A Testbed for Image Synthesis

Ben Trumbore, Wayne Lytle†, Donald P. Greenberg

Program of Computer Graphics, Cornell University, Ithaca, NY 14853
†currently at Cornell National Supercomputer Facility

Abstract

Image Synthesis research combines new ideas with existing techniques. A collection of software modules that provide such techniques is extremely useful for simplifying the development process. We describe the design and implementation of a new Testbed for Image Synthesis that provides such support. This Testbed differs from previous Testbeds in both its goals and its design decisions.

The Testbed design addresses the problems of high model complexity, complicated global illumination algorithms and coarse grain parallel processing environments. The implementation is modular, portable and extensible. It allows for statistical comparison of algorithms and measurement of incremental image improvements, as well as quantitative comparison of Testbed images and light reflectance measured from physical models.

The Testbed is designed to interface with any available modeling system. This compatibility was achieved through careful design of the data format that represents environments. The software modules of the Testbed are organized in a hierarchical fashion, simplifying application programming.

1 Purpose

The goal of *Realistic Image Synthesis* is to generate images that are virtually indistinguishable from photographs. Creating such images requires an accurate simulation of the physical propagation of light. This can be an enormous computational task, requiring sophisticated light reflection models and accurate surface descriptions. Today's algorithms, which are embedded in commercial workstation hardware, cannot be extended to handle these complex situations.

Future image synthesis techniques must incorporate global illumination algorithms, bidirectional reflectance functions, wavelength dependent surface properties and procedural textures. Thus, they will require new computer graphics system architectures. These algorithms must accurately simulate physical reality, and must also be fast enough to keep computation times reasonably low. Future hardware will also rely on parallel processing and pipelined architectures to provide the throughput necessary for interactive speeds.

For these reasons it was important to devise a *Testbed* to facilitate research in future image synthesis techniques. The Testbed which has been developed at Cornell University's Program of Computer Graphics [LYTL89] has been structured to perform the following:

1. Test new light reflection models and new global illumination algorithms so that experimental approaches can be combined in a modular fashion.
2. Render scenes and simulations of far greater complexity than what is currently being rendered.
3. Provide an environment for exploiting coarse grain parallelism for high level global illumination algorithms.
4. Provide a mechanism for comparing computer simulations with actual measured physical results from laboratory testing.

5. Reduce program development time by having appropriate modularity and interface descriptions, allowing experimental methods to be easily tested.
6. Measure algorithm performance by statistically monitoring subroutine calls and run times. These results are correlated with environment data to obtain predictive methods for computational costs.
7. Provide a mechanism for easy maintenance of large collections of graphics software.

Additionally, these goals were to be accomplished within a university research environment where personnel is constantly changing due to graduation.

Such a system is difficult to implement because it must also comply with certain existing constraints. Old modeling data must be usable with new image synthesis algorithms. The software must work on different manufacturers' products and be amenable to the rapid changes of the computer industry and graphics display technology. Lastly, the data that is used for rendering must be independent of the modeling process, and cannot be restricted to a display list structure.

Realistic images can represent existent or nonexistent environments. For photorealism it is important to simulate the behavior of light propagating throughout an environment. One must model the light arriving at each surface directly from the light sources as well as the light that arrives indirectly. Indirect light is reflected from other surfaces and potentially transmitted through the given surface. The light leaving the surface in a certain direction is readily determinable from these light sources and the known physical properties of the surface.

This short paper does not allow a comprehensive review of the various rendering approaches that simulate global illumination effects. The reader is referred to several summaries of such algorithms [GLAS89, GREE86].

Today's common rendering algorithms can be classified as belonging to one of two broad families: *Ray Tracing* and *Radiosity*. In Ray Tracing the image plane is discretized and sample points are taken at each pixel, yielding a view-dependent solution. In the Radiosity Method the environment is discretized and a view independent solution is obtained. Both methods use simplifying assumptions in an attempt to simulate the propagation of light and solve the general Rendering Equation [KAJI86].

In ray tracing, a ray is traced from the eye through each pixel into the environment [WHIT80]. At each surface struck by the ray, reflected and/or refracted rays can be spawned. Each of these must be recursively traced to establish which surfaces they intersect. As the ray is traced through the environment, an intersection tree is constructed for each pixel. The branches represent the propagation of the ray through the environment, and the nodes represent the surface intersections. The final pixel intensity is determined by traversing the tree and computing the intensity contribution of each node according to the assumed surface reflection model. Numerous methods including adaptive ray tracing [HALL83], distributed ray tracing [COOK84b], cone tracing [AMAN84], and environment subdivision methods [GLAS84, KAPL85, HAIN86, GOLD87] have subsequently reduced computation times and improved image quality.

The radiosity approach [GORA84], based on methods from thermal engineering [SPAR78], determines surface intensities independent of the observer position. The radiosity of the light energy leaving a surface consists of both self-emitted and reflected incident light. Since the amount of light arriving at a surface comes from all other surfaces and lights within the environment, a complete specification of the geometric relationships between all reflecting surfaces must be determined. To accomplish this, the environment is subdivided into a set of small, discrete surfaces. The final radiosities, representing the complete interreflections between these surfaces, can be found by solving a set of simultaneous equations. Although the approach was originally restricted to simple diffuse environments, it has subsequently been extended to complex environments [COHE85], specular surfaces [IMME86, WALL87], and to scenes with participating media [RUSH87]. Furthermore, computational times have recently been vastly reduced using progressive refinement methods [COHE88]. Testbed implementations of ray tracing and radiosity renderers, as well as a hybrid algorithm that combines both approaches, are illustrated in Section 6.

