Illustrating Physics with Ray-Traced Computer Graphics

T.J. Bensky, California Polytechnic State University, San Luis Obispo, San Luis Obispo, CA

This paper provides a brief introduction to using ray-traced computer graphics for creating illustrations to be used in physics teaching. The article focuses on Povray, a freely available ray-tracing software program. We have found that a ray-traced illustration produced with this software provides a final-image quality that is far superior to hand-drawn illustrations and those produced using standard click-and-draw computer drawing software. Techniques for illustrating time-dependent scenarios are discussed as well.

As teachers in physics (and other scientific fields), we tend to make a lot of diagrams to help get our pedagogical messages across. In the role of teaching or explaining a complicated concept, a *good* figure usually goes a long way. In fact, it has been said that "with a good figure the problem is already half solved."[1] When a "really good" (e.g. neat) figure is needed, many of us probably use a computer drawing program to produce it.

It is the intent of this paper to introduce the reader to the program called Povray, which allows for the creation of two- and three-dimensional still or animated (time evolving) graphics with very little effort. Povray produces images using a technique called "ray tracing," which in our experience creates stunning figures with attention-grabbing detail and perspective that is very difficult to obtain using other techniques. Also, Povray excels in the two cases where the usual methods of producing diagrams fail: creating anything in three dimensions or anything illustrating time-dependent phenomena. Over the past several years, we have found Povray to be indispensable in producing convincing, attention-grabbing figures and animations at all levels of physics teaching.

Table I. A basic Povray scene defined as a light source, camera, and sphere.

```
#include "colors.inc"
light_source { <2,0,-10> color White }
camera { location <0,1,-5> look_at <0,0,0> }
sphere { <0,0,0>,1 pigment {Yellow} }
```

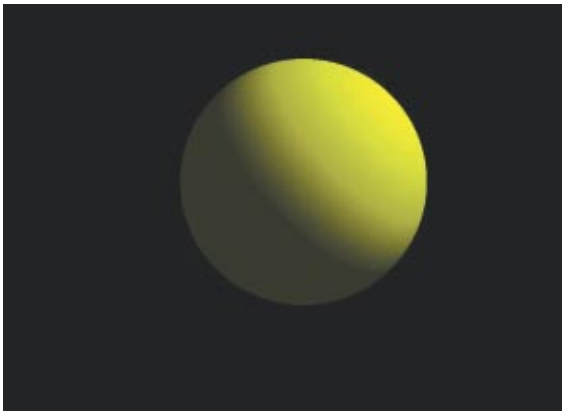

Fig. 1. A ray-traced figure of a single sphere, illuminated by a light source, which is back and to the right. The Povray code for this figure is shown above in Table I.

The Ray-Tracing Method for Producing Figures

Povray differs from the standard palette-driven computer drawing program in that it generates a

Table II. Adaptation of the code shown in Table I, which includes changing the light source and add a plane below the sphere and a box.

```
#include "colors.inc"
light_source { <5,2,-5> color White }
camera { location <0,1,-5> look_at <0,0,0> }
sphere { <0,0,0>,1 pigment {Yellow} }
box{<-3,3,3>,<-2,2,2> rotate <10,30,0> pigment
{Green}}
plane {<0,1,0>,-2 pigment {LightBlue}}
```

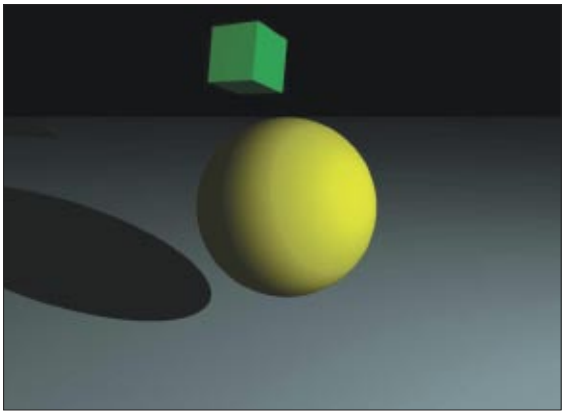

Fig. 2. An adaptation of Fig. 1, with a plane and box added. This figure was produced by the code in shown in Table II.

figure using a method called "ray tracing." Ray tracing is the computer technique companies such as Pixar[2] have used to create a flurry of animated movies released in recent years. Often a ray-traced figure is difficult to discriminate from an actual photograph. With ray tracing, one does not simply start pointing and clicking on circles and line endpoints to draw a figure. Instead, the desired scene is *defined*. A minimal scene definition includes three items. The first is the (*x,y,z*) coordinate of the object in the scene. The second is the (*x,y,z*) coordinate of the lighting source in the scene. The third is the (*x,y,z*) coordinate of the viewpoint (or camera) in the scene (additionally the direction in which the camera should "look" must also be specified). Once a scene has been defined, the ray-tracing process can begin.

Ray tracing a figure based on a scene's definition involves bouncing many thousands of computer-simulated light rays from the light source onto the objects in the scene. Where a light ray intersects an object, any law of physics for light/matter interaction may be applied in computing the characteristics (direction, intensity, color, polarization, etc.) of the outgoing ray. In the most basic form, the law of reflection and refraction is applied to calculate the next direction of the light ray. If a ray happens to enter the entrance aperture of the camera, a dot will be recorded where the ray intersects the camera's image plane (which eventually becomes the delivered image). The life of a ray must be terminated if it does not enter the camera after a predefined number of scene interactions.

Povray may be freely downloaded from http://www.povray.org. Windows, Macintosh, and Linux versions of the software are available. Once the software has been downloaded, it is either automatically installed or is installed by double-clicking on the file that is downloaded. The remainder of this article will provide a short tutorial on using Povray, ending with a discussion on illustrating a few common topics in elementary physics.

