Walter97] used Phong shading and object-specific local light sources to approximate the appearance of a non-diffuse global illumination solution using hardware rendering. Although the results were not exact, especially with regards to shadows, they produced some very effective simulations of uncluttered environments.

As geometry grows more complex and many surfaces are occluded, the naive approach of rendering the entire scene into a depth-buffer becomes too slow. Many approaches to improve depth buffer rendering performance have emerged over the years, including cell-based visibility precomputation, hierarchical depth sorting, level of detail modeling, and billboarding. Cell-based visibility algorithms work best when an environment can be neatly partitioned into large cells connected by small portals [Airey90, Teller91]. Hierarchical depth buffers [Greene93] and depth sorting [Greene96] work better in open environments, but offer little speed-up in scenes with few large occluders, such as forests. Level of detail modeling addresses this problem, but automatic algorithms for simplifying arbitrary three-dimensional objects are a challenge [Luebke97]. Billboarding takes advantage of texture-mapping hardware by replacing complex objects with stand-up impostors, but these must be recomputed frequently to avoid artifacts [Schaufler98, Shade96, Sillion97].

When geometric complexity is combined with complex lighting and shading, interactive rendering no longer seems possible. Therefore, some have thought it best to forego geometry and sample the light field directly. The image-based approach to interactive rendering has been explored by several researchers in recent years [Ashdown93, Chen95, Levoy96, McMillan95], including those who have used simplified geometry in the process [Debevec96, Gortler96, Miller98, Nimeroff96, Pighin97]. Lischinski developed a method for interactive rendering that uses two ray-traced light fields for diffuse and non-diffuse contributions [Lischinski98], and Bala developed a method for ray-traced walk-throughs with guaranteed error bounds [Bala99]. With the exception of Pighin's and Bala's work, none of these methods make use of an on-line ray tracer or other sample generator to refine the displayed image. Pighin's method does not reuse ray samples, and Bala's technique does not display an image until complete, leading to potentially long frame times and slow interaction.

In this paper, we present a new approach, which combines a holographic scene representation with a parallel, interactive ray calculation. This method most resembles the recent work by [Walter99], where samples are cached and displayed interactively in a continuous update cycle. However, where Walter et al keep no permanent record of computed rays, we assume the ray computation is costly enough that persistent storage is worthwhile. In our approach, rays are computed, cached, and eventually stored to disk using a *holodeck* data structure -- a spatial grid used to sort rays without regard to sampling density. These rays are reused for subsequent views, and additional rays may also be generated interactively. Each ray intersection distance is recorded along with the floating point color to enhance display processing. This requires a total of 10 bytes per sample in our implementation. Rays are clustered together into beams for efficient disk access, so no intermediate processing or compression is required between computing samples and using them. Typical holodeck files range from 50 Mbytes to 1 Gbyte, depending on resolution and the number of sections. Although large, these data structures may be kept on CD-ROM or other mass storage media for rapid access and rerendering, and do not need to be kept in memory.

We start by describing our method, including the holodeck representation, the three-process program design, and basic display representations. This is followed by an exposition of our results, where we give example scenes, views, and timings. Finally, we conclude with some discussion of the technique, and a few ideas for the future.

2. Method

To assure optimal reuse of ray computations, we need a data structure that allows us to rapidly store and retrieve ray samples in less time than it would take to recompute them. We begin with the observation that, although each ray has an origin point corresponding to the eye, its computed radiance is valid anywhere along its length, and may be valid behind the origin as well, so long as there are no obstructions[1]. Since our goal is to move about in a virtual environment, and motion happens most naturally in unobstructed regions, we decided to combine the notion of a hologram with an unobstructed region of free movement, which we call a *holodeck section.* Rays will pass freely through such regions, and their entry and exit points will be recorded along with their computed values. Any view point within a region will access the rays that pass near it; thus rays will be reused along their length to the greatest extent possible. This is very similar to the *light field* and *lumigraph* constructs presented in [Levoy96] and [Gortler96], except that there is no "development" step needed between computation and display -- rays are stored and retrieved interactively.

Since only a portion of the environment is viewed at any one time, we can greatly decrease our memory requirements by keeping beam samples on disk when they are not being used and bringing

[1] The physical unit of radiance is the quantity of light passing through a point in a given direction, which is expressed in watts/steradian/meter^2 in Standard International (SI) units. Radiance is constant along an unobstructed ray, which implies that there is no participating medium. Although there are ways to overcome this limitation, we will not explore them in this paper.

them back into memory as needed. Although we could leave this task to the operating system, we found that the common algorithms for virtual memory management were too expensive and inefficient for our application. We therefore created a holodeck server process that manages one or more holodeck sections, keeping the most recently used ray samples resident in a finite memory cache.

To compute ray samples, we use the *Radiance* **rtrace** program, which is freely available and does a good job computing global illumination in complicated environments [Ward94]. This program also lends itself well to parallel processing on multiprocessor and networked systems, which is important for achieving good interactivity. Although we chose to use *Radiance*, we could have picked any program that computes ray sample values. *Radiance* evaluates specific view rays, but even a pure Monte Carlo method, which generates random rays in an environment, could be used to fill a holodeck. The end result captures the full light field, unlike density estimation methods, which usually throw away directional information [Shirley95].

For each ray we trace, we store the computed radiance and the distance from the entry point to the surface intersection so that we can reproject sample points onto our displayed image. This parallax correction minimizes image blurring, which would otherwise be caused by rays not passing exactly through our current view point. There will still be some problems computing occlusion, but we can address these with some clever drawing techniques.

Overall, our system normally consists of three logical processes: a holodeck server, a sample generator, and a display process. This arrangement is diagrammed in Figure 1. The holodeck server controls access to the holodeck file, and the display process controls access to the display, keyboard, and mouse. One or more **rtrace** processes perform the actual ray tracing, and interprocess communication flows through TCP/IP sockets. The holodeck server may also be run without a sample generator if a display-only function is desired, or without the display process for filling in holodeck samples as a background calculation.

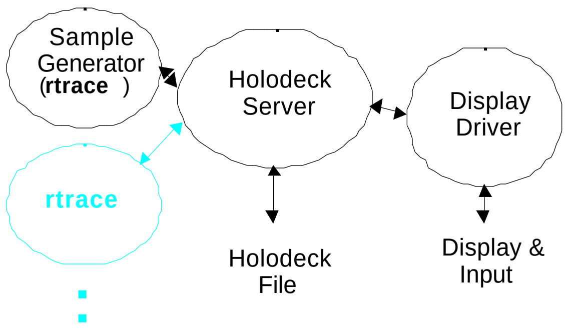

Figure 1. Schematic diagram of holodeck rendering system. Arrows show the flow of information.

In this section, we first describe the holodeck data structure and how it is set up in a scene. We then discuss the server process and how it handles VM management and different calculation modes. Third, we discuss the sample generator and describe its parallel processing and synchronization methods. Fourth, we describe the display process and detail three variations that utilize different display representations. Finally, we discuss coordination between the three logical processes.

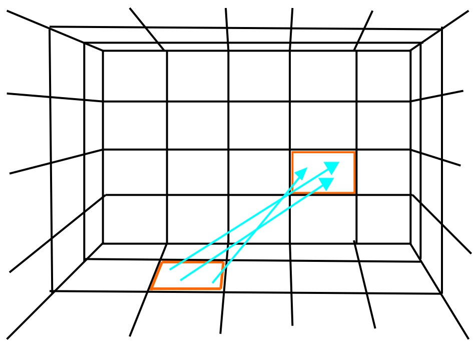

Figure 2. A holodeck section as seen from inside. A ray passing from one grid cell to another is stored together with other rays in the same beam. (A beam of three rays is shown.)

2.1 The Holodeck Data Structure

The holodeck data structure stores information for all view rays that have been computed for a particular scene. In its basic form, a holodeck section is simply a gridded box, like the one shown in Figure 2. Rays passing through a section will pass through two cells on two walls. Rays that pass through the same pair of cells in the same direction are collected into an indexed *beam*. All rays for a particular beam are stored and accessed together on disk, and a section directory records each beam's location and size. A holodeck file may contain multiple sections, which represent different regions of free movement in the scene[2]. These section boundaries and grid resolutions are set up by the user based on where they want to go and what they want to see.

The total number of beams for an W×D×H gridded holodeck section is:

$$N = 2W^2D^2 + 2W^2H^2 + 2D^2H^2 + 8W^2DH + 8WD^2H + 8WDH^2$$

If the grid is too fine, the section directory becomes large and unwieldy, taking too much room in memory and too long to update on disk. If the grid is too coarse, ray bundles will not resolve visibility very well. Thus, it is important to choose the grid dimensions wisely. We found grid sizes between 4 and 24 on a side to work best, with a target N of 50,000 to 500,000. Note that the sample density is *not* related to the number of beams in a section. Therefore, the holodeck places no limit on final image resolution. The grid resolution only affects cache coherency and the initial sample density.

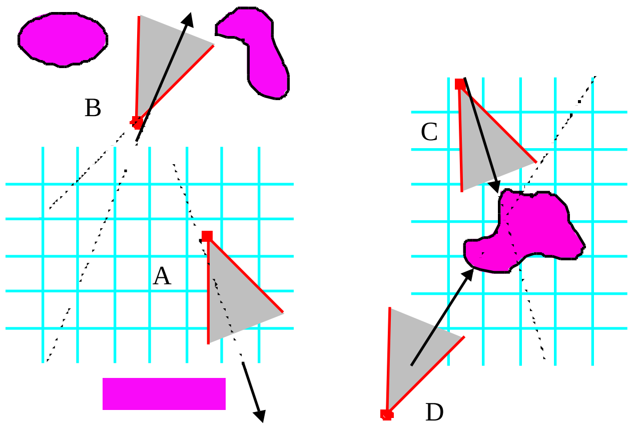

Figure 3. An interior holodeck section is shown in plan view (a) on the left, and an exterior section (b) is shown on the right, with one example ray shown for each view.

[2] In an alternative interpretation, a section may enclose complicated objects for viewing from the *outside*.

Each section has a view class, *interior* or *exterior,* which dictates where view rays originate and determines what geometry is visible. For an interior section we wish to look out of, the rays originate at the exit wall. Geometry outside such a section will be freely visible from anywhere inside. It is also possible to generate views from outside an interior section (or inside an exterior one), and this is often done with no ill effects. Figure 3a shows two possible views of an interior holodeck section. View A shows a typical perspective, taken from the inside looking outward. View B, however, is outside looking away from the section. So long as there is no intervening geometry between the outer wall and the eye point to block the rays, the beam samples taken from the holodeck will work fine.

For an exterior section we wish to look into, the rays originate at the entry wall. Geometry inside such a section will be freely visible from anywhere outside. Figure 3b shows two possible views of an exterior section. View C is positioned inside the section, which is acceptable in this case because there is no geometry behind it. View D is the only view shown that will be missing samples, since part of its frustum does not intersect the section. If the viewer needs to see objects outside the section as well, an additional section could be added nearby to cover these rays.

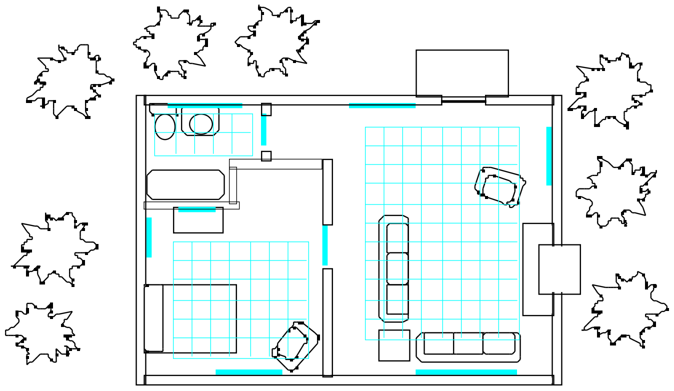

Figure 4. A cabin floor plan showing section grid placement and portals. (Portals are discussed in Section 3.1.)

A holodeck consists of one or more sections as described, which are associated only by their locations in three-space. For example, multiple interior sections might define regions of free movement in each room of a house, as shown in Figure 4. The bottom of each section begins some distance above the floor, so as to capture the geometry that is placed there. By bringing the section walls in from the room geometry, we avoid object clipping problems with minimal restriction of movement. If an object is unavoidably in our view space, we may either redefine our sections to accommodate it, or render it directly as a local object using the method described later in Section 2.4.6.

