1998

Evaluation of subdivision methods used in octree ray tracing algorithms

Scott Brown

Rochester Institute of Technology
Computer Science Department

Evaluation of Subdivision Methods used in Octree Ray Tracing Algorithms

by
Scott D. Brown

A thesis, submitted to
The Faculty of the Computer Science Department
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by: _____
Professor Nan Schaller

_____
Professor John Schott

_____
Professor Stanislaw Radziszowski

February 12, 1998

Evaluation of Subdivision Methods used in Octree Ray Tracing Algorithms

Scott D. Brown

ABSTRACT

Key words: ray tracing optimization, non-uniform spatial subdivision, octree, octtree, octree subdivision methods.

The non-uniform spatial subdivision technique refined by Andrew Glassner [Glassner 1984] minimizes facet intersection tests *(i)* by automatically generating a density dependent spatial hierarchy of facet regions (called an *octree*) and *(ii)* by only testing facets in regions along the path of the ray. Past research has addressed optimization of the octree ray tracing process by separately improving both the octree traversal method and facet intersection algorithms. This author attempted to further improve the overall approach by attempting to identify new octree construction methods that would decrease the number of traversals required to render the scene.

This research focused on the subdivision technique used in constructing the octree. The conventional Glassner algorithm utilizes cubic octants that can result in a large population of empty octants when rendering scenes containing localized regions with high facet density. Sparse octrees (containing significant numbers of empty octants) were believed to hinder performance of the facet traversal algorithm. As an alternative to the conventional cubic algorithm, the performance benefits of non-cubic octants were investigated. Octrees constructed with "rectangular" octants which more closely bound the scene were tested as one alternative to the cubic octants method. As a second alternative, this author proposed and implemented an "ideal-cut" subdivision algorithm that subdivides the parent octant through the mean location of the facets contained in the octant.

For the scenes tested, the "conventional" cubic algorithm was shown to perform better than either alternative method, although, it was also shown to suffer from memory and run-time explosions on some scenes. The "rectangular" octant algorithm consistently approached the run-times produced by the "cubic" method. Since, the "rectangular" method defaults to the "cubic" method for scenes with 1:1:1 aspect ratios, the "rectangular" method must be considered as a reasonable alternative in rendering applications. The "ideal-cut" subdivision algorithm was shown to minimize the empty octants and overall octree depth, however, an increased number of facet intersection tests were required. In an attempt to identify the algorithm or scene characteristics that gave rise to the variation in algorithm performance, an extensive correlation analysis was performed. However, an *a-priori* scene characteristic to automatically select the most efficient algorithm was not identified.

Evaluation of Subdivision Methods used in Octree Ray Tracing Algorithms

I, Scott D. Brown, hereby **grant permission** to the Wallace Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Scott D. Brown — Date: 2/12/98

Acknowledgments

I wish to thank Nan Schaller for being my advisor during the course of this Thesis and the Independent Study that started this whole project, but mostly for her patience to comment on the many revisions of this document. I also wish to thank Stanislaw Radziszowski for bringing his knowledge of computational complexity to a computer graphics related thesis.

My thanks also go to Pat Fleckenstein for helping me debug my original code.

Finally, my sincerest thanks to John Schott for supporting my efforts to complete this degree and for his participation in my Thesis Defense.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Background

1.1 Ray Tracing Optimizations

Ray tracing is a popular rendering method because it simulates the physics of image formation: rays cast from the simulated camera find the sources of photons that would enter a real camera. Using simple vector mathematics, sources of reflections and shadows can also be identified and incorporated. In a minimal algorithm, facet intersection tests must be performed on every object in the scene to correctly identify the closest surface along the the ray. For large scenes, this linear search becomes computationally significant. As a result, much of the research in the ray tracing community is focused on strategies to minimize the number of intersection tests required per ray trace.

A multi-level bounding box algorithm is commonly utilized to minimize the number of facet intersection tests (see Figure 1.1) [Clark 1976, Rubin and Whitted 1980, Weghorst *et al.* 1984]. The ray being traced is tested for intersection with all of the bounding volumes at the top of the hierarchy. If the ray intersects a volume, then further intersection tests must be performed on the sub-volumes or facets contained by the volume. If the ray does not intersect the volume, then it can be assumed that intersection with any facets within the volume is impossible, and those tests can be avoided. The burden of constructing the bounding box hierarchy is usually placed on the user, requiring additional planning and effort to create scenes.

Although this strategy does minimize intersection tests when compared to the global search approach, it has several limitations:

- The user may define "poor" bounding boxes that contain large numbers of facets or overlap other boxes, resulting in less than ideal performance. Automated methods can suffer from similar problems.

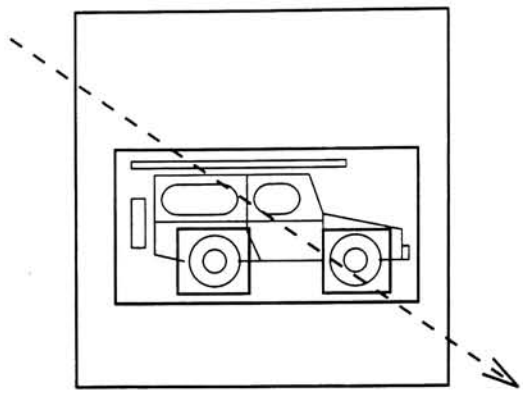

Figure 1.1: A 2-dimensional example of a user defined bounding box hierarchy and a ray being traced through the scene.

- The order that bounding box intersection tests are performed is not optimal. Because there is not a spatial ranking based on distance from the ray origin, all boxes must be checked at a given level in the hierarchy.

To avoid these limitations, a *non-uniform spatial subdivision* [Glassner 1984] algorithm can be used. This algorithm systematically subdivides the scene into axis aligned boxes or *octants* based on facet density - regions containing more facets are subdivided into smaller octants to minimize the number of intersection tests required per octant (see Figure 1.2). The resulting data structure is a spatial hierarchy of bounding boxes, called an *octree*. The depth of the octree is limited only by the amount of physical memory. However, as it will be described later, the depth of the octree does effect the computational efficiency of the algorithm.

In general, the octree algorithm is more efficient than the user defined hierarchy algorithm as it addresses many limitations of the later:

- The construction of the octree is automated, and requires minimal overhead at run-time rather than during the scene construction process.
- The number of facets contained in each octant is bounded by a user defined threshold and octants never overlap.
- Octants are progressively examined based on distance from the origin of the ray. Once an intersection has been identified within an octant, the search can be terminated.

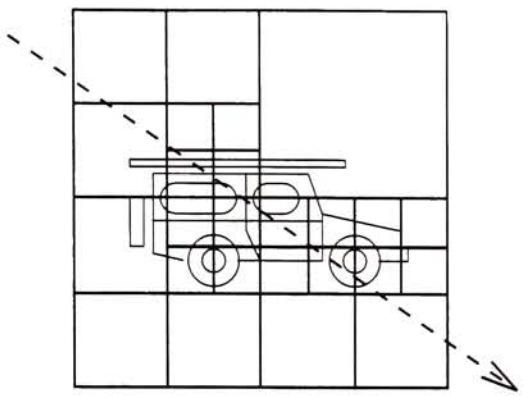

Figure 1.2: A 2-dimensional example of spatial subdivision based on facet density and a ray being traced through the scene.

1.2 Octree Creation and Ray Tracing

The spatial subdivision of the scene into an octree is the required preprocessing stage that makes the run-time tracing of the scene more efficient than many conventional ray tracing techniques. Upon completion of the scene subdivision phase, each leaf node or *leaf octant* in the octree will contain a list of all facets that intersect the space defined by that octant. The algorithm for building the octree is outlined below and is illustrated in Figure 1.3:

- The base of the octree, or the *world octant*, is constructed to bound all of the objects in the scene.
- The octree is created by adding each facet in the scene to the octree. Starting with the *world octant*, each facet is tested against each sub-octant for intersection. Each intersected sub-octant then repeats this intersection test for its sub-octants in a recursive fashion. The recursion terminates when a *leaf octant* is found, at which point a reference to the facet is added to that octant's facet list. All sub-octants of an octant intersected by the facet must be examined since the facet may extend over several branches of the octree.
- If the addition of a facet to a leaf octant exceeds the user defined subdivision threshold then that octant is split. To split an octant, eight new sub-octants are allocated and the facets contained by the original octant are sorted into each of the new sub-octants. The splitting of leaf octants continues until all octants are below the subdivision threshold.
- Facets are added in this manner until all the facets in the scene are loaded into the octree.

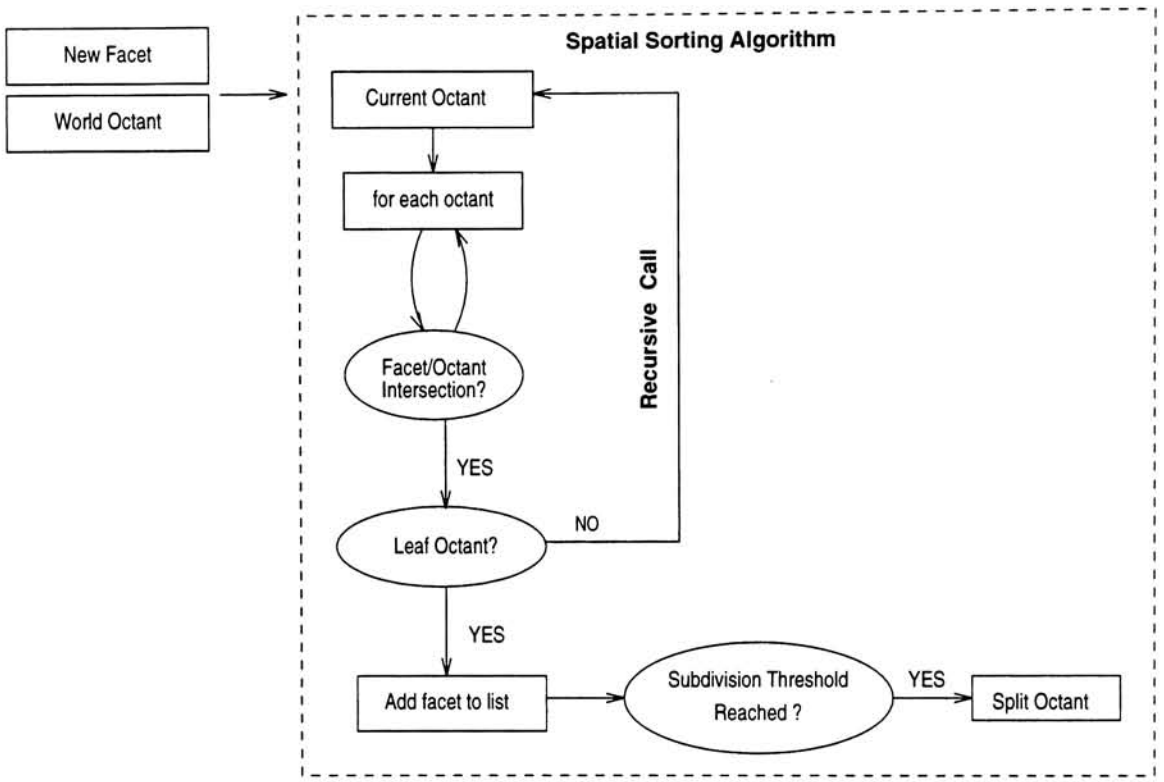

Figure 1.3: Creating an octree for efficient ray tracing.

Once the octree has been constructed for a particular scene, the procedure for tracing a ray proceeds as follows (see Figure 1.4):

- If a starting leaf octant is not provided, then the first leaf octant along the path of the ray is used as the starting octant. The ray is tested for intersection (a hit) with all the facets within an octant. To identify the closest hit to the ray origin, all hits are stored in a list sorted by distance from the ray origin. In the event that no intersections are found within the current octant, the next octant along the path must be determined.
- To find the next octant along the ray, a search point is computed in the neighboring octant along the path of the ray. First, a ray/box intersection test is performed to find the exit point from the current octant [Haines 1991, Kay 1986]. The search point is half the size of the smallest octant perpendicular to the exited face of the current octant. This point is guaranteed to be within the next octant without skipping the smallest octant. The search point is supplied to the octree search function and the octant containing the point is identified as the next octant along the ray.
- The process repeats itself until a facet is intersected or the ray leaves the world octant.

The design document of the octree ray tracing algorithm used in this research (*libRT*) is included as Appendix A. For more details on the mechanisms involved with octree construction and ray tracing this section should be consulted.

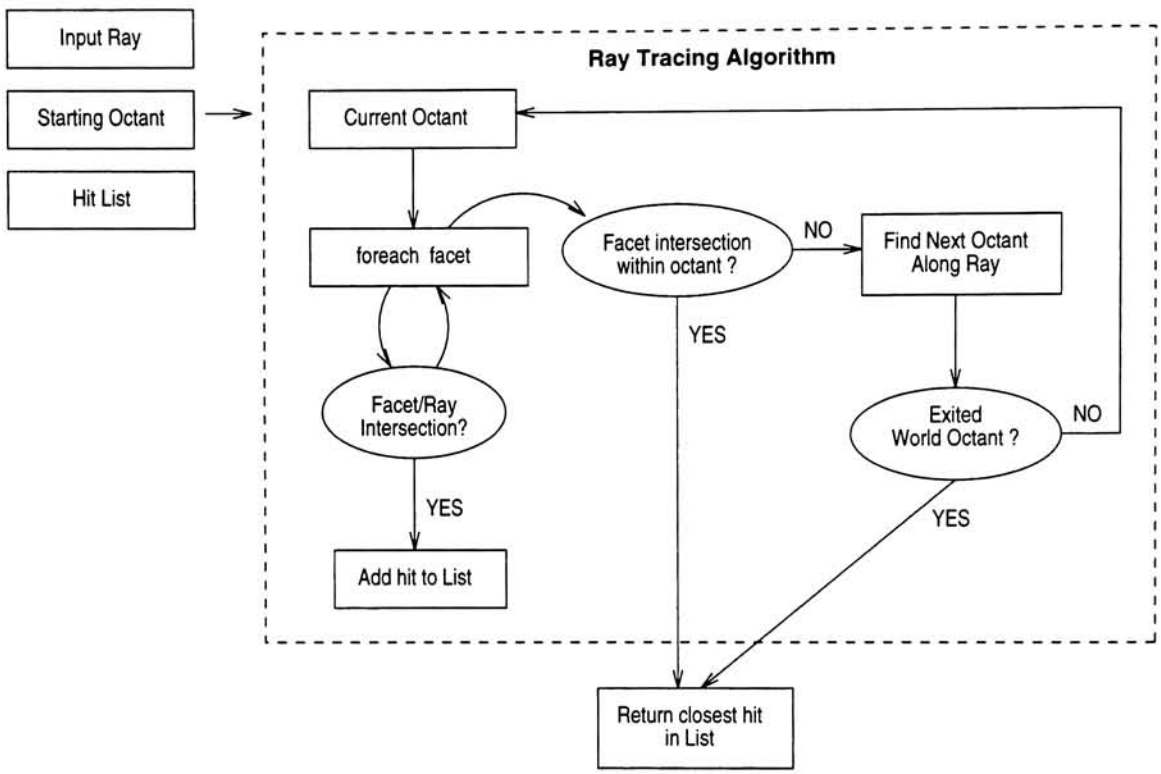

Figure 1.4: Utilizing an octree to perform ray tracing.

1.3 Preliminary Observations

The performance of the octree ray tracing method is dependent upon two primary operations: *(i)* octree traversal (moving from the current octant to the next along the path of the ray) and *(ii)* facet intersection tests. Since facet intersection algorithms have been heavily investigated [Badouel 1990, Haines 1991], this research focuses on improving the octree traversal aspect of the algorithm. Specifically, this author attempted to identify new octree construction methods that would decrease the number of traversals required to render the scene.

The conventional octree subdivision method splits the octant into eight (8) sub-octants of equal size. If the facets within an octant are extremely localized, one or more of the sub-octants will be over populated and must be split again, further increasing the octree depth. Since the octant traversal algorithm must navigate a hierarchical tree, deeper octrees result in longer traversal times. When over populated octants are generated, the remaining sub-octants are usually sparsely populated or possibly empty. This results in an increased probability of the search finding an empty octant (containing no facets to intersect the ray) and requiring another search to be started.

Figure 1.5 illustrates each of these observations. Case A illustrates a scenario where the conventional cubic octant method cuts the parent octant through the center. The ray tracing algorithm searches and identifies octant #1 as the next octant along the path of the ray, finds it empty, performs another search to find octant #2, and then performs five facet intersection tests to isolate the closest facet. In Case B, the parent octant has been cut in a slightly different manner. In this case, only one octant search is required and only three facets are tested for intersection before the hit facet is isolated. These additional computations were avoided because the parent octant in Case B was subdivided through the average facet location rather than through the middle of the octant. If the sub-octants used in this example were constructed using the Case B methodology, fewer total facets would be examined (since the first octant in Case B contains fewer facets than the second in Case A) and the time associated with the first octant traversal using a more conventional subdivision algorithm would be skipped.

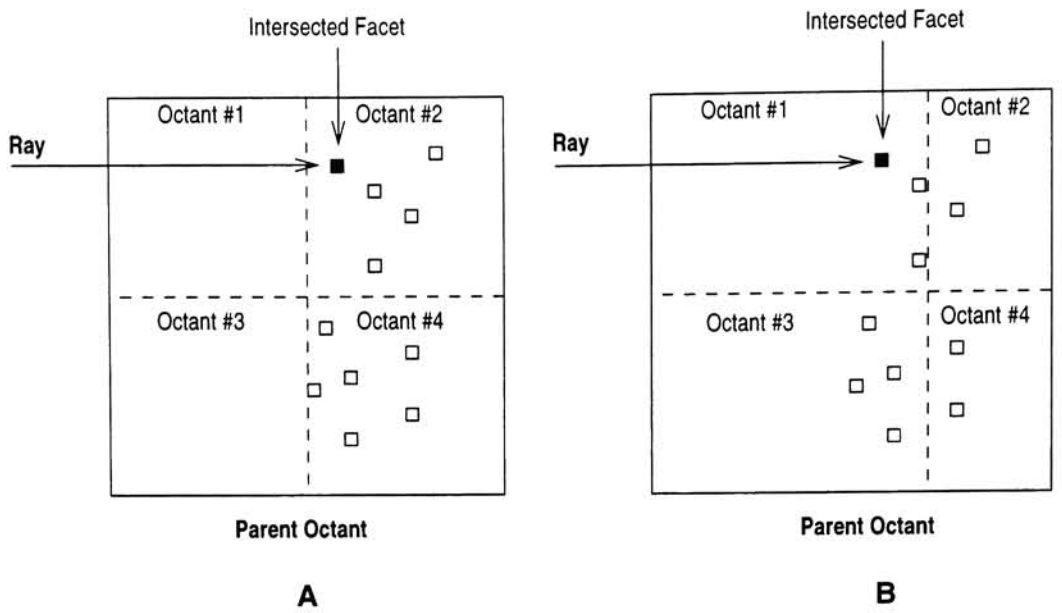

Figure 1.5: Case A requires two octants to be traversed and five facets to be checked as opposed to one octant and three facets in Case B.

This proposed subdivision method does not come free of cost. The same facet population is illustrated again in Figure 1.6, however, the ray to be traced comes from a different direction. In this scenario, some facets are tested in Case B that would otherwise be avoided in Case A. However, a change in performance is only noticed when starting in octants #1 and #3 with rays from these vertical directions. This does not affect the efficiency when starting with either octant #2 or #4 from any direction. Preliminary data showed that facet intersection tests can be 3 to 10 times faster than octant traversal (depending on the depth of the octree). Therefore, the possibility increasing overall run-times by performing additional facet intersection tests for a fraction of the possible ray tracing directions was believed to be minimal.

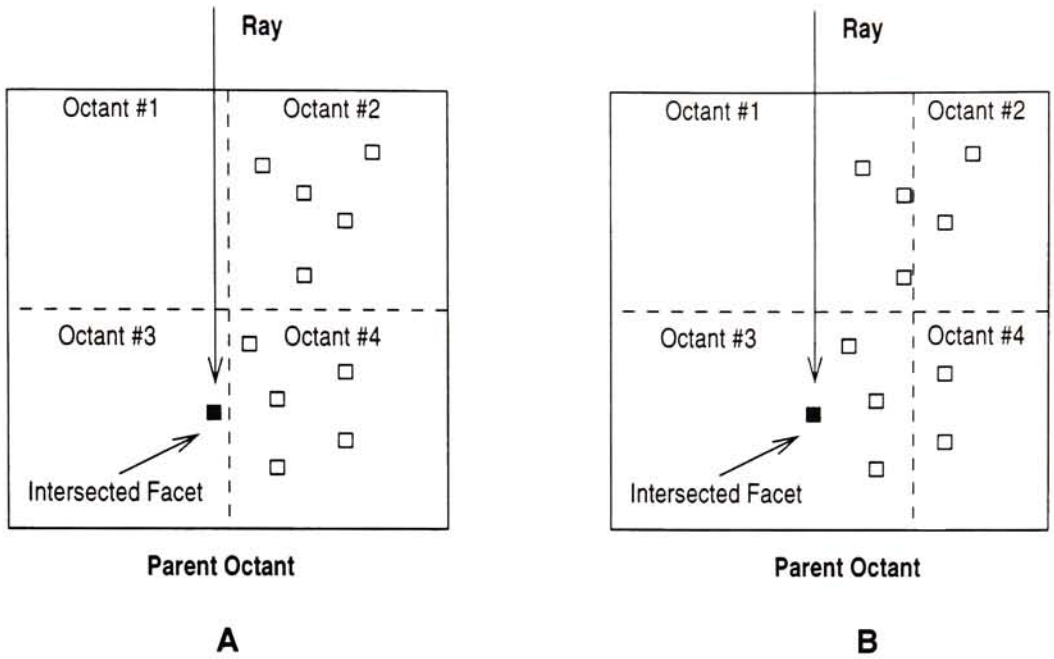

Figure 1.6: Additional facet intersection tests are performed in Case B that are avoided in Case A

Since the total number of facets is constant for a given scene, octrees using fewer octants will have a higher number of facets per octant. Although the average number of facet intersections per octant may increase slightly, it is possible to save enough time by performing fewer and faster octant traversals that the resulting overall run-time improves.

1.4 Statement of Hypothesis

This author hypothesized that a subdivision method that attempts to distribute the facets contained in an octant equally to the child octants may be more efficient. By doing so, it was believed that shallower octrees would be produced, decreasing the amount of time spent traversing the octree.

1.5 Summary of Octant Subdivision Methods

To test this hypothesis, two variations of the cubic octant style were implemented and tested. The example 2-dimensional facet distribution featured in Figure 1.7 - 1.9 is used to illustrate the potential storage and

traversal efficiency of the conventional algorithm and each subdivision variant.

1.5.1 The Conventional Algorithm

Figure 1.7 depicts the octree generated when using the conventional subdivision method described by Glassner in his original paper. The spatial subdivision of the scene begins by establishing a cube shaped octant that bounds the entire scene. Since this subdivision method bisects filled octants, all sub-octants derived from the bounding octant will also have a 1:1:1 aspect ratio.

For the facet distribution in Figure 1.7, a total of 10 octants (two of which are empty) are required using this subdivision method and a subdivision threshold of 8 facets.

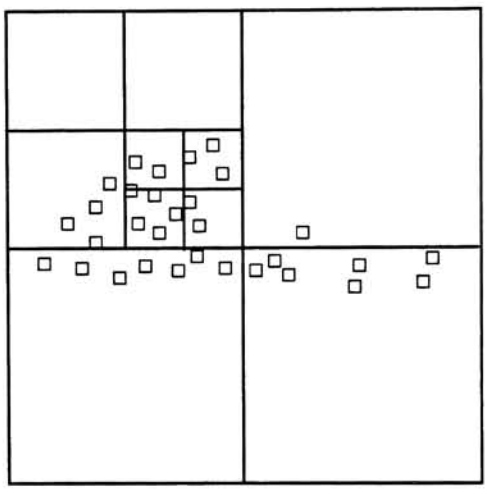

Figure 1.7: A 2-dimensional example of the conventional spatial subdivision utilizing cubic octants.

