The Holodeck: A Parallel Ray-caching Rendering System

Gregory Ward Larson
Silicon Graphics, Inc.

Abstract

This paper presents a new data structure for light field rendering, which resembles a *Star Trek* holodeck in form and function. The grid on a holodeck section acts as a four-dimensional rendering target for a ray tracing algorithm, whose goal is to update an interactive display. A holodeck server coordinates separate ray evaluation and display processes, optimizing disk and memory usage. Holodeck data may also be computed off-line, and displayed later with or without an interactive ray calculation. Since the rendering hardware is not being taxed by either geometry or lighting, the net result is interactive walk-throughs of complex spaces with arbitrary surface reflection functions.

1. Introduction

An important goal in computer graphics is to produce realistic renderings in real time, simulating a window into a virtual world that the user may alter. As a practical matter, realism and real-time interaction need to be balanced for a particular application. It may be possible in one case to achieve real-time frame rates by reducing realism to what the graphics hardware will support. In another case, we may be willing to tolerate reduced interactivity to achieve the best-looking or most accurate results. In this paper, we consider applications where physical accuracy is critical, and we want to get there as fast as we can, provided we do get there eventually. We specifically look at how we can improve the interactive visualization capabilities of a physically-based ray tracing solution to global illumination through parallel processing and view ray sample caching. The principal benefit of our method over typical image-based rendering (IBR) approaches is that the entire representation need not be precomputed before a user can begin touring the scene.

Two basic approaches to interactive ray traced imagery have emerged over the years. The first approach is to update the display progressively as rays are traced for a particular view. This can be done by simply drawing progressively smaller rectangles [Painter89], or by using more sophisticated representations, such as constrained Delaunay triangulations and texture maps [Pighin97]. The second approach is to precompute a holographic scene representation, and compress it for quick synthesis of particular views [Levoy96] [Gortler96]. The problem with the first approach is that all information about a particular view is lost once the viewer moves to a new position, where the image must be recalculated from scratch. The problem with the second approach is that all possible views must be precalculated at the outset, which is inefficient, and precludes the possibility of iterative scene changes.

In this paper, we present a third approach, which combines a holographic scene representation with a parallel, interactive ray calculation. 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, which may be refined while the view is stationary. 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 compression or "development" stage is required. 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 devices 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

[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.

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 by Levoy and Hanrahan [Levoy96] and Gortler et al [Gortler96], except that there is no development step -- rays are stored and retrieved interactively.

The lack of any development step and the need for rapid access have two important implications. First, ray samples are going to take up a lot of space -- since we cannot afford to perform coherency-based compression, everything may not fit in memory. Second, we require some kind of virtual memory (VM) management. Although we could leave this task to the operating system, it was immediately apparent that the common algorithms for VM management are too expensive and inefficient for our needs. We therefore created a holodeck server process, which 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. The ability to direct the samples is a plus, 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 to throw away directional information [Shirley95].

Along with each computed radiance, we store the ray distance so we may reproject sample points onto our displayed image. This 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 this with some clever drawing techniques.

Overall, our system normally consists of three logical processes: a holodeck server, a ray calculation, 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 an **rtrace** process if a display-only function is desired, or without the display process for background calculation.

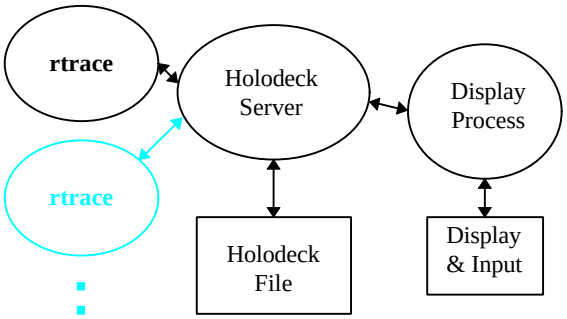

Figure 1. Schematic diagram of holodeck rendering system. Arrows show the flow of information.

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 ray calculation itself. Fourth, we describe two versions of the display process, one for X11 and one for OpenGL. Finally, we discuss coordination between our three logical processes.

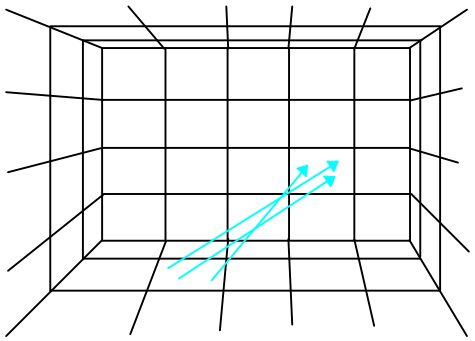

Figure 2. A holodeck section as seen from inside. A ray passing from one grid cell to another is stored together with other rays in the same beam. (A beam of three rays is shown.)

2.1 The Holodeck Data Structure

The holodeck data structure stores information for all the 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:

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

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

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 about 10^6.

The shape of a holodeck section is also important. If it is too narrow in any one dimension, it will not produce a good distribution of beams. For example, if the distance between an opposing pair of walls is the same size as the grid cells on those walls, a beam going straight across will contain rays that vary by more than 90° in their direction! The optimal section shape is a cube, but we found parallelepipeds with aspect ratios as high as 1:5 work quite well. Long, narrow passageways and large rooms with low ceilings are usually broken up into multiple sections to avoid problems. It is not necessary for each section to abut its neighbor, since views can be maintained somewhat outside of holodeck sections as well. (This is explained in the Display Process section and demonstrated in the Results section.)

