Graphic Scene Simulations

Amiga graphics so realistic

they seem to transcend reality,

conjuring a vivid surrealistic world.

By Eric Graham

BEHOLD THE ROBOT juggling silver spheres. He stands firmly on the landscape and gleams in the light. He is only a microchip phantom, yet he casts a shadow. You can see his reflection in the refined orbs he so deftly tosses. He inhabits space, in a pristine computer's dreamscape. Though he looks strangely real, he exists only in the memory of the Amiga.

This colorful automaton is not your everyday computer graphics robot. He wasn't carefully rendered with a paint program, nor was his image captured by a video frame grabber from a picture or a model. He and his surrealistic world were "automatically" created with a C program. In this article, I will describe how these graphic simulations are created through a technique called *Ray Tracing*.

A ray-tracing program, written in C and named *Graphic Scene Simulation*, accompanies this article. These images require a hold-and-modify (HAM) display. Because of space limitations, my listing doesn't include the routines for producing a hold-and-modify screen. However, my routines do form the logical core of a ray-tracing application. The procedure for creating the routines necessary for setting up and releasing HAM screens can be found in Addison-Wesley's *Amiga ROM Kernel Reference Manuals*. For those of you uninitiated into C (or who just don't care to program), details will be given at the end of this article about obtaining the program on disk with the executable code, and information on inputting data to create scenes of your own.

Gaining Perspective

Computer images are usually made up of two-dimensional shapes. In the Amiga's case, these shapes are filled with color. As in any two-dimensional artform, if the shapes are combined in the right way, the result can give the appearance of a third dimension—one inhabited by solid objects. To say it in reverse: Three-dimensional computer images can be broken down into two-dimensional shapes (i.e., the pieces that make up the whole). The final product, the three-dimensional image, has the look of substance, volume, depth, reality. The whole, as it were, becomes greater than the sum of its parts.

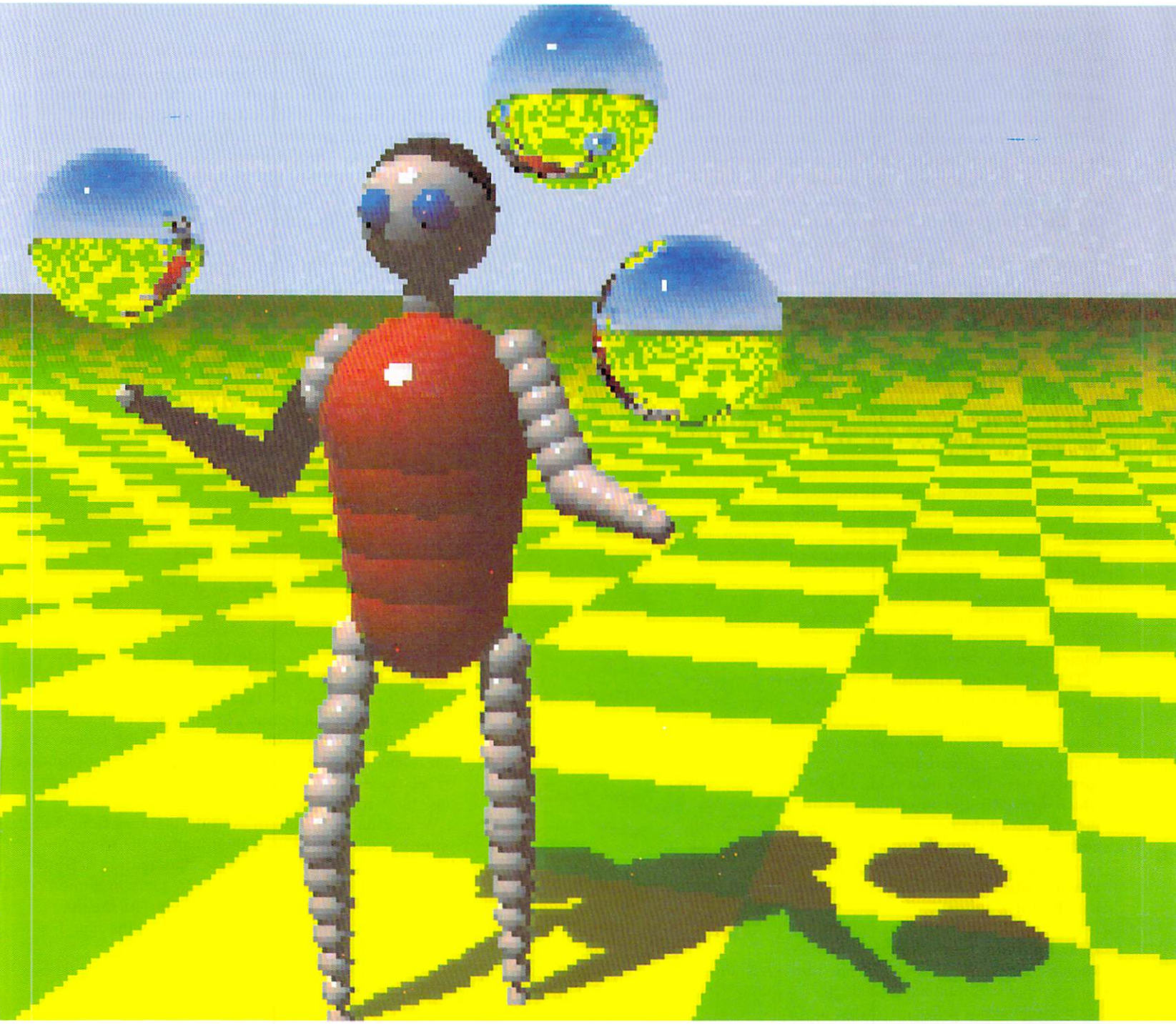
Highly sophisticated computer graphic simulations go beyond this and attempt to replace the often cartoon-like computer image with one containing all the nuances of color and shading that we see in the real world. Such graphic simulations are now commonplace; you see them before TV network news programs, movies and sports events, and as station logos. Graphic simulations have also been used in many movies, such as *Tron* and *The Last Starfighter*.

