Computer Graphics
Term Project Report
TRay - The final trace

Arvind Thiagarajan John P John

May 3, 2004

Abstract

The traditional approach to computer graphics rendering has been the forward graphics pipeline consisting of transformations, clipping and scan conversion. These are essentially object space methods. The other approach to solving this problem is ray tracing, which is an image space method and works on an illumination-per-pixel rather than per-object basis. Ray tracing makes it possible to model realistic effects like shadows and specular reflection much easier than in the traditional methods. Other effects like recursive reflection, refraction, transformations, constructive solid geometry, texture mapping, dielectric surfaces can also be modeled by ray tracing or simple extensions to it. Finally, harder to capture global illumination effects including caustics and inter-diffuse reflections are handled by path tracing, a close cousin of ray tracing. In this term project, we present the implementation of a ray tracer supporting the above features which we call *Tray*. This report describes the features we implemented, the theoretical basis for each method and the practical implementation challenges that ere faced in addition.

1 Aim and Motivation

This section will be a little informal. The whole story began one afternoon when browsing images submitted for the rendering competition on graph-

ics.stanford.edu. The addictive realism of the images and their stunning quality led us to write to a couple of students at Stanford on what term project we could take up. Huamin Wang of Stanford graciously responded by asking us to try out a ray tracer. He said that it would teach us a lot of things about graphics and that we would be able to create realistic images.

The motivation behind image space methods is that they are able to more easily simulate effects like shadows and specular highlights that cannot be easily captured by object space methods that are traditionally used in real time rendering. Further, the quality of a rendering produced by a ray tracer can be much superior to real time rendering using graphics libraries like OpenGL or DirectX.

More importantly, we think what really excited us about ray tracing was that we would be actually implementing a subset of the graphics pipeline, not using functionality provided by other libraries as mentioned above. This meant two things: one, that it would be much more challenging (as it would involve core computer graphics algorithms and implementation, rather than just the use of a higher level API) and second, that we would have more control over the rendering.

2 Introduction to Ray Tracing

In itself, ray tracing is a simple and elegant algorithm. It is basically a point sampling method that traces infinitesimal beams of light (that we shall call rays) through a model of a scene. The fundamental idea in ray tracing is that light rays can be traced back from the observer to the sources. This approach can easily handle mirror reflections, refractions and direct illumination. The advantage of ray tracing is that the geometry of the scene is treated like a black box - rays are traced into the scene, they intersect objects, and return some illumination value. This means that handling complex geometry is easier with ray tracing. It also has the disadvantage, however, that certain coherence properties of objects cannot be exploited. Further, backwards ray tracing cannot possibly generate global illumination effects like caustics, soft shadows, indirect illumination and color bleeding.

The input to a ray tracer is a description of the scene that is to be rendered. This description is normally written in a scene description language (SDL) that the ray tracer understands. The SDL file will typically include the

following: The position and parameters of the eye/camera, the specifications of objects in the scene (position, shape and size) and the parameters of the image to be generated (size, antialiasing/filtering options). Objects can be specified in a variety of ways in a rray tracer - the most common technique is primitive instancing, where the object is an instance of some primitive that the ray tracer supports natively.

The algorithm works by shooting rays from the eye through each pixel in the final image. These rays intersect objects in the scene and return a color value for that pixel. The color value returned typically includes the direct illumination for the nearest object intersected by a ray (in ray casting) but it can also include the effects of multiple specular reflections and refractions (in recursive ray tracing).

In the following sections, we present each of the modules of our ray tracer and the features we implemented in chronological order.

3 Steps in Implementing *TRay*

3.1 Ray Casting, Spheres and Ambient Light

The first step in any implementation is the basic ray casting algorithm. The idea is as follows: For each ray cast through the scene, determine the nearest object along the line of sight that intersects the ray. Each type of primitive that the ray tracer supports will need to have an intersect() routine that intersects the object with a given ray. Among the easiest primitives to intersect with a ray is a sphere - this is why ray traced images invariably contain spheres.

3.1.1 Casting a Ray

To cast a ray through the scene, one needs to transform from the coordinates as given in the scene file to image space coordinates. As in the forward graphics pipeline, the scene file will contain the world coordinate positions of the **CW** (Centre of Window) and the **COP** (Centre of Projection). In a ray tracer, the **COP** is the eye and the **CW** is normally termed "look" or "lookat" (as it is a point that the eye is supposed to be looking at). Finally, the SDL file will also contain the **VUP** (view up vector) that is used to determine the direction which is "upwards".