2 Other Testbeds

Several testbed systems, mostly facilitating rendering research, have been developed in the past. These systems can be classified in one of three ways:

- Monolithic systems with many run time options.
- Assemblies of simple nodes linked through UNIX pipes or similar mechanisms.
- Collections of isolated software libraries called by higher level user-written programs.

The new Testbed for Image Synthesis, described in this paper, fits the latter classification.

One of the first testbeds appearing in the literature was developed at Bell Laboratories in 1982 and was used for scanline rendering [WHIT82]. This testbed facilitated the construction of renderers that allowed the simultaneous processing of several object types in a single pass. Cornell's original Testbed for Image Synthesis [HALL83] was designed to aid research pertaining to the Ray Tracing rendering process, including lighting models, rendering parametric surfaces, reflections, light propagation and texture mapping. This object-oriented system was modular and allowed easy addition and testing of experimental object types.

Potmesil and Hoffert of Bell Laboratories developed the testbed system FRAMES [POTM87]. This system used UNIX filters to construct image rendering pipelines. New techniques could easily be added, and the system lent itself to experimentation with distributed rendering. In the same year Nadas and Fournier presented the GRAPE testbed system [NADA87], also based on a similar notion of "loosely coupled" nodes. GRAPE used data-flow methods to provide a more flexible architecture that was not limited to a linear flow. However, since node assemblies may not contain cycles, this system is still not flexible enought to facilitate global lighting affects.

The REYES image rendering system was developed at Lucasfilm and is currently in use at PIXAR [COOK87]. Designed more as a production system than a testbed, this system efficiently renders environments of very high model and shading complexity. REYES is an example of a "monolithic" system, which is geared toward one specific rendering technique. Brown University's BAGS system [STRA88, ZELE91] provides several rendering techniques, including scanline and ray tracing renderers. Because it consists of many software modules, BAGS appears to be a flexible system for developing new renderers. However, its modules are tightly coupled and interdependent. It might be viewed as a different sort of monolithic system that provides a greater breadth of functionality.

3 The Modeler Independent Description (MID)

It is extremely beneficial for all of a system's renderers to produce images from a single environment description. To resolve this need for unlimited modeling data formats and a single rendering data format, a Modeler Independent Description (MID) data format was defined. No display structure hierarchy is used by MID. However, MID does allow trees of primitives to be formed using the boolean set operations of union, intersection, and difference.

MID is a text format that describes a list of primitive objects and their attributes. It serves as the interface between modeling programs and rendering programs. Figure 1 depicts the high level structure of the Testbed. Modeling programs, on the left, read and write their own private data formats. These data formats can all be converted into MID, but MID files cannot be converted back to the modeler data formats. Several modelers may share the same private data format. Modelers may use the display structure of their choice to produce the interactive graphical communication used during modeling. Generally, the conversion of modeling data to MID is only used when a high quality rendering is to be produced. The design of this interface allows for independent implementation of modeling and rendering software. Because the MID format is simple yet extensible, it allows old models to be used by new renderers, and new models to be used by old renderers.

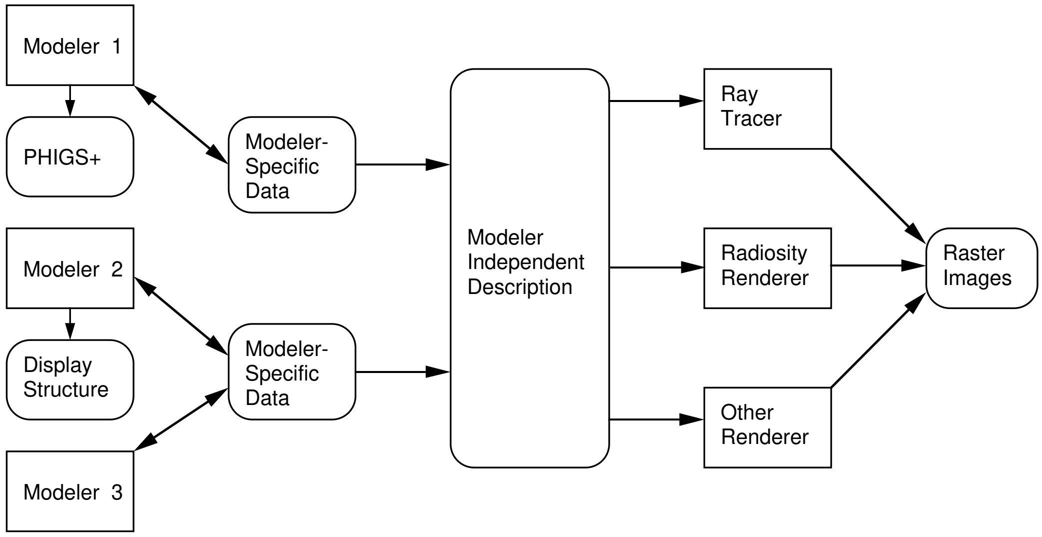

Figure 1: Testbed Structure

A single software module is used by all renderers to read MID data and create standard data structures. Renderers may use the information in these standard structures to construct their own local data structures. The local structures used by a Ray Tracer, a Radiosity renderer, and a Monte Carlo renderer are likely to be substantially different. Each of these renderers produces raster image files in a standard format (Figure 1). The rendering programs themselves are constructed using functionality from a variety of Testbed modules.

Because the Testbed must support environments of unlimited size, routines that interpret the MID format must be able to read objects sequentially. If an environment is larger than a computer's virtual memory, it cannot be read all at once. When object templates can be defined and then instanced repeatedly, a template definition could be needed at any time. This would require that the entire environment be retained in memory. For this reason, MID does not allow instancing of transformed objects. However, because some object geometries are defined by a large amount of data, several objects may reference the same geometric data (which is stored separately from MID).