Although many simulations employ supercomputers like the Cray XMP and high-powered (and expensive) graphic displays, you can use your Amiga to experiment with sophisticated graphic simulations. The Amiga is unique among modestly priced computers in that it provides the ability to display many colors and shades of brightness. Using the hold-and-modify (HAM) mode, you can use up to 4,096 colors and shades on the screen at the same time. The Amiga, as you can see in the images I have created, is very capable of producing subtle graphic simulations.

An Alternate Reality

When you perform a graphic simulation, you are given a rare opportunity to in some ways design your own universe. You obviously have such mundane choices as what colors to use for the ground and the sky (if you even want a ground or sky), but you can also choose

The robot juggler. All aspects of this HAM image were "automatically" created with a ray-tracing program.



your own laws of physics.

Because you are trying to make a visual rendering, you must decide how light is to behave in your world. Does it travel in straight lines or in curves? How does it reflect off different objects? You must also populate your universe with objects. Finally you must place an "observer" in the universe, for it is the observer's view that you wish to display on your computer screen.

The Robot's Universe

Our robot's universe is particularly simple. It has a flat ground, checkered like cheap vinyl flooring, that stretches off into infinity. It has a sky that changes from one color directly overhead, to another color at the horizon. (Being a traditionalist, I chose shades of blue, but you can choose any two colors you wish.)

The universe requires one discrete place from which to make all measurements. (Mathematicians call such a place an origin.) The objects in this universe are all the same shape; they are spheres. Each sphere has a position, a radius, a color and a surface finish. (Spheres have one very nice property: you do not have to specify ►

their orientation. Rotate a plain sphere and you cannot see the difference.) The position of each sphere is specified by its coordinates, which are a set of three numbers. If you wish, you can regard the first of these numbers as how far the center of the sphere is to the north of the origin. The second number specifies how far the sphere is to the west of the origin; the last number measures how far the sphere is above the origin.

The color of a sphere is represented by a set of three numbers also, representing the fraction of red, green or blue light that is reflected from its surface. Exactly how light is reflected from the sphere depends upon what the surface of the sphere is like. In this universe there are only three possibilities to consider: First, the surface can be dull, in which case light scatters in every direction; second, the surface finish can be shiny, with most of the reflected light being scattered, but a little being reflected in one specific direction; finally, the surface can be like a mirror, with no light being scattered and all reflected light going in one direction.

