The Ray Tracing Kernel

David Kirk* and James Arvo, Apollo Computer, Inc.,
330 Billerica, Chelmsford, MA 01824, USA.

*current address: California Institute of Technology,
Computer Science 256-80, Pasadena, CA 91125, USA

Abstract

We describe a methodology for implementing a ray tracer which provides both a convenient testbed for developing new algorithms and a way to exploit the growing number of acceleration techniques. These benefits are a natural consequence of a collection of data abstractions called the *ray tracing kernel*. By defining an *object* in a broad sense, the kernel allows a single abstraction to encapsulate a wide spectrum of concepts including geometric primitives, acceleration techniques, CSG operators, and object transformations. Through hierarchical nesting of instances of these objects we are able to construct and efficiently render complex environments.

Introduction

Ray tracing has become a popular research topic since the early work in this area [14, 21] and rapid development continues in all aspects of the algorithm. The flexibility and generality of ray tracing is exemplified by the wide assortment of geometric primitives which have been investigated. These include procedural models [11], fractals [11, 3], swept surfaces [19], bicubic patches [18], and surface tesselations [17]. The problem of efficiently rendering environments comprised of large numbers of these geometric primitives has received even more attention in recent years. The result is an equally diverse assortment of *aggregate* techniques, including bounding volume hierarchies [15, 20], octrees [7, 8], uniform grid subdivision [7], plane sets [12], directional techniques [1, 13], and grid hierarchies [17].

Even the geometric transformations applied to objects have yielded a variety of approaches. In addition to common affine transformations, surface deformations can be performed through nonlinear transformations [2]. The simulation of motion blur introduces the need for time-varying transformations [5].

These trends have motivated the design of a software environment for developing, testing, benchmarking, and combining various ray tracing techniques. Of particular interest to us are acceleration methods such as spatial subdivision algorithms. Though the need for flexible development environments has also been addressed in [6], [10], and [22], our approach has the additional focus of creating novel hybrid algorithms from diverse techniques. The ability to freely mix radically different algorithms results in a powerful tool for constructing and efficiently rendering complex scenes.

We achieve these goals by encapsulating the common features of many important techniques into a single conceptual model. Of central importance is a data abstraction which allows geometric primitives, transformations, acceleration techniques, and other mechanisms to be viewed as "black boxes" with identical interfaces. This data

[1]In *Proceedings of Ausgraph '88*, pages 75–82, Melbourne, Australia, July 1988. Recreated from original manuscript, February, 1996.

abstraction defines an *object* which is a procedural entity capable of a small number of elementary operations.

Objects and Object Classes

Within the kernel, an *object* is a data abstraction for geometrical entities and operations. Figure 1 shows the routines which are required to define a parametric family of objects, or an *object class*. Each of these routines accepts a pointer to class-specific data, or *parameters*, which select a particular member from the object class. In the case of simple geometric entities, or *primitive objects*, the parameters can range from a center and radius for the class of spheres to a set of control points for the class of bicubic patches. The object class routines are then responsible for performing operations on the specified objects in the class. In practice it is convenient to implement object classes as records of function pointers.

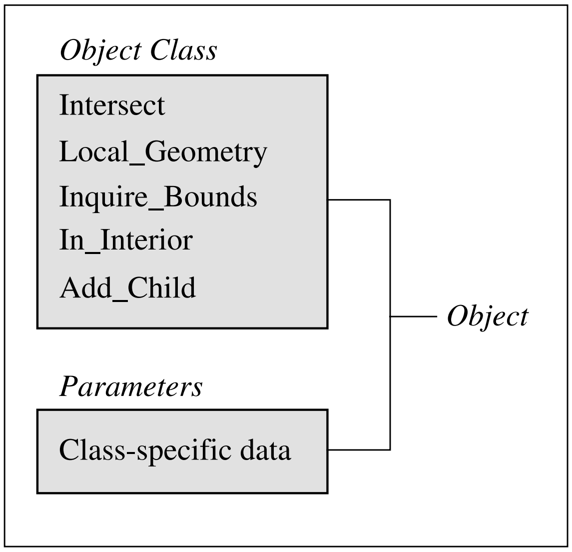

Figure 1: *Selecting an object from a class.*

The "intersect" routine of the object class checks for intersection between a given ray and an object. If the ray hits the object, the distance and point of intersection are returned. The "local geometry" routine computes additional geometric information at the point of intersection, such as the normal vector and possibly tangent vectors (for texture mapping). The "inquire bounds" routine returns a bounding box (or some other bounding volume) for an object and the "interior" routine tests whether a given point is contained within the object. Most primitive objects fit easily into this framework.