As we mentioned in the introduction, each view ray sample is encoded into ten bytes in our holodeck structure. This encoding is detailed in Table 1. The ray color is stored in the four-byte, RGBE floating-point format native to *Radiance* [Ward91]. This format covers a wide dynamic range, which is important so we can 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[3]. 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 encoded as a 16-bit unsigned integer, which is encoded as 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 3 shows a holodeck file layout with three sections. For clarity, only a few beam pointers are shown.

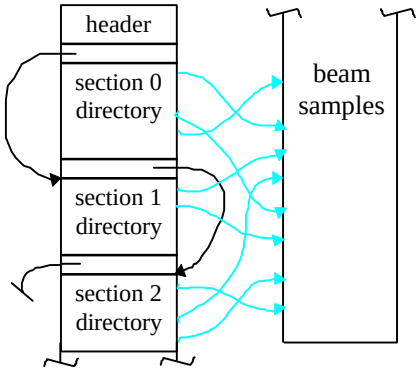

Figure 3. Typical holodeck file layout, showing offset pointers.

2.2 The Holodeck Server

The holodeck server is responsible for maintaining holodeck file consistency and keeping a cache of 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. However, there may be significant time spent waiting for disk seeks and reads, 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. Once the cache becomes full, beams are written to the holodeck file to free up memory, least-recently-used (LRU) beams first.

[3] The actual ray origin may be anywhere along the path between the entry and exit points, depending on the user-specified calculation parameters discussed in the following subsection.

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 for batch operation. List entries specify the section, index and desired number of samples for each beam. Since error is proportional to $M^{-0.5}$ for M Monte Carlo samples, the list is sorted in order of increasing computed/desired sample counts, which corresponds to a *decreasing* computed/desired error ratio. On each iteration, one or more beam requests is removed from the head of the list, and a number of new samples is assigned to the ray calculation (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. In the interim, the server sends the display process whatever rays were computed previously.

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 Ray Computation

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. The processes share memory and indirect irradiance values efficiently up to at least 16 invocations, which is as many as we have had the opportunity to try in our tests.

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(2) 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. So far, we have never witnessed any delays due to this method.

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 a 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[4]. To compute maximum queue length, we divide this number 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 after the maximum packet size, or manually by sending **rtrace** a zero direction vector. (Coordinated flushing is also necessary to avoid deadlock.)

The server's interface to our 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

[4] 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.

several packets, possibly from more than one process, which are all put in the returned list.

The actual rays traced depend not only on the selected beam, but also on a user-set parameter, called OBSTRUCTIONS. This parameter may be set to True, False, or neither. If True, each ray will begin at a random point on the beam's entry cell, and proceed toward a random point on its exit cell. This way, the ray is guaranteed to intersect any object lying between its entry and exit point. If OBSTRUCTIONS is False, then the ray *begins* at the exit point, assuring no objects contained within the holodeck section will be visible. If OBSTRUCTIONS is left unset, each ray will have its origin at some random point between the entry and exit points, so it will sometimes intersect an interior object in its path, and sometimes not.

This ability to control the visibility of interior objects may seem perverse, but it is actually quite useful. If the holodeck section encloses an object to be viewed from the outside, then setting OBSTRUCTIONS to True gives us what we want. If we plan to be inside each section and render local geometry with an alternate technique, then setting OBSTRUCTIONS to False is clearly the right thing to do. If we plan to view our holodeck from the inside, but we are not certain that we have excluded all relevant geometry from each section, then leaving OBSTRUCTIONS unset is most reasonable. That way, we will be able to see interior objects, but we will also be able to see past them, even if they block large areas. Starting a ray at a random distance means it will be more likely to intersect an object near the exit wall, which is what we want to see from an interior viewpoint. In effect, leaving this variable unset gives a soft boundary to each holodeck section. Resulting occlusion discrepancies will be cleared up in the display algorithms, discussed in section 2.4.3.

Another user parameter controls not what a ray sees, but how **rtrace** evaluates distance. If the VDISTANCE parameter 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 may 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 these user parameters in the Results section.)

Thanks to the simplicity of our queuing model and the nominal demands we place on our ray evaluation, 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 environment. For its part, the display process must do the following:

- Accept user input and view manipulation,
- Tell the holodeck server which beams to compute, and
- 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.

To simplify our discussion, we will only examine the two common graphics configurations we have implemented: 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 only discuss the image representations separately.

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 looking toward the first section if OBSTRUCTIONS is True). 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. Only view rotation, which keeps the view origin where it is, avoids the need for any 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 commands are provided to facilitate interactive scene changes. A command is provided to kill the **rtrace** process and another to restart it after some change to the scene description. The changes dissolve-fade into the displayed image 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

Each time a view is selected, the display driver must inform the holodeck server which beams it needs in order to fill in the image. It is then the server's responsibility to get as many rays to the display driver as quickly as possible so we can display a reasonable image. The server does this by first sending whatever rays it happens to have in memory, followed by whatever rays it can find on disk, followed by whatever new rays are calculated by the ray tracing process. This final stage continues until the screen resolution is achieved, with frequent updates along the way.

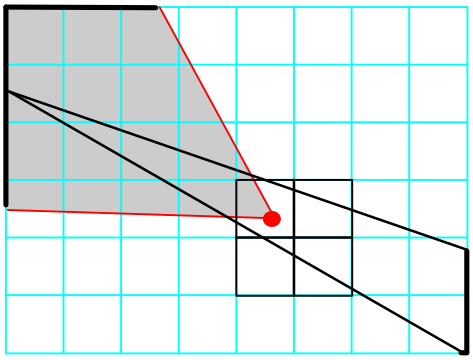

Figure 4. A plan view of a holodeck section, showing how we determine beams that contribute to a specific view. For each cell in our view volume (gray), we identify beams by drawing pyramids through voxels to find the opposite cells.