Consider a ray from the **COP** through pixel (i,j) of the image whose width is w pixels and height is h pixels (w and h are either parameters for the ray tracer or set to defaults). Let the field of view angle (also specified in the scene file) be *fov*. The direction of the ray $\mathbf{d}$ is given by the (vector) equations:

$$\mathbf{DOP} = \mathbf{CW} - \mathbf{COP}$$

$$\mathbf{DU} = \mathbf{DOP} \times \mathbf{VUP}$$

$$\mathbf{DV} = \mathbf{DOP} \times \mathbf{DU}$$

$$fl = w \div (2 * \tan fov/2)$$

$$\mathbf{CORNER} = ((fl \times \mathbf{DOP})/DOP) - w\mathbf{DU}/2 - h\mathbf{DV}/2$$

$$\mathbf{d} = \begin{bmatrix} DU.x & DV.x & CORNER.x \\ DU.y & DV.y & CORNER.y \\ DU.z & DV.z & CORNER.z \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix}$$

3.1.2 Ray - Sphere Intersection

Consider a ray whose origin is given by the position vector $\mathbf{p}$ and whose direction is given by $\mathbf{d}$. The parametric equation of the ray is given by: $\mathbf{r} = \mathbf{p} + t\mathbf{d}$ while the equation of the sphere is given by $|\mathbf{r} - \mathbf{c}| = r_0$. These two equations can be solved directly for $\mathbf{r}$, the position vector of the point of intersection, but it is normally inefficient to do so. Instead, a better approach is to first use a quick test to see if the ray is actually heading towards the sphere. This is done by determining the vector $\mathbf{c} - \mathbf{p}$, which is the line joining the origin of the ray to the centre of the sphere. The dot product of this with $\mathbf{d}$ is determined. If this is negative, the ray can never intersect the sphere and can quickly be rejected. This leads to a substantial time saving, especially for complex scenes.

There is one subtle point in the above method: testing the dot product $(\mathbf{c} - \mathbf{p}).\mathbf{d}$ will not work if the ray originates inside the sphere. This is not a

problem with ray casting as all rays originate only from the eye. However, when extended to recursive ray tracing, this approach fails. The ray sphere intersection needs to be written with more care in this case, especially for the case of refraction, where the origin of the ray can be inside a sphere.

3.1.3 Simple Ambient Lighting

The simplest model for direct illumination is to assign an intrinsic color $O_{a\lambda}$ to the surface of each object (in terms of red, green and blue components or more generally, for any λ). Each object is also assumed to have a coefficient of ambient lighting k_a that controls the ambient illumination. This model assumes uniform illumination of the object: irrespective of the point of intersection, the intensity returned by the ray is $I_a k_a O_{a\lambda}$ for each ambient light source and each color component λ. However, this is sufficient to test the simple ray caster and the sphere intersection routine, which is what it is basically meant for.

3.2 Phong Illumination and Point Sources

The next step in implementation, obviously, was to improve the primitive lighting model. We extended our ray caster in this stage to handle point light sources and directional lights. The illumination model used is called the Phong model and is a very simple, easy to implement model that is popular in most ray tracers. This is not a physically based model - it is just intended to give aesthetically pleasing results. The Phong lighting model for any surface consists of three distinct components: ambient lighting (this is given by the term mentioned in the previous section), diffuse lighting and specular lighting. The model is based on the observation that some surfaces are perfectly specular, i.e they reflect light only along one direction (e.g mirror reflectors like silver or other metals) while others scatter light uniformly in all directions (e.g walls, chalk). Further, some surfaces are in between in that they tend to scatter light in a cone about the mirror reflected ray. These are glossy surfaces (e.g glass, apples).

The diffuse lighting component is independent of the direction of the ray being cast, and depends only on the angle at which the light from the source falls on the object being illuminated. The diffuse illumination in the Phong model is therefore given by $I_p k_d O_{d\lambda} \cos(\theta)$ for a given point light source p.

In this equation, k_d is called the coefficient of diffuse reflection and $O_{d\lambda}$ is an intrinsic diffuse color for the object (in *Tray*, this is the same as the ambient color as a separate diffuse color is not really necessary).