We find this parameterization to be more natural in a walk-through setting than light slabs [Levoy96, Gortler96], which require six slabs to cover all views in a space,[3] or spheres [Camahort98], which are awkward to place in most environments. Because we adjust the sampling density on a per-beam basis in our method, we are also less concerned about parameter uniformity.

If each holodeck beam contained the same number of rays, the angular sample density would be highest in the section center and looking lengthwise down long sections (e.g., a building corridor), which corresponds well to the natural viewing tendency during walk-throughs. However, the number of rays for each beam need not be uniform, and the sampling density can, and usually does, vary substantially over different parts of the holodeck. In an interactive calculation, this density is usually driven by view history, but may be assigned in any way desired during precalculation. We often use a beam's volume as a base measure of its importance, and assign target ray densities proportionally, building up samples gradually across all beams so that the density ratios are close to our targets no matter when the precalculation is terminated.

In the holodeck cache, each view ray sample is encoded into ten bytes. This encoding is detailed in Table 1. The ray color is stored in the four-byte, RGBE floating-point format native to *Radiance*

[3] Assuming unidirectional rectangular slabs.

[Ward91]. This format covers a wide dynamic range, which enables us to compute an appropriate tone mapping at display time [Larson97]. We also record the ray entry and exit points for our section grid cells, so we can compute the exact origin and direction of each sample Since each grid cell already has a fairly specific location in the world, one byte per degree of freedom is enough to get a very accurate ray specification. Together with the ray distance (as measured from the entry cell), we can derive the surface intersection point to reproject samples for any view.

Encoded value	**Size**
floating point color	4 bytes
position in starting cell	2 bytes
position in ending cell	2 bytes
ray distance	2 bytes

Table 1. Ray sample encoding requirements.

To save space, the ray distance is stored as a 16-bit unsigned integer, which encodes a linear value over a lower range (0 to 2^{11}-1) and a logarithmic value over an upper range (2^{11} to 2^{16}-1). The exact encoding depends on the holodeck section size, whose diagonal length determines the border between the lower and upper ranges. The step size for the logarithmic range is taken to match the linear step at the border, which is $(1 + 2^{-11})$. This gives a maximum encoded distance on the order of the section diagonal times 10^{13}, with an accuracy of 0.05%.

A holodeck file consists of a global information header, followed by a file offset for the next section directory, followed by the first directory. The last section in the file is preceded by a zero offset pointer. A section directory consists of the world coordinates for the section, grid dimensions, and a file offset and sample count for each beam. After the first section, directories may appear anywhere in the file, and beam data indexed by the directories may be interspersed at random. If a holodeck file contains only one section, the first offset will be zero, and all beam data will follow the section directory. Figure 5 shows a holodeck file layout with three sections. For clarity, only a few beam pointers are shown.

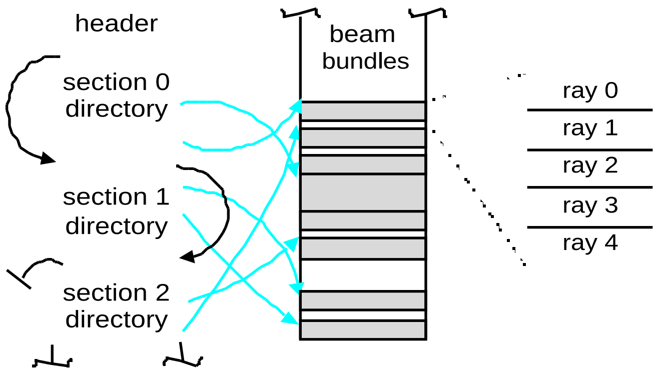

Figure 5. Holodeck file organization, showing section directories, which contain beam offsets and sizes. Each beam consists of a set of contiguous ray records.

2.2 The Holodeck Server

The holodeck server is responsible for maintaining holodeck file consistency and keeping a cache of the most recently accessed beams in memory. The server also turns out to be the most convenient place to manage the ray calculation and meet the demands of the display process, which is why it is in the center of our system diagram (Figure 1). Most of the time, the server does not require much of the CPU; it merely mediates display bundle requests and keeps the ray tracing processes busy. There may be significant time spent waiting for disk seeks and reads however, which is why we have to be clever about how samples are loaded and cached.

As we discussed earlier, holodeck beams are indexed based on grid cell pairs for each section, and a directory marks the file location and number of rays for each beam. A copy of this directory is kept resident in memory, and is updated when new ray samples are written to the file. Initially, the directory is empty. As ray samples are computed, beams are allocated from the cache. In our implementation, between 1 and 21 rays are added to a beam at a time, depending on how quickly rays are being computed.[4] Once the cache becomes full, beams are written to the holodeck file to free up memory, in least-recently-used (LRU) order.

The cache size is set appropriately for the host system, and is usually between 8 and 24 Mbytes. When this limit is reached, beams are purged from memory and written to disk as necessary. Deallocated disk slots are tracked in a fragment list, which is coalesced and reused for other beams as the calculation progresses. Nevertheless, the file may become fragmented, and beams that are adjacent in the space of a section may end up nowhere near each other on disk. Therefore, we run an optimizer after long precalculations to collate beams and eliminate fragments.

There are three basic server operation modes. In batch mode, rays are calculated and stored to the holodeck file without display. The beam computation order and density are determined by the world-volume of each beam, since beams that cross greater distances and enclose greater volumes of space are proportionately more likely to contain a randomly placed view point. In display-only mode, sample values are read from the holodeck file for display, but no ray calculation takes place. In interactive mode, ray calculation and file access are driven by the display process, which tells the server which beams it wants at any given moment.

In all three modes, the server works from a list of requested beams, taken either from the display process or derived from beam volumes in batch mode. List entries specify the section, index, and desired number of samples for each beam. The list is sorted in order of increasing computed/desired sample counts. Since error is proportional to $M^{-0.5}$ for M Monte Carlo samples, this corresponds to a *decreasing* computed/desired error ratio. On each iteration, one or more beam requests are removed from the head of the list, and a number of new samples is assigned to the sample generator (assuming there is one). The number of rays assigned depends on the average time required to compute each ray and the number of beams in each queue, so that each ray queue can be emptied within about five seconds. This was deemed important for system responsiveness -- when a user moves to a new view, they should not have to wait more than a few seconds for the computational focus to catch up. While the sample generator is catching up, the server sends the display process relevant rays from the holodeck cache.

In batch mode, the holodeck is gradually filled at a density always proportional to beam volume. Thus, there is no minimum time the calculation needs before useful information is put into the holodeck. The server may be killed at any time, and restarted in interactive mode without data loss or compromise. This differs from most rendering computations, which must proceed until they are done.

2.3 The Sample Generator

The holodeck sample generator evaluates view rays requested by the holodeck server. These rays are computed in bundles of samples corresponding to a particular beam, though the beams themselves will be in no predictable order. Ray origins and directions are sampled within each beam by choosing 4 random variables 0-255 corresponding to the subgrid coordinates. This algorithm is modified when we have an active display driver to restrict rays to pass within some specified distance of the view origin. This improves the image convergence rate without unduly burdening the holodeck server, which must derive sample locations very quickly in order to feed multiple ray evaluation processes running in parallel. (See Appendix A for a description of this modified sampling algorithm.) An adaptive sampling technique could also be applied [Walter99], though this tends to bias the result [Kirk91]. The lack of a nice 2-D image plane to work with also makes adaptive sampling trickier, since none of the more sophisticated interpolation techniques are applicable [Guo98].

[4] 21 is the number of rays that fit into a 512-byte packet, which guarantees communication flow.

Rays are evaluated by creating a two-way connection to the *Radiance* **rtrace** program, which takes ray origins and directions on its standard input and sends evaluated colors and distances to its standard output. In our implementation, multiple **rtrace** processes may be invoked on a local multiprocessor machine, with a separate duplex connection to each process.

Multiple **rtrace** processes share memory and data using a system-independent, coarse-grained technique. Static data, such as the global scene geometry and materials, are shared by parallel processes running on the same UNIX host. The first process invocation loads and initializes all scene data, then makes *fork* system calls to create an appropriate number of child processes. Since child processes created in this way share memory on a copy-on-write basis, all memory pages created before the first call will be shared so long as they are not altered. In most cases, shared scene data comprises more than 90% of the total memory requirements, meaning each additional **rtrace** invocation adds less than 10% to the single-process usage.

Of the data created during **rtrace** execution, only the indirect irradiance values cached as part of the diffuse interreflection calculation [Ward88] must be shared to maintain linear speedup (i.e., avoid redundancy in the ray calculation). This is accomplished through a semaphore-locked *ambient file* that holds all indirect irradiance values computed by **rtrace** across multiple sequential and parallel invocations. Each process flushes its newly computed values to this file periodically, using the following routine:

```
ambsync() begin
    obtain write lock on ambient file
    if file has grown since last write then
        load values added since last write
    end if
    write new values to file
    record file size for next check
    unlock file
end ambsync
```

So long as this routine is not called so frequently that it creates contention for the file lock, it will not adversely affect the performance of parallel execution. We have tested the use of shared memory and the common indirect irradiance file with as many as 24 invocations and have not found contention to be a problem. The global illumination and parallel computation algorithms employed in *Radiance* are described further in [Ward94] and [Larson98].

For each **rtrace** process, the holodeck server tracks the queue of beam packets submitted for processing. This queue has a maximum length, determined by the system's pipe buffer size. Typically, about 400 rays may be queued at one time without risking deadlock[5]. To compute maximum queue length, we divide this number of rays by maximum packet size, which is 21 in our implementation (the number of ray specifications that fit into 512 bytes). The actual size of each packet may be adjusted downward to maintain interactivity, as described in the previous section. Buffered I/O is flushed automatically when the maximum packet size has been reached, or manually by sending **rtrace** a zero direction vector. (Coordinated flushing is also necessary to avoid deadlock.)

The server's interface to the ray calculation is very simple. A single routine is given a list of packets to queue up, and it returns a list of packets that finished. The total number of packets available for queuing is determined by the pipe buffer size, the maximum packet size, and the number of processes. We write packets to the shortest queues first, and after the last packet is queued, we call the UNIX *select* function to wait for the first packet to be finished by any of our **rtrace** processes. In many cases, we will get back several packets, possibly from more than one process, which are all put in the returned list.

[5] We cannot allow the server process to block when it submits a new packet for processing, because it would be unavailable to read the **rtrace** output, which would be the only way to release the block.

Through the input parameter VDISTANCE, the user can control not what a ray sees, but how **rtrace** evaluates distance.

If VDISTANCE is set to False, then **rtrace** computes the distance to the first object that is intersected. If VDISTANCE is set to True, then **rtrace** computes the *virtual distance* for each ray. In the case of diffuse and curved surfaces, this is the same as the first intersection distance. However, when there is a flat, specular surface, such as a mirror or a pane of glass, then **rtrace** returns the distance to the object reflected in or visible through the specular surface. When this intersection point is later reprojected for display, it will give a sharper image than the first intersection, especially if the section grid is coarse and the program has little time to converge. The disadvantage of using virtual distance is that edges of specular objects may break up, and some reprojections may not be exact, especially if the specular object has a lot of refraction. (We show some effects of this in Image 12 in the Results section.)

Due to the simplicity of our queuing model and the nominal demands we place on our sample generator, it is straightforward to adapt this system to different computation environments. We could substitute another ray tracing system for *Radiance*, or use a distributed network of machines to perform our calculations rather than a multiprocessor host. Alternatively, we could employ a massively parallel computer, and communicate over a single network connection.

2.4 The Display Process

The display process is the most important component of our system, because it is responsible for what the user sees and how the user directs the simulation. Our overall goal is to provide an interactive walk-through of a realistic virtual environment. For its part, the display process must do the following:

1. Accept user input and view manipulation
2. Tell the holodeck server which beams to compute
3. Create a reasonable image from returned beam samples

Of these three tasks, only the second one is unaffected by the choice of graphics hardware. User input and view manipulation vary with the input devices available and the interaction model; a head-mounted display is different from a CAVE, which is different from a monitor with a spaceball or a mouse. Likewise, the visual representation will change from one output device to the next, especially if a stereoscopic display is available. One of the advantages of our system design is the great flexibility it offers in selecting the ray calculation and display methods.

