MIPSRAY: A MIPS Ray Tracer

EECS 314: Computer Architecture

Josh Lee, Gary Doran

April 29, 2008

1 Problem Statement

The objective of the MIPSRAY Project was to develop a Ray Tracer using the MIPS Assembly Language. Ray tracing is a technique for rendering images of three-dimensional scenes that models the physical way that rays of light behave. As most light rays do not end up reaching the camera, simulating the path of every ray of light would be computationally infeasible. Instead, the algorithm shoots rays out of the camera into the scene, then simulates the reflections and refractions in reverse.

The actual ray tracing algorithm is relatively straightforward, given a camera (specified with vectors for its position and the image frame), a scene full of objects (e.g. spheres, planes), and light sources. For each pixel in the frame, a ray is generated that starts at the camera and passes through that pixel into the scene. The intersection of the camera ray with each object in the scene is computed, and the one nearest the camera is chosen. If there are no hits, the pixel is assigned a background color. If the ray hits an object, a shadow ray is shot towards each light source from the hit to determine whether (and to what extent) the surface is illuminated. Furthermore, a reflected ray is generated at the point of intersection, and the algorithm is called recursively on this ray. In this case, the color of the pixel is determined by combining the color of the object hit, the degree of illumination, and color returned by the reflected ray. The process just described is performed for every pixel in the frame to produce an image of the scene.

2 Major Challenges

The major challenges of the MIPSRAY Project arose from attempting to implement various aspects of the Ray Tracing algorithm. First of all, the appropriate data structures had to be selected to represent various objects in the scene. Because it is relatively easy to mathematically manipulate vectors, three-vectors of double precision floating point numbers were commonly used. Thus, rays became a pair of vectors, one specifying the ray's origin and another unit vector specifying the direction of the ray. Objects such as a sphere were primarily specified by a vector indicating the sphere's center and a double for its radius, along with other information pertaining to the sphere's color (another vector) and reflectivity. Light sources were represented by a vector for position and a vector for color. Because random access of the light sources and scene objects was not necessary, linked lists were used to store these objects. Accordingly, the first word of each object/light source was either null or a pointer to the next object/light source. Of course, along with these data structures came the challenge of creating procedures to implement various operations on these data structures such as vector operations or finding an intersection between a ray and an object.

Other major challenges arose from more practical issues related to implementation of the Ray Tracing algorithm. For example, the generated image needed to be written to the hard disk in the form of a bitmap (BMP) image file. This process required knowledge of both the BMP file format, as well as the assembly instructions required to create, open and write to the file. In addition, some basic mathematic knowledge was required to implement certain aspects of the algorithm.

For example, basic vector arithmetic and the ability to find roots of polynomials were required to find the intersection of a ray with a sphere. The recursive nature of the algorithm required the stack to be used, which added a small challenge to the implementation, particularly due to booking concerning memory offset values. Furthermore, since it was difficult to test certain parts of the program individually due to both the nature of the algorithm and of assembly language, debugging was difficult while meaningless output files were produced. Only after recognizable (albeit incorrect) outputs were produced did debugging become much less challenging.

3 Key Components

3.1 vectors.s and roots.s

The first key component of the implementation was the set of mathematical tools needed to manipulate various data structures and calculate intersections. A set of procedures, located in the file "vectors.s" were written to perform vector operations, including addition, subtraction, dot product, and scalar multiplication. In addition, several procedures calculate values such as the magnitude of a vector, or return a unit vector in the direction of a given vector. These procedures were used throughout the program. Another file, "roots.s" houses procedures that find the roots of polynomials (linear and quadratic equations). These procedures were used primarily to calculate the intersections of rays with objects in the scene. The main challenges with implementing the root finder involved considering all possible cases for a given equationan equation can have either none, one, or two roots or a quadratic equation could actually be a linear equation if the coefficient of the degree-two term is zero. The procedures in "vectors.s" and "roots.s" provide a toolbox that is used heavily throughout the rest of the ray tracing program.

3.2 shapes.s

Another key component of the ray tracer required finding points of intersection, reflections, etc. involving rays and objects. The procedures in the "shapes.s" file handle these operations. Because intersections are found differently with different shapes, main procedures "intersect," "reflect," and "normal" call the appropriate shape-specific procedures depending on the type of object they are passed as an argument. The intersect procedure returns the distance from the origin of a ray to the first point a ray intersects with an object (and specifies whether such an intersection exists).

Spheres Given a ray specified by $\vec{s}$ and $\vec{d}$ (source and direction), and a sphere specified by $\vec{c}$ and r (center and radius), the intersection can be found with some basic algebra. First, assume that the ray intersects with the circle at a distance t away from its source. Then $\vec{s}+t\vec{d}$ will be the point of intersection. By the definition of a sphere, the distance squared between this point and the center of the circle will be r^2. So $\left\| \vec{s}+t\vec{d}-\vec{c} \right\|^2 = r^2$. We can write this formula as:

$$\begin{aligned}
&\left\| (\vec{s}-\vec{c})+t\vec{d} \right\| = r^2 \\
&\left((\vec{s}-\vec{c})+t\vec{d} \right) \cdot \left((\vec{s}-\vec{c})+t\vec{d} \right) = r^2 \\
&(\vec{d}\cdot\vec{d})t^2 + 2\vec{d}\cdot(\vec{s}-\vec{c})t + \left(\| \vec{s}-\vec{c} \|^2 - r^2 \right) = 0
\end{aligned}$$

This quadratic equation can then be solved for t to find the distance to the closest intersection. The equation will have no real roots if no such intersection exists, and only negative roots if the intersection occurs behind the ray—in both cases, the ray misses the object entirely.