The specular reflection component tries to model the property of glossy surfaces of light being cast in a cone about the reflected ray. First, given a point source p whose position vector with respect to the point of the ray-object intersection is $\mathbf{S}$, we compute the reflected ray using the law of reflection. The reflected ray $\mathbf{R}$ is given by:

$$\mathbf{R} = 2(\mathbf{N}.\mathbf{S})\mathbf{N} - \mathbf{S}$$

In the above equation, $\mathbf{N}$ is the normal vector to the surface at the point of intersection. The illumination due to specular reflection is given by $I_p k_s O_{s\lambda}(\mathbf{V}.\mathbf{R})^{n_s}$, where $\mathbf{V}$ is the view vector (along the reverse of the ray which just intersected the object) and n_s is a property of the surface called shininess. The higher the value of n_s, the closer the surface is to a perfect mirror. Lower values of n_s are used for imperfect reflectors like glossy surfaces.

The Phong lighting model can be easily extended to multiple light sources by summing the contributions from each light source (ambient, point or directional). Care needs to be taken to clamp the individual components of the color (red, green and blue) to a maximum of 1, however. Another point worth noting is that since computation of both diffuse and specular lighting requires knowing the value of $\mathbf{N}$, the ray tracer needs to be extended not only to compute ray-object intersections, but also the normal to the object's surface at the point of intersection. This is very easy to do for spheres, for instance.

The main limitation of simple models like this and indeed, of ray tracing itself, is that they cannot capture global illumination effects. There is no easy way to capture indirect illumination of diffuse surfaces with ray tracing. Effects like soft shadows and caustics are also impossible to render with ray tracing alone. A solution to this problem that we implemented is described at the end of the report.

3.3 Recursive Ray Tracing, Reflection and Shadows

The next feature that was added was recursive ray tracing. A simple ray caster only computes reflected illumination due to light sources: it does not compute specular reflections bouncing off other objects in the scene. This is handled elegantly by the recursive ray tracing algorithm: When a ray strikes a specular surface, a new ray (corresponding to the reflected ray) is created and is traced through the scene. This ray originates from a point just above the surface (the reason for this is explained below) and is directed in the reflected direction as computed using the law of reflection. This ray, like all other rays in the scene, returns an illumination value. The illumination due to the ray is scaled by a factor k_r which is called the reflectance of the surface. This is added to the illumination due to the light sources in the scene.

One very important point that has been ignored till now is the possibility of occlusion of a light source due to shadows cast by other objects in the scene. To handle this, a new kind of ray called shadow rays are introduced. These are traced towards the light source (if it is a point source) from a point just above the surface of the object under consideration. The same ray-object intersection routines can be reused for shadow rays to determine if the shadow ray intersects any object that is closer than the light source. If the light source is blocked, then we do not add the illumination due to that light source. It is instructive that a feature that is considered highly complex in a scanline-based forward pipeline (shadows) are so easy to implement with ray tracing.

The fact that the ray tracing algorithm has become recursive complicates things a bit, though. Some practical implementation issues that we faced in this regard were memory leaks and stack overflows due to recursion, with memory not being freed properly. A second, important issue, (hinted at earlier) is the origin for recursively spawned rays. One cannot begin tracing the ray exactly from the intersection point, mainly due to the vagaries of floating point numbers in modern computers. There is always the possibility that such a ray might intersect the very same surface again! To avoid this, the rays need to start from a point just above the surface.

3.4 SDL Parser

At this stage, we implemented a parser using *yacc* for a simple scene description language, to test the ray tracer on different scenes. The SDL used is very simple and is meant only as a tool to test the program.

3.5 Refraction and Transparency

Modeling transparent objects in a ray tracer is a bit more of a challenge - it requires implementing refraction. In principle, this is the same as reflection, with a refracted ray being created instead of a reflected ray. The equation governing the direction of the refracted ray is given by Snell's laws of refraction:

- The incident ray, refracted ray and the normal to the surface at the point of incidence all lie in the same plane.
- If i is the angle of incidence and r is the angle of refraction, then $\sin i / \sin r = n_2/n_1$, where n_1 is the refractive index of the medium of the incident ray and n_2 is the refractive index of the medium of the refracted ray.

Once the refracted ray is computed correctly, the computation of illumination is much the same as in reflection - trace the refracted ray and scale the illumination returned by k_t which is the transmission coefficient for the medium.