Each object in a MID environment is defined as a primitive type and a limitless list of attributes. These attributes are specified by name-value pairs, in which an attribute is named and its value is specified. An attribute value can be a scalar number, a text string, or a name of a data block stored separately from MID. Three attributes are predefined by the Testbed: transformations, geometric data, and rendering information. Transformation attributes define the way a primitive object is transformed from object space to world space. Geometric data attributes complete the physical description of those primitive objects that require additional parameters. For example, the definition of a torus depends on the lengths of its major and minor radii, and a polygonal object requires a list of vertices and polygons to define its shape. Rendering information includes material, surface, and emission properties. The Testbed currently provides the following primitive types:

`sphere`	`cube`	`cylinder`	`cone`
`pyramid`	`prism`	`torus`	`polygons`
`conic`	`square arch`	`round arch`	`camera`

New image synthesis algorithms using the Testbed may define other attributes as they are needed. A renderer can be written to look for certain attributes, interpreting their values as it sees fit. Renderers that do not use a given attribute will simply ignore it. This open-ended attribute system provides the basic attributes that remain consistent between renderers, but does not hinder future research.

4 The Testbed Module Hierarchy

The Testbed's modules are organized into three conceptual *levels* of functionality (Figure 2). Modules at a given level may use other modules at the same level, or any level below it. Image synthesis applications built upon the Testbed may use modules from any level.

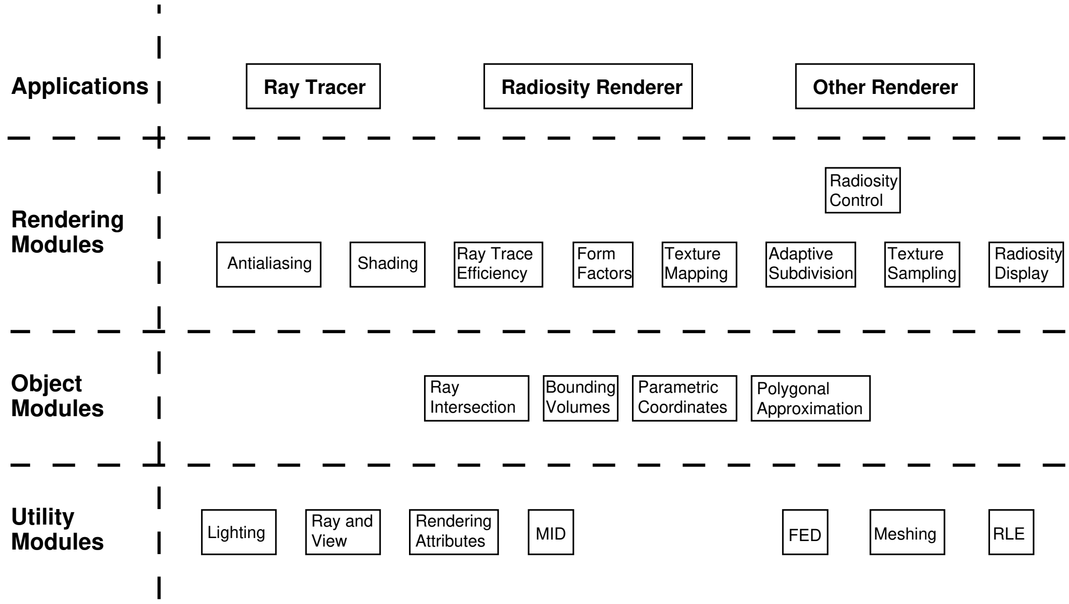

Figure 2: Testbed Module Hierarchy

The lowest level of the Testbed contains Utility modules, which provide the most basic of functionalities. Some Utility modules create and manipulate data structures, such as those for environments, attributes, polygons, and raster images. Other Utility modules perform low-level mathematical functions, such as transformation matrix operations and the solution of sets of simultaneous equations. Many of the Utility modules are useful for both rendering and modeling applications.

The middle level of the Testbed contains Object modules, which perform certain functions for all types of primitives in the Testbed. When a new primitive type is added to the Testbed, functionality for this type must be added to each Object level module. If a new Object level module is added to the Testbed, functionality must be included in that module for each primitive type in the Testbed. Object level modules include intersecting rays with primitives, creating bounding volumes for primitives, and approximating primitives with a collection of polygons. These modules allow applications and other libraries to treat all primitives the same, regardless of their type.

The highest Testbed level contains Image Synthesis modules. These modules provide simple interfaces for complex rendering techniques. They perform such rendering tasks as shading, texturing, radiosity solutions, and the use of hierarchical structures to make ray tracing more efficient. Researchers can easily implement rendering approaches by using several Image Synthesis modules. Individual modules can then be replaced with specific software the researcher has written. This minimizes development overhead and allows the researcher to concentrate on new functionality. While such a modular organization can lead to a small reduction in efficiency, the primary emphasis is to provide an environment for algorithmic experimentation.

5 Individual Testbed Library Modules

The Cornell Testbed currently includes about forty separate software modules. For simplicity, we present short descriptions of most modules. In many cases, several modules perform similar functions using different algorithms. Descriptions are brief, but they should give a sense of the role that each module plays within the Testbed. The libraries are grouped according to their Testbed level.

5.1 Utility Level Modules

Modeler Independent Description: This module reads the previously described MID data files and constructs data structures representing those files. It also reads geometric data and rendering attributes for those objects that specify them.

Polygonal Data Structures: The Testbed uses several geometric data formats, relying mainly on the Face-Edge Data structure (FED) [WEIL88]. Several libraries implement this data format, reading and writing between data structures and files. FED stores considerable information about an object's topology. This information is useful for implementing meshing and radiosity algorithms.