Everything we could possibly want to know about a sphere in this universe can be summarized by eight numbers: three for the position, one for the radius, three for the color and one for the surface type. The Amiga's memory is large enough to hold the information on tens of thousands of spheres, so we could, in principle, build quite a complex universe.

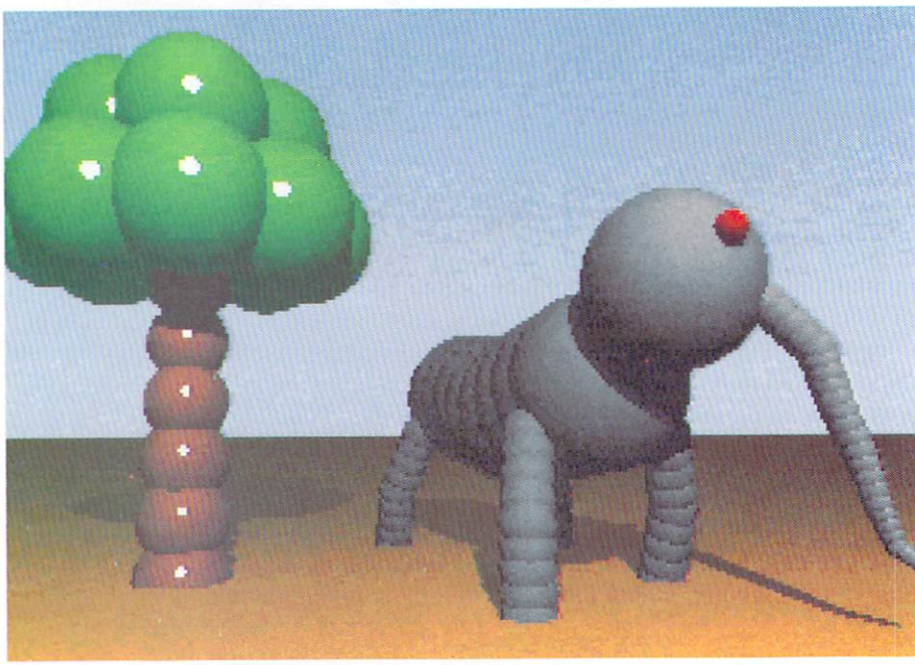
Illumination has been provided by positioning some lamps. The lamps are yet more spheres, but they give out light rather than reflect it. To soften the shadows, we have provided ambient lighting: diffused light that comes from all directions.

The last element specified is the "observer." Perhaps it is better to think of the observer as a camera. We will try to calculate what light would enter the lens of the camera, and so be able to display on the screen of our computer what picture the camera might take. We must specify where the camera is located (three numbers), in which direction it is pointing (two numbers, one for the number of degrees it is pointing away from north, and one for how many degrees it is pointing above or below the horizon), and finally how wide the camera's view is—what photographers describe as the focal length of the camera lens.

How to Generate a Picture

We have condensed an entire scene into a few numbers, so that it can be handled by the computer. Now we must go about calculating what to display on the screen. We have two ways of solving the problem. The first is called *Boundary Representation*, or B-rep by the experts. This involves considering where the apparent boundary of each object would lie on the screen and then filling in the part of the screen inside the boundary with the appropriate color. This is particularly easy on the Amiga because the system software will handle area filling. Unfortunately, boundary representation has a couple of drawbacks. The color within a border varies due to shading and reflections, so a simple area fill will not suffice for realistic scenes. The second problem is that parts of one object may be hidden by another object, so determining which parts are visible may be complicated.

The other way of generating scenes is called *Ray Tracing*. It is conceptually very simple: Imagine that you are in the position of the observer. You take a glass plate, scribed with a grid of lines that creates squares that match the pixel positions on the screen, and hold it up in front of you. The color that you see through one square of the plate is just the color that you need to display on the screen at the corresponding pixel location. Follow the light ray backward, from your eye, through the glass plate and out into the world. Once we find the first thing that this backward ray strikes, we can calculate the brightness of the object at that point by multiplying the color of the object by the light that the object receives.



An imaginary pachyderm stands by an exotic tree. The landscapes in all these images stretch back to infinity.