1.5.2 The Rectangular Octant Algorithm

The first variant uses non-cubic, or *rectangular* octants. In this implementation, the world octant is created to enclose the maximum extents of the scene. All derived sub-octants will feature the same aspect ratio as the bounding octant. Since the world octant tightly wraps the scene, the author hypothesized that any subsequent subdivisions will have a higher probability of separating localized concentrations of facets and would possibly avoid both overfilled and empty octants (see Figure 1.8).

For the facet distribution in Figure 1.8, a total of of 7 octants (one of which is empty) are required using this subdivision method and a subdivision threshold of 8 facets.

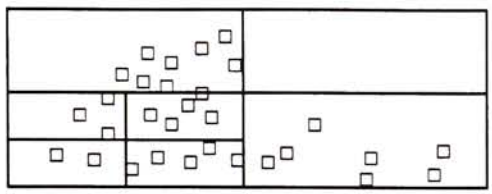

Figure 1.8: A 2-dimensional example of the spatial subdivision utilizing non-cubic octants.

1.5.3 The Ideal Octant Algorithm

The third variation splits the octant through the mean location of the facets contained by the octant, and is referred to as the *ideal-cut* algorithm. This algorithm requires maintaining the mean location of facets contained by a given octant as the octant is filled. To balance the population of facets in the sub-octants, the parent octant is divided through the average location of *all* the facets contained by the octant. Therefore, every facet must be sorted into the octants at a given level before subdivision can take place (see Figure 1.9).

The sorting approach used by the *ideal-cut* algorithm changed the design of the facet sorting aspect of octree construction since the octant cannot be properly cut until the mean of all the facets intersecting the octant is determined. In contrast, the previous two algorithms can split the octant once the octant has reached the subdivision threshold. The details of the octree creation using the *ideal-cut* algorithm is described in detail in Chapter 2.

For the facet distribution in Figure 1.9, a total of of 4 octants (none of which are empty) are required using this subdivision method and a subdivision threshold of 8 facets.

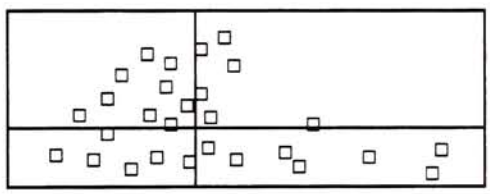

Figure 1.9: A 2-dimensional example of the spatial subdivision utilizing "ideal-cut" octants.

1.6 Statement of Work

1.6.1 Implementation

Although the octree ray tracing library had already been completed as part of another effort, additional modifications were made during the course of this research. Specifically, these modifications included:

- The addition of counters to keep track of *(i)* the total number of octants, *(ii)* the number of empty octants and *(iii)* the average number of facets per octant.
- Implementation of the "ideal-cut" subdivision method. This required keeping track of the dimensional means of facets intersecting each octant for use during octant subdivision, and implementing the new octree creation algorithm which utilizes a different sorting approach.
- Implementation of the pseudo-random scene generation tool to create test scenes (see Appendix C).

The details of the modifications required for this research are described in Chapter 2.

1.6.2 Data Creation

The modified subdivision methods were added to the *libRT* ray tracing library. The author currently has this library installed in the DIRSIG synthetic image generation model. DIRSIG is a high radiometric fidelity multi-spectral rendering environment that can require 100 - 150 rays to be traced for each rendered pixel [Schott et al.].

The three different octree algorithms (cubic octants, non-cubic octants, and ideal-cut octants) were each tested on nine (9) test scenes. Three (3) of the test scenes were statistically generated to test the various strengths of each subdivision algorithm. The remaining six (6) scenes were readily available DIRSIG scenes. A description and rendering of each scene is presented with the scene by scene analysis in Chapter 3.

1.6.3 Data Analysis

All of the collected data and resulting analysis are presented in Chapter 3. Important run-time values for each algorithm were extracted from the run-time profiler output for each scene in the test set. The values for each scene are compiled in Section 3.1 in tabular form. A summary of the general performance analysis for each algorithm an be found in Section 3.2. A complex correlation analysis of important run-time parameters was also performed. The results and discussion of this analysis is presented in Section 3.3.

Chapter 2

Implementation

The research in this effort addressed the performance of three octant subdivision methods. This chapter describes the implementation differences between the creation of an octree using either the "cubic" or "rectangular" subdivision method and the "ideal-cut" subdivision method. It should be pointed out that there are no differences between the various algorithms with respect to the ray tracing process.

2.1 The Conventional Subdivision Algorithm

At the core of the conventional "cubic" octree creation algorithm is a "spatial sorting" algorithm that adds a reference to any leaf octant that intersects a given facet. Pseudo-code for this recursive facet sorting routine is outlined below:

```
RT_Sort_Facet( RT_OCTANT *octant, RT_FACET *facet )
{
    if( Octant_Is_A_Leaf( octant )) {
        RT_Add_Facet_To_Octant( octant, facet )
    }
    else {
        foreach( suboctant ) {
            if( RT_Facet_Box_Intersect( suboctant, facet )) {
                RT_Sort_Facet( suboctant, facet )
            }
        }
    }
}
```

The routine `RT_Add_Facet_To_Octant` adds the facet reference to the current leaf octant and checks if the facet count for that octant exceeds the subdivision threshold. If it does, the octant is split into 8 sub-octants and all the facets in the original octant are re-sorted into the new sub-octants. The addition of facets to the octree then continues.

2.2 The Ideal Cut Subdivision Algorithm

The "ideal cut" subdivision method must utilize a slightly different approach to constructing the octree. To ideally cut the octant, the mean facet location *all* of the facets intersecting the octant be determined. Therefore, all of the facets intersecting an octant must be sorted into that octant before the octant can be split. In contrast, the "cubic" and "rectangular" subdivision methods can split an octant once the threshold is reached. (a detailed description of the conventional octree creation algorithm was presented in Section 1.2).

An important modification to the spatial sorting routine was to maintaining the mean location of all the facets contained in the octant. The computation of the mean location does *not* include facets that extend past the extents of the octant. This is because this algorithm attempts to split up localized facet concentrations *within* the octant.

The algorithm for the "ideal cut" subdivision method can be described by a simple recursive process. The process is seeded by adding all of the facets in the scene to the world octant.

- If the number of facets in the current octant *exceeds* the subdivision threshold, sub-octants are created that cut through the average location of the facets within the octant. The facet storage area in the sub-octants must be made large enough to hold all of the facets that intersected the parent octant (a worst case scenario). All of the facets are then sorted into the child octants, and the old facet list is de-allocated. This process is repeated for each sub-octant of the octant.
- If the number of facets in the current octant *does not* exceed the subdivision threshold, then the size of the facet list is changed to free up any unused memory from the initial conservative allocation. To do this, a new facet list for the octant is allocated that is large enough for the facets in the octant, the facet list is copied to it, and the old list is de-allocated.

The following section contains pseudo-code for the "ideal cut" subdivision algorithm previously described:

```
RT_Optimize_Octant( RT_OCTANT *octant )
{
    if( Octant_Is_Overfilled( octant )) {
        RT_Create_Sub_Octants( octant, octant->mean, octant->sub_octants )
        foreach( facet in octant->facet_list ) {
            RT_Sort_Facet( facet, octant );
        }
        Free_Facet_List( octant->facet_list );
        foreach( sub_octant in octant->sub_octants ) {
            RT_Optimize_Octant( sub_octant );
        }
    }
    else {
        New_Facet_List( octant->facet_count, new_facet_list )
        foreach( facet in octant->facet_list ){
            Add_Facet( facet, new_facet_list )
        }
        Free_Facet_List( octant->facet_list );
        octant->facet_list = new_facet_list;
    }
}
```

The actual source code (contained in the file `RT_Octant.c`) can be found in Appendix B.

Chapter 3

Results and Analysis

3.1 Test Results

The octree ray tracer was run on nine (9) test scenes to compile the results presented in this document. The first three (3) scenes consist of simple statistical distributions of facets generated by the `make_scene` utility (see Appendix C). These scenes were specifically constructed to isolate the differences in the three (3) octant subdivision methods being studied. These statistical scenes present a significant challenge to any ray tracing algorithm because they consist of extremely dense and uncorrelated fields of facets.

The remaining scenes were readily available, "real-world" scenes that this author feels represent the size and complexity of facetized scenes that may be of interest to people in the graphics field today.

The following sub-sections summarize the data collected for each scene. Each scene was run with each of the three (3) octant subdivision variants. Each section includes a brief description of the scene, an example image of the scene, and the results tabulated from the profiler output.

3.1.1 Statistical Scene #1

Description

The scene rendered in Figure 3.1 was specifically designed so that all three subdivision algorithms would perform similarly. The scene is a cube uniformly filled with 5,000 facets of various sizes and orientations.

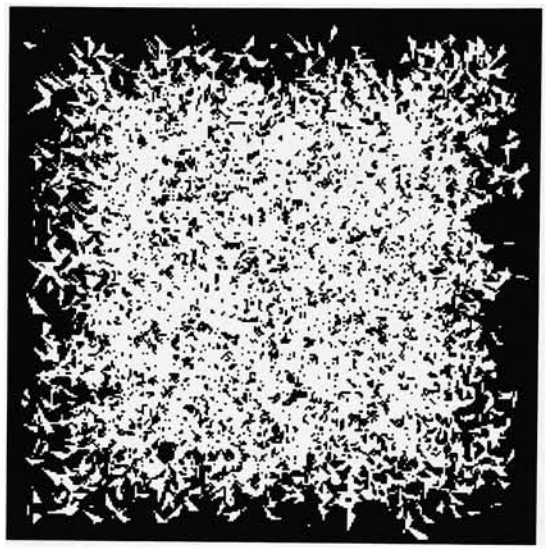

Figure 3.1: Image of the "Stat #1" scene.

This statistically generated scene was created using the following parameters extracted directly from the `make_scene` input file:

```
DIST_ENTRY_BEGIN
    TYPE = UNIFORM
    COUNT = 5000
    AVERAGE_SIZE = 500.0
    DELTA_SIZE = 250.0
    AVERAGE_X = 0.0
    DELTA_X = 10000.0
    AVERAGE_Y = 0.0
    DELTA_Y = 10000.0
    AVERAGE_Z = 0.0
    DELTA_Z = 10000.0
DIST_ENTRY_END
```

Total number of facets	10,000

Performance Metric	Octant Styles			Best Performance
	CUBE	RECT	IDEAL	
Octree build time	5.51	5.51	4.83	IDEAL
Total number of octants	2079	1855	1401	IDEAL
Number of empty octants	2	0	2	RECT
Fraction of empty octants	0.001	0.000	0.002	RECT
Number of octant searches	19,060,515	19,035,334	17,445,697	IDEAL
Octant search time	247.65	244.43	311.26	RECT
Number of facets tested	217,494,289	217,412,595	252,524,941	RECT
Facet test time	885.97	894.25	1043.80	CUBE
Total run time	1860.11	1860.14	2104.24	CUBE

Table 3.1: Summary of profiler output for the "Stat #1" scene.

Observations

Table 3.1 summarizes the performance of each algorithm for this scene. Since the scene consists of a square cube filled with facets, the world octant for all three algorithms should be the same size, thereby minimizing the differences between the CUBE and RECT algorithms. However, the size of the octrees for these two algorithms is surprisingly different. The world octant size was verified to be only slightly different because the extents of the randomly generated facet population was not exactly a cube. This small difference was found to be responsible for this rather large difference in octree size. Although the octree sizes were very different, the total time spent on octree traversal only changed by approximately 1%. For this scene, the CUBE algorithm had the best run-time, however, the run-time difference between the CUBE and RECT algorithms is negligible.

The octree created by the IDEAL algorithm was also bound by a world octant of similar size. Since the facets are uniformly distributed within the world octant, the IDEAL should cut the octants through the middle in the same manner as the CUBE and RECT algorithms. Instead, the IDEAL algorithm was found to take advantage of localized density variations, and most octants were not cut through the middle. As a result, this method minimized the number of octants required to store the scene (using significantly less octants than the other two methods). However, the overall performance of this algorithm suffers because the lower number of octants resulted in a higher number of facet intersection tests, and the worst run-time for the algorithms tested.

Summary

The observations for this scene can be summarized as:

- The `CUBE` and `RECT` methods produced comparable results.
- The `IDEAL` method produced the smallest octree but tested more facets than the other two algorithms.
- A small difference in the extents of the world octant for the `CUBE` and `RECT` methods resulted in dramatically different octree sizes. This large difference made only a slight change in the efficiency of octree traversal.

3.1.2 Statistical Scene #2

Description

The scene rendered in Figure 3.2 is approximately 4,000 units per side and contains 5,000 facets of various sizes and orientations. The density of facets decreases with distance from the center of the scene following a Gaussian distribution.

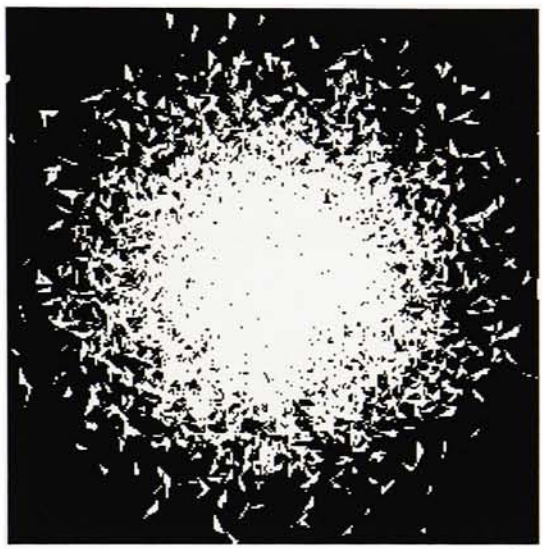

Figure 3.2: Image of the "Stat #2" scene.

This scene was generated using the following parameters extracted directly from the `make_scene` input file:

```
DIST_ENTRY_BEGIN
    TYPE = GAUSSIAN
    COUNT = 5000
    AVERAGE_SIZE = 500.0
    DELTA_SIZE = 250.0
    AVERAGE_X = 0.0
    DELTA_X = 4000.0
    AVERAGE_Y = 0.0
    DELTA_Y = 4000.0
    AVERAGE_Z = 0.0
    DELTA_Z = 4000.0
DIST_ENTRY_END
```

Total number of facets	10,000

Performance Metric	Octant Styles			Best Performance
	CUBE	RECT	IDEAL	
Octree build time	9.24	9.29	8.60	IDEAL
Total number of octants	2121	2142	2248	CUBE
Number of empty octants	30	26	0	IDEAL
Fraction of empty octants	0.014	0.012	0.000	IDEAL
Number of octant searches	17,683,759	17,549,407	17,206,180	IDEAL
Octant search time	258.43	255.60	345.21	RECT
Number of facets tested	177,663,948	174,518,591	226,710,241	RECT
Facet test time	757.66	746.85	986.01	RECT
Total run time	1636.26	1617.21	2023.31	RECT

Table 3.2: Summary of profiler output for the "Stat #2" scene.

Observations

Table 3.2 summarizes the performance of each algorithm for this scene. Because of the extents and the distribution of facets, the CUBE and RECT algorithms subdivided this scene almost identically, resulting in similar counts and times for almost every parameter profiled.

The IDEAL algorithm was designed to minimize octree size in this very case, however, this algorithm created a larger octree than the other two methods. This author suspects that the unforeseen weakness in this algorithm is that the splitting through the mean point "cuts" through a large number of facets. All cut facets must be assigned to all the octants that they overlap, resulting in the higher average number of facets per octant to test and increased octree subdivision (since the subdivision threshold will reached more often).

Summary

The observations for this scene can be summarized as:

- The CUBE and RECT methods produced comparable results.
- The IDEAL method produced the largest octree, and tested more facets than the other two algorithms.

3.1.3 Statistical Scene #3

Description

The scene rendered in Figure 3.3 is a cube approximately 10,000 units per side containing uniformly distributed facets of various sizes and orientations. There is a region of higher density in the upper-right of the image where the density of facets decreases with distance from a center point following a Gaussian distribution.

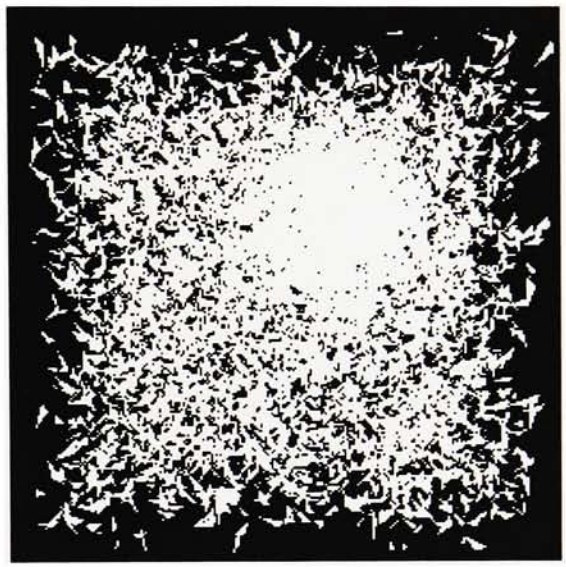

Figure 3.3: Image of the "Stat #3" scene.

This statistically generated scene was created using the following parameters extracted directly from the `make_scene` input file:

```
DIST_ENTRY_BEGIN
    TYPE = UNIFORM
    COUNT = 7500
    AVERAGE_SIZE = 200.0
    DELTA_SIZE = 100.0
    AVERAGE_X = 0.0
    DELTA_X = 10000.0
    AVERAGE_Y = 0.0
    DELTA_Y = 10000.0
```

```
    AVERAGE_Z = 0.0
    DELTA_Z = 10000.0
DIST_ENTRY_END

DIST_ENTRY_BEGIN
    TYPE = GAUSSIAN
    COUNT = 2500
    AVERAGE_SIZE = 200.0
    DELTA_SIZE = 100.0
    AVERAGE_X = 1000.0
    DELTA_X = 2000.0
    AVERAGE_Y = 1500.0
    DELTA_Y = 2000.0
    AVERAGE_Z = 2800.0
    DELTA_Z = 2000.0
DIST_ENTRY_END
```

Total number of facets	10,000

Performance Metric	Octant Styles			Best Performance
	CUBE	RECT	IDEAL	
Octree build time	7.82	7.35	7.33	IDEAL
Total number of octants	2338	1918	1758	IDEAL
Number of empty octants	9	1	0	IDEAL
Fraction of empty octants	0.0004	0.0005	0.0000	IDEAL
Number of octant searches	23,906,057	21,328,349	19,848,130	IDEAL
Octant search time	339.21	298.58	399.06	RECT
Number of facets tested	214,702,670	246,469,375	298,193,826	CUBE
Facet test time	942.95	1074.57	1301.30	CUBE
Total run time	2087.26	2178.83	2576.34	CUBE

Table 3.3: Summary of profiler output for the "Stat #3" scene.

Observations

The summary of results in Table 3.3 indicates that the CUBE algorithm performed the best. The CUBE algorithm created the largest octree and performed the largest number of octant traversals, but performed significantly fewer facet intersection tests. The IDEAL algorithm created the smallest octree but spent the most time performing octant traversals and facet intersections tests. The RECT algorithm finishes in between these two algorithms in octant size, number of octant traversals, number of facet intersection tests and total

run-time. The results from this scene amplify the balance required between octant traversal time and facet intersection time to achieve the fastest run-time.

Summary

The observations for this scene can be summarized as:

- The `CUBE` and `RECT` methods produced comparable results.
- The `CUBE` method produced the largest octree, performed the largest number of octant traversals, but tested the fewest number of facets.
- The `IDEAL` method produced the smallest octree, but performed the largest number of octant traversals and tested the largest number of facets.

3.1.4 Geometric Scene

Description

The scene pictured in Figure 3.4 consists of six geometric objects creating localized areas of high facet density.

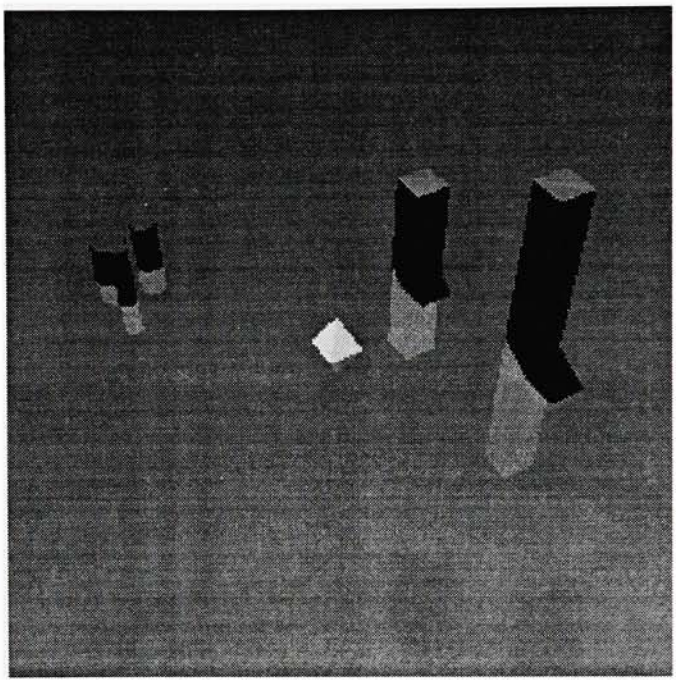

Figure 3.4: Image of the "Geometric" scene.

Using the conventional octree algorithm to render this scene was found to be extremely inefficient in comparison with a "hierarchical bounding box" algorithm. Preliminary tests on this scene lead to the philosophy behind the `IDEAL` subdivision method. Specifically, this author sought an approach to decrease the octant search/traversal time that wouldn't increase the number of facet intersection tests.

Total number of facets	162

Performance Metric	Octant Styles			Best Performance
	CUBE	RECT	IDEAL	
Octree build time	0.29	2.41	0.09	IDEAL
Total number of octants	133	861	43	IDEAL
Number of empty octants	71	465	8	IDEAL
Fraction of empty octants	0.53	0.54	0.18	IDEAL
Number of octant searches	29,483,382	147,387,325	22,231,009	IDEAL
Octant search time	271.04	4512.89	160.89	IDEAL
Number of facets tested	98,442,709	195,962,867	351,080,934	CUBE
Facet test time	326.16	604.99	1350.73	CUBE
Total run time	1272.17	8678.84	2411.37	CUBE

Table 3.4: Summary of profiler output for the "Geometric" scene.

Observations

Table 3.4 summarizes the performance of each algorithm for this scene. The CUBE algorithm produced the best time by achieving the best balance between octant traversals and facet intersection tests. The IDEAL algorithm succeeded at minimizing the size of the octree and the time spent on octant traversals. Unfortunately, the smaller total octree size greatly increased the average number of facets per octant. As a result, significantly more facet intersection tests are performed and its overall run-time does not surpass the CUBE algorithm. The RECT method created the largest octree resulting in a *significantly* higher number of octant traversals, which dominated the overall run-time.

Summary

The observations for this scene can be summarized as:

- The CUBE method produced the best overall run-time.
- The IDEAL method produced the smallest octree, performed the fewest number of octant traversals, but tested the largest number of facets.
- The RECT method created the largest octree and the resulting time spent on octant traversal was over 16 times more than the other two algorithms.

3.1.5 Bomber Scene

Description

The scene rendered in Figure 3.5 includes several aircraft constructed with moderate detail. The bomber accounts for approximately 1800 facets, and each fighter accounts for approximately 800 facets. Although this scene contains almost 8000 facets, the author considers this scene "small" when compared to the majority of scenes constructed for use with DIRSIG.