Rendering Attributes: Each object requires material and surface properties to be rendered properly. This module stores those attributes and reads them from files into data structures. This information can range from simple Red-Green-Blue color specifications to spectral database references that contain a material's reflectance at dozens of wavelengths.

Image Format: This module handles input/output of raster images. The Testbed uses the Utah Raster Toolkit's Run Length Encoded (RLE) image format for this purpose [PETE86]. These image files are used as texturing data and as the output of some high level rendering programs.

Ray and View Data Structures: Data structures supported by this module represent Ray Tracing rays and viewing specifications. The viewing specifications are derived from the MID environment, and the Ray data structure is useful throughout the Testbed.

Hash Tables: This popular data structure is implemented here to provide efficient lookup of data of any format.

Priority Queue: Another familiar data structure, the priority queue sorts its input as it is received and dispenses the sorted data upon request.

Item Buffer: Implemented in both software and hardware, this algorithm allows polygons to be scan converted to identify the portions of some *screen space* that they cover. Useful for Radiosity's *hemicube* rasterization, it can also be used to speed up the casting of view-level Ray Tracing rays.

Matrix and Vector Operations: This module defines vector and matrix data types and provides a wide variety of efficient operations for these data types.

Color Space Conversions: Rendering is often performed in color spaces that are incompatible with the display capabilities of graphics hardware. This software converts values between various color spaces, as well as between varying numbers of wavelength samples.

Root Finding Methods: Many ray/object intersection routines require solution of univariate and bivariate polynomials. These implementations provide such functionality for ray intersections and other applications.

Polygonal Meshing: Some algorithms are designed to work only with convex polygons, or perhaps only on triangles. The Testbed meshing modules operate on the FED data structure, reducing nonconvex polygons, polygons with holes and polygons that are just too large.

5.2 Object Level Modules

The Object level modules are organized by functionality, rather than by primitive type. Since many types of primitives may be used in a given model, if the organization were by primitive type a renderer would have to include all Object modules. Since renderers often require only a specific functionality for all primitive types, this functional organization is more efficient. Also, Testbed growth was expected to involve additional functionality, rather than additional primitive types. New functional modules can easily be developed for all primitive types at once without disturbing existing modules.

Bounding Volumes: This module generates bounding volumes for objects. Spheres and axis-aligned cuboids are provided. These are useful for ray tracing algorithms, or for approximating the volume an object displaces.

Ray/Object Intersection: There are many algorithms that require the intersection of a ray with a primitive object. These routines find such intersection points, the distance along the ray to the point, and the object's surface normal vector at the point [KIRK88].

Polygonal Approximation: many rendering algorithms operate only on polygonal data. This module generates polygonal approximations of varying resolutions for each primitive primitive type. This software is particularly useful for radiosity algorithms, which begin by approximating an environment with a collection of polygonal patches.

Parametric Coordinates: Each object type has a simple parameterization for its surfaces. One or more parametric *faces* are defined for the primitive type, and a (u, v) parameter space is defined for each face. This module converts (x, y, z) points in object space into face IDs and parametric coordinates between 0 and 1, and is useful for texturing and global illumination algorithms.

5.3 Rendering Level Modules

Hierarchical Bounding Volumes: To make ray casting operations efficient on large environments, several algorithms have been implemented as Testbed modules. This one uses a hierarchical tree of bounding volumes [GOLD87] to more selectively intersect only those objects near the ray.

Uniform Space Subdivision: In the same vein, this module subdivides the environment space into a regular grid. Intersection checks are performed only for objects that are in grid cells through which the ray passes [FUJI86]. Both of these efficiency schemes have identical interface specifications, allowing one to easily be substituted for another.

Antialiasing: In order to avoid image space aliasing, Ray Tracing renderers distribute view-level rays throughout each pixel that is rendered. This module coordinates such ray distributions, filters the shaded values of all rays, adaptively determines when the pixel has reached a converged value, and (when appropriate) reuses information calculated for adjacent pixels.

Radiosity: The highest level radiosity algorithm is implemented in this module. It iteratively determines which *patch* should distribute its energy throughout the environment [COHE88]. Instructions to other modules cause the environment to be meshed, textured, and eventually displayed.

Parallel Radiosity: Some rendering algorithms take on a completely different form when they are run in parallel on several computers. This module was needed to allow implementation of a parallel version of the radiosity procedure.

Adaptive Subdivision: As radiosity solutions progress, some portions of an environment will require finer meshing so the eventual solution will be more accurate [COHE86]. This software identifies those areas and instructs meshing libraries to perform the needed subdivision.

Radiosity Attributes: During the radiosity solution, each object must store temporary information at points on its surface. This module organizes such information in data structures that are used by all the radiosity modules.

Form Factor Calculation: At the heart of the radiosity technique is the ability to calculate how much energy passes from one patch in an environment to another. Calculation of these *form factors* is performed in these modules. Some algorithms use the *hemicube* and scan conversion algorithms [COHE85], while others perform the calculation using ray casts [WALL89].

Radiosity Display: Often it is desirable to display a completed radiosity solution on some common graphics hardware. This module performs this task on several different brands of graphics displays.

Light Emission Distributions: This module allows specification of non-uniform light emission distributions. Photorealism requires accurate modeling and rendering of all aspects of light transport. Not the least important of these is the initial distribution.

Shading Models: The Phong and Blinn models are among those implemented in this module. Ray Tracing applications will often use this software to generate shaded intensities once an intersected object has been found.

Texture Mapping: Texturing specifications are interpreted by this software. As rendering proceeds, this information is used to determine which texture space points correlate to the object space points that are being rendered.