We calculate the light by considering lines from the point to each of the lamps illuminating the scene. So long as nothing blocks a line, the illumination depends upon the brightness and color of the lamp and upon the distance from the lamp to the point. As the light striking an object becomes more oblique, the illumination per unit area decreases.

The ray-tracing method has a number of advantages, the foremost being that the code itself is much simpler than the code for boundary representation. It automatically handles complications like perspective, reflections, shading and shadows. It makes it easy to add new features, like differently shaped objects and different rules for reflections. Ray tracing's main drawback is that, for simple scenes at least, it is slower than boundary representation.

Hold and Modify

In order to display scene simulations with many hues and subtle shading, more than two or four, or even 16 colors must be available. The Amiga's hold-and-modify (HAM) mode, allowing up to 4,096 colors to be simulta-

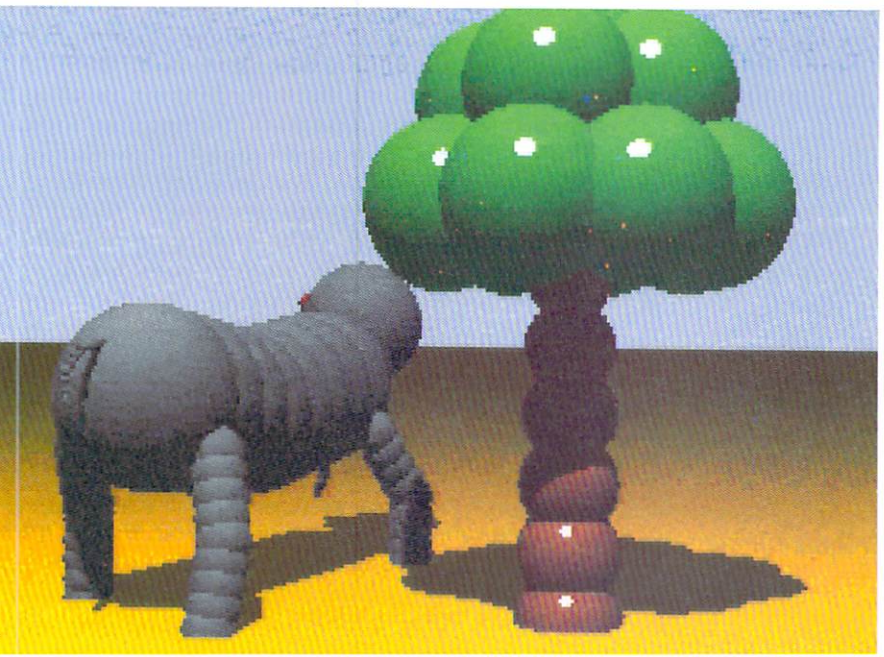
neously displayed, allows great subtlety. Colors are created on the screen by displaying a combination of the primary colors: red, green and blue. (The Amiga permits you to display 16 different levels of each primary color; these combinations produce the 4,096 colors that are possible.)

However, HAM does have one drawback due to the details of the way that it works. When you generate a screen image with HAM, you can't directly specify any color that you want. Instead, you can say how one pixel differs from the preceding one. The catch is that you can only change one color at a time. If you have just displayed an area of black (red = 0, green = 0 and blue = 0) and you want to display white (red = 15, green = 15 and blue = 15), you must, for example, first change the red value, then the green value and finally the blue value. The sequence of pixels will be black, red, yellow and white, rather than a clean transition. Therefore, brightly lit objects against a dark background will have a colored fringe around them, rather like the chromatic aberration that you see in a pair of inexpensive binoculars. The colored fringes, however, can be reduced.

HAM Tweeking

HAM does permit you to specify a color for a pixel directly, so long as it is one of 16 colors that you can select and store in the Amiga's color registers. If you choose these 16 colors wisely, then you can make sharp color transitions more cleanly and the colored fringes are hardly noticeable.

The same pachyderm and tree, but automatically recalculated and displayed from a different viewpoint.