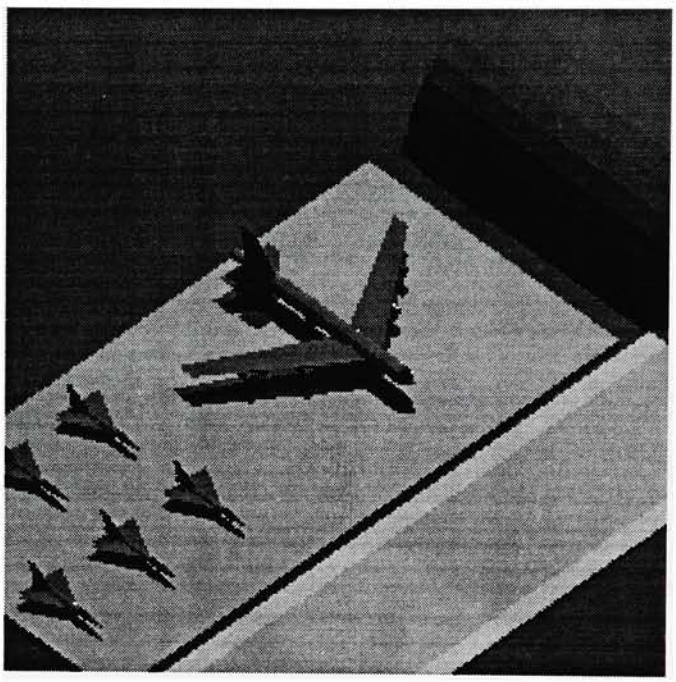

Figure 3.5: Image of the "Bomber" scene.

Total number of facets	7870

Performance Metric	Octant Styles			Best Performance
	CUBE	RECT	IDEAL	
Octree build time	17.65	43.43	21.77	CUBE
Total number of octants	3822	14539	7848	CUBE
Number of empty octants	1094	6607	3760	CUBE
Fraction of empty octants	0.28	0.45	0.47	CUBE
Number of octant searches	77,373,878	216,137,691	139,305,642	CUBE
Octant search time	1361.35	4694.43	3666.99	CUBE
Number of facets tested	207,638,721	630,578,534	761,160,694	CUBE
Facet test time	800.01	2503.31	3581.59	CUBE
Total run time	3817.23	12768.34	11522.73	CUBE

Table 3.5: Summary of profiler output for the "Bomber" scene.

Observations

Table 3.5 summarizes the performance of each algorithm for this scene. The CUBE algorithm performed best for every profiled parameter. This is the one of two scenes in the test set where a particular algorithm creates the smallest octree, spends the least amount of time traversing, and examines the fewest facets.

Summary

The observations for this scene can be summarized as:

- The CUBE method created smallest octree, tested the fewest number of facets and produced the best run-time.
- The RECT and IDEAL algorithms required significantly more octant traversals and facet intersection tests.

3.1.6 Forest Scene

Description

The scene rendered in Figure 3.6 was included in the test set because the leaf facets on the trees are transmissive as they are in the real world. Transmissive facets cannot be rejected based on their orientation with respect to the incoming ray. Instead, they must be rejected based on the slower intersection point/vertex inside-outside test. Therefore, an algorithm that can minimize the number of facets that must be tested will likely have the best overall time.

This scene is configured with textures, however, the overhead of modeling texture can assumed to be constant for all the algorithms.

Figure 3.6: Image of the "Forest" scene.

Total number of facets	7276

Performance Metric	Octant Styles			Best Performance
	CUBE	RECT	IDEAL	
Octree build time	22.31	100.91	57.39	CUBE
Total number of octants	5775	25039	15114	CUBE
Number of empty octants	1744	7785	5480	CUBE
Fraction of empty octants	0.30	0.31	0.36	CUBE
Number of octant searches	79,956,770	141,624,413	105,740,271	CUBE
Octant search time	1402.32	2754.45	2798.52	CUBE
Number of facets tested	276,077,732	493,019,557	655,108,561	CUBE
Facet test time	2080.59	3890.37	4870.64	CUBE
Total run time	5388.22	10411.57	10808.33	CUBE

Table 3.6: Summary of profiler output for the "Forest" scene.

Observations

Table 3.6 summarizes the performance of each algorithm for this scene. Again, the CUBE algorithm performed best for every profiled parameter. This is the other scene where the CUBE algorithm creates the smallest octree, spends the least amount of time traversing, and examines the fewest facets.

Summary

The observations for this scene can be summarized as:

- The CUBE method produced the best overall run-time.
- The RECT and IDEAL algorithms required significantly more octant traversals and facet intersection tests.
- The IDEAL method created a smaller octree than the RECT method. However, the average time for each octant search was much longer.

3.1.7 Terrain Scene

Description

The scene rendered in Figure 3.7 is a facetized terrain produced by an automated tool using ground elevation data from an actual location in Yuma, Arizona. To provide a perspective on scale, the rendered area is approximately 1 km. across in each direction. The darker stripes are dried river bed areas and the small objects appearing in the lower left are 30 m. x 30 m. reflectance panels used to calibrate imagery collected by aircraft flown over the actual scene.

This scene is configured with textures, however, the overhead of modeling texture can assumed to be constant for all the algorithms.

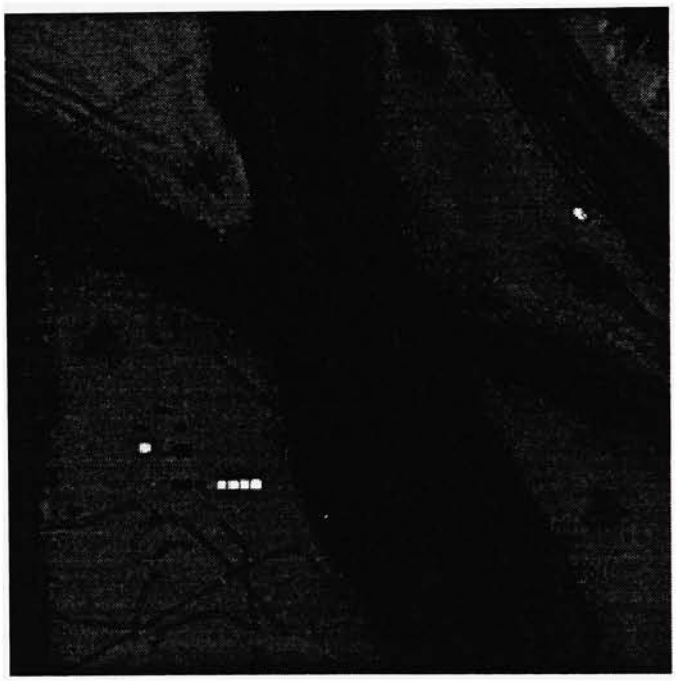

Figure 3.7: Image of the "Terrain" scene.

This scene was rendered during preliminary tests and this author hypothesized that the RECT algorithm had benefits for flat scenes (significantly smaller aspect ratio in one dimension) and somewhat regularly spaced facets. It was also hypothesized that the IDEAL algorithm might perform even better than the RECT algorithm in the regions with small variations in the facet density.

Total number of facets	5850

Performance Metric	Octant Styles			Best Performance
	CUBE	RECT	IDEAL	
Octree build time	6.43	6.91	5.67	IDEAL
Total number of octants	1925	1995	1632	IDEAL
Number of empty octants	1109	599	149	IDEAL
Fraction of empty octants	0.57	0.30	0.09	IDEAL
Number of octant searches	98,003,215	36,193,773	36,537,374	RECT
Octant search time	1473.55	498.27	601.11	RECT
Number of facets tested	426,463,608	299,791,938	368,948,835	RECT
Facet test time	715.11	448.30	559.15	RECT
Total run time	4471.48	2016.05	2337.66	RECT

Table 3.7: Summary of profiler output for the "Terrain" scene.

Observations

Table 3.7 summarizes the performance of each algorithm for this scene. The RECT algorithm produced the best overall run-time out of the three algorithms tested but minimizing both octant traversals and facet intersection tests. Surprisingly, the RECT algorithm required the fewest amount of octant traversals by using the largest octree. The IDEAL algorithm minimized octree size, however, more time was spent performing octant traversal and facet testing.

Summary

The observations for this scene can be summarized as:

- The RECT method produced the best overall run-time. The RECT octree was the largest, however, it spent the least amount of time on octree traversals.
- The IDEAL method created the smallest octree, however, the average time for each octant search was much longer.

3.1.8 Fighter Scene

Description

Each aircraft in the scene rendered in Figure 3.8 contain almost 8,300 facets. In preliminary tests, these localized areas of high facet density were subdivided with varying degrees of success by the octree algorithms.

This scene is also configured with textures, however, the overhead of modeling texture can assumed to be constant for all the algorithms.

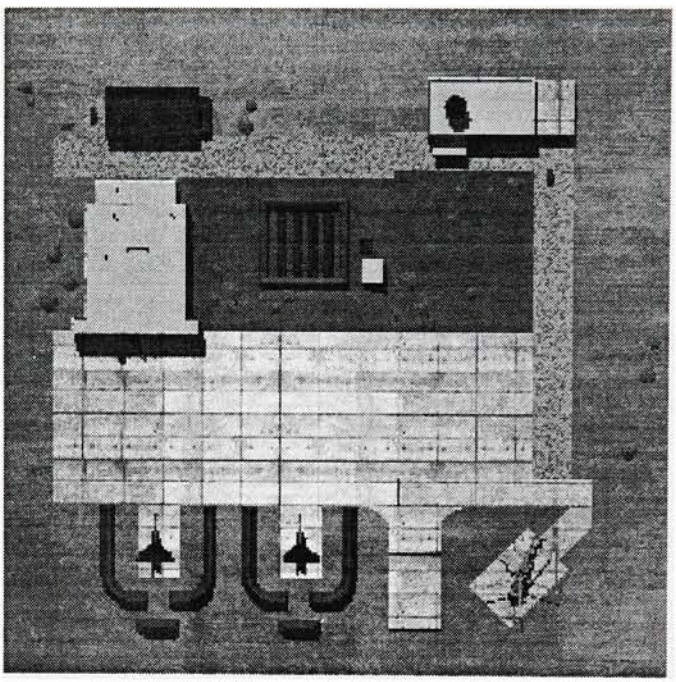

Figure 3.8: Image of the "Fighter" scene.

Total number of facets	54540

Performance Metric	Octant Styles			Best Performance
	CUBE	RECT	IDEAL	
Octree build time	724.61	144.56	445.07	RECT
Total number of octants	96005	11837	71527	RECT
Number of empty octants	31846	3212	32773	RECT
Fraction of empty octants	0.33	0.27	0.45	RECT
Number of octant searches	268,124,967	68,631,908	363,930,282	RECT
Octant search time	5959.80	1134.62	14004.46	RECT
Number of facets tested	1,581,950,537	639,406,231	2,352,114,887	RECT
Facet test time	5804.52	2927.49	10067.75	RECT
Total run time	20385.29	6181.76	36113.92	RECT

Table 3.8: Summary of profiler output for the "Fighter" scene.

Discussion

Table 3.8 summarizes the performance of each algorithm for this scene. This is another example of the RECT algorithm performing the best on scenes containing "flatter" objects with large numbers of facets (*e.g.* the aircraft). While attempting to identify the feature that resulted in the extremely large octrees and poor run-times with the other two algorithms, it was realized that the aircraft parking area consists of thousands of concrete "slabs" (each 6-sided boxes) and rubber expansion joints (the vertical lines near the two lower aircraft are hits to the rubber joints). All of these facets appear in almost the same plane, and may force the two slower subdivision methods to recursively subdivide more that the RECT algorithm.

Summary

The observations for this scene can be summarized as:

- The RECT method produced the best overall run-time.
- The CUBE and IDEAL methods created significantly larger octrees than the RECT method. Overall run-times were dominated by octant traversal time *and* facet intersection tests.
- The CUBE and IDEAL methods may "cut" a large number of facets which must then be assigned to multiple octants, increasing the average number of facets per octant.

3.1.9 Urban Scene

Description

The scene rendered in Figure 3.9 is one of the most complex DIRSIG scenes created to date. It contains over 400 buildings, 200 trees, a facetized terrain, vehicles, etc. This scene embodies all of the challenges (localized areas of high density, transmissive facets, etc.) of the previous scenes.

This scene is also configured with textures, however, the overhead of modeling texture can assumed to be constant for all the algorithms.

Figure 3.9: Image of the "Urban" scene.

Total number of facets	77693

Performance Metric	Octant Styles			Best Performance
	CUBE	RECT	IDEAL	
Octree build time	101.22	108.17	147.97	CUBE
Total number of octants	23835	25067	36653	CUBE
Number of empty octants	6202	3209	1874	IDEAL
Fraction of empty octants	0.26	0.12	0.05	IDEAL
Number of octant searches	65,374,748	53,745,434	91,776,596	RECT
Octant search time	1132.96	988.68	2617.47	RECT
Number of facets tested	344,047,269	380,182,000	750,989,877	RECT
Facet test time	2320.34	2613.48	4962.70	CUBE
Total run time	5188.21	5256.34	10718.20	CUBE

Table 3.9: Summary of profiler output for the "Urban" scene.

Discussion

Table 3.9 summarizes the performance of each algorithm for this scene. The CUBE and RECT subdivision methods were found to produce comparable results. The IDEAL algorithm suffers (again) from an oversized octree that is most likely a result of facet cutting (see Section 3.1.2). The larger octree makes the octant traversals more common and time consuming. Additionally, almost twice as many facets are tested for intersection by the IDEAL algorithm.

Summary

The observations for this scene can be summarized as:

- The CUBE method produced the best overall run-time, however, the RECT algorithm produced comparable results.
- The RECT method created a slightly larger octree, but performed fewer octant traversals.
- The IDEAL method produced the largest octree and the resulting octant traversals were slower. The IDEAL algorithm may split the octant through many facets, resulting in a higher average number of facets per octant.

3.2 General Analysis

This section summarizes the results of each subdivision method for the tested scenes. For each performance metric, the values for each method on each scene are tabulated as well as the average for all the scenes. Since each scene has a different level of complexity and facet populations, it is difficult to establish a test set normalization so that the algorithms can be quantitatively compared from scene-to-scene. As an alternative, the results are also summarized by ranking the relative performance of each algorithm for a given scene. By averaging the ranks for all of the scenes, a semi-quantitative means for examining each algorithm for the whole test set is achieved.

3.2.1 Run-Time Performance

The simplest question to address is "which algorithm is the fastest?". Table 3.10 summarizes the run-times and rankings for the various algorithms for the scene test suite. These run-times reflect the rendering time only, and do not include the octree build-time.

	Time			Rank		
Scene	CUBE	RECT	IDEAL	CUBE	RECT	IDEAL
Stat #1	1860	1860	2104	1	2	3
Stat #2	1636	1617	2023	2	1	3
Stat #3	2087	2178	2576	1	2	3
Geometric	1272	8678	2411	1	3	2
Bomber	3817	12768	11522	1	3	2
Forest	5388	10411	10808	1	2	3
Terrain	4471	2016	2337	3	1	3
Fighter	20385	6181	36113	2	1	3
Urban	5188	5256	10718	1	2	3
Average	5122.67	5662.78	8956.89	1.44	1.88	2.77

Table 3.10: Average time and rankings taken to render the test scenes.

Both the scene-by-scene and average rankings indicate that Glassner's original cubic octant method is the fastest for the scenes tested.

3.2.2 Octree Build Time and Memory Usage

Since one of the key metrics of interest in this study is the size of the octree, it is important to consider the time required to construct the octree. Table 3.11 summarizes this aspect of each algorithm by analyzing the

time required by each algorithm to build the octree for the various scenes. The table also includes a ranking for the build time for each scene (a rank of "1" corresponds the fastest build time) and the average rank for all the scenes.

	Time			Rank		
Scene	CUBE	RECT	IDEAL	CUBE	RECT	IDEAL
Stat #1	5.51	5.51	4.83	2	2	1
Stat #2	7.82	7.35	7.33	3	2	1
Stat #3	9.24	9.29	8.60	2	3	1
Geometric	0.29	2.41	0.09	2	3	1
Bomber	17.65	43.43	21.77	1	3	2
Forest	22.31	100.91	57.39	1	3	2
Terrain	6.43	6.91	5.67	2	3	1
Fighter	724.61	144.56	445.07	3	1	2
Urban	101.22	108.17	147.91	1	2	3
Average	99.45	47.61	77.62	1.89	2.44	1.56

Table 3.11: Average time and rankings taken to construct the octree.

The average ranks indicate that octrees utilizing the IDEAL octants take the shortest time to construct for the scenes tested. This was closely followed by the CUBE octant algorithm and finally by the RECT algorithm. However, these rankings do not reflect the average time for each algorithm. Based on average time, the RECT algorithm was the fastest (rather than the slowest). This is a result of the variance in the build time required by each algorithm. For example, the CUBE algorithm took *much* longer to create the octree for the "Fighter" scene, which tremendously affected its overall average.

Table 3.12 includes a summary of the size of the octree produced by each algorithm. The average rankings for the octree size produced by each algorithm matched the average rankings for octree build time. This verifies that larger octrees require more time to construct.

	Octant Styles		
Scene	CUBE	RECT	IDEAL
Stat#1	3	2	1
Stat#2	1	2	3
Stat#3	3	2	1
Geometric	2	3	1
Bomber	1	3	2
Forest	1	3	2
Terrain	2	3	1
Fighter	3	1	2
Urban	1	2	3
Average	1.89	2.339	1.78

Table 3.12: Average rankings based on size of the octree

The amount of time taken to construct the octree and the size of the octree are important aspects to consider when rendering extremely large scenes. Although the octree can be constructed once and simply read in for any rendering operation, the build time might be a factor for rapid scene development, near "real-time" applications or for creating one-time renderings. In addition, an algorithm that creates a large enough octree to cause a computer to continuously swap memory to disk may take longer to run than a "slower" algorithm that uses less memory.

3.3 Correlation Analysis

Several hypotheses were made in the proposal for this research. Of specific interest was to determine a single octant subdivision as the most efficient for most scenes, or to determine a metric that would indicate which subdivision method to use for a given scene.

After a review of the tabulated results for each scene, most readers will agree that one algorithm does not immediately lend itself as the clear winner. The CUBE algorithm was the fastest for 4 of the 6 scenes "real-world" scenes, but suffered from a memory explosion on one scene (see Section 3.1.8).

To identify the contributing factors that influence the variations in algorithm performance, the following parameters were correlated:

- Octree size/depth and run-time

- Octree size/depth and octant traversal
- Octree size/depth and the number facet of intersection tests
- Octant traversal and run-time
- Octant traversal and empty octant population
- Empty octant population and run-time

If there is some correlation between these parameters, the strengths and short comings of the various algorithms might be identified, and criteria for a better algorithm might be established. The Spearman rank order correlation statistic was used to quantify the correlation between two parameters [Kendall 1962]. The correlation value, ρ, for each parameter pair was computed using the following equation:

$$\rho = 1 - \frac{6 \sum (a_i - b_i)^2}{n^3 - n} \tag{3.1}$$

To establish the correlation between two parameters, each parameter is ranked by value for each of the three (3) octant subdivision methods These rankings are then entered into Equation 3.1, where a_i is the rank for parameter A using method i, b_i is the rank for parameter B using method i and $n = 3$. Some example computations are presented in Table 3.13:

Method	A	B	rank_A	rank_B	ρ
CUBE	100	10	3	3	
RECT	200	30	2	2	1.0000
IDEAL	500	40	1	1	
CUBE	100	10	3	3	
RECT	200	40	2	1	0.5000
IDEAL	500	30	1	2	
CUBE	100	40	3	1	
RECT	200	30	2	2	-1.0000
IDEAL	500	10	1	3	

Table 3.13: An example correlation computation.

A correlation value of 1.0 indicates that there is direct (positive) correlation between the two parameters. A value of -1.0 indicates that the parameters are negatively correlated, and a value near 0.0 indicates that there is little correlation between the values. To compute the overall correlation value, the individual correlation values for each scene were averaged.

Since only three algorithms are used in the analysis, the correlation values can change significantly in response to a small change in ranking. An example of this can be seen in the second set of entries in Table 3.13.

One mismatch in rankings (from 1-1, 2-2, 3-3 to 1-2, 2-1, 3-3) changed the correlation value from 1.0 to 0.5. Therefore, this author considers average correlation values near 0.5 deserving of further investigation.

For the reasons described earlier (see Section 3.1), average correlation values excluding the statistical scenes are sometimes computed *in addition* to the nine (9) scene average. This is because some of the statistical scenes were designed so that all three algorithms would perform similarly, and the relative ranks may emphasizes very small differences. These values are included at the bottom of the correlation analysis tables under the entry "Excluding Stats".

3.3.1 Octree Size versus Run-Time

To investigate a possible relationship between total octree size and the performance of the octree algorithms, the total number of octants in the octree (parameter A) was correlated with the run-time (parameter B). The correlation analysis for these two parameters is presented in in Table 3.14.

Analysis of the tabulated data indicates that there is possibly some correlation between the smallest octree and the fastest run-time for some of the scenes. Since, three out of the six "real-world" scenes had their best run-time when the octree was smallest, this possible connection was investigated further. Two processes dominate the generic octree ray tracing algorithm: octant traversal (finding octants along the path of the ray) and the actual facet intersection tests. The next two sections attempt to identify if either of these processes are correlated to the overall run-time.

Scene	STYLE	A	B	rank_A	rank_B	ρ
Stat #1	CUBE	2078	1860	1	3	-1.0000
	RECT	1855	1861	2	2	
	IDEAL	1401	2104	3	1	
Stat #2	CUBE	2121	1636	3	2	0.5000
	RECT	2142	1617	2	3	
	IDEAL	2248	2023	1	1	
Stat #3	CUBE	2338	2087	1	2	0.5000
	RECT	1918	2178	2	1	
	IDEAL	1758	1301	3	3	
Geometric	CUBE	133	1272	2	3	0.5000
	RECT	861	8678	1	1	
	IDEAL	43	2411	3	2	
Bomber	CUBE	3822	3817	3	3	1.0000
	RECT	14539	12768	1	1	
	IDEAL	7848	11522	2	2	
Forest	CUBE	5775	5388	3	3	0.5000
	RECT	25039	10411	1	2	
	IDEAL	15114	10808	2	1	
Terrain	CUBE	1925	4471	2	1	-0.5000
	RECT	1995	2016	1	3	
	IDEAL	1632	2337	3	2	
Fighter	CUBE	96005	20385	1	2	0.5000
	RECT	11837	6181	3	3	
	IDEAL	71527	36113	2	1	
Urban	CUBE	23835	5188	3	3	1.0000
	RECT	25067	5256	2	2	
	IDEAL	36653	10718	1	1	
All Scenes						**0.3333**
Excluding Stats						**0.5000**

Table 3.14: Analysis of correlation between total octree size and run-time performance.

3.3.2 Octree Size versus Octant Traversal

Table 3.15 includes the correlation analysis of total octree size (parameter A) and the number of octant traversals performed (parameter B).

This modest value indicates that the number of octants in the octree may be positively correlated to the number of traversals that must be performed. In reflection, this is what one would expect: a higher total number of octants implies that (on average) more octants must be traversed along a path between two points.

Scene	STYLE	A	B	rank_A	rank_B	ρ
Stat #1	CUBE	2078	19060515	1	1	1.0000
	RECT	1855	19035334	2	2	
	IDEAL	1401	17445697	3	3	
Stat #2	CUBE	2121	17683759	3	1	-1.0000
	RECT	2142	17549407	2	2	
	IDEAL	2248	17206180	1	3	
Stat #3	CUBE	2338	23906057	1	1	1.0000
	RECT	1918	21328349	2	2	
	IDEAL	1758	19848130	3	3	
Geometric	CUBE	133	29483382	2	2	1.0000
	RECT	861	147387325	1	1	
	IDEAL	43	22231009	3	3	
Bomber	CUBE	3822	77373878	3	3	1.0000
	RECT	14539	216137691	1	1	
	IDEAL	7848	139305642	2	2	
Forest	CUBE	5775	79956770	3	3	1.0000
	RECT	25039	141624413	1	1	
	IDEAL	15114	105740271	2	2	
Terrain	CUBE	1925	98003215	2	1	-0.5000
	RECT	1995	36193773	1	3	
	IDEAL	1632	36537374	3	2	
Fighter	CUBE	96005	268124967	1	2	0.5000
	RECT	11837	68631908	3	3	
	IDEAL	71527	363930282	2	1	
Urban	CUBE	23835	65374748	3	2	0.5000
	RECT	25067	53745434	2	3	
	IDEAL	36653	91776596	1	1	
All Scenes						**0.5000**
Excluding Stats						**0.5833**

Table 3.15: Analysis of correlation between the total octree size and the number of octant traversals performed.