Another type of object, one which manages a collection of subordinate objects, or *children*, is called an *aggregate object*. The simplest example of an aggregate object is the *list object* which employs a linked list to maintain the collection of references to its children. These references are called *object instances*, or simply *instances*. The "add child" routine, which is implemented only by aggregate objects, here simply inserts instances into the linked list. The intersect operation of the list performs "exhaustive" ray tracing by invoking the intersect operator of each child in the list sequentially and retaining the nearest intersection. Both the "inquire bounds" and "in interior" routines of the list object proceed by applying the corresponding operation to each of the children and processing the results.

More "intelligent" aggregate objects, such as octrees or uniform grids, can perform these operations without resorting to exhaustive search. This is the first examples of the value of the object class data abstraction. Aggregate objects present the same interface to the outside world as primitive objects and therefore hide their specific algorithms. That is, from the level of abstraction of the kernel, all objects are indistinguishable even though some contain collections of other objects. Because the method used to store and trace through a subordinate collection of objects is completely embodied inside the object class routines, all of the acceleration techniques discussed in the introduction can be easily implemented as aggregate objects. An immediate and important consequence is that seemingly unrelated algorithms can be nested to any depth in a consistent manner.

The object abstraction is also broad enough to include binary operations such as the "difference'," "intersection," and "union" operations used in constructive solid geometry (CSG). For instance, a *CSG difference object* may accept exactly two children and apply different semantics to the first and second child. In out implementation, the CSG difference object computes the difference (child#1 - child#2) by returning only intersections with the first child which are not contained within the second child. By performing this operation strictly through the object class routines we allow the possibility of subtracting aggregate as well as primitive objects.

Certain transformations can be expressed as aggregate objects which accept exactly one child and take modeling matrices as parameters. By transforming rays "going in" and data such as normals, points, and bounds "going out," these objects implement the well-known technique for transforming objects by inverse transforming the rays instead. This is applicable for all affine transformations.

This approach has many of the features of object oriented programming [9]. In particular, each object has a private memory which can only be manipulated through its set of operations, and all of the objects from a single class share the same interface. However, the kernel as we have defined it provides no mechanism for inheritance from one class to another. The operations required of all object classes are essentially the same and must be directly supported by each class. Another difference is that there is no true message passing. All operations are carried out by direct function calls and are strictly synchronous. Finally, the term "instance" has a different meaning than in the context of object oriented programming. Here it implies both selecting a member from a class and making that member a child of another object. Because the latter operation can include a geometrical transformation, and environment may contain many instances of a single object.

Shaders

In addition to object classes which encapsulate geometrical algorithms, another important role is played by *shaders*. A shader is an encapsulation of an illumination model and, in the context of ray tracing, it is the agent which creates creates a ray tree by casting rays in directions determined by the laws of reflection and refraction [21] or determined stochastically [5]. For greater generality we allow a variety of shaders to be used in the rendering of a single environment. The value of this practice is made clear in [4]. We note that the flexibility of the "shade trees" described therein could be embodied within one of potentially many shaders accessed through the kernel.

Aggregate objects are free to impose different policies for application of shaders and surface properties, or *attributes*, to their children. We have found it convenient to associate pointers to shaders and their compatible attribute blocks directly with object instances. To facilitate this, the "add child" operation, which creates an instance, must be allowed to pass additional object-specific data to the parent aggregate object. This data may also contain transform information in the case of aggregate objects which gain efficiency by implementing operations of this nature directly.