This section describes the implementation of these three tasks on two common graphics configurations: a color X11 display and an OpenGL platform, both with a standard mouse and keyboard. The input and view manipulation for these two drivers is identical, so we first discuss the common input model used for the first two tasks. We then discuss three alternate image representations we have implemented for the third task.

2.4.1 Input Model

The mouse is used to direct view movement, and the keyboard is used to enter single-letter commands in the display window. The process starts with a default view in the center of the first holodeck section (or outside for an exterior section). From there, the user usually rotates the view and starts heading in some direction. In forward motion, the view advances 10% closer per frame to whatever object is under the mouse cursor as long as the button is held down. The view direction is held constant, and the view center is adjusted so whatever started out under the cursor will stay there as the view moves. This is extremely helpful in minimizing wild, unintentional view motions as the reference object under the cursor changes from frame to frame. Similarly, backing away from or orbiting an object keeps the point under the cursor fixed. View rotation, which keeps the view origin where it is, does not require visible geometry.

Even if no geometry is visible, the display driver will draw each of the holodeck section grids during view motion to keep the user oriented. Often, only part of a new view will be drawn, since the driver does not request new rays from the server until motion has stopped. A cache of ray values is kept in the driver's memory to reduce latency and allow movement outside the current view. This cache is discussed further in the subsection on Image Representation.

Two user commands are provided to facilitate interactive scene changes. A command is provided to kill the **rtrace** process(es) and another to restart it after some change to the scene description. The corresponding changes in the image dissolve-fade into view as new ray values are entered into the holodeck data structure. A third command is provided to clear the holodeck contents. This is needed for substantial scene changes in which ray values change radically.

2.4.2 Beam Selection and Convergence

To retrieve the appropriate samples from the holodeck, the driver must first relate the user's desired view to a collection of beams to request from the server. To accomplish this, we sample jittered view rays at low density and intersect them with the holodeck section grid or grids the viewer is closest to. A holodeck intersection test tells us which beam each view sample belongs to, and we accumulate beam counts as we go. From this information, we compute target beam densities corresponding to our view pixels, which allow the server to assign calculation priorities such that our display is refined uniformly over time.

The driver must also handle image convergence as the number of samples begins to approach the number of pixels while the viewer is stationary. If the driver gets a ray sample that maps to the same (or nearly the same) pixel as a previous sample, it checks to see which sample ray passed nearest the view origin. The ray that passed nearer to the origin will be more accurate in terms of specular component, and is therefore preferable. As more and more ray samples come in, the driver begins to pick and choose among them to derive the highest quality image. This optimization is performed in parallel with the local sampling procedure described in Appendix A. Because the locally sampled rays are chosen to pass close to the view origin, this method tends to cull out ray samples generated in earlier views and batch runs.

2.4.3 Image Representation and Display

As the requested samples come in from the server, the driver must quickly assemble them into a coherent image. During progressive refinement, the driver receives on the order of a thousand samples per second. Immediately following a view change, however, the server will grab whatever beams it has in memory and on file, and send them to the display driver at interprocess transfer rates. The faster the driver can deal with these new samples, the quicker it will be able to update its display and the better interaction it will provide for the user. Since the server informs the driver when it goes into this "immediate transfer mode," we can improve performance by batching sample additions between display updates.

The most important determinant of rendering quality is the representation chosen to reconstruct the image for display. This is a special case of the classic two-dimensional image resampling problem. In our case, the samples are randomly distributed at a density typically much lower than the displayed pixel resolution. We must quickly come up with a coherent display from a sparse set of samples or we lose the interactive nature of our calculation. This is where we can take advantage of available graphics acceleration hardware.

In this section, we will look at three display driver implementations, each using a different image representation. We first introduce a simple X11 driver that does not require any 3D graphics acceleration hardware. We next describe two drivers designed to exploit available rendering hardware using OpenGL. The first driver renders a subset of the scene geometry using OpenGL during view motion, and reads back the depth buffer to resolve object silhouettes once motion has stopped. The second OpenGL driver builds a triangle mesh from beam samples, displaying Gouraud-shaded triangles both during motion and afterwards during progressive refinement.

Our X11 driver builds a quadtree on the image plane from the beam samples sent by the holodeck server. Each leaf of the quadtree contains at most one sample. To render the image, we fill the rectangle corresponding to each leaf with the color of that leaf's sample if it has one, or an average of its three siblings if it does not. An empty leaf will always have at least one sibling that either has a sample or has children with samples, which are themselves averaged if necessary. The result of applying this simple drawing algorithm on a sparsely sampled conference room is shown in Image 1.

The quality of the rendering in Image 1 is limited by two factors. First, since we draw each rectangle in a constant color, the image appears blocky. Second, we see a side effect of reprojecting each ray sample, based on its calculated world intersection point, to the expected position in this view. This parallax correction step avoids the unwanted depth of field associated with light field rendering, but brings with it the potential for multidepth sampling errors near object silhouettes, as illustrated in Figure 6. Since our sample rays do not pass exactly through the eye point, we may get a sample that is actually behind a foreground object when we reproject its location. Because our quadtree does not know how to resolve 3-D shapes, this results in image artifacts, appearing as chunky or feathered object boundaries as can be seen at the top of the chair back in Image 1.

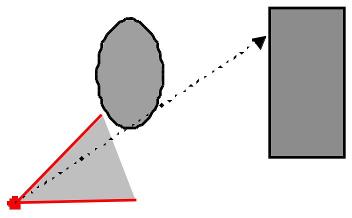

Figure 6. Multidepth values are occasionally returned by the server because beam samples do not pass exactly through the eye point. The dotted line shows that the intersected world coordinate should be occluded from this position.

Another major problem with a 2-D representation is handling motion, because all of the samples must be reprojected to each view in transit, slowing the animation. The following sections describe two drivers that overcome some of these problems through the appropriate use of 3-D graphics hardware.

2.4.4 Voronoi Representation

When 3-D rendering hardware is available, we can use it to improve both view motion and progressive rendering. During view motion, we can use the hardware to draw some or all of the scene geometry using a local lighting model, then overwrite the final view with properly shaded Voronoi regions using ray samples sent by the holodeck server. Image 2 shows our conference room rendered in OpenGL using a single light source positioned at the eye point. This is what our driver displays during view motion. Once motion has stopped, the driver reads back the depth buffer and uses it to cull incoming samples from the server. If a reprojected ray sample has a depth value more than 2% different from that computed by OpenGL, it is discarded as a multidepth sample.[6] The depth buffer is further used to constrain Voronoi regions to improve the appearance of silhouette edges.

[6] The algorithm is insensitive to the depth epsilon, so long as the OpenGL rendering meets this accuracy.

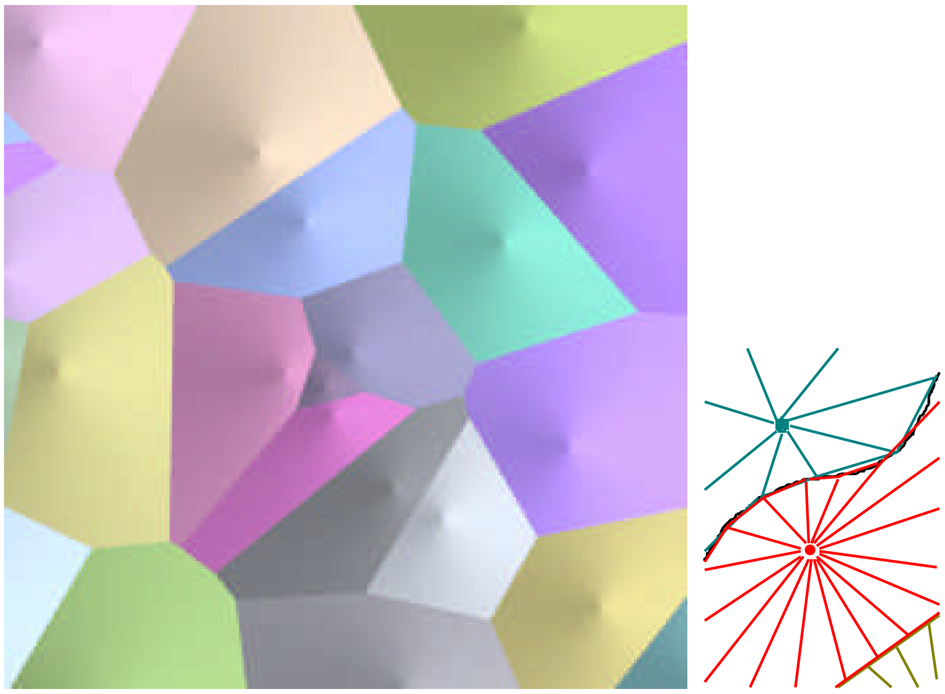

Figure 7. Drawing cones as seen from above results in a Voronoi region around each sample, as illustrated on the left. The diagram at right shows how we constrain our approximated cones along a depth discontinuity.

Our Voronoi representation thus displays a piecewise constant image using OpenGL rendering hardware. A Voronoi cell contains all points that are closer to a given sample than they are to any other sample. A simple way to create a Voronoi diagram in 2-D using 3-D graphics hardware is to draw cones, where each sample defines the apex and color of a cone, which is rendered in an orthogonal, depth-buffered mode as illustrated in Figure 7a [Haeberli90]. This method has the further advantage that it is progressive, so we can add samples continuously without having to do any special processing or ordering. To accelerate cone rendering, we sort our samples into a low-resolution grid over the display image so that we know the approximate local sample density. The density is then used to constrain the maximum radius and number of vertices for triangle fans that approximate different sized cones. Our maximum cone uses 32 triangles and covers about 10% of the screen, and our smallest uses 4 triangles and covers 0.01% of the screen.

This is our basic algorithm for drawing Voronoi regions, but as we mentioned, these regions are constrained further using discontinuities in the depth buffer retrieved from the hardware. We trace each edge of our triangle fan out from the sample position, stopping short of the maximum radius if we reach a place where the depth jumps by 2% or more, as shown in Figure 7b. In this way, we preserve silhouette boundaries, which are critical to human shape perception. A low-resolution prepass on the depth buffer allows us to determine where such tests are needed. Color discontinuities could similarly be resolved by reading back the OpenGL color buffer.

Image 3 shows our conference room samples rendered using the Voronoi representation. The edges are better defined and the samples are less blocky than with the quadtree approach. The time required to draw each sample is slightly longer, depending on the graphics hardware available, but it is still quite short compared to the time to compute a ray. When the cones are large early on in the sampling process, the driver takes a little longer due to pixel fill, but it quickly accelerates as sample density goes up and cone size goes down.

A potential problem with rendering the scene geometry during motion is that it may be too complex to allow for good interaction. We can avoid this problem by specifying a small, relevant subset of the geometry to load for each section, thereby avoiding long redraw times. However, careless application of this solution will undermine multidepth sample culling, since the depth values will be wrong where the geometry is missing. We therefore create a small set of invisible “portals” beyond which we do not draw the geometry. After view motion, we render the portal geometry into the back buffer and use it to determine where to clear our depth values. Using this method, portions of the scene behind portals will be drawn, but without multidepth culling, which results in feathered edges as seen in the background of Image 4. We may also use portal geometry to turn off depth checking on planar specular surfaces, allowing us to use virtual ray distances to reproject samples more accurately. Portals placed over the windows, mirrors and doorways of the cabin scene are shown in Figure 4 as thick, gray lines.

The strength of this display driver lies in its ability to fully exploit the available resources to produce a coherent image from the sample data, both during motion and during refinement. The cone drawing approach provides a simple, fast, and progressive means of interpolating radiance values. This driver makes efficient use of OpenGL hardware to place ray samples accurately on local geometry and provide real-time feedback during view motion, yielding clean object silhouettes and good interactivity. However, the driver still uses piecewise constant shading, and since we switch representations for view motion, we suffer an abrupt change in lighting when we start to move. Finally, since we need geometric information to cull multidepth samples, we only get clean silhouettes when this additional information is available.

Our second driver makes a different set of trade-offs by creating a triangle mesh directly from our ray samples and rendering it in hardware using Gouraud shading.