The relationship between an increased number of octant traversals (parameter A) and an increase in total run-time (parameter B) was investigated in the correlation analysis presented in Table 3.16. Since the the number of facet intersections may be correlated to the size of the octree, this relationship cannot be implied from the previous table (Table 3.15).

The average correlation value for the "real-world" scenes was 0.7500 but the overall average was lowered by the addition of the statistical scenes to the analysis (to a value of 0.3889). If the data from the statistical scenes is removed from consideration, this high correlation value indicates that minimizing octant traversals can result in improved overall run-times. However, assuming that this is the only parameter affecting run-time is naive. Some algorithms may produce fewer total octants, but time saved by fewer octant traversals may be offset by an increased number of facet intersection tests. This possibility is analyzed in the next section.

Scene	STYLE	A	B	rank_A	rank_B	ρ
Stat #1	CUBE	19060515	1860	1	3	-1.0000
	RECT	19035334	1861	2	2	
	IDEAL	17445697	2104	3	1	
Stat #2	CUBE	17683759	1636	1	2	-0.5000
	RECT	17549407	1617	2	3	
	IDEAL	17206180	2023	3	1	
Stat #3	CUBE	23906057	2087	1	2	0.5000
	RECT	21328349	2178	2	1	
	IDEAL	19848130	1301	3	3	
Geometric	CUBE	29483382	1272	2	3	0.5000
	RECT	147387325	8678	1	1	
	IDEAL	22231009	2411	3	2	
Bomber	CUBE	77373878	3817	3	3	1.0000
	RECT	216137691	12768	1	1	
	IDEAL	139305642	11522	2	2	
Forest	CUBE	79956770	5388	3	3	0.5000
	RECT	141624413	10411	1	2	
	IDEAL	105740271	10808	2	1	
Terrain	CUBE	98003215	4471	1	1	1.0000
	RECT	36193773	2016	3	3	
	IDEAL	36537374	2337	2	2	
Fighter	CUBE	268124967	20385	2	2	1.0000
	RECT	68631908	6181	3	3	
	IDEAL	363930282	36113	1	1	
Urban	CUBE	65374748	5188	2	3	0.5000
	RECT	53745434	5256	3	2	
	IDEAL	91776596	10718	1	1	
All Scenes						**0.3889**
Excluding Stats						**0.7500**

Table 3.16: Analysis of correlation between the number of octant traversals and the total run-time

3.3.3 Octree Size versus Facet Intersection Tests

The analysis in the previous section established that smaller octrees may require fewer octant traversals before identifying an intersected facet. However, since fewer octants are used to store the same amount of facets, the average number of facets that must be tested in each octant must be increase. Therefore, a performance increase can only be expected if the time saved by fewer octant traversals is not outweighed by the increase in facet intersection tests. Table 3.17 contains the correlation analysis for the size of the octree (parameter A) and the number of intersection test performed (parameter B).

The average correlation value for the "real-world" scenes was 0.2500 but the overall average was lowered by the addition of the statistical scenes to the analysis (to a value of 0.0556). For the larger "real-world" scenes, the correlation coefficient is high enough to assume that the decrease in the number of octants does come at the cost of an increase in the number of facet intersection tests that must be performed. However, the scenes featured in these test cases, and the analysis from Section 3.3.1 indicate that this increase in intersection tests is sometimes offset by a decrease in octant traversals.

Scene	STYLE	A	B	rank_A	rank_B	ρ
Stat #1	CUBE	2079	217494289	1	2	-0.5000
	RECT	1855	217412595	2	3	
	IDEAL	1401	252524941	3	1	
Stat #2	CUBE	2121	177663948	3	2	0.5000
	RECT	2142	174518591	2	3	
	IDEAL	2248	226710241	1	1	
Stat #3	CUBE	2338	214702670	1	3	-1.0000
	RECT	1918	246469375	2	2	
	IDEAL	1758	298193826	3	1	
Geometric	CUBE	133	98442709	2	3	-0.5000
	RECT	861	195962867	1	2	
	IDEAL	43	351080934	3	1	
Bomber	CUBE	3822	207638721	3	3	0.5000
	RECT	14539	630578534	1	2	
	IDEAL	7848	761160694	2	1	
Forest	CUBE	1925	426463608	2	1	-0.5000
	RECT	1995	299791938	1	3	
	IDEAL	1632	368948835	3	2	
Terrain	CUBE	5775	276077732	3	3	0.5000
	RECT	25039	493019557	1	2	
	IDEAL	15114	655108561	2	1	
Fighter	CUBE	96005	1581950537	1	2	0.5000
	RECT	11837	639406231	3	3	
	IDEAL	71527	2352114887	2	1	
Urban	CUBE	23835	344047269	3	3	1.0000
	RECT	25068	380182000	2	2	
	IDEAL	36653	750989877	1	1	
All Scenes						**0.0556**
Excluding Stats						**0.2500**

Table 3.17: Analysis of correlation between total octree size and the number of facet intersection tests performed.

3.3.4 Empty Octant Population versus Run-Time

One of the original original hypotheses concerned the correlation between empty octant population and octant traversal. The `IDEAL` subdivision method was developed as an attempt to minimize the poor subdivision of localized regions of high facets density.

The analysis of octant traversals performed (parameter A) and the percentage of empty octants (parameter B) in Table 3.18 illustrates that this hypothesis is not entirely justified using the current implementations and the tested scenes.

This moderate correlation value indicates that higher populations of empty octants to do not necessarily imply increased octant traversal. Several of the fastest runs had the largest number of empty octants, however, in other scenes the fastest runs had the smallest number of empty octants. Specifically, the data collected from the "Urban" scene indicates that the most traversals were performed on the scene with the fewest number of empty octants. This is further verified by Table 3.19, that analyzes the correlation value between the empty octant population (parameter A) and the overall run-time (parameter B).

Scene	STYLE	A	B	rank_A	rank_B	ρ
Stat #1	CUBE	19060515	0.0009	1	2	-0.5000
	RECT	19035334	0.0000	2	3	
	IDEAL	17445697	0.0014	3	1	
Stat #2	CUBE	17683759	0.0140	1	1	1.0000
	RECT	17549407	0.0120	2	2	
	IDEAL	17206180	0.0000	3	3	
Stat #3	CUBE	23906057	0.0004	1	2	0.5000
	RECT	21328349	0.0005	2	1	
	IDEAL	19848130	0.0000	3	3	
Geometric	CUBE	29483382	0.5300	2	2	1.0000
	RECT	147387325	0.5400	1	1	
	IDEAL	22231009	0.1800	3	3	
Bomber	CUBE	77373878	0.2800	3	3	0.5000
	RECT	216137691	0.4500	1	2	
	IDEAL	139305642	0.4700	2	1	
Forest	CUBE	79956770	0.3000	3	3	0.5000
	RECT	141624413	0.3100	1	2	
	IDEAL	105740271	0.3600	2	1	
Terrain	CUBE	98003215	0.5700	1	1	0.5000
	RECT	36193773	0.3000	3	2	
	IDEAL	36537374	0.0900	2	3	
Fighter	CUBE	268124967	0.3300	2	2	1.0000
	RECT	68631908	0.2700	3	3	
	IDEAL	363930282	0.4500	1	1	
Urban	CUBE	65374748	0.2600	2	1	-0.5000
	RECT	53745434	0.1200	3	2	
	IDEAL	91776596	0.0500	1	3	
All Scenes						**0.4444**
Excluding Stats						**0.5000**

Table 3.18: Analysis of correlation between octant traversals and empty octant population

Scene	STYLE	A	B	rank_A	rank_B	ρ
Stat #1	CUBE	0.0009	1860	2	3	0.5000
	RECT	0.0000	1861	3	2	
	IDEAL	0.0014	2104	1	1	
Stat #2	CUBE	0.0140	1636	1	2	-0.5000
	RECT	0.0120	1617	2	3	
	IDEAL	0.0000	2023	3	1	
Stat #3	CUBE	0.0004	2087	2	3	-0.5000
	RECT	0.0005	2178	1	2	
	IDEAL	0.0000	2576	3	1	
Geometric	CUBE	0.5300	1272	2	3	0.5000
	RECT	0.5400	8678	1	1	
	IDEAL	0.1800	2411	3	2	
Bomber	CUBE	0.2800	3817	3	3	0.5000
	RECT	0.4500	12768	2	1	
	IDEAL	0.4700	11522	1	2	
Forest	CUBE	0.3000	5388	3	3	1.0000
	RECT	0.3100	10411	2	2	
	IDEAL	0.3600	10808	1	1	
Terrain	CUBE	0.5700	4471	1	1	0.5000
	RECT	0.3000	2016	2	3	
	IDEAL	0.0900	2337	3	2	
Fighter	CUBE	0.3300	20385	2	2	1.0000
	RECT	0.2700	6181	3	3	
	IDEAL	0.4500	36113	1	1	
Urban	CUBE	0.2600	5188	1	3	-1.0000
	RECT	0.1200	5256	2	2	
	IDEAL	0.0500	10718	3	1	
All Scenes						**0.2222**
Excluding Stats						**0.4166**

Table 3.19: Analysis of correlation between empty octant population and run-time performance.

3.4 Predicting the Best Algorithm

The previous sections have shown that no single algorithm always produces the fastest run-time. This section will examine some of the scene metrics to see if the best algorithm for a given scene can be predicted, however, it should be noted that many more test scenes are required to isolate one metric with any confidence.

3.4.1 Using scene size

Table 3.20 summaries the size of the various scenes and the best algorithm for that scene. The scenes have been sorted by size.

Scene	Number of Facets	Fastest Algorithm
Geometric	162	CUBE
Terrain	5,850	RECT
Forest	7,276	CUBE
Bomber	7,870	CUBE
Stat #1	10,000	CUBE
Stat #2	10,000	RECT
Stat #3	10,000	CUBE
Fighter	54,540	RECT
Urban	77,693	CUBE

Table 3.20: Correlation between scene size and the fastest algorithm.

There is some indication that RECT octants may work better for larger scenes. Although the CUBE algorithm runs the fastest for the "Urban" scene, the RECT algorithm runs only 1.5% slower. Further testing with scenes of this size is required.

3.4.2 Using scene aspect ratio

Table 3.21 summaries the dimensional size/aspect ratios of the various scenes and the best algorithm for that scene. The scenes have been sorted by aspect ratio.

This analysis does not indicate any general trends. It is tempting to speculate that CUBE works better are scene with larger aspect ratios, however, the "Fighter" scene runs 3 times slower with the CUBE algorithm than the RECT algorithm.

Scene	Aspect Ratio	Fastest Algorithm
Stat #1	1:1:1	CUBE
Stat #2	1:1:1	RECT
Stat #3	1:1:1	CUBE
Forest	18:18:1	CUBE
Terrain	32:32:1	RECT
Urban	56:56:1	CUBE
Bomber	85:85:1	CUBE
Fighter	156:156:1	RECT
Geometric	400:400:1	CUBE

Table 3.21: Correlation between scene aspect ratio and the fastest algorithm

Chapter 4

Conclusions and Recommendations

4.1 Algorithm Summary

The CUBE algorithm rated highly in most of the analysis. However, it was shown to suffer from memory and run-time explosions with some scenes. The RECT algorithm was shown to be a consistent performer and lends itself as a reasonable alternative to the CUBE algorithm. When rendering scenes with 1:1:1 aspect ratios, the RECT method will behave the same as the CUBE method (see Section 3.1.1).

The IDEAL algorithm did not meet the expectations of this author in either octree size or overall run-times. It is suspected that this method cuts facets more often than the other algorithms. As a result, more facets appear in multiple octants and an increased number of facet intersection tests are performed. This suspicion could be verified with further testing. Another possibility is that the case depicted in Figure 1.6 may occur more often than expected. In general, the IDEAL algorithm, as it is currently defined, should not be considered for use.

In general, the low confidence in some observations arises from the discrete correlation values resulting from the limited number of subdivision methods tested and indicate that further analysis needs to be performed. To generate more data for analysis, the same methods might be run using a set of subdivision thresholds rather than a single subdivision threshold.

In addition, the correlation analysis indicated that some of the author's hypotheses were incorrect. Specifically, there is little evidence of a connection between higher populations of empty octants and slower overall run-times.

4.2 Automatic Algorithm Selection

In the pursuit of a "smart octree builder", a scene trait has *not* been identified to automatically select the subdivision method that will yield the best results. For example, the "Fighter" (see Section 3.1.8) and "Urban" (see Section 3.1.9) scenes have comparable numbers of facets, aspect ratios, etc., but a large disparity in the efficiency of the algorithms.

To identify to best subdivision algorithm for a given scene, a brute force approach is suggested. The completed scene should be run using the `CUBE` and `RECT` algorithm and the fastest one utilized for all subsequent runs.

4.3 Future Work

There is enough variation in the performance of the `CUBE` and `RECT` algorithms that a better understanding of each algorithm is required. Identification of the scene trait that "breaks" each algorithm might be discovered with careful experimental analysis on more precisely built scenes.

Octree ray tracers attempt to balance the amount of time spent traversing the octree and the time testing facets for intersection. This study examined constructing better octrees using the same subdivision threshold. In a more complex and time-consuming study, different subdivision thresholds may be used to evaluate each algorithm.

Appendix A

libRT Design Document

This appendix contains the original design and implementation document for the *libRT* ray tracing library. This library was written by the author as part of another effort. Preliminary evaluation of this library lead to the interest in subdivision methods examined in the research project.

A.1 Abstract

A new ray tracing library (named *libRT*) was implemented to replace the current ray tracer in the DIRSIG synthetic image generation program. This library uses the non-uniform spatial subdivision technique refined by Andrew Glassner [Glassner 1984]. This algorithm minimizes facet intersection tests *(i)* by automatically generating a density dependent spatial hierarchy of facet regions (called an *octree*) and *(ii)* by only testing facets in regions along the path of the ray.

To maintain compatibility with the old ray tracer, the new library supports scenes comprised of three and four sided planar facets. Although the new library is significantly more complex than the old ray tracer, the interface was designed to be simpler and more efficient. This includes mechanisms for the initialization and storage of facets in a managed database area, automated generation of a spatial hierarchy for improved run-time performance, and a simplified ray tracing routine.

One important enhancement offered by this new library over Glassner's original octree algorithm is the ability to create non-cubic or rectangular octants. The justification for this option is presented as well as some preliminary guidelines for which octant style may be most efficient for a given scene. A brief description of the round-off/precision strategy used in the library is also included.

This document is broken down into three primary sections: a discussion of the library's implementation, a preliminary performance appraisal, and an brief introduction to the library from the user's standpoint.

A.2 Introduction

A.2.1 Project Justification

Due to the complexity of the DIRSIG model, each pixel in the rendered image requires approximately 130 rays to be traced. Profiler output for the original code indicated that the ray tracing library accounted for a majority of the model's total computations. Hence, the design of a more efficient ray tracing library was easily justified by the direct impact it would have on the model's overall performance.

Like the old library, this library provides only ray tracing facilities and therefore does *not* include either a camera model or image rendering capabilities. The DIRSIG model already includes a sophisticated model for computing the rays cast from the complex optical geometries of camera/sensor platforms used in remote sensing applications [Salacain 1994], rendered by DIRSIG are produced by the radiometric submodel using the reflective and thermal data associated with the facet intersected at each pixel. In addition to determining the facet intersected at each pixel, the ray tracer is invoked by several of the rendering submodels to establish the shadowing conditions (radiometry submodel), the background/diffuse loading (radiometry submodel) and the solar loading history (thermal model) (see A.1).

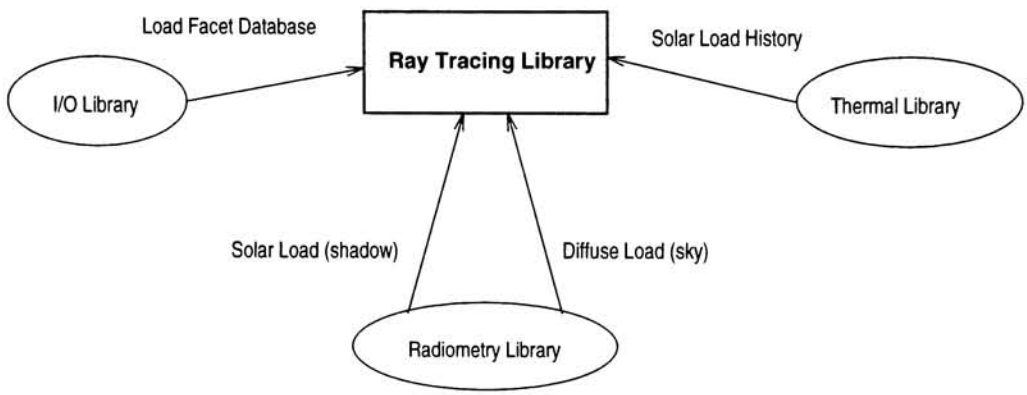

Figure A.1: Sources of invocation for the ray tracer library

A.2.2 Ray Tracing Basics

Ray tracing is a popular rendering method because it simulates the physics of image formation: rays cast from the simulated camera find the sources of photons that would enter a real camera. Using simple vector mathematics, sources of reflections and shadows can also be identified and incorporated. In a minimal algorithm, intersection tests must be performed on every object in the scene to correctly identify the closest surface along the the ray, For large scenes, this linear search becomes computationally significant. As a result, most development in the ray tracing community is focused on strategies to minimize the number of intersection tests required per ray trace.

The old DIRSIG ray tracing library utilized a three level bounding box hierarchy to minimize the number of facet intersection tests (see Figure A.2). Using this algorithm, the ray is tested for intersection with the bounding volumes at the top of the hierarchy. If the ray intersects the volume, then further intersection tests must be performed on the sub-volumes or facets contained by the volume. If the ray does not intersect the volume, then it can be assumed that intersection with any facets within the volume is not possible, and those tests can be skipped. Although this strategy does minimizes intersection tests when compared to the global search approach, it unfortunately has several limitations:

- The burden of constructing the bounding box hierarchy with the DIRSIG ray tracer was placed on the user which required additional planning and effort to create scenes.
- The user may define "poor" bounding boxes which contain large amounts of facets or which overlap other boxes, resulting in less than ideal performance.
- The order in with which bounding box intersection tests are performed is not optimal. Because there isn't any spatial ranking based on distance to the ray origin, all boxes must be checked at a given level in the hierarchy.

To avoid these limitations, a new ray tracing library utilizing *non-uniform spatial subdivision* [Glassner 1984] was designed and implemented. This algorithm systematically subdivides the scene into axis aligned boxes or *octants* based on facet density - regions containing a more facets are subdivided into smaller octants to minimize the number of intersection tests required per octant (see Figure A.3). The resulting data structure is a spatial hierarchy of bounding boxes, called an *octree*. In general, the octree algorithm should be more efficient than the user defined hierarchy algorithm by addressing all three limitations of the later:

- The construction of the octree is automated, and requires only a minimal of overhead at run-time rather than during the scene construction process.
- The number of facets contained in each octant is bounded by a user defined threshold and octants never overlap.

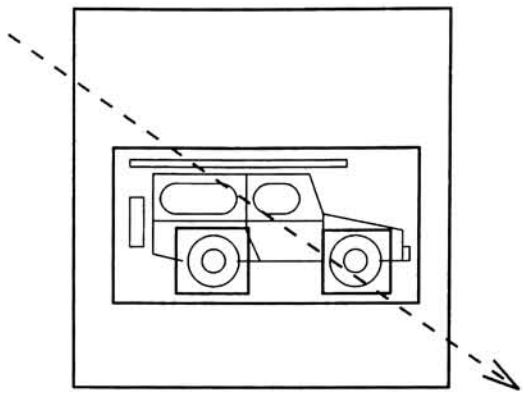

Figure A.2: A 2D example of a user define bounding box hierarchy and a ray being traced through the scene.

- Octants are progressively examined based on distance from the origin of the ray. Once an intersection has been identified within an octant, the search can be terminated.

The non-uniform spatial subdivision algorithm is slightly more complex than its uniform subdivision counterpart because the process of navigating the octants is more complicated. However, this approach was chosen because for scenes lacking a uniform distribution of facets (which characterizes the real world scenes rendered with DIRSIG), the non-uniform subdivision algorithm will create fewer empty boxes and fewer overfilled boxes.

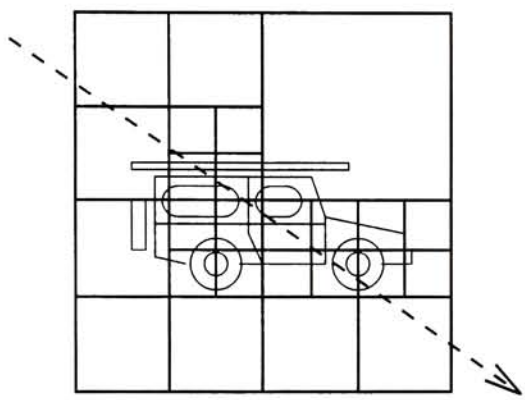

Figure A.3: A 2D example of spatial subdivision based on facet density and a ray being traced through the scene.

A.3 Implementation Description

The *libRT* library provides three general facilities: *(i)* a simplified interface which allows the user to add objects to a managed storage area, *(ii)* a way to create an octree for the loaded objects using a spatial subdivision algorithm and *(iii)* a way to determine objects intersecting a ray using the octree. A general introduction and a detailed description of the implementation is provided for each of these primary facilities. The names and a brief description of internal support routines are also included to clarify the logical operation of the primary functions. These internal functions can be identified by the prefix `RT_` in the function name. A detailed descriptions of all the internal routines is provided in **Appendix A: Module Descriptions**.

A.3.1 Supporting Data Types

At the core of the library are the basic geometric data types for points and vectors. Both `RT_POINT` and `RT_VECTOR` are implemented as fixed-length arrays to facilitate more streamlined algorithms. The type `RT_RAY` is a structure consisting of an `RT_POINT` to store the ray origin and a normalized `RT_VECTOR` to define the direction of the ray. The number of dimensions (length of the array) is defined by `NUM_DIMS` in `RT.h`. Macros are provided to easily copy points and vectors.

Although the *libRT* library was written in ANSI C (for portability and speed), the library was specifically designed to reflect the trend towards object oriented interfaces (see Figure A.4). At the heart of this design, is the *world* data structure (type name `RT_WORLD`). The world data structure stores all entities relevant to a specific scene including all the facets, the control parameters for the octree and the octree itself. The self contained design of the `RT_WORLD` structure is deliberate so that a program can simultaneously create, manage and ray trace several "worlds" on an individual basis.

Scenes consist of facets (data type `RT_FACET`) which include geometry data (data type `RT_GEOMETRY`) and rendering attributes (data type `RT_ATTRIBUTES`). The geometry data includes the 3D coordinates for each vertex, the normal to the facet, the offset of the plane equation, and a 3x3 orientation matrix for redirecting exiting vectors. The attribute data structure is meant to store user specific rendering data. For use with DIRSIG, the attribute data structure was modified to store the spectral reflectance and surface specularity information used by the radiometry model to render the image.

The facet database area in the `RT_WORLD` structure consists of a static array of pointers to facet blocks (data type `RT_FACET_BLOCK`). As more facet blocks are needed, a new block is allocated and the pointer is stored in the next slot in the array.

The base and any node in the octree is called an octant (data type `RT_OCTANT`). Data in the octant structure includes the space it occupies, the size of the octant, and a pointer to a list of sub-octants or facets (depending on it's location in the octree).