In principle, after you know all the color transitions that your picture requires (up to 64,000 of them), you could make your selection of what to put in the 16 color registers in order to minimize the fringes. That would be a lot of work. You could pick the 16 to span the spectrum of colors without regard for the picture content. What I have done in my program is to employ what is called a heuristic method (i.e., an approach to problem solving whereby updated solutions are found, while a program is runnimg, by continually building on results or feedback). At first I don't use any of my 16 direct colors. When I first encounter a sharp transition, I assign the color value to one of the free registers. As the display progresses, when I encounter another color transition I look to see if the contents of one of the color register values is close enough; if it is, then I use that value rather than use up one of the remaining registers. All you have to do is fine-tune the meaning of "close enough." If you are too picky, you will have used up all your registers early and you won't be able to cleanly handle a color change combination that you encounter at the end of the display.

One other problem that is shared by all raster displays is called aliasing, or more commonly, the "jaggies." This is caused by the finite size of the pixels. The closer a slanted edge is to horizontal, the more the grid of pixels creates a staircase effect. The human eye is expert at spotting these aberrations. Sophisticated anti-aliasing schemes have been devised to recognize the jaggies and then modify the image in their vicinity to make them less apparent. A simpler scheme is to slightly blur the image by mixing the color of each pixel with the colors of its neighbors.

A more subtle problem arises when you have very gradual shading. Since the colors can only change in discrete steps, an effect appears like that on a contour map, with rings of colors painfully evident. The most effective solution is to purchase a computer with a higher resolution display, and with more than 16 levels for each color. For those of us without deep pocketbooks, a simple trick is to slightly randomize the colors so as to break up the regularity of the contours and make them less evident. This gives the final picture a slightly grainy look that is not unpleasant.

Speeding up the Display

On the Amiga, with a resolution of 320 × 200, we have to calculate the brightness of 64,000 pixels. In the ray-tracing method, there is one ray for each pixel. If the scene contains 50 objects, we have to see if the ray intersects an object. If it strikes several objects, we must know which one is the closest to the observer. Then we have to evaluate the brightness at the nearest object. This involves checking to see if another object lies between the first object and each lamp. Each intersection test involves a few dozen floating-point arithmetic operations. Since the Amiga cannot perform floating-point operations directly, each operation must be broken into many simpler operations. The time required to generate a single picture this way could be as long as a month!

Whenever you are faced with the problem of optimizing code to increase speed, you should step back from the details and see what is going on. Obviously a lot of arithmetic is being done here. How can it be made faster? Recasting the equations from floating point to integer form would be possible, but would be extremely unpleasant. A simpler approach would be to replace ►

the usual Amiga arithmetic routines with the *Motorola Fast Floating Point Package*. (The program I used to generate the pictures with this article differs from the listing printed here in that it has the Motorola routines included.) The Motorola routines are less accurate but run nearly ten times faster. Now our picture only takes a few days to generate.

The next step in optimization is to pretend to be a computer and to follow through each stage of the calculation. The first thing that you will see is that most of the calculations are quite unnecessary. Many rays will not intersect any objects at all. Many raster scan lines will also miss the objects. If you can detect this, then a lot of time can be saved, even though your code gets more complicated.

Putting in these features can speed up the generation of a typical scene by a factor of one hundred. Now we can make a picture in an hour. Doubtless some further improvements are possible, but they would require considerable programming effort. Professional scene simulations on a Cray supercomputer (on a finer raster grid, and with more complicated objects) take several minutes, so we are not doing badly with our Amiga!

The ray-tracing method permits one further trick. Suppose we generate a small picture, only one quarter of the size of the screen; now we only have to calculate the brightness of a sixteenth of the normal number of pixels. These mini pictures only take a minute or two to generate, and you can use them to make sure that the observer is in the right place and the lamps are at the right brightness.