The beams we need for our display are the ones that pass close by our view point, whose rays are directed inside our view volume. To identify these beams, we calculate which cells are intersected by rays passing through the grid cells in our view and the eight closest voxels around our viewpoint. Figure 4 shows a plan view of an example holodeck section with a user's view volume. We draw a pyramid from each visible cell backwards through each voxel, and determine which cells it encloses on the opposite side. All beams between the grid cells in the view volume and their opposing partners will potentially contribute to this view.

To minimize the cost of recomputing the beam list for small view changes, we compute it in eight parts, one for each neighborhood voxel. For small changes in the view position, we only have to compute beams for whatever new voxels are added to our neighbor list, dropping ones that fall off. If a movement doesn't cross a voxel center plane, we won't have to change voxel lists at all.

For small changes in the view direction, we only need to recompute beams for cells near the image borders. In our implementation, we make a list of cells visible in the previous view and cells visible in the new view for each voxel shared by the two views. We then compute the beams that were in the last view but not in this one, and take them off our list. Beams that are in this view but weren't in the last one are added. The rest we can leave alone. If fewer than half of the old cells are shared by the new view, it's faster to just clear the voxel list and compute all the new cell beams.

What happens when our view is very close to one of the section walls, or outside the holodeck section altogether? In general, we should be able to see anywhere our view rays penetrate the holodeck, no matter where our vantage point is. Even when we look away from the holodeck, there will be some rays that can be applied towards our view. Figure 5 shows a few examples. View A, though it is directed *away* from the holodeck section, can still pick up beam samples that happen to be leaving the holodeck in its direction. View B is looking towards the section, and poses no particular problem. View C, however, will only form a partial image if we are working from this one section, since some ray directions do not intersect the holodeck at all. To compute the beams for exterior view points like A, B and C, we extend the algorithm shown in Figure 4 by allowing voxels to exist outside a holodeck section. In our implementation, we reduce the number of neighborhood voxels to four, two or one, depending on which section wall planes contain the viewpoint. This reduces the number of beam computations that add little information to the view. In general, a holodeck consists of multiple sections, and we can choose whichever section is closest to get our beams. In some cases, we pull beams from more than one section for a view that lies between sections.

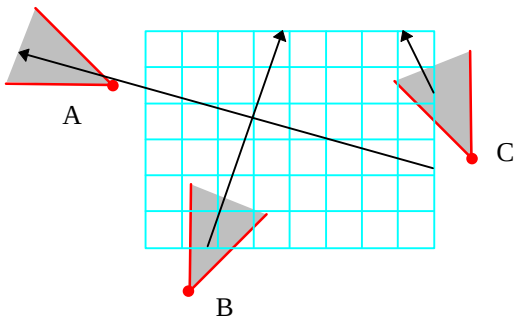

Figure 5. Three possible views of a holodeck section. One example ray is shown for each view.

2.4.3 Image Representation

Once it has informed the holodeck server which beams it wants, the display process converts the returned ray samples into a displayed image. Drawing the rays as points doesn't work because there aren't enough of them to cover more than a fraction of the screen's pixels. A progressive Delaunay triangulation would work, but would be time-consuming if applied to every ray. Rather than using a more sophisticated technique such as a constrained Delaunay triangulation with textured regions [Pighin97], we decided to start with a simple quadtree representation. The advantage of a quadtree is that it's easy and quick to update, and gives us a region to draw for each ray sample. We aren't forced to draw rectangles, but are free to choose

whatever representation works best on our display. Under X11, we decided to draw rectangles. Under OpenGL, we draw Dirichlet domains (a.k.a. Voronoi polygons). We will discuss these two choices and their implications at the end of this section.

Because the holodeck's ray samples pass *near* our view point rather than through it, we need to reproject the intersection points to get the correct image position for each ray. This is a straightforward matrix transformation on the world intersection point, computed from the values given in Table 1. The image quadtree is built by subdividing a quadrant whenever it contains more than one sample point, down to some minimum node size, which is usually a few pixels on a side. At terminal nodes, we only replace a leaf sample if the new ray intersection is in front of the old one. If two samples are about the same distance, we choose the ray that passes closest to our view point. In no case will we use a sample that passes beyond some maximum angle from our view point, which is 20° in our implementation. Even so, we may still have occlusion errors due to ray samples that pass further from our viewpoint than from some occluding object. Since our methods for reducing such artifacts depend on our drawing technique, we will discuss them at the end of this section, also.

The ray colors handed to us by the server are floating point, world radiance units. Since our display is not capable of reproducing the dynamic range of a real environment, we need to map these values to lie within our CRT monitor's gamut. To accomplish this, we use a fast integer implementation of the visibility matching tone reproduction operator developed by Larson et al [Larson97]. We keep a record of image brightnesses (log luminances), and use them to compute a new brightness mapping each time the screen is redrawn. (The user may also force a redraw at any time.) Our tone mapping has two variations. The first variation computes the optimal colors for display, so as to maximize visible areas on the screen by minimizing contrast compression of large regions. The second variation attempts to reproduce human visual sensitivity by adjusting both contrast and color visibility according to the eye's local adaptation. The latter technique produces dim, weakly colored simulations of dark environments, which shows what would be visible to a human observer in the real world. For well-lit environments, the two techniques produce similar results. In fact, the human contrast algorithm may actually produce brighter, more visible displays for outdoor simulations, since the eye's sensitivity is greater outdoors than under typical CRT viewing conditions.