An Example

As a concrete example, we will describe the functions required to implement the simple list object mentioned earlier. These are:

- List_Add_Child(data, child, inst_data)
- List_Intersect(data, ray, hit_info)
- List_Local_Geom(data, hit_info, geom)
- List_Inq_Bounds(data, type, bounds)
- List_In_Interior(data, point)

Both List_Intersect and List_In_Interior return a boolean value as their function value indicating success or failure. Here "data" is a pointer to private data (i.e. the parameters) which is accessed and modified exclusively through these operations. The list object's private data will contain a pointer to the head of the list of children, and this is updated as new children are added. The argument encoding object-specific instance data, "inst_data", may be omitted by some aggregate objects, though it is commonly used for transform and attribute information. The "hit_info" argument is used to store information for computing the local geometry at the point of intersection if it is needed. For most aggregate objects, this information will include the child which produced the closest intersection and the "hit_info" returned by its intersect operator. The aggregate's local geometry routine then invokes the corresponding routine of that child, passing it the appropriate "hit_info". We found it useful to implement "hit_info" as a stack, allowing the intersect operators to return arbitrary amounts of data.

The forms of the remaining arguments are fixed among all the object classes, and their definitions constitute part of the kernel. The "child" argument consists of two pointers: one to the object class (a record of function pointers) and one to the object-specific parameters. The "ray" argument encodes a 3D origin, a unit vector, and other information such as "time" and minimum and maximum distance limits. The "geom" argument encodes the surface normal, tangents, etc., and is used strictly as input to a shader. Making this uniform allows all shaders to be applied to all objects.

The "type" argument selects which bounding volume is to be returned an, in some cases contains additional data (e.g. plane-set normals [12]). Aggregate objects may require any or all of these bounding volumes for a child and may perform a wide variety of operations on them. For instance, an aggregate object implementing a bounding volume hierarchy may construct additional volumes enclosing two or more of these volumes, a BSP tree object may need to determine if a volume is intersected by a plane, and an octree object may need to identify voxels which are intersected by a volume. Rather than anticipating and encapsulating these operations into a procedural interface, the exact encodings of these bounding volumes are made public as part of the kernel definitions. Four bounding volume types are supported in our implementation: boxes, spheres, and two forms of polyhedra (hull points and intersections of slabs).

The Trace Procedure

The ray tracing kernel attempts to separate the high-level operations of ray tracing from the pure mechanics of specific algorithms. The kernel itself implements no specific objects, transformations, or shaders, and has no policy concerning how a collection of objects is stored or rendered. In fact, at the level of the kernel there is only one object exporting the generic interface. This single object is responsible, directly or indirectly, for managing the rest of the objects in the environment. This is most vividly demonstrated by the one procedure which is formally part of the kernel; the procedure *Trace*:

- Trace(Ray, Object, Color)

This procedure intersects a given ray with an object and returns the resulting color. It requires almost no actual code outside of the objects and shaders themselves and serves essentially as a guide for how shaders communicate with objects. The kernel is therefore little more than a set of interface definitions and data types which allow all of the details specific to geometry and illumination to be hidden within the objects and shaders.

Hybrid Algorithms

The kernel abstractions allow us to easily add new object classes and modify existing objects without affecting the rest of the ray tracer. More importantly, however, by allowing us to mix diverse acceleration techniques as easily as we can mix diverse primitive objects, it also furnishes a new approach to dealing with complex environments.

For example, we can combine any of the acceleration techniques listed in the introduction into a *meta hierarchy* for ray tracing a single environment. Though Snyder and Barr [17] described a restricted form of this nesting, the ray tracing kernel allows any aggregate object to be the child of any other aggregate object. The nested aggregate appears essentially as a bounding volume and intersection technique to its parent and is therefore handled as easily as a primitive object.

Different aggregate objects will invariably present different trade-offs in terms of space and time. Techniques whose memory usage grows rapidly with the number of objects can be given a coarser world by making use of more levels of hierarchy.

The role of spatial subdivision techniques is to change large problems into small problems within the voxels. Though it is common practice to use "exhaustive" ray tracing on the small collection of objects found within the voxels, this needn't be the case. Bounding volume hierarchies as well as any other optimization technique can also be applied in this context. Object nesting provides a simple means of accomplishing this. Care must be taken, however, to ensure that objects which intersect more than one voxel are not intersected multiple times with a single ray. If the object happens to be an aggregate, this intersection can be arbitrarily expensive.

Nested Transforms

An interesting benefit which can be derived from nested objects is that it is possible to take better advantage of *sparse* transformations. That is, we can transform a ray or normal vector with fewer arithmetic operations by taking advantage of matrices which contain many zeros. This is a consequence of being able to apply transformations at multiple levels in an instancing hierarchy, thereby creating a hierarchy of coordinate spaces. Though this can mean transforming each ray several times before it even reaches a primitive object, frequently the transforms which are lower in the hierarchy are very simple, consisting of scale and translate operations. There are cases when a single *dense* transform places the ray in a coordinate space relative to which most of the objects have sparse transforms.