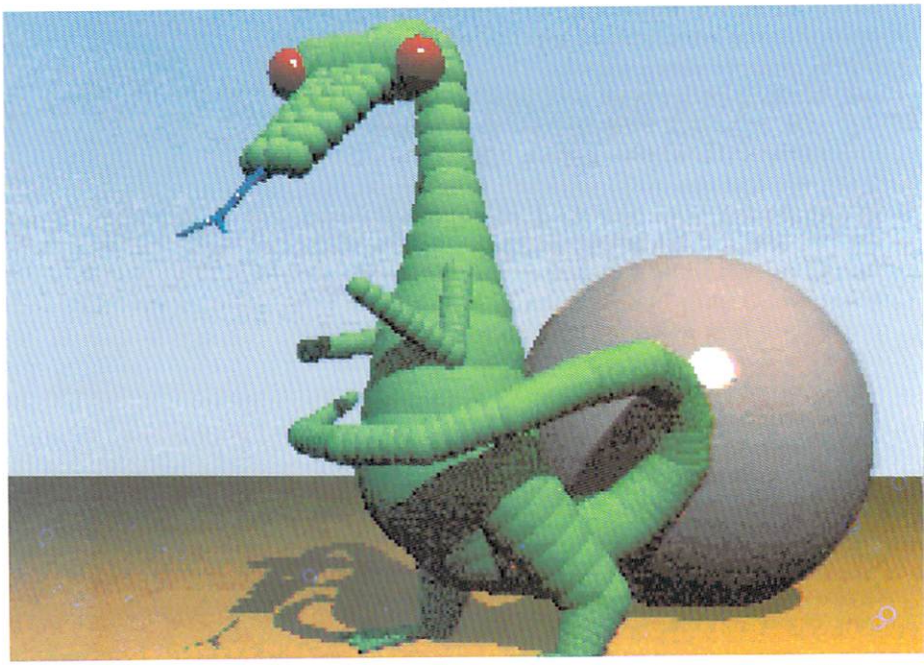
A Ray-Tracing Program

The first part of the accompanying listing, Graphic Scene Simulation, defines the data structures describing the world of spheres. The comments in the code should suffice to explain how it works, but a few remarks about the underlying mathematics may make it more comprehensible. I have used vector arithmetic to handle three-dimensional geometry, so you don't have to be bothered with trigonometry or other complications.

A vector is an object that has both a magnitude and a direction. Think of it as an arrow. It can be pointed in any direction, and the length of the arrow represents its magnitude. A vector can be represented by an array of three numbers, or components. Vectors can be added or subtracted, simply by adding or subtracting their components. In this program, the function *vecsub()* performs vector subtraction.

Vectors are involved in multiplication in three ways. If a vector is multiplied by a number, then each component is multiplied by the number. Two vectors can be multiplied to produce a single number: This is sometimes called a dot product; the function *dot()* performs this operation. A dot product is equal to the products of the magnitudes of the vectors multiplied by the cosine of the angle between the vectors. (As you can see, *dot()* is easier to implement than it is to describe!) One useful result is that the magnitude of a vector, b, is given by *sqrt(dot(b,b))*. Vectors whose magnitude is equal to 1.0 are called unit vectors, and are very useful when you just want to indicate a direction. The vectors *uhat*[] and *vhat*[] are unit vectors that are used to represent the orientation of the screen from the observer's viewpoint.

The third way of multiplying vectors is called a vector product, because the result is also a vector. The function *vecprod()* implements this operation. The result is a vector at right angles to the two vectors being multiplied, with a magnitude equal to the products of the magnitudes and the sine of the angle between the vectors. This is useful in the function *reflect()*, which calculates the direction that a ray takes after being reflected.



A recently-hatched dragon and a dragon egg? As are all the inhabitants of this computed reality, the dragon is made up entirely of spheres.

The rays that we are using are straight lines, but how are these to be represented in a computer? We will use something called the parametric equation of a line. It is really very simple. The coordinates of a point that lies on a line are calculated from seven numbers. Six of them serve to define the line (three to locate the position of the line and three to specify the direction of the line). The seventh number is called the parameter, and it tells you where you are on the line. The array *line*[6] is used to represent a line, and the function *genline()* calculates the six components of a line. The function *point()* takes a line and a value for the parameter—I usually call the parameter *t*—and calculates the coordinates of the point on the line.

Another important geometric function is *intsplin()*, which tests to see if a given line hits a particular sphere. Since all the points on the surface of a sphere are the same distance from its center, we only have to see if any point on the line is also that distance from the center of the sphere. Usually the line either misses altogether or hits the sphere twice, once going in and once emerging. Our use of the parametric equation for a line lets us distinguish these cases. We are interested in the intercept closest to the observer, because that is what he sees.

Three other functions are important in the ray-trac- ►

ing process. The function *pixline()* generates the equation for a line corresponding to a particular pixel on the screen. The function *raytrace()* takes this line and sees where it winds up. The line could intercept a sphere or a lamp, or it could hit the ground. What matters is what the ray hits first, because that is what is seen. If it misses everything, then it must be heading towards the sky, so we just paint the pixel with an appropriate sky color. The last function to be described is *pixbrite()*, which calculates the brightness of a pixel associated with a ray that hits a sphere or hits the ground. We then have to calculate the illumination at that point on the surface of the sphere or the ground.