2.4.5 Triangle Mesh Representation

In this version of the display driver, the rendering representation is a dynamic, 3-D triangle mesh whose vertices correspond to the world space sample points. Image 5 shows the conference room samples rendered with a triangle mesh. The advantage of such a representation is threefold. Firstly, triangles are rendered and Gouraud-shaded by the graphics hardware, providing barycentric interpolation of radiance between spatially adjacent samples. Secondly, since the mesh explicitly contains the 3-D information for each sample, and not just a representation of the projections for a particular view, the rendering representation can be re-used for subsequent frames. Finally, this representation can be utilized in environments where scene geometry is not readily available, because it is derived entirely from the sample data.

In the Voronoi representation, the boundaries of the cone regions in the image plane form the Voronoi diagram of the sample point set. In this representation, we utilize the straight-line dual of the Voronoi diagram: the Delaunay triangulation. Given a single view, we can construct a two-dimensional Delaunay triangulation of the projected samples on the image plane [Darsa96, Pighin97, Shirley95]. While this basic approach provides a better interpolation of the radiance values, the samples must be reprojected and the mesh reconstructed each frame to provide a coherent image during viewer motion. If we utilize the projected points to determine the mesh topology in two dimensions, but retain the original 3-D information in the vertices, the result is a 2.5-D mesh representation, which can be rendered from alternate viewpoints [Sillion97, Darsa97]. This allows only limited view motion; artifacts will appear as soon as the viewer moves enough to reveal new areas in the scene. Image 6 illustrates the 2.5-D nature of the resulting mesh.

From a fixed vantage point there is a one-to-one mapping between visible world space points and their projection onto a sphere centered at that viewpoint. In the mesh representation, we further exploit this 2.5-D nature of the data: a Delaunay triangulation constructed on the sphere provides the mesh topology, with the vertices coming from the 3D sample coordinates. We maintain a Delaunay condition on the mesh to improve image quality and robustness of the representation. In addition to providing a reasonable interpolation, such a triangulation has the property of maximizing the minimum angle and therefore minimizes rendering artifacts caused by long, thin triangles. Such triangles can also prove problematic during the computation and manipulation of the mesh, as they are prone to producing round-off error and inconsistencies in the calculations.

Given the initial view, a unit sphere is centered at the eye point. This initial point is called the "canonical view point" and may or may not coincide with the current eye location as the simulation progresses. A base icosahedral mesh is created that covers the surface of the sphere. Given a new sample, we perform point location to find the existing spherical mesh triangle that its projection intersects. If the projection of the new sample falls within some epsilon distance of an existing sample, the sample whose direction passes closest to the current view direction is maintained to improve convergence. We also perform a heuristic test based on the relative depth of the nearest neighbors to reject possible multidepth samples that could produce visibility errors if introduced into the mesh. If accepted, the sample point is inserted into the triangle, creating three new triangles if the new sample falls interior to an existing triangle, or four if it falls on an edge. The Delaunay condition is tested, and reasserted if necessary. The algorithms we have developed are designed to be as efficient as possible while still maintaining a good quality representation that is also robust. We discuss the relevant steps below.

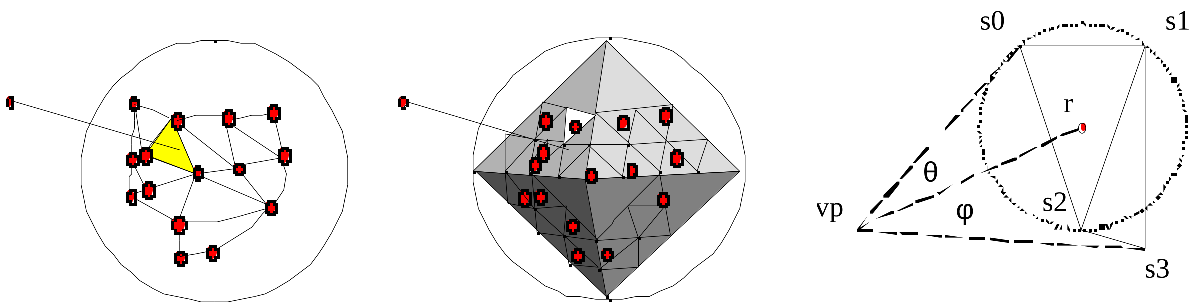

Figure 8. a) Spherical mesh and b) quadtree structure. The projection of a sample point onto these two structures is shown. The sample from the highlighted quadtree cell will be chosen for the walk, since the intersected cell is empty. c) Delaunay point-in-cone test: TRUE if ($\varphi < \theta$), i.e. s3 is interior to cone defined by vp,s0,s1 and s2.

We maintain a separate quadtree data structure to accelerate point location. Associated with the view sphere is an octahedron in canonical form (origin at the canonical viewpoint, corners aligned with coordinate axes) which subdivides the sphere surface into eight uniform spherical triangles. A triangular quadtree is created on each octahedral face. Figure 8a shows a simple spherical mesh, and 8b is a representation of the corresponding quadtree. Each quadtree cell contains a list of the samples that fall in that cell. Given a new sample point, we first convert it into integer barycentric coordinates relative to the triangle forming the quadtree root for the appropriate octant. The quadtree is then traversed until the appropriate leaf is located. This operation is efficient, requiring only integer shifts and adds at each level.

Once the leaf is located and before the new sample is inserted, we take the first sample stored in that leaf's set and from that vertex begin a walk along the mesh surface in search of the spherical triangle containing the new sample. In the case where the leaf cell contains no samples, the search recursively pops back up in the quadtree traversal until an occupied leaf is found (see Figure 8b). Each mesh sample contains a pointer to one of its adjacent triangles. This triangle is retrieved and tested to see if it encloses the new sample. The position of the point is compared against the planes of the great circles forming the spherical triangle. If the point lies inside of all three planes, it is accepted as being inside the triangle. If any one of the tests fail, the search traverses to the triangle that is adjacent across the plane for which the test failed. We are guaranteed that there will be an enclosing triangle because the surface of the view sphere is completely tiled with spherical triangles and similarly, due to the 2D nature of the topology, we are guaranteed to converge upon the correct triangle, without revisiting any triangles along the way.

After any changes to the mesh topology, we check to see if the Delaunay condition still holds. A triangle satisfies the Delaunay condition if the circumcircle of the triangle contains no other sample points [Preparata85]. In two dimensions, this condition can be verified with a point-in-circle test; in the spherical environment, we utilize a point-in-cone test. Given a triangle with vertices *v0,v1,v2* and its adjacent neighbor with vertices *v2,v1,v3*, the test is whether *v3* lies within the cone formed by the canonical view point *vp*, and *v0,v1,v2*. The triangle vertices are projected onto the view sphere, yielding points *s0,s1,s2,s3* (see Figure 8c). A point lies in the cone if the angle between the cone center ray *r* and *s3* is less than the angle between *r* and one (any) of the vertices *s0,s1,s2*. The ray defining the cone center is equivalent to the normal to the plane passing through *s0,s1,s2* .

After the addition of each new sample, the point-in-cone test is performed against the sample point and all adjacent triangles. If the test fails, an edge swap is performed in the quadrilateral formed by the two adjacent triangles [Guibas85, Lischinski94]. The worst case for swaps performed at insertion is O(n) where *n* is number of samples in the mesh. On average, we have found the number of edge swaps per insertion to be around 2.8, independent of the number of samples.

Sample points may also be deleted from the mesh when a better sample is found for the same image location. In this case, the sample is removed, as well as all of its adjacent triangles. The resulting hole is re-triangulated, with the Delaunay condition reasserted for all new triangles.

One of the advantages of this representation is that it is a valid 3-D triangle mesh that can be transformed with new views and rendered directly by the hardware. After each view change, the current mesh is first rendered. To optimize the rendering, we utilize software view frustum culling and OpenGL display lists. Both techniques make use of the spherical quadtree data structure. To perform culling, each of the 6 faces of the current view frustum is projected onto the spherical quadtree. All of the samples stored in each cell that is intersected by one of the frustum faces, and all triangles adjacent to those samples, are marked as potentially visible. This produces approximate visibility: it will include triangles that are not visible, and may leave out visible triangles, but only small triangles around the periphery of the view frustum. We have not noticed any visible artifacts from omitted triangles in practice, and therefore employ the relatively inexpensive culling to optimize the rendering. For each of the visible cells, the associated triangles are rendered into an OpenGL display list. As the view changes, the display lists for previously seen portions are re-used, and new display lists are rendered and stored for subsequent use as new cells become visible. For lower end machines, we have also implemented an approximate rendering scheme based on the quadtree, which is invoked if the frame rate drops below a specified rate. The approximation traverses the quadtree to a specified level and renders the triangles forming the quadtree cells, using an average of the samples stored in the cell as an approximation for the vertex depth and color. The level is chosen dynamically to give as much detail as possible while maintaining a minimum frame rate.

When the view is static, the mesh and the display are incrementally updated as more sample points are received for the same view. Each batch of new triangles will overwrite the image of the triangles that they are replacing. We utilize a painter's approach in the incremental display algorithm: the depth buffer is disabled, the new triangles are depth-sorted, then rendered back-to-front. When the canonical viewpoint corresponds to the current eye point, depth testing is unnecessary, but these two points do not always coincide. The mesh constructed from a canonical viewpoint is valid for all viewing configurations with the same eye position. For small view motions relative to the distance to the viewed sample points the mesh is re-used. With this approximation, errors will appear in the image where portions of the environment are occluded relative to the canonical viewpoint, but visible to the current eye point (or vice versa). We minimize these errors by reconstructing the mesh once the viewer moves a substantial distance away from the canonical viewpoint. This process can be slow if there are many samples, and therefore we first cull the samples to the new view frustum and only re-insert those samples that are relevant to the current view.

The mesh representation provides a reasonable interpolation of the sparse sample set available at start-up and the image is progressively refined as more information is received. In the limit, the image converges to the screen resolution as the mesh triangles refine to sub-pixel size. The most notable artifact in the resulting images is the lack of sharp edges, which is noticeable even in higher resolution images. Running on an SGI O2 workstation, about 10,000 samples can be added to the mesh per second. This rate is sufficient to keep up with a single ray-trace process during progressive refinement but is not able to update the display and representation as fast as the server can deliver samples in the case of cached rays, or multiple ray tracing processes.

Compared to the Voronoi version of the display driver, the triangle mesh gives us a smoother and more consistent representation, especially during view motion, and avoids the need for separate scene geometry. On the other hand, the triangle mesh construction can become a performance bottleneck, especially after large view motions. The appropriate choice of driver depends on the needs of the application.

2.4.6 Display Options

Each driver supports an option for tone mapping based on human color and contrast sensitivity, so that the display corresponds more closely to what a person would be able to see in a real environment [Larson97]. The Voronoi and triangle mesh drivers also support options for stereo display and local, dynamic objects.

Before it writes to the display, the driver must map the original floating point colors it gets from the server into RGB coordinates. This is a dynamic version of the image tone-mapping problem [Tumblin93]. For the sake of speed, and because it can account for human visual response and extremes of dynamic range, we apply the histogram adjustment procedure of [Larson97] in our drivers. In *camera* mode, we use an optimized brightness mapping that attempts to make all luminance levels visible over the current image, regardless of whether or not they would be visible in reality. We gather the requisite brightness histogram as samples come in from the server, and update the computed tone mapping at the user's request/convenience (e.g., on full screen redraws). In *human* tone-mapping mode, our method takes into account human color and contrast sensitivity, presenting a display that roughly corresponds to the real-world visibility a human observer would experience in the environment being simulated.

Stereo display requires adding a second eye point to the computation of beam requests, and handling the returned samples appropriately. In the case of the Voronoi representation, the samples are inserted into two lists, one for the left eye and one for the right eye. Most samples go into both lists, but the different left and right depth maps will affect which samples are used and how they are drawn near silhouettes. In the triangle mesh driver, the returned samples are all treated the same and used to build a single mesh, which is then rendered in stereo. It is easy to see how we could extend these algorithms to handle CAVEs and other multi-view environments [Cruz-Neira93].