To minimize delays in redrawing the screen, we keep a cache of recently sent rays in a list, which is referred to by leaf nodes in our quadtree. During view movement, we rebuild our quadtree at reduced resolution from this cache of values, rather than asking the server to resend everything over the interprocess connection. We also draw the section grids over the displayed image, for better feedback and as a reference frame when the cache has nothing to show for a particular view. Once the new view is selected by releasing the mouse, we draw a full resolution quadtree of whatever leaves we have, and ask the server to send over rays for beams that have changed since the previous view. For small view changes, selective updating usually results in considerable savings. This is similar to the frameless rendering idea proposed by Bishop et al [Bishop94].

We can also minimize processing time by storing the exact information we need for each leaf. The stored leaf samples and their byte requirements are shown in Table 2. The world coordinate takes the most space, but encoding it and decoding it would add too much to our redisplay time. We can, however, efficiently encode the world direction, which we need for determining the best ray for each terminal node. We do this by encoding the two smaller vector components in 14 bits each, plus the sign bit for the largest component. (One bit of 32 is wasted.) The encoded result allows us to compare vector angles with a tolerance of about 15 seconds of arc. (A typical pixel covers over 100 arc seconds.)

Stored value	Size
world coordinate	12 bytes
encoded world direction	4 bytes
encoded world brightness	2 bytes
display chromaticity	3 bytes
current tone-mapped color	3 bytes

Table 2. Recorded values for each leaf in the display quadtree.

The storage requirement for our quadtree is 24 bytes per leaf, plus 4 bytes for each node index, which is multiplied by the allocated cache size. This is usually equal to the display resolution divided by three or four, since the target resolution is generally smaller than the display resolution, and we don't need to keep much more than one screenful of cached leaves around. For a 1280×1024 display, this adds up to about 16 Mbytes of main memory needed by the display process.

As we mentioned before, the actual image representation of our quadtree is different in our two driver implementations. Under X11, it is easiest to draw rectangles, whereas OpenGL allows us to be a little more creative. We discuss the ramifications in the following two subsections.

2.4.3.1 2D driver (X11)

Since the display model in X11 is strictly two-dimensional, it is easiest to just draw our quadtree leaves as rectangles. Unfortunately, we cannot rely on every leaf node in the quadtree to have a corresponding value. Since our ray samples are randomly distributed over the image, some quadtree nodes may contain only two or three values. This situation is illustrated in Figure 6.

Besides filling in missing values, we also want to reduce visible occlusion errors in our displayed image, which can occur when a ray passes close by an object, but not as close by our view point. Figure 7 shows one such example. To minimize visible artifacts, we look at all leaf nodes at each quadtree level. If any leaf is further than some fraction of the closest leaf's distance, we do not use that value, but treat it instead as an empty quadrant. In our tests, we found

a depth epsilon of 5% to work reasonably well. Because the closer leaf values tend to be in the majority at nodes where occlusion errors occur, this technique reduces the severity of artifacts.

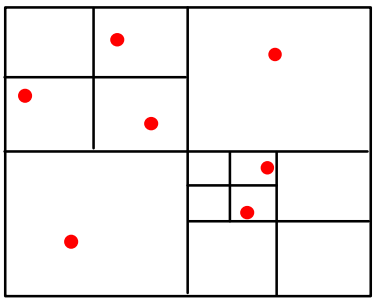

Figure 6. A random sampling pattern stored in a quadtree with no more than one sample per leaf node.

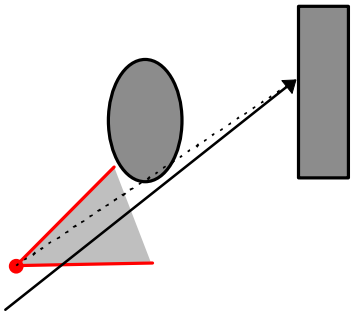

Figure 7. Occlusion error caused by using a ray not passing through the view point. The dotted line shows the intersected world coordinate is not actually visible from this position.

How do we fill in missing quadrants due to random sampling and occlusion culling? As we draw our valid leaves, we keep a sum of their values. For any subtrees we draw, we get the returned average and add it to our sum. Then, we go back and fill in any missing quadrants with the averaged color. In almost all cases, this produces a satisfactory image.

Figure 8a shows a low-resolution image produced without gap filling or occlusion detection. Occlusion errors can be seen as white pixels near the podium silhouette. Figure 8b shows the same image drawn by our X11 driver. Note how there are still some occlusion errors visible at this resolution. Using our reduction technique, roughly 25% of the occlusion errors present in the quadtree are actually drawn.

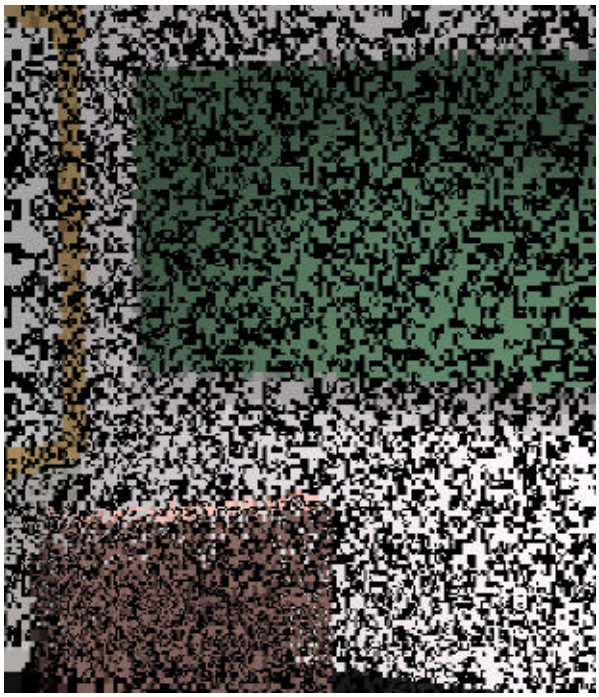

Figure 8a. An image drawn of the quadtree leaves that contain some low-resolution sample values, without considering occlusion.