The function *pixbrite()* checks to see which, if any, lamps are shining on the point because they could be eclipsed by other objects. The illumination depends on the color and brightness of the lamp, upon its distance and on the angle between the lamp and the surface. The remainder of the code adds straightforward detail. For example, *gingham()* calculates the checkerboard pattern of the ground and *skybrite()* calculates the brightness of the sky.

As mentioned earlier, I have omitted the code for setting up and releasing the HAM screen; the procedure is described in the *Amiga ROM Kernel Reference Manuals*. Also, it is up to you to set up the data describing the observer and the objects in your universe.

If you would like a disk containing the complete source code for a ray tracer, *including* the HAM routines, send your name and address to me at the address listed at the end of this article and enclose $15 to cover duplication and shipping charges. The disk contains versions of the source in both C and Amiga Basic. Also included is the version of the program used to generate the images for this article (in case you only want to construct new scenes, rather than get involved with any programming) and an explanation of how you can input your own data into the program.

Listing 1. *Graphic Scene Simulation.*

```
#define BIG 1.0e10
#define SMALL 1.0e-3
#define DULL    0
#define BRIGHT  1
#define MIRROR  2

double dot();   /* Vector dot product */
struct lamp {
    double pos[3];      /* position of lamp */
    double color[3];    /* color of lamp */
    double radius;      /* size of lamp */
};

struct sphere {
    double pos[3];      /* position of sphere */
    double color[3];    /* color of sphere */
    double radius;      /* size of sphere */
    int type;   /* type of surface, DULL, BRIGHT or MIRROR */
};

struct patch {          /* a small bit of something visible */
    double pos[3];      /* position */
    double normal[3];   /* direction 90 degrees to surface */
    double color[3];    /* color of patch */
};

struct world {  /* everything in the universe, except observer */
    int numsp;          /* number of spheres */
    struct sphere *sp;  /* array of spheres */
    int numlmp;         /* number of lamps */
    struct lamp *lmp;   /* array of lamps */
    struct patch horizon[2]; /* alternate squares on the ground */
    double illum[3];    /* background diffuse illumination */
    double skyhor[3];   /* sky color at horizon */
    double skyzen[3];   /* sky color overhead */
};

struct observer {       /* now the observer */
    double obspos[3];   /* his position */
    double viewdir[3];  /* direction he is looking */
    double uhat[3];     /* left to right in view plane */
    double vhat[3];     /* down to up in view plane */
    double fl,px,py;    /* focal length and pixel sizes */
    int nx,ny;          /* number of pixels */
};

main()
{
    double line[6],brite[3];
    struct observer o;  struct world w;
    int i,j,ii,jj,skip; short int si,sj;
```

Listing continued on p. 92.

Future Simulations

The Amiga is a very capable machine for scene simulation; I have only hinted at the possibilities. It would not be difficult to replace the spheres in our example with more complicated objects. Or how about spending a week of your Amiga's time generating a 10-second movie with graphic simulations? Each HAM image takes 48K bytes, so you will need extra memory if you want to construct a long movie. I have been able to compress the images so that even a 512K Amiga can be used to display moving scene simulations. At 30 frames per second, the interplay of moving shapes, shadows and reflections gives an eerie impression of reality, or should I say super-reality—that surrealistic alternate universe that exists within your Amiga.□

Eric Graham is an ex-astronomer and software developer living in the mountains of New Mexico, surrounded by computers. Write to him at PO Box 579, Sandia Park, NM 87047. Eric's ray-tracing program (with a full Intuition interface) will be available from Byte by Byte this summer.

REFERENCES

Denken, Joseph, *Computer Images: State of the Art* (Stewart, Tabori & Chang, 1983). This book contains more than 250 color images that involve varying degrees of computer intervention.

Sorensen, Peter R., "Simulating Reality with Computer Graphics" (*Byte*, March 1984, pp.106–134). This is an excellent review of professional graphic simulations, including examples and interviews with some of the masters of the craft.