The ray tracer routine introduces two more types: `RT_HIT` and `RT_HIT_LIST`. The `RT_HIT` type is capable of storing a pointer to the intersected facet, a pointer to the octant containing the facet, the hit point, the

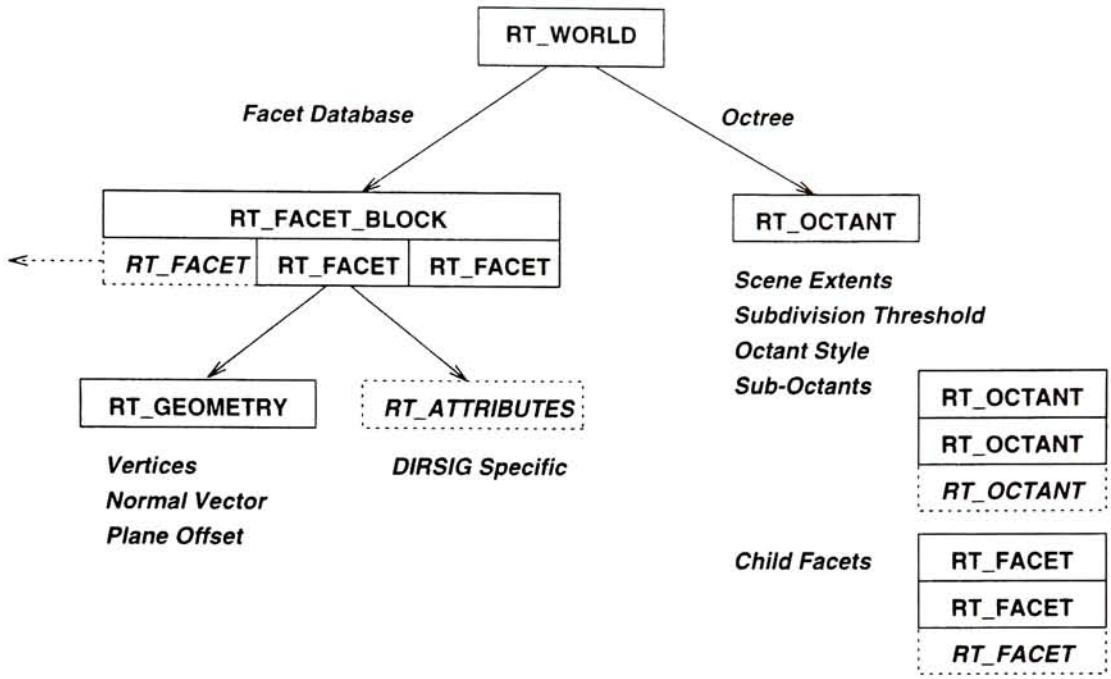

Figure A.4: The core data structures used in *libRT*.

distance from the origin of the ray to the hit point and the angle from the facet normal to the incident ray. A `RT_HIT_LIST` consists of a static array of hits, and a counter to maintain the number of hits currently stored in the list.

Actual definitions of the data types can be found in **Appendix A** or `RT.h`.

A.3.2 Object Storage

The object storage facilities provided by the library are primitive at the time of this initial release. The only geometric construct defined by the library at this time are three and four sided facets. The facets are read in one at a time from the input data file using an existing file reading routine in the DIRSIG I/O library. For each facet in the file, the I/O routine reads in the geometric attributes (including the location of vertices and the normal vector definition) and stores them in a temporary `RT_FACET` definition. This facet definition is then added to the specified world by calling a loading routine which computes additional information (including a 3D orientation matrix and the plane equation offset) before copying the facet into the storage area.

The library was constructed to allow the user to manage several different "worlds" simultaneously. Each world must be initialized through the use of the `Initialize_World()` routine. This function initializes the `RT_WORLD` structure to a known state for the facet loading and octree creation routines.

A facet definition is added to a given world data structure using the `Add_Facet_To_World()` routine. Internally, the library uses a block type allocation scheme to store the facets loaded into the storage area. If the addition of a facet exceeds the current storage capacity of the specified world, another block of facets is allocated and its reference added to the block list. The maximum number of facet blocks that can be allocated and the number of facets in each block is defined at compile time by the constants `MAX_FACET_BLOCKS` and `FACET_BLOCK_SIZE` in the include file `RT_World.h`. The current settings for these values allow for 512K facets to be stored.

As each facet is added to the world data structure, each vertex is checked to see if the minimum and maximum dimensional extents of the scene need to be updated. These dimensional extents are later used by the octree building routine to construct the world octant.

A.3.3 Octree Creation

The spatial subdivision of the scene into an octree is the required preprocessing stage which makes the run-time tracing of the scene more efficient than conventional ray tracing techniques. Upon completion of the scene subdivision phase, each leaf node or *leaf octant* in the octree will contain a list of all facets that intersect the space defined by that octant (see Figure A.5). The algorithm for building the octree is outlined below:

- The base of the octree, or the *world octant*, is constructed to bound all the objects in the scene using the scene extents determined during the loading stage.
- The octree is created by adding each facet in the world to the octree one at a time. Starting with the *world octant*, the new facet is tested against each sub-octant for intersection. Each intersected sub-octant then repeats the intersection test for its sub-octants in a recursive fashion. The recursion terminates when a *leaf octant* is found, at which point a reference to the facet is added to that octants facet list. All sub-octants of an octant intersected by the facet must be examined since the facet may extend over several branches of the octree.

- If the addition of a facet to a leaf octant exceeds the user defined subdivision threshold then the octant is split. To split an octant, eight new sub-octants are allocated and the facets contained by the original octant are sorted into each of the new sub-octants. The splitting of new leaf octants will continue until all octants are below the subdivision threshold.
- Facets are added in this manor until all the facets in the scene are loaded into the octree.

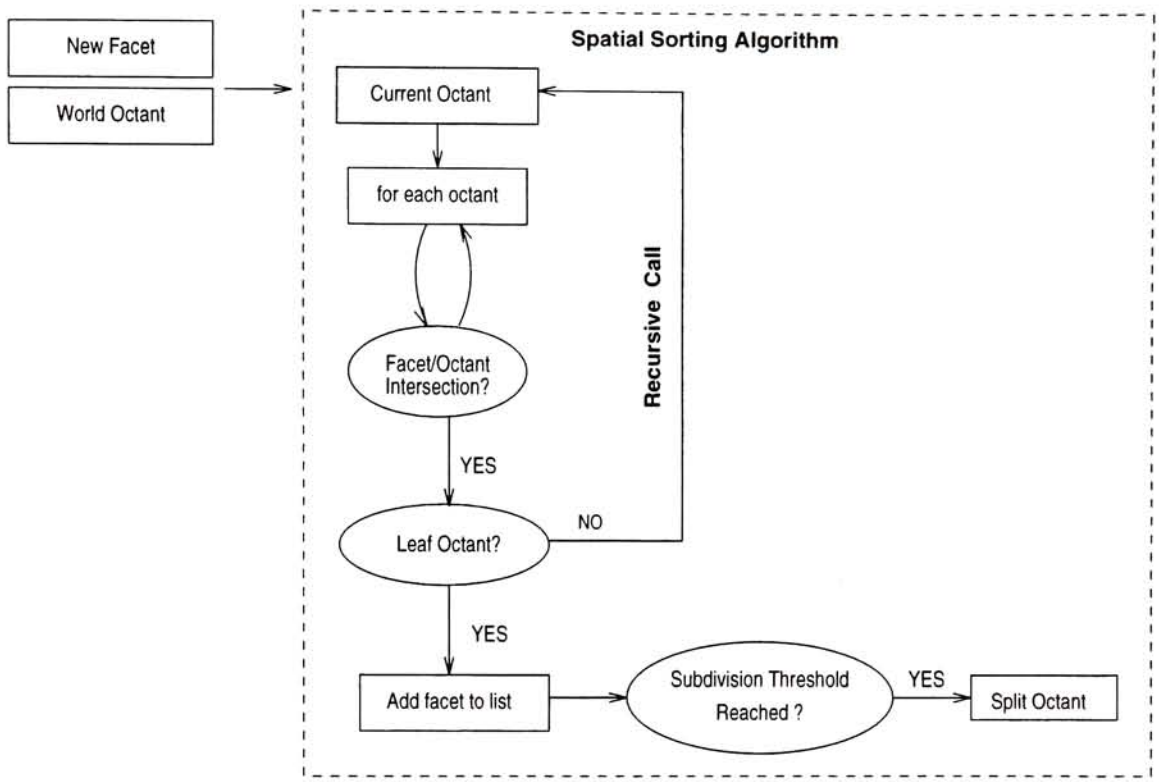

Figure A.5: Creating an octree using a Spatial Sorting Algorithm.

Once the world has been loaded with all the desired facets, the octree must be created. This is achieved through a call to the `Create_Octree()` routine. Since this library supports an additional octant style not described by Glassner, the calling function must specify which world octant convention to use. The octant convention defines the shape (either cube or rectangle) of the world octant, and hence the shape of every sub-octant created within it. The enumerated data type `RT_OCTANT_STYLE` has two possible values:

`CUBE_STYLE`
The "cube-like" octant convention described by Glassner in his original paper using octants that are the

same length in all dimensions. In this case, the size of world octant is set to the largest dimension from the minimum and maximum extents of the loaded scene.

`RECTANGLE_STYLE`
A non-cubic octant convention featuring octants which may have *different* lengths in each dimension. In this case, the world octant is set to the the minimum and maximum extents of the loaded scene. For scenes that are extremely large in one dimension, this convention may produce fewer empty octants and improve performance.

The "spatial sorting" algorithm used to build the octree relies on a facet-box intersection algorithm which is described later in this section. Each facet stored in the world facet storage area is passed to the sort routine which adds a reference to any leaf octant that intersects it. Pseudo-code for the recursive sorting routine is outlined below:

```
RT_Sort_Facet( RT_OCTANT *octant, RT_FACET *facet )
{
    if( Octant_Is_A_Leaf( octant )) {
        RT_Add_Facet_To_Octant( octant, facet )
    }
    else {
        foreach( suboctant ) {
            if( RT_Facet_Box_Intersect( suboctant, facet )) {
                counter = 0;
                RT_Sort_Facet( suboctant, facet, counter )
                if( counter == 0 )
                    Issue_Error();
            }
        }
    }
}
```

Note the special counter that is also passed to the sorting routine which keeps track of how many leaf octants the facet was added to. In the event that a facet which intersects an octant fails to intersect any of the sub-octants, an error is issued.

The facet-box intersection routine (see `RT_Facet_Box_Intersect()`) utilizes four checks to test for facet intersection (Glassner[1997]):

CASE #1: If any vertices are inside the cube, the facet must intersect the cube.

CASE #2: If all the vertices are to one side of the cube, the facet cannot intersect the cube.

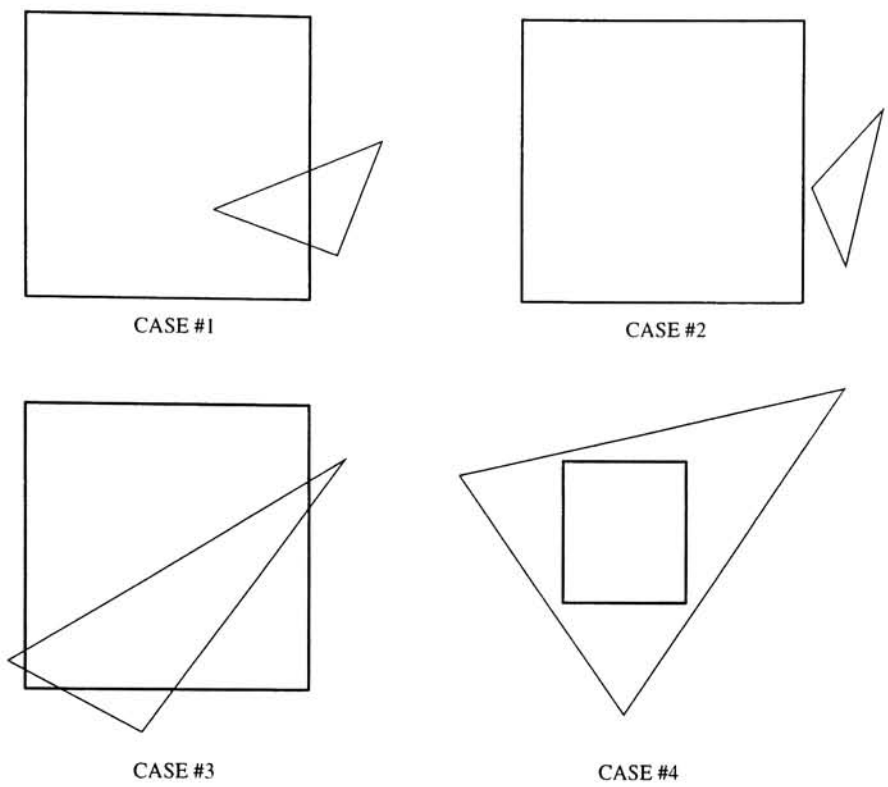

Figure A.6: A 2D representation of the four tests scenarios used in the facet-box intersection algorithm.

CASE #3: If any edge intersects the cube, the facet must intersect the cube

CASE #4: If the cube cuts the plane of the facet, the facet must intersect the cube.

The `RT_Add_Child_To_Octant()` routine adds the facet to the child list for that octant. If the new facet exceeds the threshold for subdivision, the octant is subdivided by the `RT_Split_Octant()` routine. This routine allocates and initializes the eight new sub-octants, and then sorts the facets in the original octant into the sub-octants.

A.3.4 Ray Tracing

Having built the octree in the preprocessing stage of the algorithm, the procedure for ray tracing now becomes simpler. For any given ray, the algorithm proceeds as such:

- If a starting leaf octant is not provided, then the first leaf octant along the path of the ray is used as the starting octant. The ray is tested for intersection with all the facets within an octant. To identify the closest hit to the ray origin, all hits are stored in a list sorted based on distance from the ray origin. In the event that no intersections are found within the this octant, the next octant along the path must be determined.
- To find the next octant along the ray, a new point is computed in the neighboring octant along the path of the ray. The new point is half the size of the smallest octant perpendicular from the exited face of the current octant. This point is guaranteed to be within the next octant without skipping the smallest octant.
- The process repeats itself until a facet is intersected or the ray leaves the world octant.

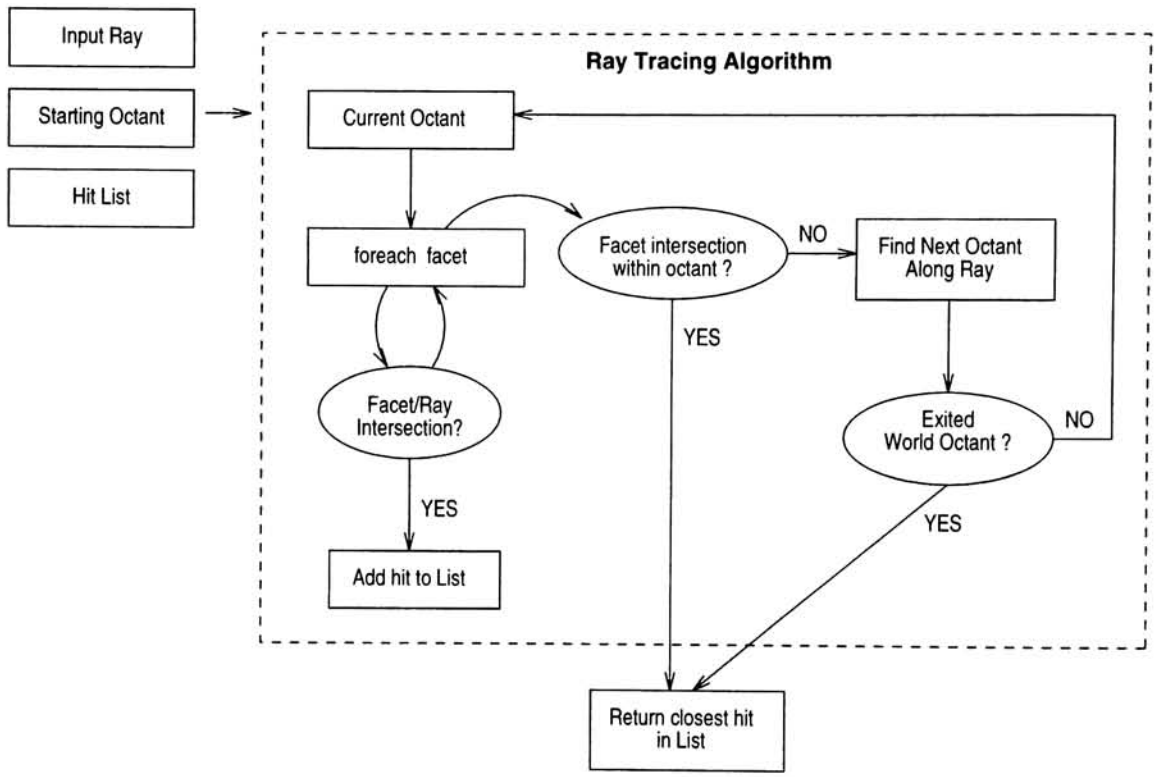

Figure A.7: Ray tracing with the use of an octree.

Ray tracing a ray is accomplished by the calling the `Trace_Ray()` routine. This function is passed the ray to be traced, the world to trace within and a list to store the facet intersection information (see Figure A.8). In addition, a variable of enumerated data type `RT_TRACE_MODE` is passed to indicate which of the two run modes

to use. When rendering an image, the option flag must be set to `CLOSEST_HIT_FLAG` so that the ray tracer returns after the facet closest to the ray origin has been determined. When this flag is set to `FIRST_HIT_FLAG`, the ray tracer will return immediately upon intersecting a facet. The later mode is useful in shadowing or obstruction cases where continuing to search for the closest facet obstructing the path does not provide any additional benefit.

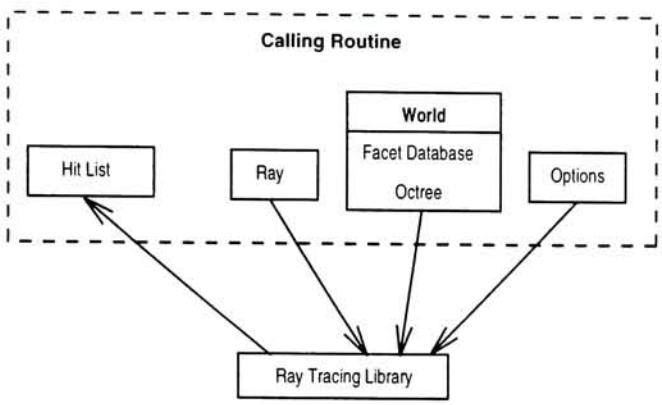

Figure A.8: The ray tracing interface to *libRT*

The primary function of `Trace_Ray()` is to initialize the hit list and to determine the starting octant to be used by the octree tracing algorithm, `RT_Trace_Octree()`. If the ray origin is outside the world octant, the starting octant is the first octant along the ray inside the world octant. To find this octant, the intersection of the ray with the world octant is determined and a *search point* just inside the world octant is computed. If the origin is within the world octant, the search point is simply set to the origin. In either case, the search point is passed to the `RT_Find_Octant()` routine to find the starting octant.

In special cases, the starting octant may be passed in by the routine calling `Trace_Ray()`. For ray traces originating from a previously determined hit point (*i.e.* shadow traces, background traces, etc.) the starting octant can be set to the octant within which the hit occurred (this octant is recorded in the hit list entry for this specific purpose). For DIRSIG, this can avoid repeating the search for the starting octant approximately 100 times per pixel. Avoiding this many searches becomes computationally significant when deep octrees are created.

The `RT_Trace_Octree()` function is responsible for tracing all the octants along the ray until a facet is intersected or the ray leaves the world octant. This search begins with the starting octant passed in by `Trace_Ray()`. The facets within each octant are traced by the `RT_Trace_Facets()` routine. If the ray does not intersect any facets within the current octant, then the next octant along the path must be identified. To find this octant, a new search point is determined using a modified ray-box intersection algorithm (see `RT_Ray_Box_Intersect()`) that returns the exit point and which octant face the ray exited from. The new search point is half the length of the smallest octant perpendicular from the exited face. If the ray exits at

an edge or a corner, then the appropriate adjustments are made to produce the correct search point.

All the facets within an octant are checked for intersection by the `RT_Trace_Facets()` routine. The facet intersection test consists of two steps: *(i)* find the intersection point of the ray with the plane of the facet (see `RT_Ray_Plane_Intersect()`) and *(ii)* check if the intersection point is within the bounds of the facet vertices (see `RT_Check_Vertices()`). Any facet passing these two tests is added to the hit list using the `RT_Add_Hit()` routine that utilizes an insertion sort to order the hits in increasing distance from the ray origin.

Glassner's original paper discusses a mechanism to avoid performing multiple intersection test on objects that span multiple octants, however, it does not discuss the "false termination" case depicted in Figure A.9.

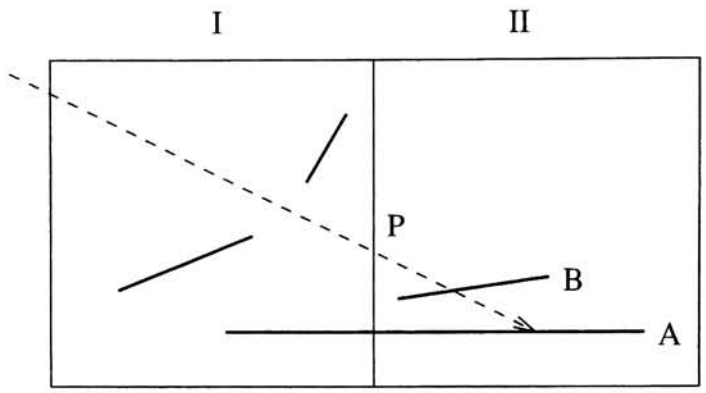

Figure A.9: A 2D representation of a "false termination" case.

The tracing of the octree is supposed to terminate when an intersected facet is found while testing facets contained in the *current* octant. Facet A is intersected while testing octant I, however, it should not terminate the search for the closest facet because the hit was outside the octant. Testing if the hit point is within the current octant can be accomplished by comparing the distance to the hit with the distance to the exit point, P, of the box. In this case, facet B is actually the closest facet which will be found while examining octant II. The alternative to this testing strategy is to cut facets which span more than one octant at the octant bounds. Since this is assumed to be a commonly occurring scenario, the overhead of facet clipping is cannot be justified when compared to the minimal cost of the distance test presented.

To avoid repeating intersection tests with a facet that spans multiple octants along the path of the ray (*i.e.* facet A in Figure A.9), Glassner suggests generating a unique key for each ray to mark facets which have been tested by the ray. This mechanism was integrated into the library using a simple key generation scheme. Ray keys are generated by the function `RT_Get_Ray_Key()` which returns the value of a large unsigned counter stored in the `RT_WORLD` data structure. The counter is incremented with every call to the `RT_Get_Ray_Key()`, and a initialization routine resets all the facets in the database when the counter overflows. The facet tracing routine compares the current ray key with last ray key stored in the facet, and if they are the same, then the intersection tests are skipped.

After all the facets have been examined in the current octant, the search termination case must be checked: If the closest hit (the first in the hit list) is closer than the distance to the exit point for the current octant, then the routine returns the value `RT_COMPLETE` to indicate that the search for the closest facet is complete. Otherwise, the routine returns `RT_SUCCESS` to indicate normal operation, or `RT_FAILURE` if errors are encountered.

A.3.5 Round-off and Precision Strategy

Since most of the intersection algorithms produce metrics for intersection through a series of arithmetic operations, a general strategy to deal with accumulated round-off error and inexact comparisons had to be defined. For example, the expected value from a series of operations may be 0, however, the cumulative errors in computations might result in a value of 1.345E-15. Although these two values should be equal, the equivalence operator "==" will indicate that they are not.

To solve this problem, all the intersection routines test if a value falls into a small range rather than direct or exact equivalence. The width of this range is defined as `RT_SMALL_VALUE` in the include file `RT.h`. This value is currently set to 1E-6. In effect, this delta value extends the size of facets and octants in the ray tracer very slightly. For example, in the Ray/Box intersection test (see `RT_Ray_Box_Intersection()`) the octants overlap at the seams so that coplanar facets appear in both neighboring octants.