The real difficulty in implementing refraction is not in this computation, but rather in keeping track of the refractive indices ! When a ray enters a medium, we need to keep track of the refractive index of the medium it has just entered. However, when it leaves the medium and enters a medium, there is no easy way to know the refractive index of the new medium. The solution is to store the refractive indices in a hierarchy based on the geometry of the scene - however, this becomes very complicated even for simple scenes. We have sidestepped this issue by forbidding nesting of refractive indices, that is, the only possible interfaces are between air and some other medium. Supporting nested refractive indices in a ray tracer is much more of a challenge and is a possible feature that could be added.

Another interesting case is **total internal reflection**, where a ray gets trapped inside a refracting medium with high refractive index. This can be

handled similar to reflection, by spawing a new totally reflected ray and tracing it. However, there are cases when the recursion never terminates, and the ray remains trapped within the object. To handle this problem and prevent stack overflows, one needs to limit the recursion depth. Beyond this limit, we do not trace recursive reflected/refracted rays. This also significantly saves on rendering time for complex scenes with reflection. Even a depth of as low as 3 is enough to capture most effects in a scene.

3.6 Transformations

Transformations like translation, rotation and scaling are useful in a ray tracer for modeling more complex scenes. This enables modelers to visualize the object they wish to model in its own object coordinate system, and then position it in its environment using an appropriate combination of translation, rotation and scaling. Another advantage is that more complex objects can sometimes be obtained by transforming primitives. For example, an ellipsoid is simply a scaled version of a sphere!

Transformations are implemented in *TRay* using a transformation matrix stored with each object. When the SDL specifies a transformation to be applied to an object, we multiply the transformation matrix for that object with the matrix for that transformation. Homogeneous coordinates are used to unify translation with rotation and scaling. When we want to intersect a ray with a transformed object, the trick is to apply the inverse transformation to the ray and proceed with the intersection as we normally would. The result (intersection point and normal) are in object coordinates and need to be transformed back to world coordinates.

Transforming the intersection point is straightforward, but the normals are trickier to handle. The idea used to transform normals is to transform the plane perpendicular to the normal, and to get back the new normal. Alternatively, it is possible to argue that translation does not affect the normal's direction, rotation simply rotates the normal by the same amount, and scaling scales the normal by the inverse factor. We have used the second approach as it is more efficient.

One final point is that the transformation matrix, its inverse and the normal transformation matrix are all precomputed for efficiency. We have used LUP decomposition in *TRay* to invert the 4x4 matrix.

3.7 Other Objects - Planes and Boxes

The next feature that was added to *TRay* was support for objects other than spheres. We started off with planes and boxes as the next two objects to consider. Both of them are quite easy to intersect with a ray.

For a plane, the standard Cyrus Beck formulation is used:

$$t = \mathbf{N}.(\mathbf{p} - \mathbf{p_e})/ - \mathbf{N}.\mathbf{d}$$

where $\mathbf{p}$ and $\mathbf{d}$ are the origin and direction of the ray, as before, $\mathbf{N}$ is the plane normal and p_e is a point on the plane (this with $\mathbf{N}$ actually specifies the plane in the SDL file).

For boxes, the approach is to simply intersect the ray with each of the 6 bounding planes and use the minimum positive value of t so obtained. Back face culling seems to be a method for optimization, but when one remembers that rays can originate inside objects, it fails miserably.

Infinite planes with mirror surfaces intersecting at different angles can be used to create excellent looking scenes with multiple reflections. We did try out such fun experiments with *TRay*!

3.8 Constructive Solid Geometry

Another feature present in *TRay* is CSG (Constructive Solid Geometry) that enables objects to be modeled as the union, intersection or difference of primitives (or other CSG objects). While such a feature is very hard to implement in the conventional scanline models, it is much easier with ray tracing. As explained earlier, ray tracing is more or less independent of the geometry of the scene: all that is needed to render an object with ray tracing is a ray-object intersection routine and a normal computation method. In a way, these are the interfaces an object needs to implement.

A CSG object is normally represented by a tree, whose leaves are primitive objects and whose internal nodes are boolean set operators (union, intersection and difference). This representation is particularly convenient in ray tracing, as explained below.

To intersect a ray with a CSG object (represented by a node in the tree), all we do is to recursively intersect the ray with each of its children. These children could themselves be CSG objects, or primitives (leaves). The trick

is to do extra bookkeeping: instead of just keeping track of the nearest intersection of the ray with each object, we keep track of all the intersections. These intersections are stored as a sequence of intervals that represent the ray entering and leaving the object respectively.