Texture Sampling: A point or area in texture space is sampled by this module. Texture filtering and efficient storage schemes are also implemented by these routines.

6 Rendering Examples

The following sections provide examples of applications that use Testbed modules. The three examples included are a ray tracing program, a radiosity program and a hybrid program that combines ray tracing and radiosity. Each example begins with a description of the algorithm (as implemented in the Testbed), followed by pseudocode that outlines the actual code the user would need to write. The figures that accompany the descriptions show which modules are called by the application program and how those modules then call other Testbed modules.

6.1 Ray Tracing

6.1.1 Ray Tracing Description

This ray tracing application first calls the Modeler Independent Description (MID) module to read the environment data. MID uses the Rendering Attribute module to read the surface properties that have been assigned to the objects in the environment. MID also uses the Face-Edge Data Structure modules (FED) to read geometry information for those non-primitive objects that require additional data. This sample program makes heavy use of data structures and access macros associated with the Ray/View and Matrix Transformation modules.

The data structure that is returned by MID is then passed to the Lighting, Ray and View, and Ray Tracing Efficiency modules. These modules identify the camera and light sources in the MID structure, creating and returning their own data structures. The Ray Tracing Efficiency module uses the Bounding Volume and Ray Intersection modules to create data structures for efficient intersections of rays with the entire environment. The Run-length Encoding module (RLE) is used to initialize the output image file.

After initialization, a recursive ray tracing procedure is performed for each pixel of the image. At each pixel the Antialiasing module determines the rays to be traced. The shaded intensity of each ray is reported to the the Antialiasing module, which filters the values and decides when enough rays have been cast. This pixel value is written out by the RLE module. This scheme allows coders to select intersection methods and shading algorithms independent of the oversampling and filtering methods that are used.

The recursive tracing routine first uses the Efficiency module to determine the closest object along the given ray. The object is shaded and any reflected and transmitted rays spawned at the surface are recursively cast. If the surface has been texture mapped (or bump mapped), the Texture Mapping and Texture Sampling modules calculate the altered surface color or normal vector. The object's surface point is converted into texture space, and that texture space point is used to sample the object's texture data. The Lighting module generates rays and energy values used for shading and shadow tests for each light source.

Figure 3 depicts how a ray tracer uses the Testbed modules. Plates 1 and 2 are images produced by the Testbed ray tracer.

6.1.2 Ray Tracing Pseudocode

```
Ray Tracing Mainline:
   MID module reads environment
   Lighting module initializes light data structures
   Ray and View module initializes the camera data structure
   Efficiency module initializes ray intersection data structures
   RLE module initializes image output
   Antialiasing module initializes antialiasing parameters
   for each pixel {
      Antialiasing module adaptively selects primary rays to trace
         recursive tracing routine returns shaded value for each ray
      RLE module outputs completed pixel value
   }

Recursive Tracing Routine:
   Efficiency module finds closest object along ray (return if none)
   Texture Mapping module converts object space point to texture space
   Texture Sampling module samples texture value in texture space
   Lighting module selects light rays to test {
      Efficiency module eliminates light rays that are shadowed
      Shading module shades object using unobstructed light rays
   }
   if surface is reflective
      recursive tracing routine evaluates all reflection rays
   if surface is transparent
      recursive tracing routine evaluates all transmitted rays
```

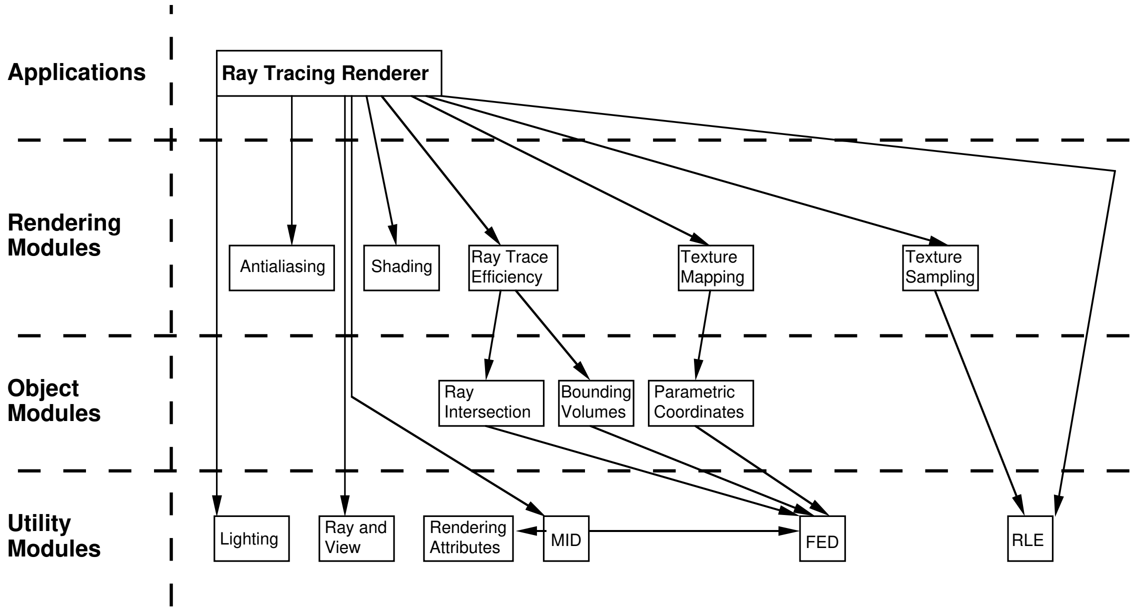

Figure 3: A Testbed Ray Tracer

6.2 Radiosity

6.2.1 Radiosity Description

The radiosity program presented in this example uses the progressive refinement method [COHE88]. In this algorithm, energy is *shot* from patches and dispersed throughout the environment. When a converged solution is achieved, polygons are displayed with their resultant radiosities.