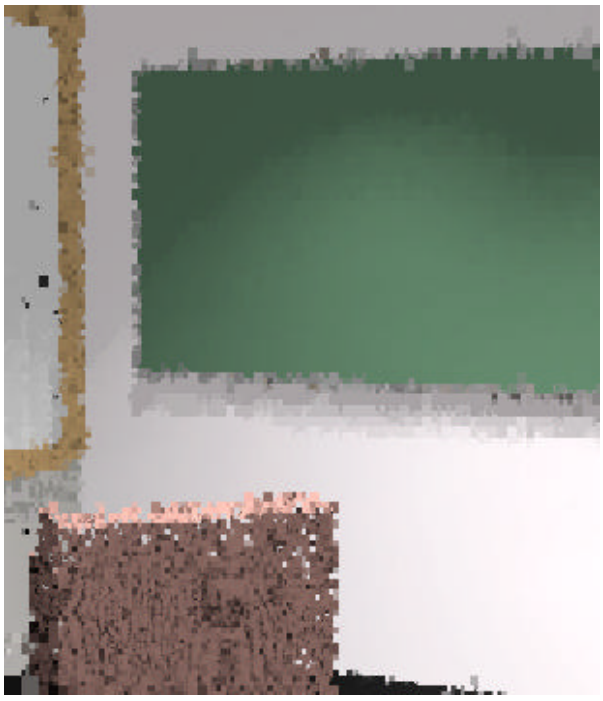

Figure 8b. An image of the same data produced by our X11 driver, with gap-filling and occlusion error reduction.

2.4.3.2 Depth-buffered triangles (OpenGL)

Using 3D rendering hardware, we can display a better representation of our quadtree. Rather than drawing the quadrants our samples just happened to land in, we can draw Dirichlet domains around each computed point. A Dirichlet domain, also known as a Voronoi polygon, is the area on a plane that is closer to a particular point than it is to any other point. As a piecewise constant approximation,

the Dirichlet map (i.e., Voronoi diagram) is the least biased representation possible.

Creating a progressive Dirichlet map is as simple as drawing a flat-shaded, depth-buffered cone for every sample point [Haeberli90]. Each cone is drawn as seen looking from directly above, with the apex at the sample point and the base past the edges of the image. If it were fully shaded with shadows from some angled light source, we would see something much like the range of very round mountains shown in Figure 9. Since we draw each cone flat-shaded, what we see instead is the Dirichlet domain around each sample.

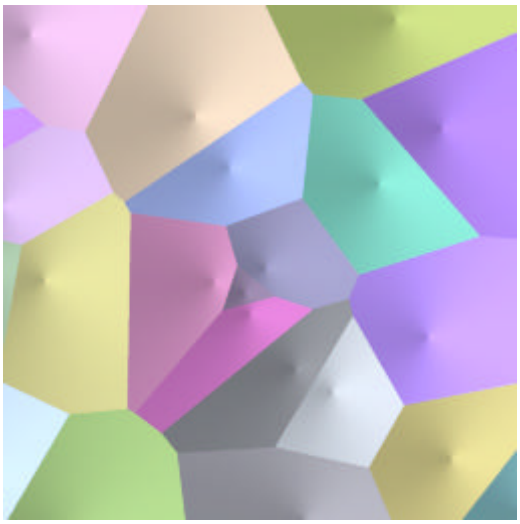

Figure 9. Drawing cones as seen from above results in a Dirichlet domain around each apex [Haeberli90].

Drawing cones the size of the screen for every sample point is slow and wasteful. Since we know the rough sample coverage from our quadtree, we can limit each cone radius to the width of its corresponding quadrant. The amount of overlap will be minimal, but it is still rather expensive to render high-quality cones. We could precompute a depth map for the cones and render using the stencil buffer, but it is cheaper and faster to approximate each cone as a triangle fan and render them that way. In our implementation, we vary the resolution of the triangle fan depending on the target cone radius, such that a full-screen cone is drawn with 32 base vertices, and the smallest cone is drawn as a four-sided pyramid.

Figure 10a shows the same sample data as before, rendered with the simple cone-drawing algorithm just described. Note that there are some missing pixels, though not nearly as many as the unfilled quadtree in Figure 8a. However, the original occlusion errors are back.

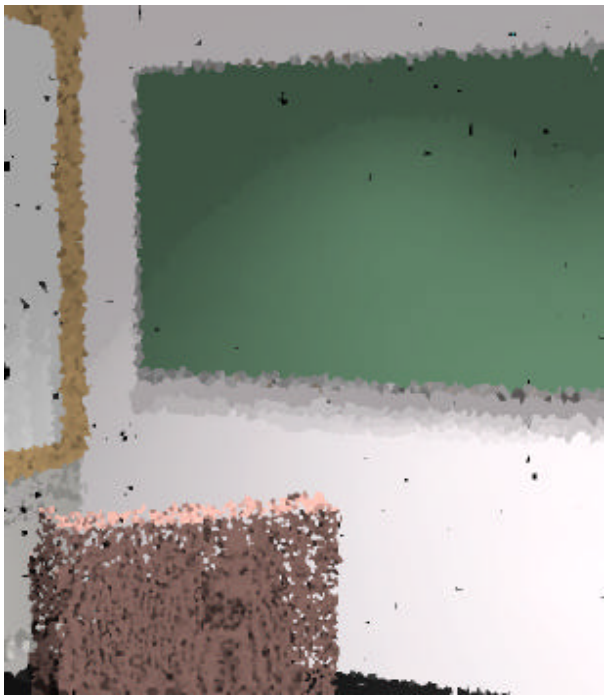

Figure 10a. Our scene samples rendered with cones to create Dirichlet domains.