One drawback of our holodeck implementation is that the ray cache is only valid if the scene is static. However, we at least partially overcome this limitation by implementing local, dynamic objects in the drivers themselves. These objects are rendered directly in OpenGL, using local lighting derived from the holodeck environment, similar to [Walter97]. For each object in an interior section, we query the server for all precomputed beams that pass through the object's center, then average and group the brightest of these beams into 8 light sources, putting the rest into an ambient component. Since each position gets its own local lighting, the rendered objects will exhibit the appropriate brightness distribution. By caching this information at each distinct object position, we can quickly handle small object motions or rotations without having to requery the server. The objects do not cast shadows, either back on themselves or into the holodeck environment, but they otherwise appear consistent with the rendered samples from our static scene.

Image 7 shows a wastebasket thrown into a cabin scene and illuminated by local light sources approximated from holodeck samples. Since all the light in this scene enters through the windows, the sides facing away from the window are darker. We can rotate this object in place using the computed lighting, but if we translate it significantly, we must recalculate the lighting from the holodeck based on its new position. For specular materials, we could also compute a sphere map or cube map from the relevant holodeck samples to further improve realism.

2.5 Process Coordination

Good coordination between the holodeck server, the sample generator, and the display process is needed to keep everything running smoothly. We could easily deadlock by waiting for a process that is waiting for us. To insure against this, we use the following process model:

1. The server waits for ray values to come back from **rtrace**, and checks the display process for any requests using a non-blocking read once new rays have been delivered. If there are no further beams to compute, the server waits for input from the display process.
2. The display process waits for input from the holodeck server and the user with equal priority, updating the image before each call to *select*.

3. The display process is permitted to send short, intermittent requests to the server. If the display process has a long request to make, it first puts in a request for the server's attention. While waiting for an acknowledgment, the display process continues to load packets sent by the server.
4. The display process may request a shut down, but the server makes the final decision. Once the display process receives a order to shut down, it must quit immediately.

The above rules are modified if there is no calculation process or no display process. If there is no ray calculation, the server waits on the display process alone, sending it whatever relevant rays it finds in the holodeck file. If there is no display process, the server creates its own list based on beam volumes.

While the user is changing views with the mouse, the server process may stall because its socket to the display backs up. This isn't a problem, though, because the display process will get back to reading from the server once motion has ceased, and there may be no need for the old beams in the new view, anyway.

A typical interactive calculation with all three logical processes is detailed in Appendix B.

3. Results

Image 8 shows the grid for an exterior holodeck section surrounding a 3-dimensional chess game. What is enclosed by the section grid will be visible from the outside, since the section type is set to *exterior*. Image 9a shows a view of the holodeck using the Voronoi driver, generated from scratch on a single processor SGI Octane in about ten seconds. Image 9b shows the same view after a minute. Models such as the virtual sculpture in Image 10 simply cannot be rendered using radiosity or multipass methods [Diefenbach96] because it contains curved, refracting surfaces, and most of the detail we are interested in is inside a solid block of glass. This image was rendered from scratch in a few minutes on an SGI O2 workstation. The same rendering can been produced in this time by the *Radiance* **rview** program, but as soon as the view changes even slightly, the rendering must be started over completely. The holodeck reuses rays whenever and wherever possible, providing interactive movement and realistic feedback. Thus, the user does not hesitate to move about freely and explore the space. This combination of accuracy and free movement is not provided by any other rendering method.

In Image 7, the scenery outside the window would be difficult to render directly even on the most advanced graphics hardware, because it contains over 500 million triangles. Because we only need to draw the visible samples returned by the holodeck server, displaying this image takes a few seconds on an entry level system.

Image 11 shows what happens with the holodeck when we take a vantage point that is outside all sections using the quadtree driver. Here we see our cabin model from a position over and between the living room section and the bedroom section. We see the result of rays that begin at the top or outside of each section and intersect the wall or floor below. Because the sections are within the room boundaries, the walls and ceiling are invisible, giving us a kind of X-ray vision. The geometry between sections is also invisible, so we see nothing of the wall and doorway that lie between the two rooms.

Image 12a shows a low resolution view of the bathroom mirror with the VDISTANCE variable set to False. Because the ray distance to the mirror itself is returned by **rtrace**, our display driver reprojects points on the mirror, regardless of what they reflect. By setting VDISTANCE to True, the distance to reflected objects is returned by **rtrace** instead, and we get the sharper reflection shown in Image 12b. In cases where the reflecting objects are very small, silhouettes may break up due to multidepth samples at the mirror edges as described in Section 2.4.3. This is one reason we might want to set VDISTANCE to False, especially with the quadtree and mesh drivers, which are more sensitive to this problem.

Image 13 shows multiple interior section grids in a proposed redesign of the Office of Environmental Policy at the White House. Note how the section walls intersect geometry, and extend into the hallway. In the hallway itself, the user can move from one office section to the next, possibly passing between sections. Because we can draw from sections behind as well as in front of us, this works fine. Image 14 shows an image taken from the hallway, where samples are being retrieved from a holodeck section lying just behind our view point. Because this scene contains specular surfaces and an indirect lighting system, it would be extremely difficult to render it in hardware, and although the geometry is not very solid at this early stage, the lighting and overall feel of the space are beginning to emerge.

Image 15a shows a terminal in the end office with a poor task lighting arrangement. We cannot really tell how bad it is, though, until we move our view point to that shown in Image 15b, where the specular reflection becomes more visible. For a rotation this large (above 20°), the quadtree and mesh drivers ignore their locally cached samples and uses only new ones sent by the holodeck server. For smaller moves, the display process would gradually update the image with new rays sent by the server, and the view-dependent highlight would dissolve out of its old position and into its new one. (The Voronoi driver shown in this image always gets new samples from the server, since it has no local cache.)

Most of the figures shown in this paper were generated on a single-processor workstation with low-end graphics. Using a multiprocessor platform with faster graphics hardware, we can achieve better interactivity in more challenging environments. For example, we employed a 16-processor Onyx to compute a daytime holodeck of the same OEP office space. To resolve the complicated interreflections, we ran 14 copies of **rtrace** for 20 hours to calculate about 83 million view rays, which went into an 820 Mbyte holodeck file. In all, over 1.4 billion rays were traced to compute the light field, and 235 thousand *indirect irradiance* values were recorded [Ward88,Ward94]. Viewing the holodeck interactively, our server accessed an average of 41,000 view rays per second from the holodeck for each new view, and computed 1200 rays/second continuously from its 14 **rtrace** processes. In both batch and interactive mode, CPU utilization was over 99% for each running copy of **rtrace**. In interactive mode, the other processors got light duty from the holodeck server and display process, except during and immediately after view changes, when the display process was quite busy.

Image 16 shows an interactive sequence taken from a walk-through of the daylight OEP office using the triangle mesh driver. Image 16a uses samples taken from the precomputed holodeck. Image 16b was captured during movement to a new view. Image 16c is what we see immediately after releasing the mouse; since the view has rotated more than 20°, the mesh driver ignores much of its local cache. Half a second later, the server has retrieved some better samples from the holodeck file; the resulting image is shown in Image 16d.

The air traffic control tower shown in Image 17 was rendered using the Voronoi driver on a 24-processor SGI Onyx2 with IR graphics. In a precalculation that achieved 96% linear speedup running 24 rtrace processes over 2 hours, we produced a 300 Mbyte holodeck file that contains as much detail as shown Image 17a at every eye point within the interior section, covering most of the control room. Each new view takes 3 seconds to retrieve and display 100,000 precalculated samples, and given another 30 seconds, improves to the resolution shown in Image 17b. Though this model is fairly complex, with 650,000 surfaces and many large textures, the real problem for traditional hardware rendering is the scale. The mountains on the horizon are 8 miles away, and we want to render them with foreground objects closer than a foot. That’s a ratio of 50,000:1, which is more than most depth buffer hardware can handle without serious errors. Because the holodeck samples have a depth complexity near 1, this problem is avoided. However, the most important benefit for this particular design application is the accurate prediction of lighting and visibility as provided by our global illumination and tone mapping methods. These features enable the user to make important design decisions regarding tower equipment and window shades, which could ultimately affect air traffic safety.

Table 1 shows interactive display rates for various hardware, drivers, and scenes. From these numbers, we see that the holodeck recall rate is not strongly affected by scene complexity or the presence of a progressive ray calculation. However, there is a cost associated with improving the display representation, as we move from drawing quadtrees to Voronoi regions to a Gouraud-shaded mesh.

Mach.	Driver	Scene	#proc.	Render	Recall
O2	QTree	Conf.	0	0	15,000
O2	QTree	Conf.	1	140	13,000
O2	Vor	Conf.	1	130	6500
O2	Mesh	Conf.	1	112	3200
O2	Vor	Cabin	1	410	5900
O2	Vor	Tower	1	180	9200
Onyx2	Vor	Tower	1	360	30,000
Onyx2	Vor	Tower	21	6500	35,000

Table 1. Holodeck performance for different configurations and scenes. (See conf. in Image 1, cabin in Image 4, tower in Image 17.) Interactive rendering and recall rates are in samples/second. Rendering rates include ray-tracing and display time. Precomputed sample recall rates are averaged over many view motions, and include disk, cache and interprocess transfer time.

4. Conclusions

The holodeck ray cache provides an interactive framework for progressive, view-independent ray tracing with hardware rerendering. By decoupling the sampling method from the display method, we can take full advantage of the available processor, graphics, and display hardware. Global illumination is computed by a physically-based ray tracer and cached for quick rerendering, providing an accurate impression of lighting from any view point.

Using the holodeck, a designer can interactively navigate a scene, evaluate the lighting, look for sources of glare, and make design decisions with the confidence of knowing that the displayed image is a reliable predictor of reality. Unlike progressive radiosity, we are not limited to simple, diffuse models, and unlike light field rendering, we do not need to compute the entire holodeck before viewing it. The user spends only as much time rendering as they need to make their evaluations, and they can always go back for more. Other rendering methods do not offer this combination of physical accuracy and user convenience.

Due to the modularity of our system, it is easy to add or improve the different components. We intend to further refine the triangle mesh driver to make it faster, provide mesh segmentation and layering to improve object silhouettes, and extend it to handle continuous motion for head-mounted displays. We would also like to run the holodeck on a massively parallel machine, such as the Cray T3E, to tackle some currently intractable lighting problems. Finally, we plan to experiment with captured rather than ray-traced holodeck samples, and mix synthetic and real worlds with mutual illumination.

5. Acknowledgments

The authors would like to thank Dan Baum, Celeste Fowler, Sara McMains, Gary Meyer, Holly Rushmeier, Eddie Saxe and Carlo Séquin for their helpful comments. Thanks to SGI and ONR (MURI grant #FDN00014-96-1-1200) for funding this work, and permitting us to make the software available as an add-on to LBNL's *Radiance* rendering package. To get the latest copy of *Radiance*, go to

http://radsite.lbl.gov/radiance, and to http://positron.cs.berkeley.edu/gwlarson/hd for the holodeck overlay.

6. References

[Airey90] Airey, John M., “Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations,” PhD. Thesis, UNC Chapel Hill, 1990.

[Ashdown93] Ashdown, Ian, “Near-field photometry: a new approach,” *Journal of the Illuminating Engineering Society,* Winter 1993, 22(1), pp. 163-180.

[Bala99] Bala, Kavita, Julie Dorsey, and Seth Teller, “Error-bounded interactive ray tracing,” Tech. Report 748, MIT Laboratory for Computer Science, March 1998.

[Bishop94] Bishop, Gary, Henry Fuchs, Leonard McMillan, Elen Scher Zagier, “Frameless Rendering: Double Buffering Considered Harmful,” *Computer Graphics (Proceedings of SIGGRAPH 94),* ACM, 1994.

[Camahort98] Camahort, Emilio, Apostolos Lerios, Donald Fussell, “Uniformly Sampled Light Fields,” *Rendering Techniques '98,* G. Drettakis and N. Max (eds.), SpringerWeinNewYork, 1998.

[Chen95] Chen, Shenchang Eric, “Quicktime VR An image-based approach to virtual environment navigation,” *Computer Graphics (Proceedings SIGGRAPH 95),* ACM, August 1995, pp. 29-38.

[Cruz-Neira93] Cruz-Neira, Carolina, Daniel Sandin, Thomas DeFanti, “Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE,” *Computer Graphics (Proceedings of SIGGRAPH 93),* 1993.