This precision range is only applied to comparisons within the intersection routines. If the octant searching algorithm, for example, also accounted for the precision range, the compounded precision variance could possible identify the wrong octant containing the search point.

Appendix B

Source Code

This appendix contains the selected source code (from `RT_Octree.c`) for the "ideal cut" subdivision method. These sections of code feature the only changes made to `libRT` to implement this alternate subdivision method. These code sections can implement any of the investigated methods

B.1 Create_Octree

```
/*
 * NAME:         Create_Octree
 * PURPOSE:      Creates an octree from the facets loaded in the facet list
 *               stored in the RT_WORLD variable.
 * INPUTS:       This routine has the following inputs:
 *                   1. The world to build the octree for (type = RT_WORLD)
 * OUTPUTS:      The base of the octree created by this routine is stored in
 *               the world variable passed in.
 * RETURN:       The functions returns RT_SUCCESS under normal operation,
 *               and RT_FAILURE when errors arise.  All errors are reported
 *               in detail by the subroutines that encounter them.
 * SUBROUTINES: This function makes calls to the following routines:
 *               RT_Sort_Facet()
 * AUTHOR:       S.D. Brown
 */
int Create_Octree( RT_OCTANT_STYLE      style,
                   RT_WORLD             *world )
```

```
{
    int                 j;
    double              size, max_size = 0;
    RT_POINT            mid_point;
    RT_INDEX            block_index, facet_index;
    RT_FACET            *facet = NULL;

    printf( "Creating octree ... " );
    fflush( stdout );

    /* check what style of octant to create */
    if( style == CUBE_STYLE ) {

        /* find the maximum dimension of the world */
        for( j = 0; j < NUM_DIMS; j++ ) {

            /* compute the size of this dimension */
            size = fabs( world->extents.max[ j ] - world->extents.min[ j ]);

            /* check if this is the biggest dimension */
            if( size > max_size )
                max_size = size;

            /* compute the mid point for this dimension */
            mid_point[ j ] =
                ( world->extents.min[ j ] + world->extents.max[ j ] ) / 2.0;
        }

        /* set the min and max of the bounding octant */
        for( j = 0; j < NUM_DIMS; j++ ) {
            world->octant.box.min[ j ] = mid_point[ j ] - ( max_size / 2.0 );
            world->octant.box.mid[ j ] = 0.0;
            world->octant.box.max[ j ] = mid_point[ j ] + ( max_size / 2.0 );
        }
    }

    /* set the size of the master octant */
    if( style == RECTANGLE_STYLE ) {

        /* find the maximum dimension of the world */
        for( j = 0; j < NUM_DIMS; j++ ) {
```

```
        /* compute the size of this dimension */
        size = fabs( world->extents.max[ j ] - world->extents.min[ j ]);

        /* assign the appropriate values to the world octant */
        world->octant.box.min[ j ] = world->extents.min[ j ];
        world->octant.box.mid[ j ] = 0.0;
        world->octant.box.max[ j ] = world->extents.max[ j ];
    }
}

/* initialize the world variables */
world->max_depth = 1;
world->num_total_octants = 1;
world->num_empty_octants = 1;
world->octant.depth = 1;

/* compute the size for each dimension */
for( j = 0; j < NUM_DIMS; j++ ) {
    world->octant.size[ j ] =
        fabs( world->octant.box.max[ j ] - world->octant.box.min[ j ]);
}

/* initialize the size of the smallest octant */
RT_VECTOR_COPY( world->octant.size, world->min_size );

/*
 * The octree generation process starts by allocating enough
 * space for all the facets to fit in the world octant.
 */
world->octant.atts = FACET_CHILDREN;
if(( world->octant.facets = ( RT_POINTER * )
        calloc((size_t) world->facet_count, sizeof( RT_POINTER * ))) == NULL ) {
    printf( "\nCreate_Octree: ALLOC ERROR\n" );
    printf( "\tCould not allocate storage for facets!\n" );
    printf( "\tRequested %d facets at depth of %d\n",
            world->facet_count, world->octant.depth );
    return( RT_FAILURE );
}
world->octant.sub_octants = NULL;
```

```
/*
 * Add all the facets to the world octant
 */
for( block_index = 0; block_index < world->num_blocks; block_index++ ) {

    /* check each facet in this block */
    for( facet_index = 0;
         facet_index < world->facet_blocks[ block_index ]->num_facets;
         facet_index++ ) {

        /* a short-cut to the current facet */
        facet =
            &( world->facet_blocks[ block_index ]->facets[ facet_index ]);

        /* now add the facet to the list */
        world->octant.facets[ world->octant.facet_count ] =
                ( RT_POINTER )facet;
        world->octant.facet_count += 1;

        /*
         * All the facets are bound by the world octant
         * Update the mean point of the octant
         */
        world->octant.bound_count += 1;
        world->octant.box.mid[ 0 ] += facet->geometry.avg_vertex[ 0 ];
        world->octant.box.mid[ 1 ] += facet->geometry.avg_vertex[ 1 ];
        world->octant.box.mid[ 2 ] += facet->geometry.avg_vertex[ 2 ];

        if( world->octant.facet_count == 1 )
            world->num_empty_octants -= 1;

    }
}


/*
 * Start the spatial sorting process by checking if the world
 * octant must be recursively split.
 */
RT_Optimize_Octant( world, &( world->octant ));
```

```
    printf( "done.\n" );
    printf( "    Minimum scene extent = %0.4f, %0.4f, %0.4f\n",
            world->extents.min[ 0 ],
            world->extents.min[ 1 ],
            world->extents.min[ 2 ]);
    printf( "    Maximum scene extent = %0.4f, %0.4f, %0.4f\n",
            world->extents.max[ 0 ],
            world->extents.max[ 1 ],
            world->extents.max[ 2 ]);
    printf( "    Minimum octant size = %0.4f, %0.4f, %0.4f\n",
            world->min_size[ 0 ],
            world->min_size[ 1 ],
            world->min_size[ 2 ]);
    printf( "    Facets/octant threshold = %d\n", world->max_facets );
    printf( "    Total number of octants = %d\n", world->num_total_octants );
    printf( "    Number of empty octants = %d\n", world->num_empty_octants );
    printf( "    Maximum octree depth = %d\n", world->max_depth );

    return( RT_SUCCESS );
}
```

B.2 RT_Optimize_Octant

```
/*
 * NAME:        RT_Optimize_Octant
 * PURPOSE:     This routine supports the "delayed ideal octant"
 *              mode of this ray-tracer.  It checks the facet population
 *              of the octant passed in and either subdivides it through
 *              the mean point of the facets in it or optimizes
 * INPUTS:      This routine has the following inputs:
 *                  1. A world containing this octant (type = RT_WORLD)
 *                  2. The octant to optimize (type = RT_OCTANT)
 * OUTPUTS:     The are no outputs for this function.
 * RETURN:      The functions returns RT_SUCCESS if the octants are
 *              allocated and initialized correctly and RT_FAILURE in
 *              other cases.
 * SUBROUTINES: This function does not call any subroutines.
 * AUTHOR:      S.D. Brown
 */
int RT_Optimize_Octant( RT_WORLD        *world,
```

```
                        RT_OCTANT       *octant )
{
    char                intersect_flag = RT_TRUE;
    int                 sort_count;
    unsigned int        j, index;
    RT_FACET            *facet = NULL;
    RT_FACET            **facet_list = NULL;
    RT_OCTANT           *new_octant = NULL;
    RT_OCTANT           *sub_octants = NULL;

#if USE_FREE
    /* check if there are any facets in this octant */
    if( octant->facet_count == 0 ) {
        free( octant->facets );
        octant->facets = NULL;
        return( RT_SUCCESS );
    }
#endif

    /* check if this octant must be subdivided */
    if( octant->facet_count > world->max_facets ) {

        for( j = 0; j < NUM_DIMS; j++ ) {

            /* compute the mid point of the octant */
            if( octant->bound_count != 0 ) {
                octant->box.mid[ j ] /= octant->bound_count;
            }
            else {
                octant->box.mid[ j ] =
                    octant->box.min[ j ] + octant->size[ j ] / 2.0;
            }

            /* adjust the midpoint if it is outside the octant */
            if(( octant->box.mid[ j ]-octant->box.min[ j ] < RT_SMALL_VALUE ) ||
               ( octant->box.max[ j ]-octant->box.mid[ j ] < RT_SMALL_VALUE )) {
                octant->box.mid[ j ] =
                    octant->box.min[ j ] + octant->size[ j ] / 2.0;
            }
        }
```

```
/* allocate a new octant with octant children */
if( RT_Create_Sub_Octants( &( octant->box ),
                           octant->size,
                           world->min_size,
                           octant->facet_count,
                           &sub_octants ) == RT_FAILURE ) {
    return( RT_FAILURE );
}

/* set the depth of the new sub-octants */
for( index = 0; index < MAX_OCTANTS; index++ ) {
    sub_octants[ index ].depth = octant->depth + 1;
}

/* attach the sub-octants to this octant */
octant->atts = OCTANT_CHILDREN;
octant->sub_octants = ( RT_POINTER )sub_octants;

/* update the info stored in the world */
if( octant->depth + 1 > world->max_depth ) {
    world->max_depth = octant->depth + 1;
}

/* update the number of octants being used */
world->num_total_octants += 7;
world->num_empty_octants += 8;

/* add all the facets in the original octant to the new octant */
for( index = 0; index < octant->facet_count; index++ ) {

    /* get the reference to this facet */
    facet = ( RT_FACET * )octant->facets[ index ];

    /* add this facet to the new octant */
    sort_count = 0;
    if( RT_Sort_Facet( facet, world, octant,
                       intersect_flag, &sort_count ) == RT_FAILURE )
        return( RT_FAILURE );

    /* check if this facet made it into at least one sub-octant */
    if( sort_count == 0 ) {
```

```
                printf( "\nRT_Optimize_Octant: SORT ERROR\n" );
                printf( "\tFacet was not in any of the sub-octants\n" );
                fflush( stdout );
                return( RT_FAILURE );
            }
        }

#if USE_FREE
        /* free the facet list for this octant */
        free( octant->facets );
        octant->facets = NULL;
#endif
        /* now optimize each sub-octant */
        for( index = 0; index < MAX_OCTANTS; index++ ) {
            new_octant = &((( RT_OCTANT * )( octant->sub_octants ))[ index ]);
            RT_Optimize_Octant( world, new_octant );

#ifdef _DEBUG
            if ( new_octant->facet_count > octant->facet_count
            || new_octant->facet_count < 0 ) {
                printf( "\nRT_Optimize_Octant: PANIC\n" );
                printf( "\tStrange things are afoot at the Circle-K!(TM)\n" );
                fflush( stdout );
                return( RT_FAILURE );
            }
#endif
        }

        octant->facet_count = 0;
        octant->bound_count = 0;

    }
#if USE_FREE
    else {

        /* allocate a list that is the right size */
        if(( facet_list = ( RT_FACET ** )
                calloc((size_t) octant->facet_count, sizeof( RT_FACET * ))) == NULL ) {
            printf( "\nRT_Optimize_Octant: ALLOC ERROR\n" );
            printf( "\tCould not allocate storage for facets!\n" );
            printf( "\tRequested %d facets at depth of %d\n",
```

```
                octant->facet_count, octant->depth );
            fflush( stdout );
            return( RT_FAILURE );
        }

        /* copy the facets from the old list to the new one */
        for( index = 0; index < octant->facet_count; index++ ) {
            facet_list[ index ] = ( RT_FACET * )octant->facets[ index ];
        }

        /* free the facet list for this octant */
        free( octant->facets );

        /* place the new list in this octant */
        octant->facets = ( RT_POINTER * )facet_list;
    }
#endif

    return( RT_SUCCESS );
}
```

B.3 RT_Sort_Facet

```
/*
 * NAME:        RT_Sort_Facet
 * PURPOSE:     Updates the entire octree by checking if octants intersect
 *              the new facet.
 * INPUTS:      This routine has the following inputs:
 *                  1. The facet to add (type = RT_FACET)
 *                  2. The octant to check (type = RT_OCTANT)
 * OUTPUTS:     All changes are to internal data structures.
 * RETURN:      The functions returns a RT_SUCCESS flag under normal
 *              operation, and RT_FAILURE when errors arise.  All errors
 *              are reported in detail by the subroutines that encounter
 *              them.
 * SUBROUTINES: This function makes calls to the following routines:
 *              RT_Intersect_Octant()
 *              RT_Add_Child_To_Octant()
 *              RT_Split_Octant()
 * AUTHOR:      S.D. Brown
```

```
 */
int RT_Sort_Facet( RT_FACET     *facet,
                   RT_WORLD     *world,
                   RT_OCTANT    *octant,
                   char         intersect_flag,
                   int          *sort_count )
{
    int         index;
    RT_BOX      box;
    RT_OCTANT   *sub_octant = NULL;
#ifdef _DEBUG
    int j;
#endif

    /* check if this octant is a leaf node (children are facets) */
    if( octant->atts == FACET_CHILDREN ) {
        *sort_count += 1;
        RT_Add_Facet_To_Octant( facet, world, octant );
        if( intersect_flag == RT_BOUND ) {
            /* update the mean point of the octant */
#ifdef _DEBUG
            for ( j = 0; j < NUM_DIMS; j++ ) {
                if ( facet->geometry.avg_vertex[ j ] < octant->box.min[ j ]
                || facet->geometry.avg_vertex[ j ] > octant->box.max[ j ] ) {
                    printf( "\nRT_Sort_Facet: WARNING\n" );
                    printf( "\tAverage Vertex outside of box\n" );
                    printf( "\tbox.min = (%f,%f,%f)\n", octant->box.min[0],
                        octant->box.min[1], octant->box.min[2] );
                    printf( "\tbox.max = (%f,%f,%f)\n", octant->box.max[0],
                        octant->box.max[1], octant->box.max[2] );
                    printf( "\tavg point = (%f,%f,%f)\n",
                        facet->geometry.avg_vertex[0],
                        facet->geometry.avg_vertex[1],
                        facet->geometry.avg_vertex[2] );
                    fflush( stdout );
                    return( RT_FAILURE );
                }
            }
            if ( octant->bound_count == 0 ) {
                for ( j = 0; j < NUM_DIMS; j++ ) {
                    if ( octant->box.mid[ j ] < -RT_SMALL_VALUE
```

```
                    || octant->box.mid[ j ] > RT_SMALL_VALUE ) {
                        printf( "\nRT_Sort_Facet: WARNING!\n" );
                        printf( "\tMID did not start at zero\n" );
                        printf( "\tbox.mid = (%f,%f,%f)\n", octant->box.mid[0],
                            octant->box.mid[1], octant->box.mid[2] );
                        fflush( stdout );
                        return( RT_FAILURE );
                    }
                }
            }
#endif
            octant->box.mid[ 0 ] += facet->geometry.avg_vertex[ 0 ];
            octant->box.mid[ 1 ] += facet->geometry.avg_vertex[ 1 ];
            octant->box.mid[ 2 ] += facet->geometry.avg_vertex[ 2 ];
            octant->bound_count += 1;
        }
        return( RT_SUCCESS );
    }

    /* check the facet against each sub-octant */
    for( index = 0; index < MAX_OCTANTS; index++ ) {

        /* create a shortcut */
        sub_octant = &((( RT_OCTANT * )( octant->sub_octants ))[ index ]);

        /* the Facet/Box intersection is destructive, copy the box */
        RT_POINT_COPY( sub_octant->box.min, box.min );
        RT_POINT_COPY( sub_octant->box.max, box.max );

        /* test if the facet intersects this octant's box */
        intersect_flag = RT_Facet_Box_Intersect( facet, &box );

        /* check if facet intersected this octant */
        if( intersect_flag != RT_FALSE ) {
            /* sort it into this sub_octant */
            if( RT_Sort_Facet( facet, world, sub_octant,
                               intersect_flag, sort_count ) == RT_FAILURE ) {
                return( RT_FAILURE );
            }
        }
    }
```

```
    return( RT_SUCCESS );
}
```

B.4 RT_Find_Octant

```
/*
 * NAME:         RT_Find_Octant
 * PURPOSE:      Given a point, this routine finds the sub-octant that the
 *               point is in.
 * INPUTS:       This routine has the following inputs:
 *                   1. The search point (type = RT_POINT)
 *                   1. The starting octant (type = RT_OCTANT)
 * OUTPUTS:      The function updates the the pointer "found_octant" to
 *               point to the octant containing the search point (if one
 *               was found).
 *                   1. The found octant (type = RT_OCTANT)
 * RETURN:       The functions returns RT_SUCCESS if an octant is found
 *               which contains the search point and RT_FAILURE otherwise.
 * SUBROUTINES: This function makes calls to the following routines:
 *               RT_Octant_Index()
 * REFERENCES:   The octant implementation used here stores pointers to
 *               the sub-octants in each octant so that they may be accessed
 *               directly.  The original published implementation
 *               (Glassner[1984]) uses a complex 4-step hashing lookup.
 *               See the "RT_README" file for the complete reference.
 */
int RT_Find_Octant( RT_POINT    point,
                    RT_OCTANT   *octant,
                    RT_OCTANT   **found_octant )
{
#if _DEBUG
    int                 j;
#endif
    int                 index = -1;
    RT_OCTANT           *new_octant;

    /* keep looping if the children are not facets */
    while(( octant->atts & FACET_CHILDREN ) == RT_FALSE ) {
```

```
        /* figure out the index to the desired child */
        index = RT_Octant_Index( point, &( octant->box ), octant->size );

        /* if the index is negative, the point is not in an octant */
        if( index < 0 )
            return( RT_FAILURE );

        /* assign the octant */
        new_octant = &((( RT_OCTANT * )( octant->sub_octants ))[ index ]);

#if _DEBUG
        /* sanity check */
        for( j = 0; j < NUM_DIMS; j++ ) {
            if(( point[ j ] < new_octant->box.min[ j ]) ||
               ( point[ j ] > new_octant->box.max[ j ])) {
                printf( "\nRT_Find_Octant: ERROR\n" );
                printf( "\tIncorrect octant found!\n" );
                return( RT_FAILURE );
            }
        }
#endif
        octant = new_octant;
    }

    /* if the world octant has facet children, check the point location */
    if( index < 0 ) {
        if( RT_Octant_Index( point, &( octant->box ), octant->size ) < 0 )
            return( RT_FAILURE );
    }

    /* return this octant as the one containing the point */
    *found_octant = octant;

    /* life is good */
    return( RT_SUCCESS );
}
```

B.5 RT_Add_Facet_To_Octant

```
/*
```

```
 * NAME:         RT_Add_Facet_To_Octant
 * PURPOSE:      Adds a facet to an octant's child list.  The routine
 *               first scans the list to make sure that the facet is not
 *               already in the list (which may result after splitting a
 *               octant).
 * INPUTS:       This routine has the following inputs:
 *                   1. The facet to add (type = RT_FACET)
 *                   2. The octant (type = RT_OCTANT)
 * OUTPUTS:      All changes are to internal data structures.
 * RETURN:       The functions returns a RT_SUCCESS if the facet is
 *               successfully added to the child list and RT_FAILURE if not.
 * SUBROUTINES: This function calls the following subroutines:
 *               (none)
 * AUTHOR:       S.D. Brown
 */
int RT_Add_Facet_To_Octant( RT_FACET    *facet,
                            RT_WORLD    *world,
                            RT_OCTANT   *octant )
{

#if _DEBUG
    /* check if this is a legal octant */
    if( octant->atts != FACET_CHILDREN ) {
        printf( "\nRT_Add_Facet_To_Octant: ERROR\n" );
        printf( "\tAttempting to add a facet to bad octant!\n" );
        return( RT_FAILURE );
    }
#endif

    /* now add the facet to the list */
    octant->facets[ octant->facet_count ] = ( RT_POINTER )facet;
    octant->facet_count += 1;

    if( octant->facet_count == 1 )
        world->num_empty_octants -= 1;

    return( RT_SUCCESS );

}
```

B.6 RT_Create_Sub_Octants

```
/*
 * NAME:         RT_Create_Sub_Octants
 * PURPOSE:      Creates/allocates an octant and initializes the internal
 *               values depending on the type of octant requested
 * INPUTS:       This routine has the following inputs:
 *                   1. The box defining the parent (type = RT_BOX)
 *                   2. The size of the parent box (type = RT_VECTOR)
 *                   3. The number of facets to allocate (type = unsigned int)
 * OUTPUTS:      The functions sets the pointer passed in to the location
 *               of the newly allocated and initialized sub-octants.
 * RETURN:       The functions returns RT_SUCCESS if the octants are
 *               allocated and initialized correctly and RT_FAILURE in
 *               other cases.
 * SUBROUTINES: This function makes calls to the following subroutines:
 *               RT_Octant_Index()
 * AUTHOR:       S.D. Brown
 */
int RT_Create_Sub_Octants( RT_BOX       *box,
                           RT_VECTOR    size,
                           RT_VECTOR    min_size,
                           unsigned int num_facets,
                           RT_OCTANT    **sub_octants )
{
    int         x_index, y_index, z_index;
    RT_INDEX    index;
    RT_OCTANT   new_octant;
    RT_POINT    tmp_point;
    static int  key_index = 0;
#ifdef _DEBUG
    int         suboctant_mask = 0;
#endif

    /* allocate the sub-octants */
    if((( *sub_octants ) = ( RT_OCTANT * )
            calloc((size_t) MAX_OCTANTS, sizeof( RT_OCTANT ))) == NULL ) {
        printf( "\nRT_Create_Sub_Octants: ALLOC ERROR\n" );
        printf( "\tCould not allocate sub-octants!\n" );
        printf( "\tRequested %d octants\n", MAX_OCTANTS );
        return( RT_FAILURE );
```

```
}

/* allocate the child octants to hold facets */
for( z_index = 0; z_index < 2; z_index++ ) {

    /* set this dimension of the size */
    if( z_index == 0 ) {
        new_octant.size[ 2 ] = box->mid[ 2 ] - box->min[ 2 ];
        new_octant.box.min[ 2 ] = box->min[ 2 ];
        new_octant.box.mid[ 2 ] = 0.0;
        new_octant.box.max[ 2 ] = box->mid[ 2 ];
    }
    else {
        new_octant.size[ 2 ] = box->max[ 2 ] - box->mid[ 2 ];
        new_octant.box.min[ 2 ] = box->mid[ 2 ];
        new_octant.box.mid[ 2 ] = 0.0;
        new_octant.box.max[ 2 ] = box->max[ 2 ];
    }
    tmp_point[ 2 ] =
        new_octant.box.min[ 2 ] + new_octant.size[ 2 ] / 2.0;
    if( new_octant.size[ 2 ] < min_size[ 2 ] )
        min_size[ 2 ] = new_octant.size[ 2 ];

    for( y_index = 0; y_index < 2; y_index++ ) {

        /* set this dimension of the size */
        if( y_index == 0 ) {
            new_octant.size[ 1 ] = box->mid[ 1 ] - box->min[ 1 ];
            new_octant.box.min[ 1 ] = box->min[ 1 ];
            new_octant.box.mid[ 1 ] = 0.0;
            new_octant.box.max[ 1 ] = box->mid[ 1 ];
        }
        else {
            new_octant.size[ 1 ] = box->max[ 1 ] - box->mid[ 1 ];
            new_octant.box.min[ 1 ] = box->mid[ 1 ];
            new_octant.box.mid[ 1 ] = 0.0;
            new_octant.box.max[ 1 ] = box->max[ 1 ];
        }
        tmp_point[ 1 ] =
            new_octant.box.min[ 1 ] + new_octant.size[ 1 ] / 2.0;
        if( new_octant.size[ 1 ] < min_size[ 1 ] )
```

```
        min_size[ 1 ] = new_octant.size[ 1 ];

    for( x_index = 0; x_index < 2; x_index++ ) {

        /* set this dimension of the size */
        if( x_index == 0 ) {
            new_octant.size[ 0 ] = box->mid[ 0 ] - box->min[ 0 ];
            new_octant.box.min[ 0 ] = box->min[ 0 ];
            new_octant.box.mid[ 0 ] = 0.0;
            new_octant.box.max[ 0 ] = box->mid[ 0 ];
        }
        else {
            new_octant.size[ 0 ] = box->max[ 0 ] - box->mid[ 0 ];
            new_octant.box.min[ 0 ] = box->mid[ 0 ];
            new_octant.box.mid[ 0 ] = 0.0;
            new_octant.box.max[ 0 ] = box->max[ 0 ];
        }
        tmp_point[ 0 ] =
            new_octant.box.min[ 0 ] + new_octant.size[ 0 ] / 2.0;
        if( new_octant.size[ 0 ] < min_size[ 0 ] )
            min_size[ 0 ] = new_octant.size[ 0 ];

        /* this octant has facet children */
        new_octant.atts = FACET_CHILDREN;
        new_octant.key = key_index;
        key_index++;
        new_octant.facet_count = 0;
        new_octant.bound_count = 0;

        /* allocate the the points to the sub-octants for this octant */
        if(( new_octant.facets = ( RT_POINTER * )
                calloc((size_t) num_facets,
                        sizeof( RT_POINTER * ))) == NULL ) {
            printf( "\nRT_Create_Sub_Octants: ALLOC ERROR\n" );
            printf( "\tCould not allocate storage for facets!\n" );
            printf( "\tRequested %d facets\n", num_facets );
            return( RT_FAILURE );
        }

        /* set the octant pointer to NULL */
        new_octant.sub_octants = NULL;
```

```
                /* assign this sub-octant to the parent octant */
                index = RT_Octant_Index( tmp_point, box, size );
                ( *sub_octants )[ index ] = new_octant;
#ifdef _DEBUG
                suboctant_mask |= (1 << index);
#endif

            }
        }
    }

#ifdef _DEBUG
    if ( suboctant_mask != (1 << MAX_OCTANTS)-1  ) {
        printf( "\nMissed some suboctants = %02x\n", suboctant_mask );
        return RT_FAILURE;
    }
#endif

    return( RT_SUCCESS );
}
```