From the sequence of intervals for each child of the original object in the tree, we reconstruct the intervals for the original object by applying the appropriate operation (union, intersection or difference) to the set of intervals. Thus, with ray tracing, the complicated process of applying Boolean set operators to solid objects is replaced by the much simpler process of performing the same operators on sets of intervals. We have used an efficient O(n) algorithm for performing the CSG operations on the intervals. This was done by a technique similar to k-way merging of lists.

However, there are other, more subtle issues in CSG: those of ensuring the validity of the resultant object. For example, it is difficult, if not impossible to support infinite/open objects like planes in CSG as the ray-object intersection interval can extend to infinity. This means such boundary cases need to be handled properly by the ray tracer, which really complicates the CSG implementation.

3.9 Antialiasing using Simple Supersampling

Since ray tracing is inherently a point sampling process, it suffers from aliasing problems. These problems are often exacerbated by the fact that ray tracing is "unaware" of the scene geometry and cannot exploit object or scanline coherence to solve aliasing problems. Thus, the image produced by the ray tracer needs to be antialiased to eliminate or reduce effects like jaggies and irregular object boundaries.

The simplest approach to antialiasing is naive supersampling, where instead of firing just one ray per pixel of the image, multiple rays are fired through different positions in the pixel. The contribution from each of these rays is averaged to get the illumination for that pixel. For 4 rays, one each through the corners of each pixel, the extra cost involved is negligible but the improvement in visual appearance is often marked. However, all artifacts cannot be eliminated completely by naive supersampling. Casting more rays than 4 can improve image quality, but substantially slows down rendering. A better solution that we implemented later is described in this report.

3.10 Triangles and Meshes

More complex objects are normally represented as triangle meshes or B-reps. This means that ray tracing complex objects reduces to being able to compute ray-triangle intersections efficiently. The triangle primitive was implemented in *TRay* using an efficient algorithm called the MT algorithm developed by Moller and Trumbore in 1997. This method not only computes the ray triangle intersection, but also computes some other parameters which are useful for shading.

Shading a wireframe or mesh object is more complicated. The three methods traditionally used are flat shading, Gouraud shading and Phong shading. The first approach simply assumes that all points in the interior of a triangle have the same normal and uses this normal for the shading calculations. While simple and fast, this has the shortcoming that it leads to **Mach bands** - discontinuities across triangles that are actually visible in the final rendering. To improve on this, Gouraud shading interpolates the illumination at the vertices of the triangle to compute the illumination at an interior point. The best method is Phong shading, which interpolates the normals rather than the illumination values at the vertices. *TRay* supports flat and Phong shading. The interpolation is done using a nice trick: The weight of a vertex is simply the area of the triangle formed by the opposit edge with the interior point. These areas are used by MT for intersection and are precomputed by the MT algorithm, which makes the approach all the more efficient.

Let $\mathbf{V_1}$, $\mathbf{V_2}$ and $\mathbf{V_3}$ be the position vectors of the vertices of the triangle and let $\mathbf{N}$ be normal to the plane of the triangle. Let $\mathbf{e_1}$ and $\mathbf{e_2}$ be the edges sharing vertex $\mathbf{V_1}$. All of the above parameters can be precomputed to save time. Let the ray to be intersected with the triangle originate at $\mathbf{p}$ and have direction $\mathbf{d}$. The Moller - Trumbore test for triangle intersection is as follows:

$det = \mathbf{d}.\mathbf{N}$
if(det = 0) then return false
else compute:
$u = \mathbf{d}.(\mathbf{e2} \times (\mathbf{p} - \mathbf{V_1}))/\det$
if($u \leq 0$ or $u \geq 1$) then return false
$v = -\mathbf{d}.(\mathbf{e1} \times (\mathbf{p} - \mathbf{V_1}))/\det$

if($v \leq 0$ or $u + v \geq 1$) then return false
else $t = -(\mathbf{p} - \mathbf{V_1}).\mathbf{N}/\det$

Interestingly, u and v turn out to be the areas one can use for Phong shading and interpolation, as also for texture mapping as described later.

We have ray traced a teapot with 1560 triangles using *TRay*. The rendering took a really long time of 75 minutes and led us to try to optimize the ray object intersections.

3.11 Bounding Volume Hierarchies

We have implemented the bounding volume hierarchy technique proposed by Kajiya to speed up the ray object intersection calculations. The idea is to bound each object in the scene by an axis aligned bounding box. These boxes are themselves bounded in groups by other, bigger bounding boxes. The boxes are thus arranged in a hierarchy with the root bounding box bounding the entire scene. Each box contains all of its children. The leaves of the tree bound actual primitives or CSG objects.