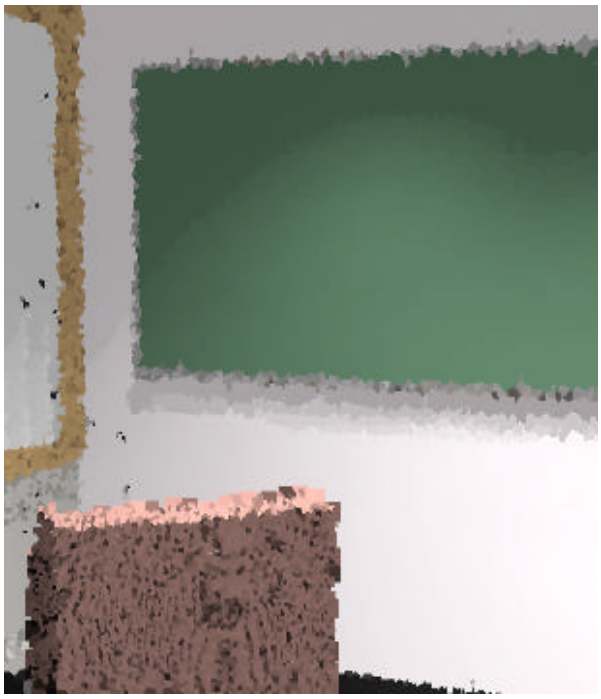

Figure 10b. The same samples rendered using variable cone heights and background filling.

We could employ the same algorithm used by the X11 driver to reduce the occlusion artifacts, but there is a better way. Instead of drawing all the cones at the same height to create true Dirichlet domains, we can vary the cone height based on the sample depth value. By varying the heights as little as 6% over the range of sample depths, we can eliminate the vast majority of occlusion errors. We do introduce a new artifact, however, which is a slight growth in some object silhouettes, but this is visually much less objectionable than disintegrating edges. Figure 10b shows our samples rendered with this algorithm, and with empty

quadtree leaves filled in using average-color, minimum-height cones. Silhouette growth is evident around the podium, but overall this is a very good impression of our scene.

2.5 Process Coordination

Good coordination between the holodeck server, the ray calculation, 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:

- 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.
- The display process waits for input from the holodeck server and the user with equal priority, updating the image before each call to *select*.
- 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.
- 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 the Appendix.

3. Results

Figure 11 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 OBSTRUCTIONS variable has been set to True. Figure 12a shows a view of the holodeck generated from scratch on a single processor SGI O2 in about ten seconds. Figure 12b shows the same view after a minute.

Figure 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. By leaving the OBSTRUCTIONS variable unset, the calculation will begin each ray at some random point within a section. In some cases, the ray may intersect interior geometry, but since it is mostly transparent (i.e., glass) or near the section boundaries, this will not interfere much with our visibility. In the hallway itself, the user will 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. Figure 14 shows a very impressionist 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.

Figure 15a shows a terminal in the end office with a poor task lighting arrangement. We can't really tell how bad it is, though, until we move our view point to that shown in Figure 15b, where the specular reflection becomes more visible. For a rotation this large (about 40°), our display process ignores its 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.

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.

Figure 16 shows an interactive sequence taken from a walkthrough of the daylight OEP office. Figure 16a uses samples taken from the precomputed holodeck. Figure 16b was captured during movement to a new view. Figure 16c is what we see immediately after releasing the mouse; since the view has rotated more than 20°, the display process ignores most of its cache. Half a second later, the server has retrieved some better samples from the holodeck file and we see what is shown in Figure 16d.

What happens when we take a vantage point that is outside all sections, with substantial portions of the view intersecting no section at all? Figure 17 shows a holodeck rendering of a cabin model, where we've moved our position so

we have both the living room's section and the bedroom's section below and in front of us. What we see are the rays that begin at the top or outside of each section and intersect the wall or floor below. Since the sections are within the wall and ceiling boundaries, that geometry is 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. The bathroom's section, which lies behind the bedroom, is not visible in this rendering, because we are not close enough to it for the display process to consider it interesting.

Figure 18a 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 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 image shown in Figure 18b. Although the reflections are now sharp, we see some other peculiarities around the mirror frame. These are occlusion errors due to the discrepancy between the mirror frame's distance and the reported distance of the reflection inside, which we didn't see in Figure 18a because the mirror and the frame were reported as having about the same distance. Both of these images would eventually converge, and in the end would resemble each other closely. The reflection with VDISTANCE set to False would never be quite as sharp, however, which is why we set it to True in this environment. In cases where the reflecting objects are very small, their breakup can be quite annoying, which is one reason we might want to set VDISTANCE to False.