B.7 RT_Octant_Index

```
/*
 * NAME:         RT_Octant_Index
 * PURPOSE:      Given a point, this function returns the index to the
 *               sub-octant.
 * INPUTS:       This routine has the following inputs:
 *                   1. A point within the parent (type = RT_POINT)
 *                   2. The box defining the parent (type = RT_BOX)
 *                   3. The size of the paret box (type = RT_VECTOR)
 * OUTPUTS:      The are no outputs for this function.
 * RETURN:       The functions returns the index to the sub-octant
 *               containing the point passed in.  In the event that the
 *               point is outside any dimensional bound of the box, an index
 *               of -1 is returned.
 * SUBROUTINES: This function does not call any subroutines.
 * AUTHOR:       S.D. Brown
 */
```

```
int RT_Octant_Index( RT_POINT   point,
                     RT_BOX     *box,
                     RT_VECTOR  size )
{
    int         j;
    int         index = 0;

    /* build the index based on the each dimension */
    for( j = 0; j < NUM_DIMS; j++ ) {

        /* start to forge the index */
        index *= 2;

        /* check which side of the mid-point it is on */
        if(( point[ j ] >= box->min[ j ] ) &&
           ( point[ j ] < box->mid[ j ] ))
            /* index += 0 */;
        else if(( point[ j ] <= box->max[ j ] ) &&
                ( point[ j ] >= box->mid[ j ]))
            index += 1;
        else
            return( -1 );
    }

    return( index );

}
```

Appendix C

make_scene Design Document

This program and this appendix were created and documented using Norman Ramsey's literate programming tool `noweb(1)`. This tool allows the programmer to break their source code into logical "chunks" and document each "chunk". Each code chunk begins with a "chunk name" and the section of code associated with that chunk. At the beginning of the chunk, the page number and "chunk number" for the code section are autmatically generated in the left hand margin. The "chunk name" for that section of code can then be referenced from any other chunk. When a chunk is referenced, the page number and that the code section appears on is automatically printed so that the reader can quickly find it.

C.1 Introduction

This program was written to generate pseudo-random scenes featuring various spatial distributions of facets. All facets are 3-sided with various sizes and rotations. The final locations of the facets is determined by the appropriate spatial location and distribution parameters specified in the input file.

C.2 Program Definition

The complete program source can be extracted from this `noweb` file using the chunk definition `Program`:

⟨*Program* 95⟩≡
- ⟨*Include Files* 96a⟩
- ⟨*Constant Definitions* 96c⟩
- ⟨*Type Definitions* 97a⟩
- ⟨*Function Prototypes* 98⟩
- ⟨*Main Program* 99a⟩
- ⟨*Support Routines* 101⟩
- ⟨*Input Reading Routines* 113⟩

C.2.1 Include Files

In addition to standard C include requirements, this program relies on the ray-tracing library (libRT) for types and some geometric support functions.

96a ⟨*Include Files* 96a⟩≡ (95) 96b▷

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "DIRSIG.h"
#include "RT.h"
```

The TAG file routines defines a set of TAG delimitation characters which are used by the `strtok()` function.

96b ⟨*Include Files* 96a⟩+≡ (95) ◁96a

```
#define NEED_TAG_DELIMS
#include "TAG_File.h"
```

C.2.2 Constant Definitions

The following constants are defined that specify the mean and variance in the rotation angles for the facets generated by this program. In any dimension, the angles used will have a mean of 180 degrees plus/minus 180 degrees.

96c ⟨*Constant Definitions* 96c⟩≡ (95) 96d▷

```
#define AVERAGE_X_ANGLE         M_PI
#define DELTA_X_ANGLE           M_PI
#define AVERAGE_Y_ANGLE         M_PI
#define DELTA_Y_ANGLE           M_PI
#define AVERAGE_Z_ANGLE         M_PI
#define DELTA_Z_ANGLE           M_PI
```

The constant `MAX_DISTS` defines the maximum number of distributions that this tool can generate in a single run.

96d ⟨*Constant Definitions* 96c⟩+≡ (95) ◁96c

```
#define MAX_DISTS               32
```

Defines:
`MAX_DISTS`, used in chunk 99b.

C.2.3 Type Definitions

A simple enumerated type is created to define the type of facet distribution to be generated.

⟨*Type Definitions* 97a⟩≡ (95) 97b▷

```
typedef enum _dist_type {
    UNIFORM,
    GAUSSIAN
} DIST_TYPE;
```

Defines:
DIST_TYPE, used in chunks 97b, 98, and 104.

The parameters for a given facet distribution in the input file are stored in the following type:

⟨*Type Definitions* 97a⟩+≡ (95) ◁97a

```
typedef struct _facet_distribution {
    unsigned int        index;

    /* size of the facets */
    float               average_size;
    float               delta_size;

    /* distribution of the facets */
    DIST_TYPE           dist_type;
    float               average_x;
    float               delta_x;
    float               average_y;
    float               delta_y;
    float               average_z;
    float               delta_z;

    /* storage area for facets */
    unsigned int        count;
    RT_FACET            *facets;

} FACET_DISTRIBUTION;
```

Defines:
FACET_DISTRIBUTION, used in chunks 98, 99b, 101, 110, 114, 116, and 118.
Uses DIST_TYPE 97a.

C.2.4 Internal Function Prototypes

This program source code contains the main routine and an additional routine that constructs the scene being generated.

98 ⟨*Function Prototypes* 98⟩≡ (95)

```
void Make_Facets( FACET_DISTRIBUTION    *dist );

double Random( DIST_TYPE        type );

int Rotate_Facet( RT_FACET              *facet,
                  double                x_angle,
                  double                y_angle,
                  double                z_angle );

int Translate_Facet( RT_FACET   *facet,
                     RT_VECTOR  translate_vector );

int Add_Facet_Normal( RT_FACET  *facet );

int Write_GDB_File( char                filename[],
                    unsigned int        num_dists,
                    FACET_DISTRIBUTION  dists[] );

int Read_Input_File( char               filename[],
                     unsigned int       *num_dists,
                     FACET_DISTRIBUTION facet_dists[] );
```

Uses Add_Facet_Normal 108, DIST_TYPE 97a, FACET_DISTRIBUTION 97b, Random 104, Read_Input_File 114, Rotate_Facet 105, Translate_Facet 107, and Write_GDB_File 110.

C.3 Main Routine

The program itself consists of four distinct stages, each of which will be defined in the following sections:

99a ⟨*Main Program* 99a⟩≡ (95)

```
int main( int   argc,
          char  *argv[] )
{
    ⟨Declarations 99b⟩
    ⟨Read the input file 99c⟩
    ⟨Generate the facets in the scene 100a⟩
    ⟨Output the scene to a file 100c⟩

    return( 0 );
}
```

C.3.1 Declarations

This program defines a set of variables to create the facets within the scene and to define the statistical distribution of the facets in the scene.

99b ⟨*Declarations* 99b⟩≡ (99a)

```
unsigned int                index;
unsigned int                num_dists = 0;
static FACET_DISTRIBUTION   facet_dists[ MAX_DISTS ];
```

Uses FACET_DISTRIBUTION 97b and MAX_DISTS 96d.

C.3.2 Reading the Input File

The input file is a TAG based file that can be read using the TAG file format routines developed by myself for another project.

99c ⟨*Read the input file* 99c⟩≡ (99a)

```
if( argc != 3 ) {
    printf( "usage: make_scene <input file> <output file>\n" );
    exit( -1 );
}
Read_Input_File( argv[ 1 ], &num_dists, facet_dists );
```

Uses Read_Input_File 114.

C.3.3 Generating the Scene

The facets in the scene are generated using the uniform distribution random number generator available in most UNIX development environments. Therefore, the first step is to initialize the random number generator. The use of a constant seed allows scenes to be recreated for the time being.

100a ⟨*Generate the facets in the scene* 100a⟩≡ (99a) 100b▷

```
srandom( 1 );
```

The core of the program is a main loop that constructs random facets that fit the statistical distribution(s) defined by the user.

100b ⟨*Generate the facets in the scene* 100a⟩+≡ (99a) ◁100a

```
printf( "Creating scene ..." );
fflush( stdout );
for( index = 0; index < num_dists; index++ ) {
    Make_Facets( &( facet_dists[ index ]));
}
printf( "done.\n" );
```

C.3.4 Outputting the scene to a file

When all of the facets have been created, the scene is output to a DIRSIG compatible (GDB format) file that can be used for testing the libRT ray-tracer.

100c ⟨*Output the scene to a file* 100c⟩≡ (99a)

```
Write_GDB_File( argv[ 2 ], num_dists, facet_dists );
```

Uses Write_GDB_File 110.

C.4 Constructing a Facet Distribution

The algorithm for constructing each facet is very basic. Each facet begins from a base size (specified by the user) and shape. The facet is then randomly rotated about all three primary axis, and then translated to its final position.

⟨*Support Routines* 101⟩≡ (95) 104▷

```
void Make_Facets( FACET_DISTRIBUTION    *dist )
{
    unsigned int        index;
    float               size;
    float               x_angle, y_angle, z_angle;
    RT_VECTOR           translate_vector;
    RT_FACET            *facet = NULL;

    if(( dist->facets = ( RT_FACET * )
            calloc( dist->count, sizeof( RT_FACET ))) == NULL ) {
        printf( "\nmake_scene: ALLOC ERROR\n" );
        printf( "\tCould not allocate %d facets!\n", dist->count );
        printf( "\tExiting ...\n" );
        exit( -1 );
    }

    for( index = 0; index < dist->count; index++ ) {
        ⟨Compute facet vertices 102a⟩
        ⟨Rotate facet 102b⟩
        ⟨Translate facet 103a⟩
        ⟨Computing the Normal Vector 103b⟩
        ⟨Add facet attributes 103c⟩
    }
}
```

Defines:
  Construct_Facet, never used.
Uses FACET_DISTRIBUTION 97b.

C.4.1 Computing the Facet Vertices

The size of the facet is computed using the mean and variance of the size specified by the user. The vertices are then computed and stored in the facet.

102a ⟨*Compute facet vertices* 102a⟩≡ (101)

```
facet = &( dist->facets[ index ]);

size = dist->average_size +
    ( dist->delta_size * (( random() / ( float )RAND_MAX ) - 0.5 ));

facet->geometry.num_vertices = 3;
facet->geometry.vertices[ 0 ][ 0 ] = sqrt( 2.0 ) * size;
facet->geometry.vertices[ 0 ][ 1 ] = 0.0;
facet->geometry.vertices[ 0 ][ 2 ] = 0.0;

facet->geometry.vertices[ 1 ][ 0 ] = size;
facet->geometry.vertices[ 1 ][ 1 ] = -size;
facet->geometry.vertices[ 1 ][ 2 ] = 0.0;

facet->geometry.vertices[ 2 ][ 0 ] = -size;
facet->geometry.vertices[ 2 ][ 1 ] = -size;
facet->geometry.vertices[ 2 ][ 2 ] = 0.0;
```

C.4.2 Rotating the Facet

The random rotation angles for each dimension are computed using constant distribution specifications. These rotation angles are then applied to the facet.

102b ⟨*Rotate facet* 102b⟩≡ (101)

```
x_angle =
    AVERAGE_X_ANGLE + ( DELTA_X_ANGLE * Random( UNIFORM ));
y_angle =
    AVERAGE_Y_ANGLE + ( DELTA_Y_ANGLE * Random( UNIFORM ));
z_angle =
    AVERAGE_Z_ANGLE + ( DELTA_Z_ANGLE * Random( UNIFORM ));

Rotate_Facet( facet, x_angle, y_angle, z_angle );
```

Uses Random 104 and Rotate_Facet 105.

C.4.3 Translating the Facet

The facet is then translated to its final location determined using the specifications supplied by the user.

103a ⟨*Translate facet* 103a⟩≡ (101)

```
translate_vector[ 0 ] =
    dist->average_x + ( dist->delta_x * Random( dist->dist_type ));
translate_vector[ 1 ] =
    dist->average_y + ( dist->delta_y * Random( dist->dist_type ));
translate_vector[ 2 ] =
    dist->average_z + ( dist->delta_z * Random( dist->dist_type ));

Translate_Facet( facet, translate_vector );
```

Uses Random 104 and Translate_Facet 107.

C.4.4 Computing the Facet Normal

Since the scenes are going to be rendered by DIRSIG, we need to compute the normal vector and assign basic rendering attributes to the facet.

103b ⟨*Computing the Normal Vector* 103b⟩≡ (101)

```
Add_Facet_Normal( facet );
```

Uses Add_Facet_Normal 108.

C.4.5 Adding the Facet Attributes

103c ⟨*Add facet attributes* 103c⟩≡ (101)

```
sprintf( facet->name, "FACET_%d_%d", dist->index+1, index+1 );
sprintf( facet->attributes.material_name, "steel" );
facet->attributes.material_code = 1;
facet->attributes.default_temperature = 0.0;
facet->attributes.thickness = 1.0;
```

C.5 Support Routines

The support routines for this scene generation tool include an interface to the standard random number generator that allows the user to produce both uniform and Gaussian distributed random numbers between 0.0 and 1.0. Additionally, two support routines are defined to do simple geometric transformations including the rotation of facets (`Rotate_Facet`) and facet translation (`Translate_Facet`). Additionally, a routine to compute the normal vector of a facet is defined (`Add_Facet_Normal`) which uses a traditional right hand rule.

C.5.1 Random Number Generation Routine

This scene generation tool allows the user to define spatial distributions that are either uniform or Gaussian distributed. It returns a random number between 0 and 1 based on the distribution type provided.

104 ⟨*Support Routines* 101⟩+≡ (95) ◁101 105▷

```
double Random( DIST_TYPE        type )
{
    double      val1, val2;
    double      noise;

    /* compute a random value */
    val1 = random() / ( double )RAND_MAX;
    val2 = random() / ( double )RAND_MAX;


    /* modify the random number based on the distribution type */
    switch( type ) {
        case UNIFORM:
            noise = val1 - 0.5;
            break;
        case GAUSSIAN:
            noise = 0.5 *
                ( sqrt( -2.0 * log( val1 )) * cos( 2.0 * M_PI * val2));
            break;
    }

    return( noise );
}
```

Defines:
Random, used in chunks 98, 102b, and 103a.
Uses DIST_TYPE 97a.

C.5.2 Facet Rotation Routine

This routine rotates a facet by computing a 3x3 transformation matrix (based on the dimensional rotation angles) and applies it to each vertex.

105 ⟨*Support Routines* 101⟩+≡ (95) ◁104 107▷

```
int Rotate_Facet( RT_FACET              *facet,
                  double                x_angle,
                  double                y_angle,
                  double                z_angle )
{
    int                         vertex;
    static RT_VECTOR            point_vector;
    static RT_VECTOR            new_point_vector;
    static RT_MATRIX            rotation_matrix;

    /* set up the rotation matrix */
    rotation_matrix[ 0 ][ 0 ] =
            cos( z_angle ) * cos( y_angle );

    rotation_matrix[ 0 ][ 1 ] =
            ( cos( z_angle ) * sin( y_angle ) * sin( x_angle )) -
            ( sin( z_angle ) * cos( x_angle ));

    rotation_matrix[ 0 ][ 2 ] =
            ( cos( z_angle ) * sin( y_angle ) * cos( x_angle )) +
            ( sin( z_angle ) * sin( x_angle ));


    rotation_matrix[ 1 ][ 0 ] =
            sin( z_angle ) * cos( y_angle );

    rotation_matrix[ 1 ][ 1 ] =
            ( sin( z_angle ) * sin( y_angle ) * sin( x_angle )) +
            ( cos( z_angle ) * cos( x_angle ));

    rotation_matrix[ 1 ][ 2 ] =
            ( sin( z_angle ) * sin( y_angle ) * cos( x_angle )) -
            ( cos( z_angle ) * sin( x_angle ));


    rotation_matrix[ 2 ][ 0 ] =
            -1.0 * sin( y_angle );
```

```
    rotation_matrix[ 2 ][ 1 ] =
            cos( y_angle ) * sin( x_angle );

    rotation_matrix[ 2 ][ 2 ] =
            cos( y_angle ) * cos( x_angle );


    /* now rotate the facet, one vertex at a time */
    for( vertex = 0; vertex < facet->geometry.num_vertices ; vertex++ ) {

        /* assign a vector pointing to the point */
        RT_POINT_COPY( facet->geometry.vertices[ vertex ], point_vector);

        new_point_vector[ 0 ] =
           (( rotation_matrix[ 0 ][ 0 ] * point_vector[ 0 ]) +
            ( rotation_matrix[ 0 ][ 1 ] * point_vector[ 1 ]) +
            ( rotation_matrix[ 0 ][ 2 ] * point_vector[ 2 ]));

        new_point_vector[ 1 ] =
           (( rotation_matrix[ 1 ][ 0 ] * point_vector[ 0 ]) +
            ( rotation_matrix[ 1 ][ 1 ] * point_vector[ 1 ]) +
            ( rotation_matrix[ 1 ][ 2 ] * point_vector[ 2 ]));

        new_point_vector[ 2 ] =
           (( rotation_matrix[ 2 ][ 0 ] * point_vector[ 0 ]) +
            ( rotation_matrix[ 2 ][ 1 ] * point_vector[ 1 ]) +
            ( rotation_matrix[ 2 ][ 2 ] * point_vector[ 2 ]));

        /* set this facet vertex to the new location */
        RT_POINT_COPY( new_point_vector, facet->geometry.vertices[ vertex ]);

    }

    /* return good tiddings of joy ... */
    return( RT_SUCCESS );
}
```

Defines:
Rotate_Facet, used in chunks 98 and 102b.

C.5.3 Facet Translation Routine

This routine adds the translation offsets to each vertex in the facet.

107 ⟨*Support Routines* 101⟩+≡ (95) ◁105 108▷

```
int Translate_Facet( RT_FACET   *facet,
                     RT_VECTOR  translate_vector )
{
    int         vertex;

    /* translate the facet, one vertex at a time */
    for( vertex = 0; vertex < facet->geometry.num_vertices ; vertex++ ) {
        facet->geometry.vertices[ vertex ][ 0 ] += translate_vector[ 0 ];
        facet->geometry.vertices[ vertex ][ 1 ] += translate_vector[ 1 ];
        facet->geometry.vertices[ vertex ][ 2 ] += translate_vector[ 2 ];
    }

    /* return good tiddings of joy ... */
    return( RT_SUCCESS );
}
```

Defines:
`Translate_Facet`, used in chunks 98 and 103a.

C.5.4 Facet Normal Computation Routine

This routine computes the normal to a facet by constructing a vector for two edges and then utilizing a cross product operation to compute the perpendicular vector.

108 ⟨*Support Routines* 101⟩+≡ (95) ◁107 110▷

```
int Add_Facet_Normal( RT_FACET  *facet )
{
    double      zenith, azimuth;
    RT_VECTOR   vector1;
    RT_VECTOR   vector2;
    RT_VECTOR   facet_normal;
    RT_VECTOR   earth_normal;

    /* set vector #1 */
    vector1[ 0 ] =
        facet->geometry.vertices[ 2 ][ 0 ] - facet->geometry.vertices[ 1 ][ 0 ];
    vector1[ 1 ] =
        facet->geometry.vertices[ 2 ][ 1 ] - facet->geometry.vertices[ 1 ][ 1 ];
    vector1[ 2 ] =
        facet->geometry.vertices[ 2 ][ 2 ] - facet->geometry.vertices[ 1 ][ 2 ];

    /* set vector #2 */
    vector2[ 0 ] =
        facet->geometry.vertices[ 1 ][ 0 ] - facet->geometry.vertices[ 0 ][ 0 ];
    vector2[ 1 ] =
        facet->geometry.vertices[ 1 ][ 1 ] - facet->geometry.vertices[ 0 ][ 1 ];
    vector2[ 2 ] =
        facet->geometry.vertices[ 1 ][ 2 ] - facet->geometry.vertices[ 0 ][ 2 ];

    /* normalize the two edge vectors */
    RT_Normalize_Vector( vector1 );
    RT_Normalize_Vector( vector2 );

    /* compute the normal vector */
    RT_Cross_Product( vector1, vector2, facet_normal );
    RT_Normalize_Vector( facet_normal );

    /* assign the normal to the facet */
    RT_VECTOR_COPY( facet_normal, facet->geometry.normal );

    /* compute zenith and azimuth of the facet */
    earth_normal[ 0 ] = 0.0;
```

```
    earth_normal[ 1 ] = 0.0;
    earth_normal[ 2 ] = 1.0;
    zenith = acos( RT_DOT_PRODUCT( facet->geometry.normal, earth_normal));
    if(( facet_normal[ 0 ] != 0.0 ) && ( facet_normal[ 1 ] != 0.0 ))
        azimuth = atan2( facet_normal[ 0 ], facet_normal[ 1 ]);
    else
        azimuth = 0.0;

    /* copy the zenith and azimuth */
    facet->geometry.zenith = RT_DEGREES( zenith );
    facet->geometry.azimuth = RT_DEGREES( azimuth );
    if( facet->geometry.azimuth < 0.0 )
        facet->geometry.azimuth = 360.0 - facet->geometry.azimuth;

    /* tell everyone who cares, things are OK */
    return( RT_SUCCESS );
}
```

Defines:
Add_Facet_Normal, used in chunks 98 and 103b.

C.5.5 GDB File Writing Routine

This routine writes out a list of facets to a DIRSIG GDB-format file.

110 ⟨*Support Routines* 101⟩+≡ (95) ◁108

```
int Write_GDB_File( char                filename[],
                    unsigned int        num_dists,
                    FACET_DISTRIBUTION  facet_dists[] )
{
    unsigned int        i, j;
    unsigned int        vertex;
    FILE                *fp = NULL;
    FACET_DISTRIBUTION  *dist = NULL;
    RT_FACET            *facet = NULL;

    printf( "Writing the scene to the output file ... " );
    fflush( stdout );

    if(( fp = fopen( filename, "w" )) == NULL ) {
        printf( "\nWrite_GDB_File: OPEN ERROR\n" );
        printf( "\tCould not open GDB file!\n" );
        printf( "\tFilename = %s\n", filename );
        printf( "\tExiting ...\n" );
        exit( RT_FAILURE );
    }

    fprintf( fp, "OBJECT\n" );
    fprintf( fp, "MAKE_SCENE_OBJ\n" );
    fprintf( fp, "1-0-0\n" );
    fprintf( fp, "PART\n" );
    fprintf( fp, "MAKE_SCENE_PART\n" );
    fprintf( fp, "1-1-0\n" );