[Darsa96] Darsa, Lucia, Bruna Costa Silva, “Multi-resolution representation and reconstruction of adaptively sampled images”, SIBGRAPI, 1996, pp. 321-328

[Darsa97] Darsa, Lucia, Bruno Costa Silva, Amitabh Varshney,“Navigating Static Environments Using Image-Space Simplification and Morphing”, *Proceedings ACM Symposium on Interactive 3D Graphics*, 1997, pp. 25-34

[Debevec96] Debevec, Paul, Camillo Taylor, Jitendra Malik, “Modeling and Rendering Architecture from Photographs: A hybrid geometry- and image-based approach,” *Computer Graphics (Proceedings of SIGGRAPH 96),* ACM, 1996.

[Debevec97] Debevec, Paul, Jitendra Malik, “Recovering High Dynamic Range Radiance Maps from Photographs,” *Computer Graphics (Proceedings of SIGGRAPH 97),* ACM, 1997.

[Diefenbach96] Diefenbach, Paul J., “Multi-pass Pipeline Rendering: Interaction and Realism through Hardware Provisions,” PhD. Thesis, University of Pennsylvania, 1996.

[Fournier95] Fournier, Alain, “From Local to Global Illumination and Back,” 6th Eurographics Workshop on Rendering, Dublin, Ireland, June 1995.

[Funkhouser93] Funkhouser, Thomas, Carlo Séquin, “Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments,” *Computer Graphics (Proceedings of SIGGRAPH 93),* ACM, 1993.

[Gortler96] Gortler, Steven, Radek Grzeszczuk, Richard Szeliski, Michael Cohen, “The Lumigraph,” *Computer Graphics (Proceedings of SIGGRAPH 96),* ACM, 1996.

[Greene96] Greene, Ned, “Hierarchical polygon tiling with coverage masks,” *Computer Graphics (Proceedings SIGGRAPH 96),* ACM, August 1996, pp. 65-74.

[Greene93] Greene, Ned, Michael Kass, and Gavin Miller, “Hierarchical Z-buffer visibility,” *Computer Graphics (Proceedings SIGGRAPH 93),* ACM, August 1993, pp. 231-238.

[Guibas 85] Guibas, L, and J. Stolfi, “Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams”, *ACM Transactions on Graphics*, 44(2), 1985, pp. 74-123.

[Guo98] Guo, Baining, “Progressive Radiance Evaluation Using Directional Coherence Maps,” *Computer Graphics (Proceedings SIGGRAPH 98)*, ACM, July 1998, pp. 255-266.

[Haeberli90], Haeberli, Paul, “Paint by Numbers: Abstract Image Representations,” *Computer Graphics*, 24(4), August 1990.

[Kirk91] Kirk, David, James Arvo, “Unbiased Sampling Techniques for Image Synthesis,” *Computer Graphics*, 25(4), July 1991.

[Larson97] Larson, Greg Ward, Holly Rushmeier, Christine Piatko, “A Visibility Matching Tone Reproduction Operator for High Dynamic Range Scenes,” *IEEE Transactions on Visualization and Computer Graphics*, December 1997.

[Larson98] Larson, Greg Ward, Rob Shakespeare, *Rendering with Radiance: The Art and Science of Lighting Visualization*, Morgan Kaufmann, 1998.

[Levoy96] Levoy, Marc and Pat Hanrahan, “Light Field Rendering,” *Computer Graphics (Proceedings of SIGGRAPH 96),* ACM, 1996.

[Lischinski94] Lischinski, Dani, “Incremental Delaunay Triangulation”, in *Graphics Gems IV*, edited by Paul Heckbert, Academic Press, 1994, pp. 47-49

[Lischinski98] Lischinski, Dani and Ari Rappoport, “Image-based rendering for non-diffuse synthetic scenes,” 9th Eurographics Workshop on Rendering, Vienna, Austria, June 1998.

[Luebke97] Luebke, David and Carl Erikson, “View-dependent simplification of arbitrary polygonal environments,” *Computer Graphics (Proceedings SIGGRAPH 97),* ACM, August 1997, pp. 199-208.

[McMillan95] McMillan, Leonard and Gary Bishop, “Plenoptic modeling: an image-based rendering system,” *Computer Graphics (Proceedings SIGGRAPH 95)*, ACM, August 1995, pp. 39-46.

[Miller98] Miller, Gavin, Steven Rubin, and Dulce Poncelen, “Lazy decompression of surface light fields for pre-computed global illumination,” 9th Eurographics Workshop on Rendering, Vienna, Austria, June 1998.

[Nimeroff96] Nimeroff, Jeffry, Julie Dorsey and Holly Rushmeier, “Implementation and Analysis of a Global Illumination Framework for Animated Environments,” *IEEE Transactions on Visualization and Computer Graphics*, 2(3), December 1996.

[Ofek98] Ofek, Eyal, Ari Rappoport, “Interactive Reflections on Curved Objects, *Computer Graphics (Proceedings SIGGRAPH 98),* ACM, July 1998, pp. 333-342.

[Painter89] Painter, James and Kenneth Sloan, “Antialiased Ray Tracing by Adaptive Progressive Refinement,” *Computer Graphics (Proceedings of SIGGRAPH 89)*, 23(3), July 1989.

[Pighin97] Pighin, Frédéric, Dani Lischinski, David Salesin, “Progressive Previewing of Ray-Traced Images Using Image-Plane Discontinuity Meshing,” 8th Eurographics Workshop on Rendering, Saint-Etienne, France, June 1997.

[Schaufler98] Schaufler, Gernot, "Per-Object Image Warping with Layered Imposters," *Rendering Techniques '98*, G. Drettakis and N. Max (eds.), SpringerWeinNewYork, 1998.

[Shade96] Shade, Jonathon, Dani Lischinski, David H. Salesin, Tony DeRose, and John Snyder, “Hierarchical image caching for accelerated walkthroughs of complex environments,” *Computer Graphics (Proceedings SIGGRAPH 96),* ACM, August 1996, pp. 75-82.

[Shirley95] Shirley, Peter, Bretton Wade, Philip Hubbard, David Zareski, Bruce Walter, Donald Greenberg, “Global Illumination via Density-Estimation Radiosity,” 6th Eurographics Workshop on Rendering, Dublin, Ireland, June 1995.

[Sillion94] Sillion, Francois X. and Claude Puech, *Radiosity and Global Illumination*, Morgan Kaufmann Publishers, San Francisco, 1994.

[Sillion97] Sillion, Francois, George Drettakis, Benoit Bodelet, "Efficient Imposter Manipulation for Real-Time Visualization of Urban Scenery", *Proceedings Eurographics, Computer Graphics Forum*, 1997, pp. 207-218

[Teller91] Teller, Seth and Carlo H. Séquin, "Visibility preprocessing for interactive walkthroughs," *Computer Graphics (Proceedings SIGGRAPH 91),* July 1991, 25(4), pp. 61-69.

[Tumbline93] Tumblin, Jack and Holly Rushmeier. "Tone Reproduction for Realistic Images," *IEEE Computer Graphics and Applications*, November 1993, 13(6).

[Walter97] Walter, Bruce, Gun Alppay, Eric P. F. Lafortune, Sebastian Fernandez, and Donald P. Greenberg, "Fitting virtual lights for non-diffuse walkthroughs," *Computer Graphics (Proceedings SIGGRAPH 97),* ACM, August 1997, pp. 45-49.

[Walter99] Walter, Bruce, George Drettakis, Steven Parker, "Interactive Rendering using the Render Cache," Proceedings of the 10th Eurographics Workshop on Rendering, June 1999.

[Ward88] Ward, Greg, Francis Rubinstein, Robert Clear, "A Ray Tracing Solution for Diffuse Interreflection," *Computer Graphics (Proceedings of SIGGRAPH 88)*, 22(4), 1988.

[Ward91] Ward, Greg, "Real Pixels," in *Graphics Gems II*, edited by James Arvo, Academic Press, 1991.

[Ward94] Ward, Greg, "The RADIANCE Lighting Simulation and Rendering System," *Computer Graphics (Proceedings of SIGGRAPH 94)*, ACM, July 1994.

Appendix A: Viewpoint Proximity Sampling

During batch holodeck calculation, rays are generated randomly within a beam by choosing uniformly distributed points on the entry and exit cells and taking the ray that passes between them. This is a quick initialization requiring essentially four calls to the random library function per ray. During an interactive calculation, however, we need a sampling algorithm that keeps rays within a desirable distance of the eye point to accelerate image convergence. The simplest such algorithm is rejection sampling, where we reject any candidate ray that does not pass within the specified distance. Unfortunately, rejection sampling can be quite expensive for certain viewpoints that have a relatively small allowed cell intersection area, such as the one shown in Figure A1. If we have to sample many rays to come up with one valid candidate, we will spend longer determining which rays to trace than we spend tracing them. This is in fact what happened when we first tried this technique. We decided that a more intelligent, direct sampling method was needed.

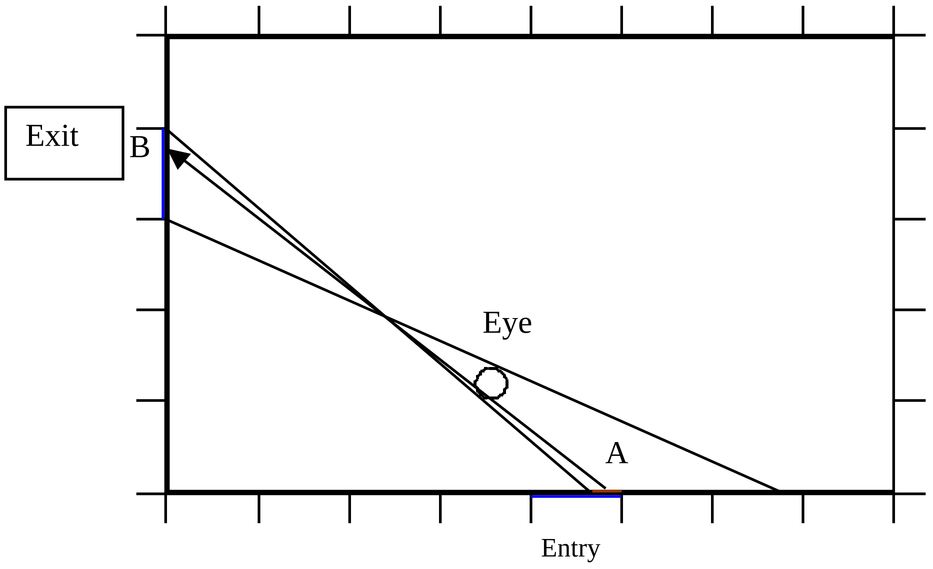

Figure A1. Extremal rays showing for a cell pair corresponding to a beam, the region where legal rays might originate within the entry cell (thickest line segment). A single example sample ray is shown.

Ideally, we would like to know the convex hull of all rays that intersect the entry and exit cells and also pass through a specified sphere around the eye point. We might then be able to sample rays restricted to lie within this convex hull. Unfortunately, neither the determination of such a hull or the sampling of rays within it are tractable problems, so we employ an approximation. Our compromise sampling algorithm proceeds as follows:

1. Intersect a cone defined by one corner of the exit cell and the sphere around the eye point with the entry wall.
2. Find the minimum and maximum extents of the resulting ellipse within the bounds of the entry cell.
3. Repeat step 1 using the opposite corner of the exit cell.
4. Find the minimum and maximum extents of the new ellipse and use them to extend the region found in step 2.
5. If the above results in a null region, default to uniform beam sampling.
6. Choose a ray origin within the determined entry region (point A in our figure).
7. Intersect the cone defined by this entry point and the eye sphere with the exit wall.
8. If the resulting ellipse does not share area with the exit cell, increment a counter and loop back to step 6 if counter is less than N, otherwise default to uniform sampling.
9. Choose an exit point within the determined region, and this defines our ray ($A \rightarrow B$ in our figure).

The breaks in steps 5 and 8 are required because this algorithm is imperfect, beginning with step 1. We only use two corners of the exit cell when in fact we should use a continuous sampling along all four edges to determine the exact candidate entry area. However, two corners are usually enough, and much cheaper than a large number of ellipse sample calculations. The failure to include the whole area sometimes results in failure at step 5, in which case we fall back on the original uniform sampling algorithm, which does not guarantee the ray will pass within the desired radius of the eye point.