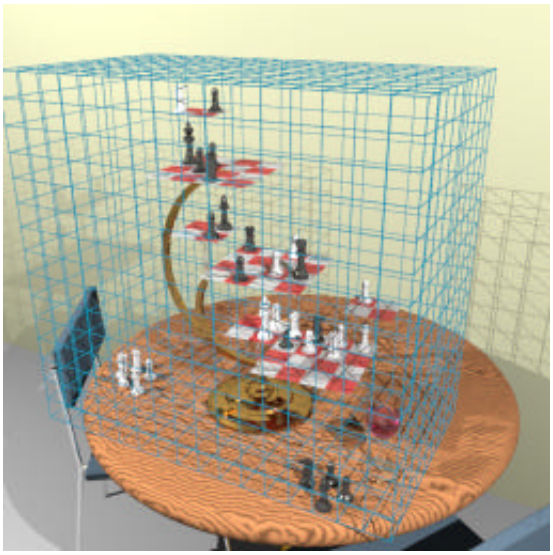

Figure 11. A grid for a holodeck section that is meant to be viewed from the exterior.

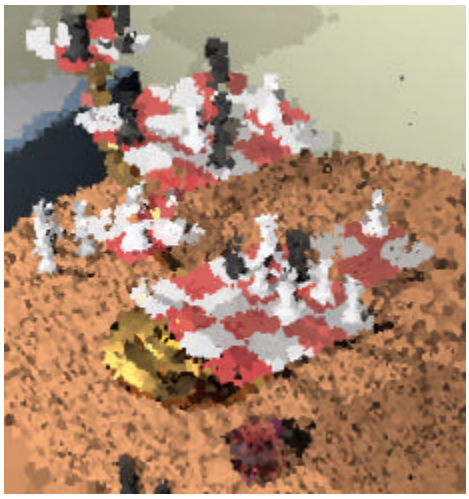

Figure 12a. An interactive rendering of the chess scene after 10 seconds on an SGI O2.

Figure 12b. The same view after 1 minute.

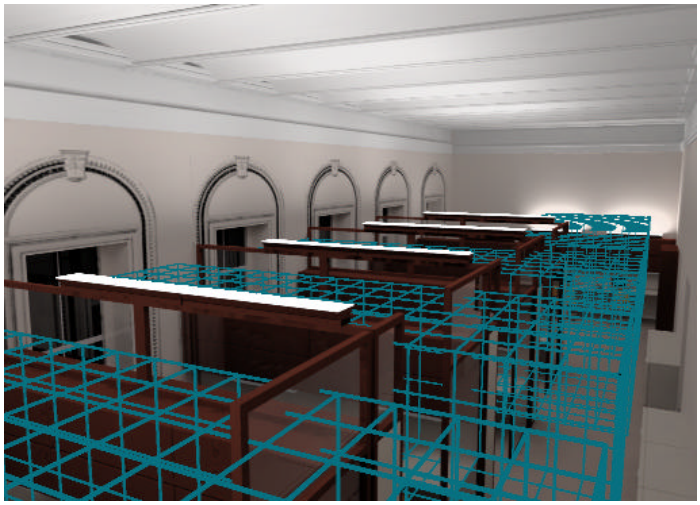

Figure 13. An overview of the OEP office space with multiple holodeck sections.

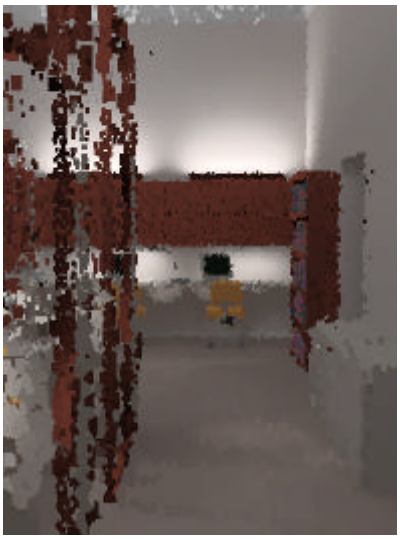

Figure 14. A very low resolution view of the OEP hallway, taken from between two sections.

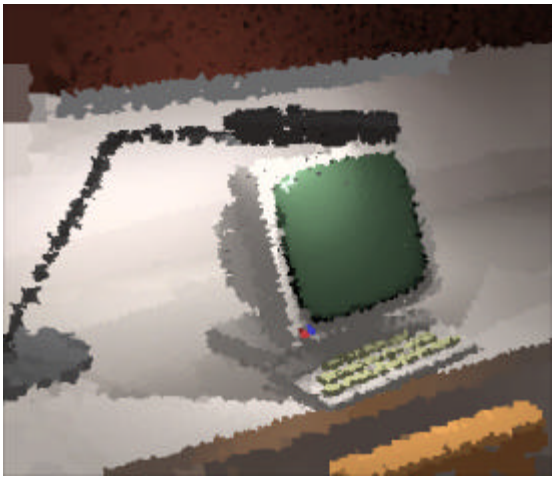

Figure 15a. A close-up of a workspace terminal with poor task lighting.

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

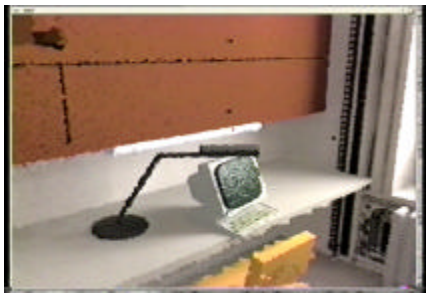

Figure 16a. A daylight version of the space precomputed in 20 hours on 14 Onyx processors.

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

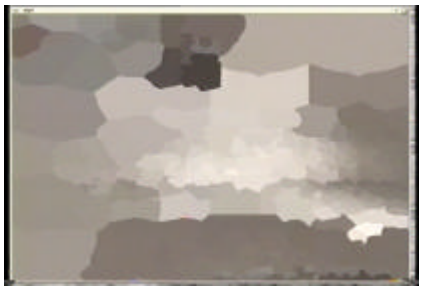

Figure 16c. The image displayed immediately after releasing the mouse. We have moved so much that the display cache contains few useful samples.