If N objects are tested before finding the closest intersection, the same operation can sometimes be done using one dense transform followed by N sparse transforms instead of N dense transforms. This is particularly true for a complex object which is built largely from scaled and translated primitives and then rotated into some arbitrary final orientation. However, if it is feasible to make N very small on average, it may be more efficient to pre-concatenate the transforms and place the autonomous objects along with the resulting dense transforms directly into the parent object. The nesting mechanism is not without its own cost, so it must be used judiciously.

Nesting of transforms can be very advantageous when dealing with motion blur and other situations requiring time-dependent transforms. A single time-dependent transform object can be used to transform a collection of objects undergoing rigid motion. That is, by creating an aggregate from objects which are moving as a group but not relative to one another, then applying a single time-dependent transformation, we avoid multiple applications of a potentially expensive transform.

Building a Hierarchy

Thus far we have focused on the high-level abstractions of the ray tracing kernel and have seen how this allows convenient implementation and use of objects. We now describe how the object hierarchy is built. Since the kernel provides no assistance here, we introduce another software layer called the *shell*. Given a set of shaders and a set of object classes such as spheres, polygons, patches, and so on, the shell allows us to construct a hierarchy whose root is the "world" object initially passes to the procedure "Trace."

The object hierarchy actually forms a directed acyclic graph, or *DAG*. The hierarchy is not necessarily a tree since objects may have more than one parent. Note that we can actually relax the requirement that the graph be acyclic if the object class intersectors have a mechanism for terminating recursion. Such a mechanism can be based upon a ray generation counter, for example.

The process of creating the DAG can be phrased in terms of a succession of *open*, *create*, *instance*, and *close* operations which perform elementary bookkeeping functions. Opening an aggregate object means that subsequent instance operations will create children of this object. When an open operation is performed, the object is pushed onto a stack, superseding the previously opened object. The close operation pops the stack. These operations are meaningless to the kernel and merely serve to create the structure which it will recognize. Note that the create operation must handle the object-specific parameters of the created object, and the instance operation must handle the object-specific data associated with the add child operation.

Since the shell is entirely in control of placing the objects in the hierarchy, it can also implement dynamic loading of objects. In such a scheme, the ray tracer contains only the kernel software, and the shell loads at runtime only those objects which are used in the scene. This mechanism is visible only to the shell.

It is useful to have another construct which is visible only to the shell. This is a modeling hierarchy used only as a convenience and not reflected in the final object hierarchy seen by the kernel. In other words, the shell provides "dummy" aggregate and transform objects which function as macros. The shell must assume the burden of copying objects and concatenating modeling transforms when these objects are instanced.

Applications

As an example of the benefits of the kernel we describe a 2.75 minute animated film entitled "Fair Play" [16] which was rendered using a ray tracer built upon the kernel concept. The environment consisted of an amusement park with a number of fairly detailed rides and attractions. Also included were trees, a fractal landscape, and two characters whom we follow through the park. Figures 2 through 4 are frames from the film. Figure 2 is the view from high atop a ride and gives a sense of the overall layout of the park. Figure 3 is a closeup of the characters beneath one of the rides. The red balloon and portions of the characters show reflections of the rest of the park. Figure 4 shows the interior of the house of mirrors. Some of the "corridors" are formed by reflection paths more than 30 deep. The exterior of this attraction is seen in the distance in Figures 2 and 3.

There are approximately 10,000 object instances in the park environment, not including the fractal mountains. Though this can no longer be considered a "large" database in view of the models reported in [1] and [17] containing millions of objects, it nonetheless represents a significant challenge for creating a several-minute ray traced animation sequence. "Fair Play" required almost 3,000 unique anti-aliased frames at 512x384 pixel resolution. Efficient rendering was crucial despite the fact that we employed a network of over 500 Apollo workstations for final production.

The organization of the amusement park environment suggested three very natural levels of detail: I) The entire park, consisting of rides, trees, and mountains, II) The individual rides, and III) small but detailed elements of the rides, such as the horses on a merry-go-round. Clearly a single uniform grid [7] would not have performed well here because of the scale involved. Large numbers of primitive objects would have been collected in a small number of voxels. Octrees, on the other hand, can deal with this problem through adaptive subdivision but cannot pass rays through empty voxels with the efficiency of uniform grids. This suggested a compromise.