Because we are usually sampling several rays at a time within the same beam, we do not need to repeat steps 1-5 for each ray, but can use our region computed in step 4 to run through steps 6-9 repeatedly. When step 8 fails for a given ray, we iterate rather than giving up right away, as there is still a good chance of getting our sample the next time around. We set the iteration limit to the number of desired sample rays plus 2 for the entire beam packet to ensure that we do not waste too much time resampling when failure dominates.

Appendix B: Process Coordination Example

The startup sequence for a typical interactive session with the three logical processes, holodeck server, sample generator and display, proceeds as follows:

1. The user starts the program, specifying two **rtrace** processes and the desired driver.
2. The holodeck server opens the holodeck file, opens the display driver, and starts two **rtrace** processes.
3. The first **rtrace** process loads all of its scene files and octree and initializes its data structure, then forks itself.
4. The second **rtrace** process attaches its i/o descriptors to the child of the first **rtrace**, so the processes effectively share memory on a copy-on-write basis.
5. The display driver gets the holodeck section grids from the server, and sets up a default view. It computes the relevant beams for this view, and prepares a long request for the server.
6. The server, having no beams to work on yet, has been waiting on the display process for input.
7. The display process requests the servers attention, and the server sends an acknowledgment.
8. The display process gets the acknowledgment, and sends its list of beams.
9. The server gets the list of beams, and checks to see what it can satisfy from the holodeck file. It sorts the beams in file order to minimize disk access time, and sends rays to the display process as it loads them from the file into memory.
10. The display process loads rays from the server and puts them into its quadtree, updating the displayed image every 50,000 samples (if there are that many).
11. Once the server has exhausted the supply in the holodeck file, it flushes the data to the display process and assigns beams to **rtrace** on a least-filled/most-requested priority basis.
12. After it has read all the beams sent immediately by the server, the display process updates the displayed image and calls *select* to wait for user input or more server packets.
13. The server, meanwhile, has called *select* to wait for one of the **rtrace** processes to send it some results.
14. One of the **rtrace** processes finishes a beam packet and flushes it to the server.
15. The server stores the beam packet in memory, freeing memory as necessary by writing beams to disk using an LRU scheme.
16. The server flushes the computed samples on to the display process and checks it for input.
17. If there is no request from the display, the server queues a new beam packet to **rtrace** and calls *select* again.

The server continues in this manner, interrupting its tending of **rtrace** only to fill display requests and manage holodeck file caching. The display process continues handling input from the server and the user and updating the displayed image. When the display process makes a shut down request, the server flushes its queue and closes **rtrace**, then flushes data to the holodeck file and sends a final shut down directive to the display. It then waits for the display process to finish before exiting itself.

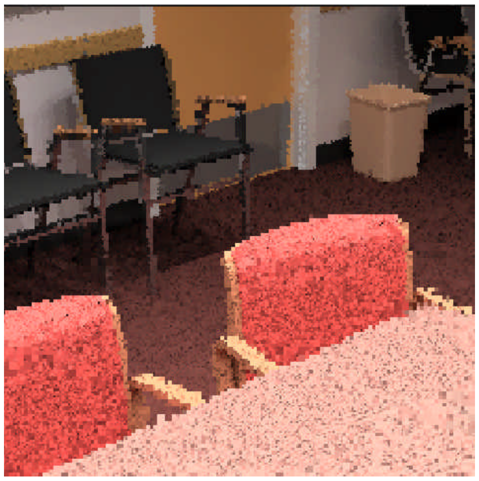
Image 1. A low-resolution, rectangle-filled quadtree rendering of a conference room.

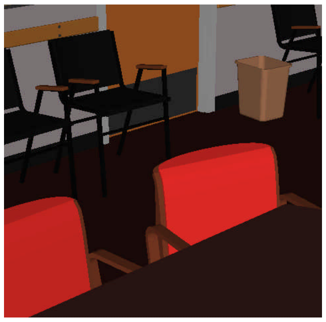
Image 2. An OpenGL rendering of the conference room geometry using a hardware lighting model.

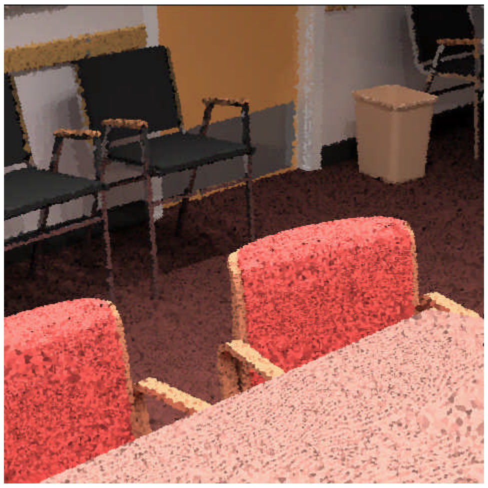
Image 3. The same conference room samples drawn with the Voronoi driver using the geometry above to clarify objects.

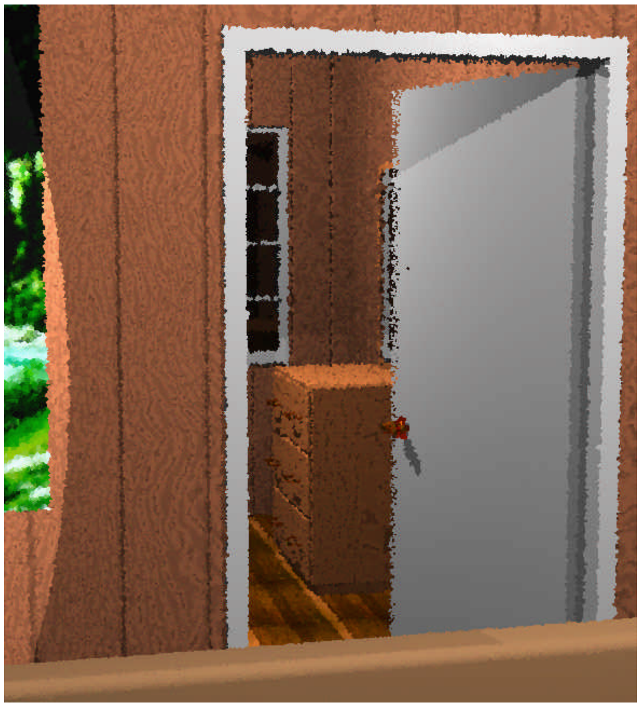
Image 4. An invisible portal separates the loaded geometry in this room's section to the unloaded geometry in the next. We see the effect in the sharp silhouettes in the foreground versus the fuzzy ones in the next room.

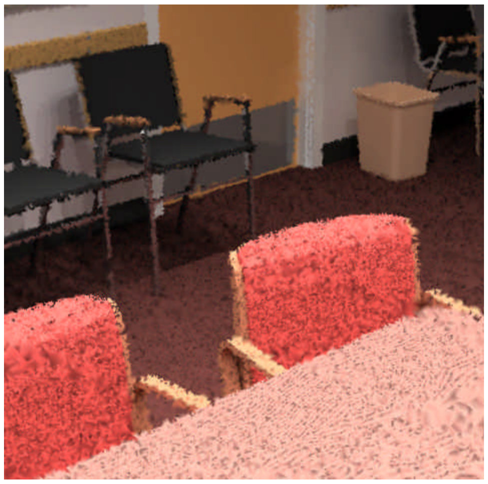
Image 5. The same conference room samples drawn with the triangle mesh driver.

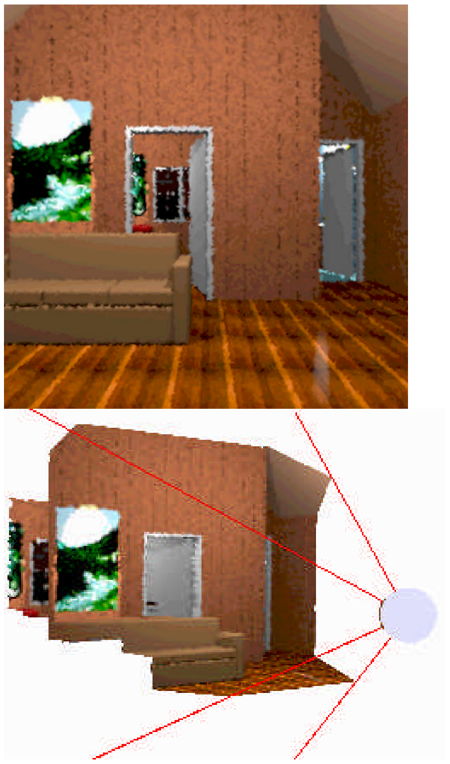

Image 6. The top image (a) shows an example view produced by the mesh driver. The bottom image (b) shows the representation from the side, illustrating its 2.5-D nature.

Image 7. A local, dynamic object inserted in our scene and rendered using OpenGL lights approximated from holodeck beam samples (Voronoi driver).

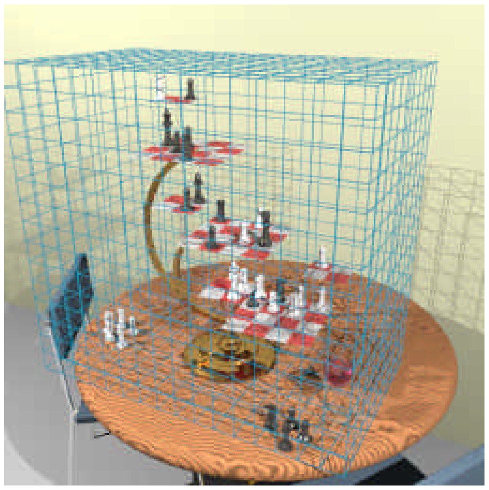

Image 8. A grid for a holodeck section that is meant to be viewed from the exterior.

Image 9a. An interactive rendering of the chess scene after 10 seconds on an SGI Octane (Voronoi driver).

Image 9b. The same view after 1 minute.

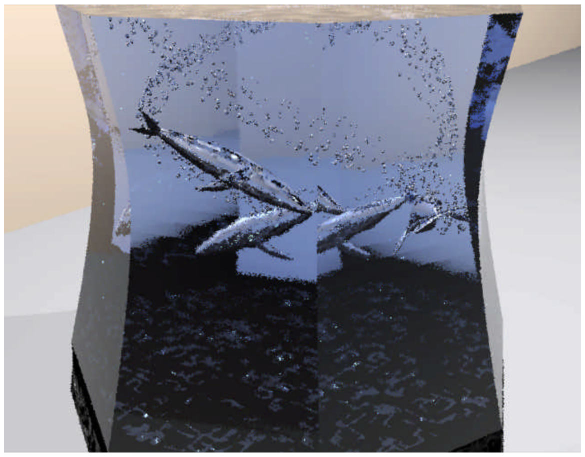

Image 10. A solid glass sculpture with whales and bubbles carved out of air and rendered using the holodeck (Voronoi driver).

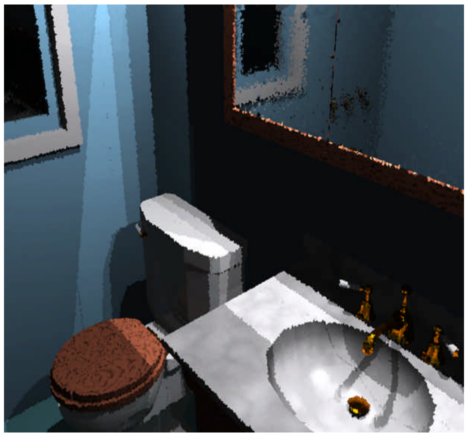

Image 12a. A low resolution rendering of a bathroom mirror with VDISTANCE=False, showing the resulting lack of definition in the reflection (Voronoi driver).

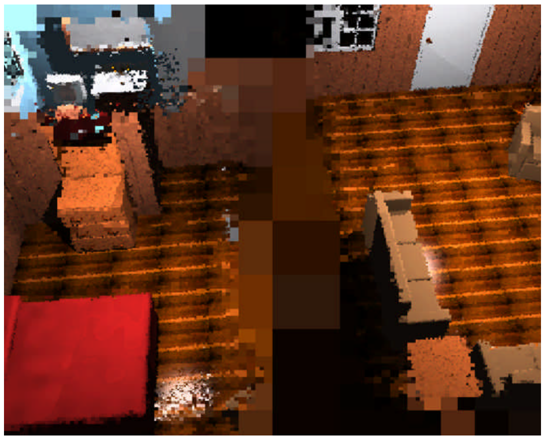

Image 11. A view from above and between two holodeck sections, with invisible regions (quadtree driver).