As with the ray tracing example, this program begins by having the MID module read the environment, attributes, and geometry. This environment is passed to the Radiosity module, which creates the data structures used to calculate radiosities. It calls the Adaptive Subdivision Algorithm module, which performs four functions on every object in the environment. These functions are:

- The Polygonal Approximation module creates a FED structure to represent the geometry of an object.
- A Meshing module subdivides the object's faces into quadrilaterals and triangles.
- Another Meshing module continues to mesh the faces to the desired size.
- The Radiosity Attribute module initializes the information about each face. This information includes the initial energy found at a face, and the accumulated radiosity at each face.

These operations are transparent to the application program.

Next, the Form Factor Calculation module initializes the data structures associated with the type of form factors that will be used by the application. The program now repeats an operation that shoots from the patch with the largest remaining energy. The repetition ends when the largest energy is smaller than a given threshold. The Radiosity module performs this shooting operation in the following way:

- The patch with the highest unshot energy is found using the Radiosity Attributes.
- The Form Factor module distributes the patch's energy to the other patches in the environment. The Ray Tracing Efficiency and Ray Intersection modules perform this function.
- The patch radiosities are updated to include the new energy.

The radiosity solution polygons can be displayed using either software or hardware display techniques. This display can be performed periodically during the solution process or after the solution has converged.

Figure 4 depicts how a radiosity renderer uses the Testbed modules. Plate 3 is an image produced by the Testbed radiosity renderer.

6.2.2 Radiosity Pseudocode

```
Radiosity Mainline:
   MID module reads the environment
   Radiosity module initializes data structures
   Form Factor module initializes form factor calculations
   repeat until solution is converged
      Radiosity module shoots patch with highest remaining energy
   display solved polygons
```

6.3 Hybrid

The hybrid rendering program in this example combines ray tracing and radiosity algorithms to produce more realistic images. It first completes a radiosity solution, and then ray traces the environment to generate the display. Whenever the ray tracing shader needs diffuse lighting values, they are taken from the previously computed radiosity solution.

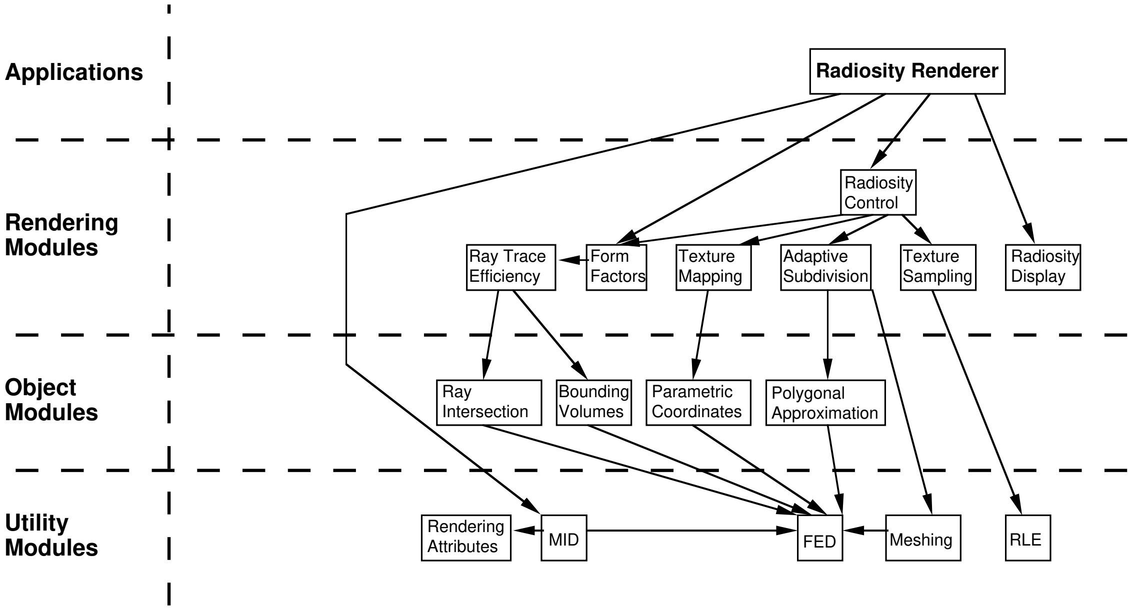

Figure 4: A Testbed Radiosity Render

6.3.1 Hybrid Description

This algorithm is substantially like the union of the radiosity and ray tracing examples. In this case, the MID module need only read the environment once. The radiosity solution then proceeds as in the second example. After radiosities are calculated, the ray tracer renders each pixel much as in the first example. The only difference in the recursive tracing routine is the additional use of the Radiosity module. The program's internal data structures must maintain a correlation between objects in the radiosity data structures and corresponding objects in the ray tracing data structures. In this way, when a ray intersects an object the radiosity energy for that object can be accessed.

Figure 5 depicts how a hybrid ray tracing/radiosity renderer uses the Testbed modules. A renderer such as the one in this example can exercise nearly all of the Testbed's modules.