Our initial approach was to place a coarse uniform grid around the entire park, and another uniform grid or octree around each ride. Frequently we placed a bounding box hierarchy around small clusters of primitive objects which would have fallen entirely within a voxel of the second-level grid. The low-level bounding box hierarchies were also a way of grouping repeated sub-structures into objects which could be instanced without replicating all the data.

Concurrently with the movie production we developed and tested a new acceleration technique [1]. This was integrated almost effortlessly into the ray tracer as another aggregate object indistinguishable from the others. Therefore, we were able to immediately substitute the new aggregate object for the top level uniform grid. This increased overall efficiency because the new algorithm took advantage of directional information and performed additional optimizations on first-generation rays.

Because the new algorithm performed adaptive subdivision in five-dimensional space, it required large amounts of virtual memory when processing complex environments. Therefore, we sometimes gave this object a coarser view of the world by grouping primitive objects into larger aggregates in a fashion similar to building a bounding box hierarchy. This drastically cut down the amount of storage consumed, allowing the ray tracer to run well on smaller machines. This strategy has the disadvantage of preventing the parent object from differentiating between objects within its aggregate children. This is typical of space/time trade-offs in which one form of optimization must be sacrificed in order to avoid the greater penalty of paging. The ability to easily nest various objects provided the flexibility to make trade-offs of this nature.

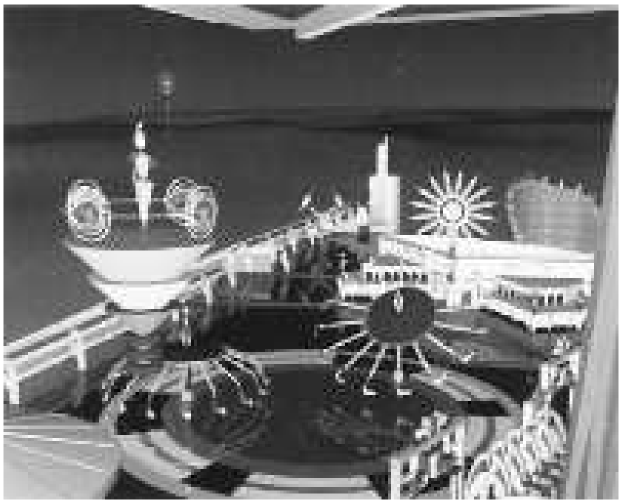

Figure 2: *A view of the amusement park.*

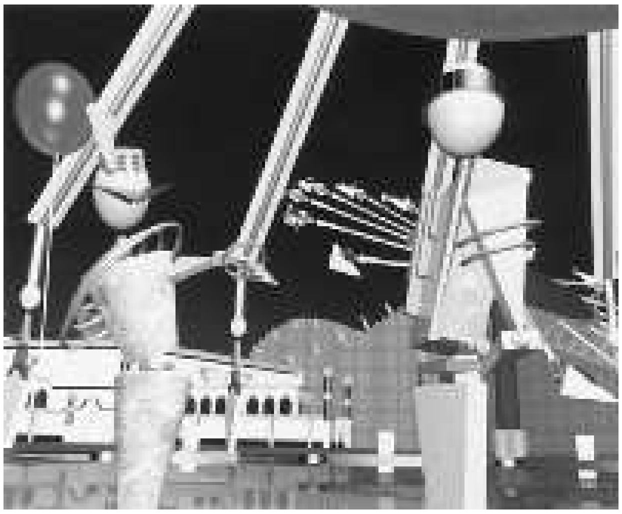

Figure 3: *The characters beneath a ride.*

Summary

The ray tracing kernel approach provides both a flexible research platform which can easily accommodate new features, and a production tool which can take advantage of many acceleration techniques simultaneously in the rendering of a single environment. A ray tracer built using the kernel paradigm was used to produce a large-scale ray traced animation through a combination of acceleration algorithms. It also served as an effective tool for the development and benchmarking of a new acceleration algorithm.

References

[1] James Arvo and David Kirk. Fast ray tracing by ray classification. *Computer Graphics*, 21(4):55–64, July 1987.