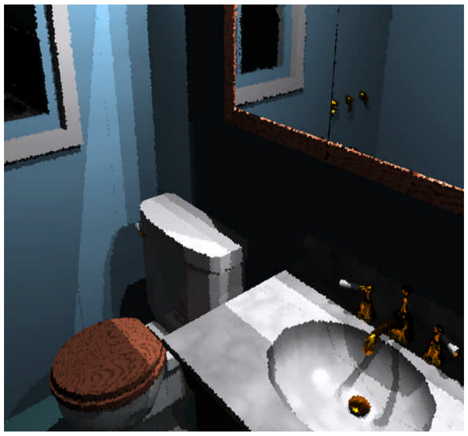

Image 12b. By setting VDISTANCE=True, we get a sharper image in our mirror.

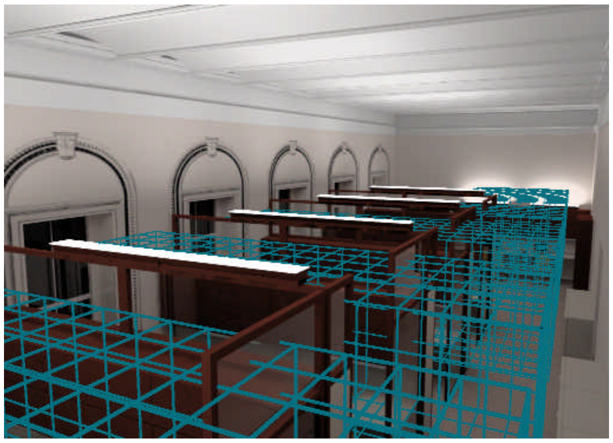

Image 13. An overview of the OEP office space with multiple holodeck sections.

Image 14. A low resolution view of the OEP hallway, taken from between two sections (Voronoi driver).

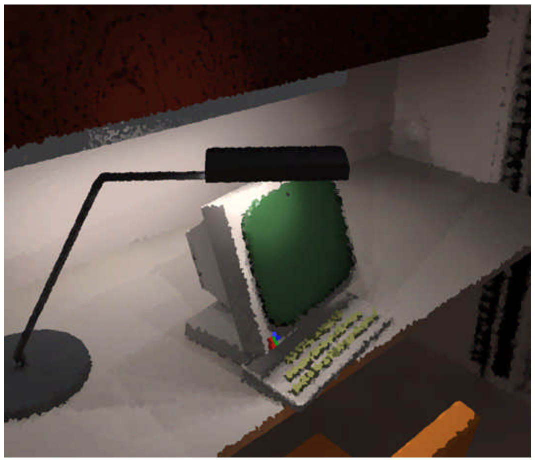
Image 15a. A close-up of a workspace terminal with poor task lighting (Voronoi driver).

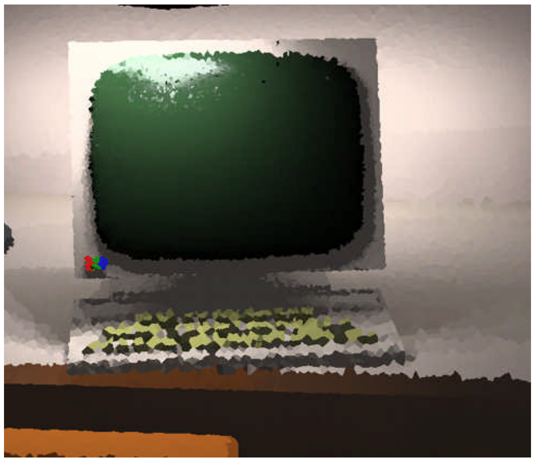
Image 15b. From another view, we can better see the problem with specular reflection off the screen.

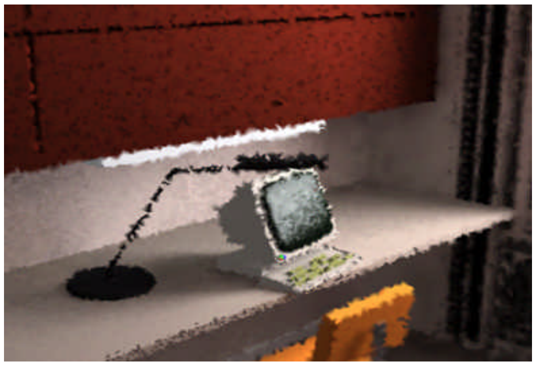
Image 16a. A daylight version of the space precomputed in 20 hours on an Onyx using14 processors (mesh driver).

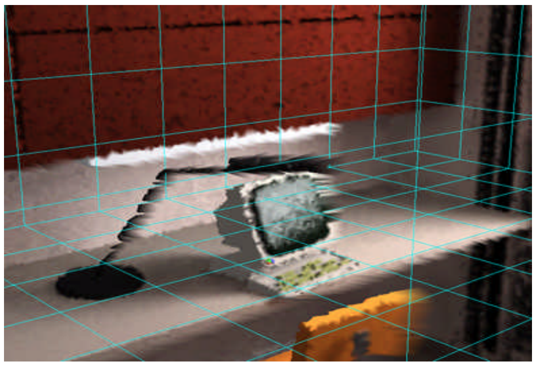
Image 16b. The low-resolution display and section grid drawn for feedback during mouse-controlled view movement.

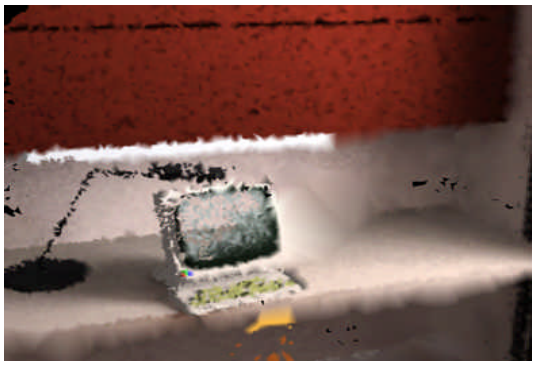
Image 16c. The image displayed immediately after releasing the mouse. We have moved so much that the display cache is missing some samples.

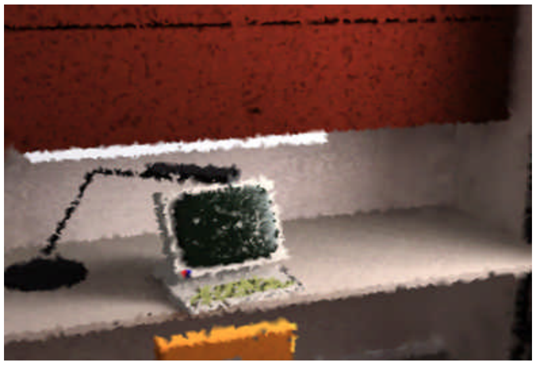
Image 16d. The same view after half a second, during which time the server has retrieved some more relevant samples from the holodeck file.

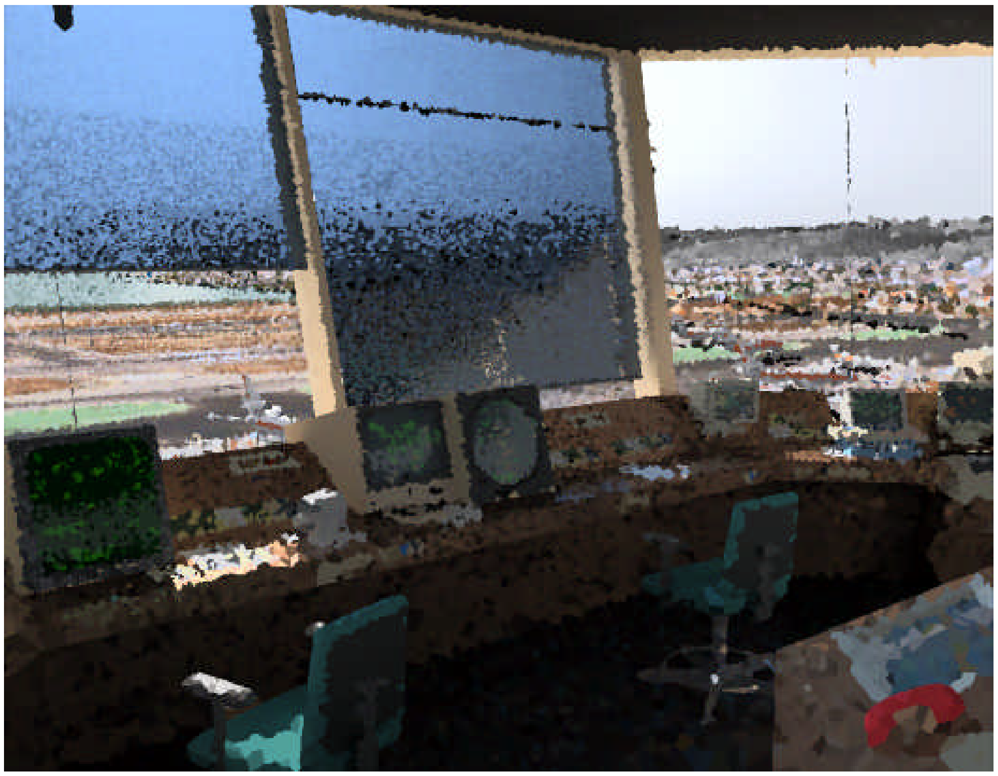

Image 17a. An air traffic control tower cab displayed from a precalculated holodeck file (Voronoi driver).

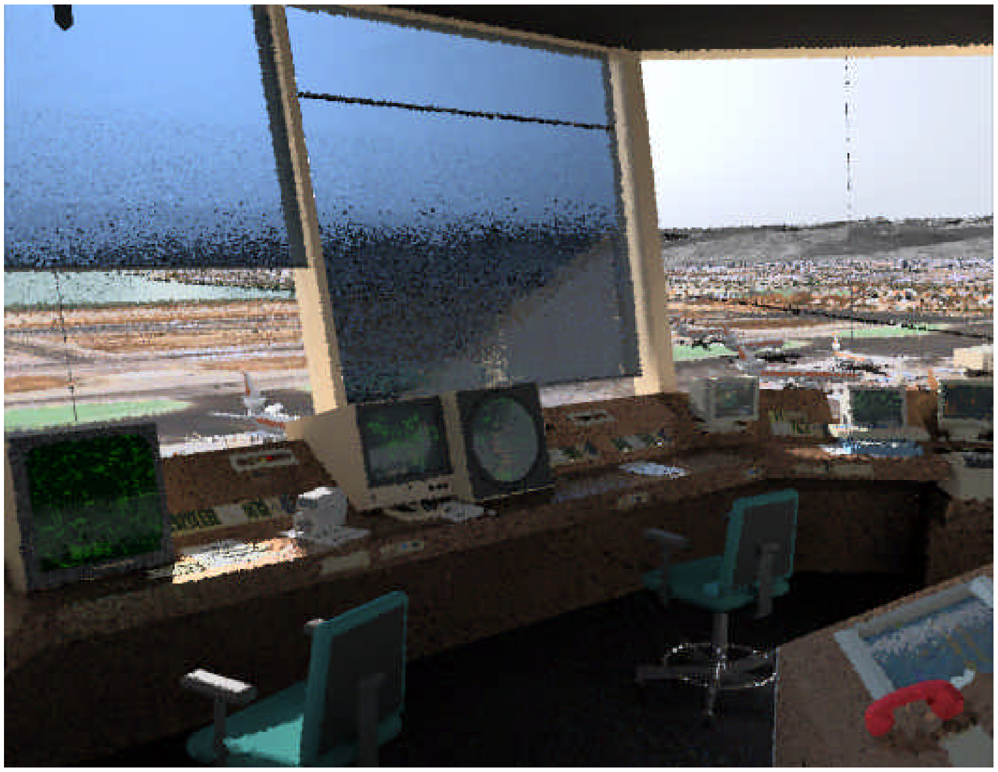

Figure 17b. The same simulation after 30 seconds of progressive ray tracing on 21 processors.