6.3.2 Hybrid Pseudocode

```
Hybrid Mainline:
   MID module reads the environment
   Radiosity module initializes data structures
   Form Factor module initializes form factor calculations
   repeat until solution is converged
      Radiosity module shoots patch with highest remaining energy
   Efficiency module initializes ray intersection data structures
   Lighting module initializes light data structures
   Ray and View module initializes the camera data structure
   Antialiasing module initializes antialiasing parameters
   RLE module initializes image output
   for each pixel {
      Antialiasing module adaptively selects primary rays to trace
         recursive routine returns shaded value for each ray
      RLE module outputs completed pixel value
   }
```

```
Recursive Hybrid Tracing Routine:
   Efficiency module finds closest object along ray (return if none)
   Texture Mapping module converts object space point to texture space
   Texture Sampling module gets texture value from texture space point
   Lighting module selects light rays to test {
      Efficiency module eliminates light rays that are shadowed
      Shading module calculates specular intensity of unshadowed rays
   }
   Radiosity module provides diffuse lighting at surface point
   if surface is reflective
      recursive tracing routine evaluates all reflection rays
   if surface is transparent
      recursive tracing routine evaluates all transmitted rays
```

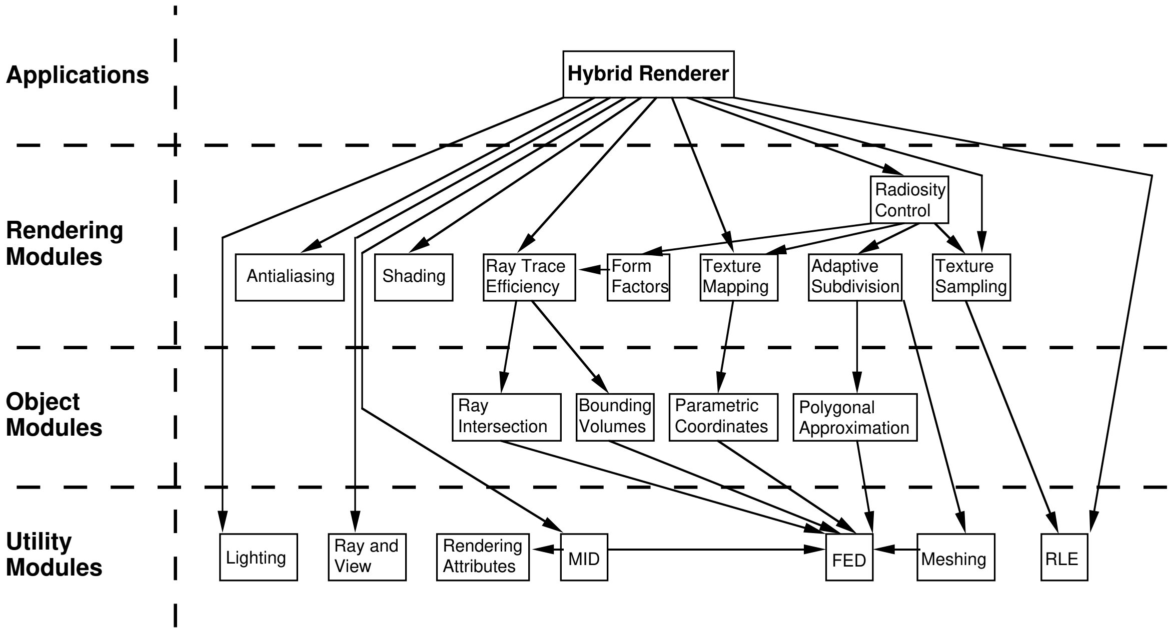

Figure 5: A Hybrid Testbed Renderer

7 Conclusion

The Testbed described in this paper has been under development for about four years. It currently consists of 40 modules and over 100,000 lines of C source code. A dozen research projects rely on this software for rendering support and as a platform for developing new algorithms. As many of these projects are completed they will be incorporated into the Testbed, contributing to its continued growth.

Images, animations, and simulations have been generated using models from two sophisticated modeling programs. One of these modelers is designed to provide complex interior and exterior architectural models [HALL91]. Testbed software is being used for parallel computations on powerful workstations. Clusters of Hewlett Packard 835 workstations and Digital Equipment 5000 and 3100 workstations have been used to develop new parallel rendering algorithms. These computations are yielding statistical information that will be used to analyze algorithm performance and to measure the accuracy of photorealistic rendering.

This research was funded by NSF grants #DCR8203979 and #ASC8715478. The generous support of the Hewlett Packard Corporation and the Digital Equipment Corporation is greatly appreciated. The Cornell Program of Computer Graphics is a member of the National Science Center for Computer Graphics and Visualization.

References

[AMAN84] Amantides, J. "Ray Tracing with Cones," Proceedings of SIGGRAPH'84, in *Computer Graphics*, 18(3), July 1984, pages 129–135.

[COHE85] Cohen, Michael F. and Donald P. Greenberg. "The Hemi-Cube: A Radiosity Solution for Complex Environments," Proceedings of SIGGRAPH'85, in *Computer Graphics*, 19(3), July 1985, pages 31–40.

[COHE86] Cohen, Michael F., Donald P. Greenberg, and David S. Immel. "An Efficient Radiosity Approach for Realistic Image Synthesis," *IEEE Computer Graphics and Applications*, 6(2), March 1986, pages 26–35.

[COHE88] Cohen, Michael F., Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. "A Progressive Refinement Approach to Fast Radiosity Image Generation," Proceedings of SIGGRAPH'88, in *Computer Graphics*, 22(4), August 1988, pages 75–84.

[COOK84a] Cook, Robert L. "Shade Trees," Proceedings of SIGGRAPH'84, in *Computer Graphics*, 18(3), July 1984, pages 223–231.

[COOK84b] Cook, Robert L., Tom Porter, and Loren Carpenter. "Distributed Ray Tracing," Proceedings of SIGGRAPH'84, in *Computer Graphics*, 18(3), July 1984, pages 137–145.