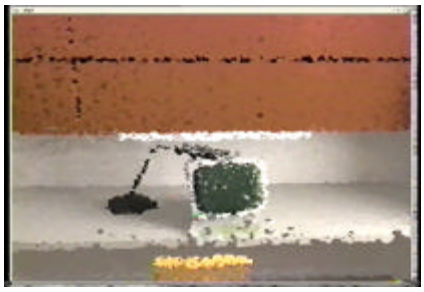

Figure 16d. The same view after half a second, during which time the server has retrieved some more relevant samples from the holodeck file.

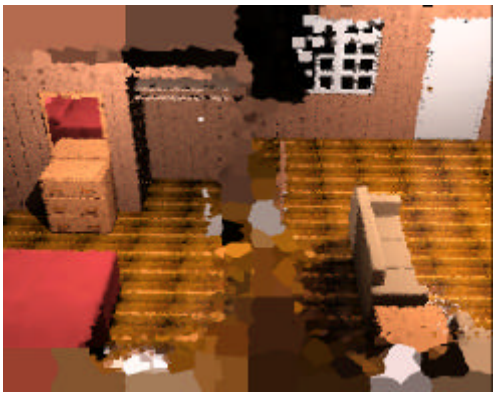

Figure 17. A view from above and between two holodeck sections, with invisible regions.

Figure 18a. A low resolution rendering of a bathroom mirror with VDISTANCE=False, showing the resulting lack of definition in the reflection.

Figure 18b. By setting VDISTANCE=True, we get a sharper image in our mirror, but the mirror's edges begin to break up a bit.

4. Conclusions

In this paper, we have presented a new method for demand-driven rendering of a 4-dimensional light field, which we store in a dynamic *holodeck* data structure that facilitates the rapid generation of new views. Compared to other image-based rendering techniques, our approach avoids the need for any preprocessing or development step, and permits rays to be calculated and stored at variable densities. Since rays are bundled on disk and cached in memory, memory size does not limit a light field's size or resolution, though we can take advantage of more memory when available.

Unlike most IBR methods, we expect to be given the world intersection point and radiance value for each sample, which is why we employ a physically-based ray tracing calculation to generate our data. With this information, we can produce more accurate displays that better represent what a person would actually see in a real scene. Using a dynamic implementation of a visibility preserving tone mapping function, we display our world radiances in a way that accounts for local adaptation as well as human color and contrast sensitivity. This correspondence is critical for reproducing visibility and visual comfort in design and training environments. Without it, there is no way to tell if the objects visible on the screen would be visible in real life, or vice versa.

Our present resampling techniques for displaying ray values are rather crude. It might be better to use a piecewise linear representation, such as Gouraud-shaded triangles, rather than our current piecewise constant approximation. However, a disadvantage to a "properly filtered" low-resolution image being displayed on a high-resolution monitor is that the human eye needs high frequency data to stay focused. An ideal solution would use pupil-tracking to follow the user's view center, keeping high frequency data in the foveal region, and allowing the rest of the image resolution to fall off according to the off-axis acuity function.

There are many avenues open for future exploration of holodeck rendering. The near future might include writing display drivers for stereo monitors, head-mounted displays, and CAVEs [Cruz-Neira93], trying the ray tracing engine out on a massively-parallel processor such as the Cray T3E, or using measurements such as stereo range data and radiance maps [Debevec96] [Debevec97] to create holodeck backgrounds for virtual worlds. There are also a few obvious optimizations we have not yet tried. One is to prefetch beams based on a user's current trajectory, similar to [Funkhouser93]. Another is to perform adaptive sampling based on beam variance, though doing this right is tricky [Kirk91]. A third is to avoid resampling distant geometry in parallel beams, since their radiance function varies only with angle, not holodeck position. Also, we might investigate quick ways to "compand" beams stored on disk. Finally, we would like to explore new ways of representing holographic information, such as time-varying data for animation, and material data to facilitate mutual illumination of local objects.

5. References

[Bishop94] Bishop, Gary, Henry Fuchs, Leonard McMillan, Elen Scher Zagier, "Frameless Rendering: Double Buffering Considered Harmful," *Computer Graphics Proceedings, Annual Conference Series*, 1994.

[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, Annual Conference Series*, 1993.

[Debevec96] Debevec, Paul, Camillo Taylor, Jitendra Malik, "Modeling and Rendering Architecture from Photographs: A hybrid geometry- and image-based approach," *Computer Graphics Proceedings, Annual Conference Series*, 1996.

[Debevec97] Debevec, Paul, Jitendra Malik, "Recovering High Dynamic Range Radiance Maps from Photographs," *Computer Graphics Proceedings, Annual Conference Series*, 1997.

[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, Annual Conference Series*, 1993.

[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, Annual Conference Series*, 1996.

[Gortler96] Gortler, Steven, Radek Grzeszczuk, Richard Szeliski, Michael Cohen, "The Lumigraph," *Computer Graphics Proceedings, Annual Conference Series*, 1996.

[Painter89] Painter, James and Kenneth Sloan, "Antialiased Ray Tracing by Adaptive Progressive Refinement," *Computer Graphics*, 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.

[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.

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

[Ward88] Ward, Greg, Francis Rubinstein, Robert Clear, "A Ray Tracing Solution for Diffuse Interreflection," *Computer Graphics*, 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, Annual Conference Series*, July 1994.

6. Appendix

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

1. The user starts the program with two **rtrace** processes and an X11 driver.
2. The holodeck server opens the existing holodeck, 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.