    ⟨Print out each distribution 111a⟩

    fprintf( fp, "END\n" );
    fclose( fp );

    printf( "done.\n" );

    return( RT_SUCCESS );
}
```

Defines:

Write_GDB_File, used in chunks 98 and 100c.
Uses FACET_DISTRIBUTION 97b.

111a ⟨*Print out each distribution* 111a⟩≡ (110)

```
for( i = 0; i < num_dists; i++ ) {
    dist = &( facet_dists[ i ]);
    for( j = 0; j < dist->count; j++ ) {
        ⟨Print out an individual facet 111b⟩
    }
}
```

111b ⟨*Print out an individual facet* 111b⟩≡ (111a)

```
facet = &( dist->facets[ j ]);

fprintf( fp, "FACE\n" );
fprintf( fp, "%s\n", facet->name );
fprintf( fp, "1-1-%d\n", j );
fprintf( fp, "%s\n", facet->attributes.material_name );
fprintf( fp, "%ld\n", facet->attributes.material_code );
fprintf( fp, "%s\n", facet->name );
fprintf( fp, "%0.4f\n", facet->attributes.default_temperature );
fprintf( fp, "%0.4f\n", facet->attributes.thickness );
fprintf( fp, "-1.0000\n" );
fprintf( fp, "-1.0000\n" );
fprintf( fp, "null\n" );
fprintf( fp, "null\n" );
fprintf( fp, "null\n" );
fprintf( fp, "%d\n", facet->geometry.num_vertices );
for( vertex = 0; vertex < facet->geometry.num_vertices; vertex++) {
    fprintf( fp, "%0.10e %0.10e %0.10e\n",
        facet->geometry.vertices[ vertex ][ 0 ],
        facet->geometry.vertices[ vertex ][ 1 ],
        facet->geometry.vertices[ vertex ][ 2 ]);
}
fprintf( fp, "%0.15e %0.15e %0.15e\n",
    facet->geometry.normal[ 0 ],
    facet->geometry.normal[ 1 ],
    facet->geometry.normal[ 2 ]);
fprintf( fp, "%0.10e\n", facet->geometry.zenith );
fprintf( fp, "%0.10e\n", facet->geometry.azimuth );
fprintf( fp, "%e\n", 0.0 );
```

C.6 Input File

The input file for this scene generation tool consists of one or more entries that describe a facet distribution. Each entry describes the distribution type, the number of facets in the distribution and the extents of the distribution.

C.6.1 Example Input File

This section contains an example distribution entry from an input file. An input file may contain up to `MAX_DISTS` number of entries.

112 ⟨*Example Input File* 112⟩≡

```
DIST_ENTRY_BEGIN
    TYPE = GAUSSIAN
    COUNT = 1000
    AVERAGE_SIZE = 6.0
    DELTA_SIZE = 4.0
    AVERAGE_X = 0.0
    DELTA_X = 10000.0
    AVERAGE_Y = 0.0
    DELTA_Y = 10000.0
    AVERAGE_Z = 0.0
    DELTA_Z = 10000.0
DIST_ENTRY_END
```

C.6.2 TAG Definitions

To read the input file, the following TAGS are defined. Each TAG definition includes the string and an index identifier for the TAG.

113 ⟨*Input Reading Routines* 113⟩≡ (95) 114▷

```
static char     *_Input_TAGS[] = {
                                "DIST_ENTRY_BEGIN",
#define DIST_ENTRY_BEGIN_TAG    0
                                "TYPE",
#define TYPE_TAG                1
                                "COUNT",
#define COUNT_TAG               2
                                "AVERAGE_SIZE",
#define AVERAGE_SIZE_TAG        3
                                "DELTA_SIZE",
#define DELTA_SIZE_TAG          4
                                "AVERAGE_X",
#define AVERAGE_X_TAG           5
                                "DELTA_X",
#define DELTA_X_TAG             6
                                "AVERAGE_Y",
#define AVERAGE_Y_TAG           7
                                "DELTA_Y",
#define DELTA_Y_TAG             8
                                "AVERAGE_Z",
#define AVERAGE_Z_TAG           9
                                "DELTA_Z",
#define DELTA_Z_TAG             10
                                "DIST_ENTRY_END",
#define DIST_ENTRY_END_TAG      11
                                NULL
};
```

Defines:
_Input_TAGS, used in chunks 114 and 116.

C.6.3 Input File Reading Routine

This section contains all the routines related to reading the input file.

114 ⟨*Input Reading Routines* 113⟩+≡ (95) ◁113 116▷

```
int Read_Distribution_Value( char              *value_str,
                             TAG_ID            tag_id,
                             FACET_DISTRIBUTION *dist );

int Read_Distribution( TAG_FILE               *file,
                       FACET_DISTRIBUTION     *dist );

int Read_Input_File( char              filename[],
                     unsigned int      *num_dists,
                     FACET_DISTRIBUTION facet_dists[] )
{
    char               *tag_str = NULL;
    char               *value_str = NULL;
    TAG_ID             tag_id;
    TAG_FILE           file;
    static char        line[ MAX_STRING ];


    /* initialize the values */
    *num_dists = 0;

    /* check if the name of a map file was specified */
    if( filename == NULL )
        return( RT_SUCCESS );

    printf( "\nLoading input file ... " );
    fflush( stdout );

    /* open the MAP file */
    if( Open_Tag_File( filename, &file ) != DIRSIG_SUCCESS ) {
        printf( "\nRead_Input_File: OPEN ERROR\n" );
        printf( "\tCould not open mappings file!\n");
        printf( "\tFilename = %s\n", filename );
        printf( "\tExiting ...\n" );
        exit( RT_FAILURE );
    }

    /* read in the file */
```

```
    while( Read_File_Line( &file, line ) != DIRSIG_EOF ) {

        /* extract the TAG from the string */
        tag_str = strtok( line, tag_delims );

        /* figure out what kind of entry to read */
        if(( tag_id = Parse_Tag_String( tag_str, _Input_TAGS )) == NO_MATCH ) {
            printf( "\nRead_Input_File: FORMAT ERROR\n" );
            printf( "\tUnknown TAG read in file!\n" );
            printf( "\tTAG = %s\n", tag_str );
            printf( "\tFilename = %s\n", filename );
            printf( "\tLine number = %d\n", file.line_number );
            return( RT_FAILURE );
        }

        /* extract out the value string */
        value_str = strtok( NULL, tag_delims );

        switch( tag_id ) {
            case DIST_ENTRY_BEGIN_TAG:
                Read_Distribution( &file, &(facet_dists[ *num_dists ]));
                *num_dists += 1;
                break;

            default:
                break;
        }
    }

    printf( "done (%d distributions in file).\n", *num_dists );

    /* return all kinds of success */
    return( RT_SUCCESS );

}
```

Defines:
  Read_Input_File, used in chunks 98 and 99c.
Uses FACET_DISTRIBUTION 97b, _Input_TAGS 113, and Read_Distribution 116.

116 ⟨*Input Reading Routines* 113⟩+≡ (95) ◁114 118▷

```
int Read_Distribution( TAG_FILE                 *file,
                       FACET_DISTRIBUTION       *dist )
{
    char        line[ MAX_STRING ];
    char        *tag_str = NULL;
    char        *value_str = NULL;
    TAG_ID      tag_id;

    /* read and parse the rest of the tags */
    do {

        /* read the next line from the file */
        if( Read_File_Line( file, line ) == RT_EOF ) {
            printf( "\nRead_Distribution: FORMAT ERROR\n" );
            printf( "\tUnexpected EOF reached at line #%d\n",
                file->line_number );
            return( RT_FAILURE );
        }

        /* extract the TAG from the string */
        tag_str = strtok( line, tag_delims );

        /* parse the string out of the file */
        if(( tag_id = Parse_Tag_String( tag_str, _Input_TAGS )) == NO_MATCH ) {
            printf( "\nRead_Distribution: FORMAT ERROR\n" );
            printf( "\tUnknown tag found in file!\n" );
            printf( "\tTAG = %s\n", tag_str );
            printf( "\tFilename = %s\n", file->filename );
            printf( "\tLine number =  %d\n", file->line_number );
            return( RT_FAILURE );
        }

        /* read the value */
        value_str = strtok( NULL, tag_delims );
        Read_Distribution_Value( value_str, tag_id, dist );

    } while( tag_id != DIST_ENTRY_END_TAG );

    /* return success of material entry */
    return( RT_SUCCESS );
```

```
}
```

Defines:
Read_Distribution, used in chunk 114.
Uses FACET_DISTRIBUTION 97b and _Input_TAGS 113.

118 ⟨*Input Reading Routines* 113⟩+≡ (95) ◁116

```
int Read_Distribution_Value( char              *value_str,
                             TAG_ID            tag_id,
                             FACET_DISTRIBUTION *dist )
{

    /* read the value for the given TAG */
    switch( tag_id ) {

        case DIST_ENTRY_BEGIN_TAG:
            break;

        case TYPE_TAG:
            if( strncmp( value_str, "GAUSSIAN", 8 ) == 0 )
                dist->dist_type = GAUSSIAN;
            else if( strncmp( value_str, "UNIFORM", 7 ) == 0 )
                dist->dist_type = UNIFORM;
            else
                return( RT_FAILURE );
            break;

        case COUNT_TAG:
            sscanf( value_str, "%d", &dist->count );
            break;

        case AVERAGE_SIZE_TAG:
            sscanf( value_str, "%f", &dist->average_size );
            break;

        case DELTA_SIZE_TAG:
            sscanf( value_str, "%f", &dist->delta_size );
            break;

        case AVERAGE_X_TAG:
            sscanf( value_str, "%f", &dist->average_x );
            break;

        case DELTA_X_TAG:
            sscanf( value_str, "%f", &dist->delta_x );
            break;
```

```
        case AVERAGE_Y_TAG:
            sscanf( value_str, "%f", &dist->average_y );
            break;

        case DELTA_Y_TAG:
            sscanf( value_str, "%f", &dist->delta_y );
            break;

        case AVERAGE_Z_TAG:
            sscanf( value_str, "%f", &dist->average_z );
            break;

        case DELTA_Z_TAG:
            sscanf( value_str, "%f", &dist->delta_z );
            break;

        case DIST_ENTRY_END_TAG:
            break;

        default:
            return( RT_FAILURE );
            break;

    }

    /* we should not make it here */
    return( RT_SUCCESS );
}
```

Uses FACET_DISTRIBUTION 97b.

Appendix D

User's Guide

D.1 Introduction

A new ray tracing library (named *libRT*) was implemented using the non-uniform spatial subdivision technique refined by Andrew Glassner (Glassner[1982]). This algorithm minimizes facet intersection tests *(i)* by automatically generating a density dependent spatial hierarchy of facet regions (called an *octree*) and *(ii)* by only testing facets in regions along the path of the ray.

The library can be easily incorporated into any ray tracing renderer. The library provides three general facilities: *(i)* a simplified interface which allows the user to add objects to a managed storage area, *(ii)* a way to create an octree for the loaded objects using a spatial subdivision algorithm and *(iii)* a way to determine objects intersecting a ray using the octree. The library does *not* include a camera model or color rendering features, so support for generation of rays to be traced and rendering of the intersected surfaces must be provided by the user.

Only four external functions are required by the user:
`Initialize_World()`
`Add_Facet_To_World()`
`Create_Octree()`
`Trace_Ray()`
All library related data types and function prototypes are available in the include file `RT.h`. The library achieve is called `libRT.a`.

D.2 Basic Data Types

At the core of the library are the basic geometric data types for points and vectors. Both `RT_POINT` and `RT_VECTOR` are implemented as fixed-length arrays to facilitate more streamlined algorithms. The type `RT_RAY` is a structure consisting of an `RT_POINT` to store the ray origin and a normalized `RT_VECTOR` to define the

direction of the ray. The number of dimensions (length of the array) is defined by NUM_DIMS in RT.h. Macros are provided to easily copy points and vectors.

Although the *libRT* library was written in ANSI C (for portability and speed), the library was specifically designed to reflect the trend towards object oriented interfaces. At the heart of this design, is the *world* data structure (type name RT_WORLD). The world data structure stores all entities relevant to a specific scene including all the facets, the control parameters for the octree and the octree itself. The self contained design of the RT_WORLD structure is deliberate so that a program can simultaneously create, manage and ray trace several "worlds" on an individual basis.

Scenes consist of facets (data type RT_FACET) which include geometry data (data type RT_GEOMETRY) and rendering attributes (data type RT_ATTRIBUTES). The geometry data includes the 3D coordinates for each vertex, the normal to the facet, the offset of the plane equation, and a 3x3 orientation matrix for redirecting exiting vectors. The attribute data structure is meant to store user specific rendering data. For use with DIRSIG, the attribute data structure was modified to store the spectral reflectance and surface specularity information used by the radiometry model to render the image.

The base and any node in the octree is called an octant (data type RT_OCTANT). Data in the octant structure includes the space it occupies, the size of the octant, and a pointer to a list of sub-octants or facets (depending on it's location in the octree).

The ray tracer routine introduces two more types: RT_HIT and RT_HIT_LIST. The RT_HIT type is capable of storing a pointer to the intersected facet, a pointer to the octant containing the facet, the hit point, the distance from the origin of the ray to the hit point and the angle from the facet normal to the incident ray. A RT_HIT_LIST consists of a static array of hits, and a counter to maintain the number of hits currently stored in the list.

D.3 Object Storage

The library was constructed to allow the user to manage several different "worlds" simultaneously. Each world must be initialized through the use of the Initialize_World() routine. This function initializes the RT_WORLD structure to a known state for the facet loading and octree creation routines.

A facet definition is added to a given world data structure using the Add_Facet_To_World() routine. Internally, the library uses a block type allocation scheme to store the facets loaded into the storage area. The maximum number of facet blocks that can be allocated and the number of facets in each block is defined at compile time by the constants MAX_FACET_BLOCKS and FACET_BLOCK_SIZE in the include file RT_World.h. The current settings for these values allow for 512K facets to be stored.

D.4 Octree Creation

Once the world has been loaded with all the desired facets, the octree must be created. This is achieved through a call to the Create_Octree() routine. Since this library supports an additional octant style not

described by Glassner, the calling function must specify which world octant convention to use. The octant convention defines the shape (either cube or rectangle) of the world octant, and hence the shape of every sub-octant created within it. The enumerated data type `RT_OCTANT_STYLE` has three possible values:

`CUBE_STYLE`
The "cube-like" octant convention described by Glassner in his original paper using octants that are the *same* length in all dimensions. In this case, the size of world octant is set to the largest dimension from the minimum and maximum extents of the loaded scene.

`RECTANGLE_STYLE`
A non-cubic octant convention featuring octants which may have *different* lengths in each dimension. In this case, the world octant is set to the the minimum and maximum extents of the loaded scene. For scenes that are extremely large in one dimension, this convention may produce fewer empty octants and improve performance.

`IDEAL_STYLE`
Another non-cubic octant convention featuring octants which may have *different* lengths in each dimension. The world octant is set to the the minimum and maximum extents of the loaded scene. All child octants are created by splitting the parent octant through the mean location of all the facets contained in the octant.

D.5 Ray Tracing

Ray tracing a ray is accomplished by the calling the `Trace_Ray()` routine. This function is passed the ray to be traced, the world to trace within and a list to store the facet intersection information. In addition, a variable of enumerated data type `RT_TRACE_MODE` is passed to indicate which of the two run modes to use. When rendering an image, the option flag must be set to `CLOSEST_HIT_FLAG` so that the ray tracer returns after the facet closest to the ray origin has been determined. When this flag is set to `FIRST_HIT_FLAG`, the ray tracer will return immediately upon intersecting a facet. The later mode is useful in shadowing or obstruction cases where continuing to search for the closest facet obstructing the path does not provide any additional benefit.

D.6 Prototypes for User Accessible Functions

```
/*
 * NAME:         Initialize_World
 * PURPOSE:      Set all the variables in the world structure to a known state.
 * INPUTS:       This routine has the following inputs:
 *                   1. The world (type = RT_WORLD)
 * OUTPUTS:      Changes are to the variable passed in.
 * RETURN:       The functions returns a RT_SUCCESS if the facet is
 *               successfully added to the child list and RT_FAILURE if not.
 * SUBROUTINES: This function does not call any subroutines.
 * AUTHOR:       S.D. Brown
 */
int Initialize_World( RT_WORLD  *world );

/*
 * NAME:         Add_Facet_To_World
 * PURPOSE:      Adds a facet to the world facet list.  If the currently
 *               allocated list is full, an additional block is allocated.
 *               Facets are added to the end of the list.
 * INPUTS:       This routine has the following inputs:
 *                   1. The facet to add (type = RT_FACET)
 *                   2. The world (type = RT_WORLD)
 * OUTPUTS:      All changes are to internal data structures.
 * RETURN:       The functions returns a RT_SUCCESS if the facet is
 *               successfully added to the child list and RT_FAILURE if not.
 * SUBROUTINES: This function calls the following subroutines:
 * AUTHOR:       S.D. Brown
 */
int Add_Facet_To_World( RT_FACET        *facet,
                        RT_WORLD        *world );
```

```
/*
 * NAME:         Create_Octree
 * PURPOSE:      Creates an octree from the facets loaded in the facet list
 *               stored in the RT_WORLD variable.
 * INPUTS:       This routine has the following inputs:
 *                   1. The style of octants to use (type = RT_OCTANT_STYLE)
 *                   2. The world to build the octree for (type = RT_WORLD)
 * OUTPUTS:      The base of the octree created by this routine is stored in
 *               the world variable passed in.
 * RETURN:       The functions returns RT_SUCCESS under normal operation,
 *               and RT_FAILURE when errors arise.  All errors are reported
 *               in detail by the subroutines that encounter them.
 * SUBROUTINES: This function makes calls to the following routines:
 *               RT_Sort_Facet()
 * AUTHOR:       S.D. Brown
 */
int Create_Octree( RT_OCTANT_STYLE      style
                   RT_WORLD             *world );

/*
 * NAME:         Trace_Ray
 * PURPOSE:      Trace a ray in the scene passed in through the world
 *               variable.  To seed the process, the ray is intersected with
 *               the world octant and the origin perturbed
 * INPUTS:       This routine has the following inputs:
 *                   1. The ray to trace (type = RT_RAY)
 *                   2. The world to trace (type = RT_WORLD)
 *                   3. The tracing mode (type = RT_TRACE_MODE)
 * OUTPUTS:      The function fills a list of hits with facets intersected
 *               until a terminating condition is met.
 *                   1. The list of hits (type = RT_HIT_LIST)
 * SUBROUTINES: This function makes calls to the following routines:
 *               RT_Ray_Box_Intersect()
 *               RT_Trace_Octant()
 * AUTHOR:       S.D. Brown
 */
int Trace_Ray( RT_RAY        *ray,
               RT_OCTANT     *octant,
               RT_WORLD      *world,
               RT_TRACE_MODE *mode,
               RT_HIT_LIST   *hit_list );
```

D.7 Basic Code Example

How to use these facilities is presented as a tutorial example below. The example covers all required initialization steps, adding facets to the managed storage area, building an octree for this world and ray tracing a ray. The means by which facets are created or read in from data files is not included in this discussion since it is handled by the user. The rays cast into the scene must also be created by user supplied functions. All routines return either RT_SUCCESS or RT_FAILURE. These return values are ignored here for the sake of code clarity.

```
int trace_example( void )
{
    RT_WORLD            world;          /* the world we will be ray tracing */
    RT_FACET            facet;          /* a facet to add to the world */
    RT_OCTANT           *start_octant;  /* octant to start ray tracing from */
    RT_OCTANT_STYLE     style;          /* the style of octants to use */
    RT_TRACE_MODE       mode;           /* the ray tracing mode */
    RT_HIT              closest_hit;    /* the first hit in the hit list */
    RT_HIT_LIST         hit_list;       /* to store facet intersections */

    /* you must initialize any RT_WORLD variables */
    Initialize_World( &world );

    /* the user defines or reads in a facet */
    facet = user_function();

    /* add this facet to the managed storage area in this "world" */
    Add_Facet_To_World( &facet, &world );

    /* define what type of octants to create the octree with */
    style = CUBE_STYLE;

    /* create the octree for the facets currently loaded into the "world" */
    Create_Octree( style, &world );

    /* the user creates the ray to be traced */
    ray = user_function();

    /* set the mode to be used when ray tracing */
    mode = RT_CLOSEST_HIT;

    /*
     * Trace this ray into "world" and place intersections in the hit
```

```
     * list.  The second argument is a pointer to the starting octant if it
     * is known.  Since it is not, a NULL pointer is passed.
     */
    Trace_Ray( &ray, NULL, &world, &mode, &hit_list );

    /* the hit list is sorted by distance from the ray origin */
    closest_hit = hit_list.hits[ 0 ];

    /* create a second ray to trace from the last facet intersection point */
    RT_POINT_COPY( ray.origin, closest_hit.point );
    ray.direction = user_function();

    /* the starting octant can be grabbed from the last hit also */
    start_octant = closest_hit.octant;

    /* trace this second ray into "world" */
    Trace_Ray( &ray, start_octant, &world, &mode, &hit_list );
}
```

Bibliography

[Badouel 1990] D. Badouel, "An efficient ray-polygon intersection", *Graphics Gems*, Glassner (editor), New York: Academic Press

[Brown 1997] S.D. Brown and N. Schaller, "Implementation of a Non-Uniform Spatial Subdivision Ray Tracer", Project Document for ICSG 898, May 1997

[Clark 1976] J.H Clark, "Hierarchical geometric models for visible surface algorithms" *Communications of the ACM*, Vol. 13 No. 2, pp.547-54

[Fujimoto 1986] A. Fujimoto, T. Tanaka, K. Iwata, "ARTS: Accelerated Ray-Tracing System", *IEEE Computer Graphics and Applications* April 1986, pp. 16-26

[Gargantini 1982] I. Gargantini, "Linear octrees for fast processing of three-dimensional objects", *Computer Graphics and Image Processing*, Vol. 20, No. 4, 1982, pp. 265-274

[Glassner 1984] A.S. Glassner, "Space subdivision for fast ray tracing", *IEEE Computer Graphics and Applications*, October 1984, pp. 15-22

[Haines 1991] E. Haines, "Essential ray tracing algorithms", *An introduction to ray tracing*, Glassner (editor) New York: Academic Press

[Jenkins 1982] C.L. Jenkins and S.L. Tanimato, "Octtrees and their use in representing three-dimensional objects", *Computer Graphics and Image Processing*, Vol. 14, No. 3, 1982, pp. 249-270

[Kay 1986] D.S. Kay and J.T. Kajiya, "Ray tracing complex scenes", *Computer Graphics*, Vol. 17, No. 3, 1982, pp. 377-388

[Kendall 1962] M.G. Kendall, Rank Correlation Methods, Hafner Publishing Company, New York, 1962

[Meagher 1982] D. Meagher, "Geometric modeling using octtree encoding", *Computer Graphics and Image Processing*, Vol. 19, No. 2, 1982, pp. 129-147

[Rubin and Whitted 1980] S. Rubin and T. Whitted, "A three-dimensional representation for faster rendering of complex scenes", *Computer Graphics*, Vol. 14, No. 8, 1980, pp. 110-116

[Salacain 1994] J. Salacain, "Simulation of Complex Focal Planes for DIRSIG", MS Thesis, Rochester Institute of Technology, December 1994.

[Schott et al] J.R. Schott, S.D. Brown and R.V. Raqueño

[Weghorst *et al.* 1984] H. Weghorst, G. Hooper and D. Greenberg, "Improved computational methods for ray tracing", *ACM Trans. Graph.*, Vol. 3, No. 1, 1984, pp. 223-237