[COOK87] Cook, Robert L., Loren Carpenter, and Edwin Catmull. "The Reyes Image Rendering Architecture," Proceedings of SIGGRAPH'87, in *Computer Graphics*, 21(4), July 1987, pages 95–102.

[FUJI86] Fujimoto, Akira, Tanaka Takayuki, and Iwata Kansei. "ARTS: Accelerated Ray-Tracing System," *IEEE Computer Graphics and Applications*, 6(4), April 1986, pages 16–26.

[GLAS84] Glassner, Andrew S. "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications*, 4(10), October 1984, pages 15–22.

[GLAS89] Glassner, Andrew S., editor. *An Introduction to Ray Tracing*, Academic Press, Inc., San Diego, California, 1989.

[GOLD87] Goldsmith, Jeffrey and John Salmon. "Automatic Creation of Object Hierarchies for Ray Tracing," *IEEE Computer Graphics and Applications*, 7(5), May 1987, pages 14–20.

[GORA84] Goral, Cindy M., Kenneth E. Torrence, and Donald P. Greeberg. "Modeling the Interaction of Light Between Diffuse Surfaces," Proceedings of SIGGRAPH'84, in *Computer Graphics*, 18(3), July 1984, pages 213–222.

[GREE86] Greenberg, Donald P., Michael F. Cohen, and Kenneth E. Torrance. "Radiosity: A Method for Computing Global Illumination," *The Visual Computer*, 2(5), September 1986, pages 291–297.

[HAIN86] Haines, Eric A. and Donald P. Greenberg. "The Light buffer: a Shadow Testing Accelerator," *IEEE Computer Graphics and Applications*, 6(9), September 1986, pages 6–16.

[HALL83] Hall, Roy A. and Donald P. Greenberg. "A Testbed for Realistic Image Synthesis," *IEEE Computer Graphics and Applications*, 3(8), November 1983, pages 10–20.

[HALL91] Hall, Roy A., Mimi Bussan, Priamos Georgiades, and Donald P. Greenberg. "A Testbed for Architectural Modeling," in Eurographics Proceedings '91, September 1991.

[IMME86] Immel, David S., Michael F. Cohen, and Donald P. Greenberg. "A Radiosity Method for Non-Diffuse Environments," Proceedings of SIGGRAPH'86, in *Computer Graphics*, 20(4), August 1986, pages 133–142.

[KAJI86] Kajiya, James T. "The Rendering Equation," Proceedings of SIGGRAPH'86, in *Computer Graphics*, 20(4), August 1986, pages 143–150.

[KAPL85] Kaplan, Michael R. "Space-Tracing, A Constant Time Ray-Tracer," SIGGRAPH'85 State of the Art in Image Synthesis seminar notes, July 1985.

[KIRK88] Kirk, David and James Arvo. "The Ray Tracing Kernel," in Proceedings of Ausgraph '88, Melbourne, Australia, July 1988, pages 75–82.

[LYTL89] Lytle, Wayne T. *A Modular Testbed for Realistic Image Synthesis*, Master's thesis, Program of Computer Graphics, Cornell University, Ithaca, New York, January 1989.

[NADA87] Nadas, Tom and Alain Fournier. "GRAPE: An Environment to Build Display Processes," Proceedings of SIGGRAPH'87, in *Computer Graphics*, 21(4), July 1987, pages 75–84.

[PETE86] Peterson, J. W., R. G. Bogart, and S. W. Thomas. *The Utah Raster Toolkit*, Technical Report , Department of Computer Science, University of Utah, Salt Lake City, Utah, 1986.

[POTM87] Potmesil, Michael and Eric M. Hoffert. "FRAMES: Software Tools for Modeling, Rendering and Animation of 3D Scenes," Proceedings of SIGGRAPH'87, in *Computer Graphics*, 21(4), July 1987, pages 85–94.

[RUSH87] Rushmeier, Holly E. and Kenneth E. Torrance. "The Zonal Method for Calculating Light Intensities in the Presence of a Participating Medium," Proceedings of SIGGRAPH'87, in *Computer Graphics*, 21(4), July 1987, pages 293–302.

[SPAR78] Sparrow, E. M. and R. D. Cess. *Radiation Heat Transfer*, Hemisphere Publishing Corp., Washington D.C., 1978.

[STRA88] Strauss, Paul S. *BAGS: The Brown Animation Generation System*, Technical Report CS-88-27, Department of Computer Science, Brown University, Providence, Rhode Island, May 1988.

[WALL87] Wallace, John R., Michael F. Cohen, and Donald P. Greenberg. "A Two-Pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods," Proceedings of SIGGRAPH'87, in *Computer Graphics*, 21(4), July 1987, pages 311–320.

[WALL89] Wallace, John R., Kells A. Elmquist, and Eric A. Haines. "A Ray Tracing Algorithm for Progressive Radiosity," Proceedings of SIGGRAPH'89, in *Computer Graphics*, 23(3), July 1989, pages 315–324.

[WEIL88] Weiler, Kevin J. *Topological Structures for Geometric Modeling*, PhD dissertation, Rensselaer Polytechnic Institute, Troy, New York, August 1988.

[WHIT80] Whitted, Turner. "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, 23(6), June 1980, pages 343–349.

[WHIT82] Whitted, T. and S. Weimer. "A Software Testbed for the Development of 3D Raster Graphics Systems," *ACM Transactions on Graphics*, 1(1), January 1982, pages 43–58.

[ZELE91] Zeleznik, Robert C. et. al. "An Object-Oriented Framework for the Integration of Interactive Animation Techniques," Proceedings of SIGGRAPH'91, in *Computer Graphics*, 25, July 1991.