The ray object intersection can be sped up by first testing a ray against the root of the hierarchy. If the test fails, we can save on further ray object tests. If the ray intersects the root, we push all its children into a heap order by distance from the ray. From this, we examine the closest bounding box and repeat the procedure. If this test fails, we try with the next child in the heap, and so on. This procedure is extremely efficient and saves on a huge number of intersection calculations.

Constructing a good hierarchy in itself was a challenge - we have used a heuristic search based on Goldsmith and Salmon's cost function (1987) to determine a good hierarchy.

After implementing bounding volume hierarchies, the time to render the same teapot was cut to a mere 7 minutes.This represents a speedup by a factor of 10. We consider the implementation of bounding volumes as the hardest stage in this term project.

3.12 Other optimizations

To speed up *TRay* to the maximum extent possible, we have profiled the code in detail and code tuned by hand all the important routines. One such

routine is the ray - bounding box intersection routine. Even though rays can originate inside bounding boxes, it is safe to use back face culling for bounding boxes as once a ray is found to originate inside, one has to consider all the children of that box for intersection anyway.

Another optimization we have implemented is exploiting shadow ray coherence. Once a shadow ray intersects an object, we cache the object that just intersected this shadow ray. The next time a shadow ray intersection is done, we first test with the object in the cache, if any.

Finally, we have implemented separate shadow ray intersection routines for each object. These are based on the fact that we only need to determine if there is an intersection for a shadow ray, not necessarily the closest intersection.

With all of the above optimizations, the rendering time for the teapot scene came down to a minute. We consider this to be an important achievement in our term project.

3.13 Adaptive Supersampling

The next thing we did was to try to improve antialiasing using an adaptive rather than a naive algorithm. An adaptive algorithm shoots more rays in those areas of the image that are more prone to aliasing, and thus saves time in other areas. This is done based on the variance in the values returned from the 4 corners of a pixel. The pixel is recursively subdivided into 4 sub areas and the process of firing rays is repeated recursively. Once the difference in illumination is smaller than a cutoff, we stop the recursion. Adaptive supersampling considerably improves image quality but also slows down the ray tracer. The teapot rendered with this method with a cutoff depth of 5 took 7 minutes to render, a slowdown of about 6-7 times.

3.14 Texture Mapping

Texture mapping is a feature that is very easy to add to a ray tracer. The idea is to map parametric u-v coordinates on an object's surface to **texels** in the texture image. For a sphere, u and v are simply the latitude and longitude expressed as fractions between 0 and 1. These map onto texels based on the width and height of the texture file.

One issue encountered in texture mapping is that of filtering. One texel may map onto multiple pixels in the output image which leads to blockiness in the output image. A solution to this is to use some filtering method. We have used bilinear filtering by interpolating texels based on the fractional parts of the u and v coordinates returned by the ray object intersection routine.

3.15 Monte Carlo Path Tracing

Ray tracing cannot simulate full global illumination, including features like caustics, soft shadows, indirect illumination on diffuse surfaces and color bleeding. This is because only specular rays are traced and not diffuse rays. One solution is to use a Monte Carlo probabilistic method to evaluate diffuse illumination: The idea is to trace a random ray at each diffuse surface and add the contributed illumination. To be effective, a large number of sample light paths need to be used for each pixel.

We have rendered a room with an area light source and two specular spheres, and diffuse walls and ceiling using the Monte Carlo method with 10, 100 and 1000 paths per pixel. For fewer samples, the main problem is one of noise in the output image. At a higher number of paths per pixel, the noise artifacts disappear. We have successfully been able to render soft shadows and area lighting, and a caustic (if a bit feeble!) using the path tracing algorithm.

4 Results and Conclusion

We have successfully implemented a ray tracer *TRay* with support for refraction, transformations, planes, CSG, adaptive antialiasing, meshes, bounding volumes, textures, area lights and soft shadows using path tracing. Possible extensions include nested refractive indices, ray tracing torii and quadrics, photon mapping, subsurface scattering and better illumination models (e.g physically based illumination models).

References

[1] http://graphics.lcs.mit.edu/classes/6.837

[2] http://graphics.stanford.edu

[3] Computer Graphics - Second Edition in C - Foley, Van Dam, Feiner and Hughes, 1991

[4] Realistic Image synthesis using Photon Mapping - Henrik Wann Jensen, 1997

[5] Ray Tracing Complex Scenes - Kajiya, 1986