[2] A. H. Barr. Global and local deformations of solid primitives. *Computer Graphics*, 18(3):21–30, July 1984.

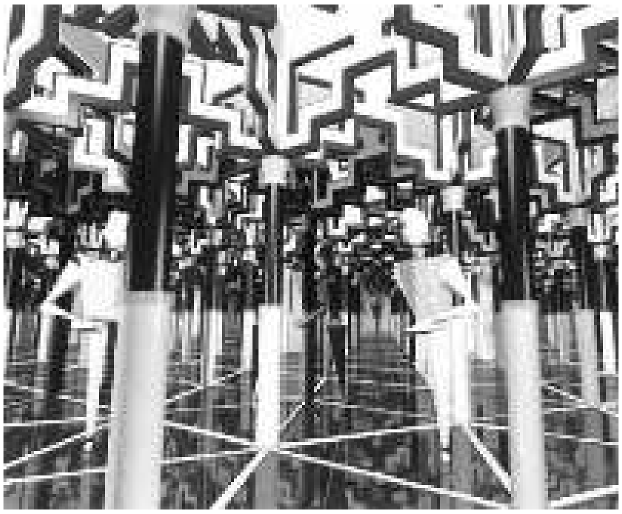

Figure 4: *In the house of mirrors.*

[3] Christian Bouville. Bounding ellipsoids for ray-fractal intersection. *Computer Graphics*, 19(3):45–51, July 1985.

[4] Robert L. Cook. Shade trees. *Computer Graphics*, 18(3):223–231, July 1984.

[5] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. *Computer Graphics*, 18(3):137–145, July 1984.

[6] F. C. Crow. A more flexible image generation environment. *Computer Graphics*, 16(3):9–18, July 1982.

[7] Akira Fujimoto and Takayuki Tanaka. ARTS: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, April 1986.

[8] Andrew Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[9] Adele Goldberg and David Robinson. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[10] R. A. Hall and D. P. Greenberg. A testbed for realistic image synthesis. *IEEE Computer Graphics and Applications*, 3(10):10–20, November 1983.

[11] James T. Kajiya. New techniques for ray tracing procedurally defined objects. *ACM Transactions on Graphics*, 2(3):161–181, July 1983.

[12] Timothy L. Kay and James Kajiya. Ray tracing complex scenes. *Computer Graphics*, 20(4):269–278, August 1986.

[13] M. Ohta and M. Maekawa. Ray coherence theorem and constant time ray tracing algorithm. In T. Kunii, editor, *Computer Graphics 1987*, pages 303–314. Springer-Verlag, New York, 1987. Proceedings of *CG International '87*.

[14] S. D. Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18:109–144, 1982.

[15] Steve Rubin and Turner Whitted. A three-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, July 1980.

[16] M. Sciulli, J. Arvo, M. White, J. Gilby, D. Kirk, K. Lefebvre, M. Marderosian, G. Hawks, A. Topeka, P. Toohil, A. Coppola, R. Palmer, P. Neal, C. Scofield, M. Shebell, C. Bremser, E. Peters, J. Francis, S. Reber, N. Benovich, E. Wunderlich, G. Rose, N. Zawistowski, O. Lathrop, V. Odryna, R. Morrison, J. Graber, A. Lopez, T. Crane, J. Bowbeer, S. Costello, J. Beck, J. Parsons, D. Penfield, G. Odryna, M. Duhamel, L. Leary, F. Cozza, M. Poklemba, A. Markuvitz, M. Allard, R. O'Neal, and B. Hashizume. Fair play. Apollo Computer, 1987. A production of the *Midnight Movie Group*.

[17] John M. Snyder and Alan H. Barr. Ray tracing complex models containing surface tessellations. *Computer Graphics*, 21(4):119–128, July 1987.

[18] Daniel L. Toth. On ray tracing parametric surfaces. *Computer Graphics*, 19(3):171–179, 1985.

[19] J. J. van Wijk. Ray tracing objects defined by sweeping planar cubic splines. *ACM Transactions on Graphics*, 3(3):223–237, July 1984.

[20] Hank Weghorst, Gary Hooper, and Donald Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, January 1984.

[21] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 32(6):343–349, June 1980.

[22] Turner Whitted and David M. Weimer. A software testbed for the development of 3D raster graphics systems. *ACM Transactions on Graphics*, 1(1):43–58, January 1982.