Planes Given a similar ray, and a plane specified by a point $\vec{p}$ on the plane and a unit normal to the plane $\vec{n}$, an intersection between a ray and a plane can be found. At the point of intersection, $\vec{s} + t\vec{d} = \vec{p} + \vec{v}$, where $\vec{v}$ is some vector that lies on the plane (i.e. is parallel to the plane). Since $\vec{v}$ and $\vec{n}$ are necessarily at right angles to one another, $\vec{v} \cdot \vec{i} = 0$. Using this fact, the equation above can be rewritten as:

$$\begin{aligned}
\vec{n} \cdot (\vec{s} + t\vec{d}) &= \vec{n} \cdot (\vec{p} + \vec{v}) \\
(\vec{n} \cdot \vec{s}) + (\vec{n} + \vec{d})t &= (\vec{n} \cdot \vec{p}) + 0 \\
(\vec{n} \cdot \vec{d})t + \vec{n} \cdot (\vec{s} - \vec{p}) &= 0
\end{aligned}$$

This linear equation can be solved for t as with the sphere to find the distance to the closest intersection with a plane. In addition to finding intersection points, "shapes.s" also contains a procedure to return the unit normal vector to a point of intersection. This procedure is useful for calculating the intensity of light at a given point on an object. Furthermore, "shapes.s" contains a procedure that generates a reflected ray from an intersection between a point and an object. To do this, it uses the fact that the starting point is the point of intersection and the direction is given by $\vec{r} = \vec{d} - 2(\vec{n} \cdot \vec{d})\vec{n}$, where $\vec{n}$ is the unit normal to intersection and $\vec{d}$ is the direction of the original ray.

The starting point of the reflected ray is actually set a small distance along the direction of $\vec{r}$, because otherwise subsequent calls to the procedure "intersect" would return a value of zero (since it would think that the ray intersected with the object at its origin). Finally, the "nearestHit" procedure loops through each object in the scene with a given ray and finds the distance to intersection with each object. It determines whether a hit occurs with any object, and returns the object that is first hit by the ray. All of these procedures are critical components of the ray tracer implementation.

3.3 bmp.s

To save images to a bitmap file, a module was required to first write the bitmap file header, then output each pixel as it was rendered. As the BMP files were only written, not read, a full-featured image loader was not necessary. A BMP file begins with a fixed-length header specifying, among other fixed values, the image's width, height, and size in bytes. The width and height are specified in the scene file, so the bitmap module can compute them at the beginning, then write the entire header out to the file. The "bmp.write_header" procedure fills in the header variables based on the width and height given. The "bmp.write_pixel" procedure takes three bytes, one for each of red, green, and blue, and writes them to the file. Each row of a BMP file ends with up to 3 bytes of padding for word alignment purposes, so this procedure also increments a counter to see if the end of the row is reached.

3.4 main.s

The last major component of the MIPSRAY project was the actual implementation of the Ray Tracing Algorithm. The algorithm was implemented in the file "main.s" in three major procedures. The first procedure, "main," create an image file, loops through pixels in the frame, and generates a

ray passing through each pixel. It then calls the "raytrace" procedure on this ray, and writes the returned color to the image file. The "raytrace" procedure is recursive, and returns immediately if the proper level of recursion is reached. Otherwise, it looks for the nearest hit object. If no object is hit by the ray, a background color (e.g. black) is returned. If an object is hit, the "shadow" procedure calculates the intensity of the color at that point given light sources and obstructions. Next, a reflected ray is generated and "raytrace" is called recursively on this ray. The contributions from the light and the reflected ray are added together, and this value is returned. The "shadow" procedure works by looping through each light source in the scene and determining whether another object obstructs the path to the light source. If so, then only a basic, ambient amount of light is illuminating the point of intersection. Otherwise, the intensity of the light from that light source is given by the cosine of the angle between the direction to the light source and the unit normal to the intersection. This value can be found using a dot product. The contributions are summed over all lights, and this value is returned to the "raytrace" procedure. All of these components are necessary to implement the Ray Tracing Algorithm.

4 Integration of Components

Because various components of the ray tracer contain different sets of tools, they are relatively easy to integrate. Procedures in "main.s" depend directly on functions supplied by the files "shapes.s," "bmp.s," and "vectors.s." The procedures in "shapes.s" also rely heavily on procedures in "roots.s" and "vectors.s." Major challenges in integration arose during debugging, when it was sometimes impractical to test functions of "lower-level" procedures except by their role in "higher-level" procedures in "main.s." In such cases, identifying the root cause of the bug became difficult. Otherwise, integration of components was relatively straightforward.

5 User Interface

The input and output to the ray tracer are primarily the scene and the image file, respectively. Various aspects of the scene are specified in a file called "scene.s." The scene file specifies the output filename, the dimensions of the image, the camera, the objects, the light sources, and the shading parameters. The two parameters, "ambient" and "diffuse", respectively determine how much of an object's illumination comes from external light sources, and how much it will be seen when it is completely in shadow. The background color, used when no object is hit by a ray, is also specified here.

To render a scene, all of the source files (bmp, vectors, roots, shapes, main, scene) are loaded into the SPIM simulator. When the program is executed, the scene is automatically rendered and saved to the scene's filename.