Using Povray

When using Povray, the scene is defined by typing a description of the scene into a text file that is then fed through the Povray program to commence the ray-tracing process. A basic Povray scene definition is shown in Table I. This scene contains a light source, camera, and a yellow sphere as the sole object. The line "#include colors.inc" tells Povray to include symbolic definitions for colors, so one may refer to colors using symbolic names like "red" for red, etc. In the following lines, notice the consistent structure of the code blocks, which consist of a name like "light_source," "camera," or "sphere," each followed by a balanced set of grouping symbols { and }. In this case, the light source is put at (*x,y,z*) location (2,0,-10) and illuminates the scene with white light. The camera (or viewpoint) is at position (0,1,-5) and is oriented to look at the point (0,0,0). A yellow sphere with a radius of 1 (arbitrary units) is placed at (0,0,0).[3] This completes an example of the most basic scene description possible: a light source, camera, and an object.

Table III. A point charge with 200 electric field lines emanating from it.

```
#include "colors.inc"
light_source { <10,0,-1> color White shadowless }
light_source { <0,0,-10> color White shadowless }

camera {location <2,2,-5> look_at <0,0,0> }
sphere { <0,0,0>,0.4 pigment {Red} }
#declare R1=seed(0);
#declare R=2.5;
#declare c=0;
#while(c <= 200)
   #declare theta=pi*rand(R1);
   #declare phi=2.0*pi*rand(R1);
   #declare xp=R*sin(theta)*cos(phi);
   #declare yp=R*sin(theta)*sin(phi);
   #declare zp=R*cos(theta);
   cylinder {<0,0,0>,<xp,yp,zp>,0.01 pigment {Yellow}}
   #declare c=c+1;
#end
```

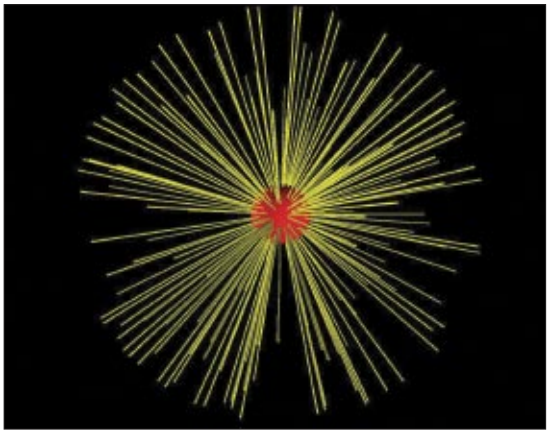

Fig. 3. A point charge with 200 electric field lines emanating from it. This figure was produced using the code shown in Table III.

When Povray is told to render this scene, the image shown in Fig. 1 is produced. Although the scene is simple, the visual effects that are produced by ray tracing are already apparent. Lighting and shadowing effects inherent to ray optics provides a perspective and visual quality that is virtually impossible to achieve using a palette-driven drawing program. If this simple figure were projected in a classroom using a computer and projection system, the audience's attention would be drawn in to the figure; it *looks very professional.*

Changing the light source position to (5,2,-5) and adding a plane and a box results in the scene definition shown in Table II. In the "plane" code block, a normal to the plane is defined as the unit vector <0,1,0> (the +*y* direction), and the plane is positioned along the normal at $y = -2$. Opposite corners of the box are given in the "box" code block, as well as a bit of rotation about the *x* (10 degrees) and *y* (30 degrees) axes. The rendered scene is shown in Fig. 2, where realistic lighting of the sphere and box is apparent, as is the sphere's shadow.

With this simple introduction to image creation using Povray, two examples from elementary physics are discussed: Gauss's law and projectile motion.

Povray and Gauss's Law

To begin, we'll return to the lone sphere in Table I and imagine it to be a point charge. The code in Table III adds some electric field lines emanating from it. Referring to it, two light sources are used to make the figure brighter. The camera and sphere (point charge) are placed as before. The "#declare" statements are used to define some variables. *R*1 is a list of pseudo-random numbers between 0 and 1 (inclusive) the program may draw from, with zero as a seed. *R* is the radius or length (in spherical coordinates) of the electric field lines, and *c* is a counter, initialized to zero. The "#while" and "#end" statements are a loop structure, similar to those found in other languages like C and Java. Theta and phi, the usual angles in spherical coordinates, are chosen at random, and from them, an (*x,y,z*) coordinate is calculated as *xp*, *yp*, and *zp*. To actually plot the electric field line, a very thin (0.01) radius cylinder is drawn, yellow in color from <0,0,0> to <*xp,yp,zp*>. The variable *c* is incremented and in this case the loop is repeated 200 more times. The eventual output is shown in Fig. 3.

Adding a Gaussian surface can be done by adding this line

```
sphere { <0,0,0>,1.5 pigment {Blue transmit 0.5} }
```

below the "#while–#end" block of the code in Table III. This will draw a blue sphere, centered at the origin, with a radius of 1.5. The color of the sphere is rendered such that half (0.5) of the light rays that interact with it get transmitted, allowing us to see inside of the Gaussian surface. The result is shown in Fig. 4. Changing the camera position to <2, 2, -1>, for instance, brings the view closer to the whole object, so one may discuss other issues dealing with Gauss's law, such as normals to the surface and the angle between the normal and the electric field lines.

Table IV. Povray code to show a ball being pushed off of a table.

```
#include "colors.inc"

light_source { <0,50,0> color White }
light_source { <10,2.5,0> color White }
light_source { <0,5,-50> color White }

camera { location <5,3,-10> look_at <1,3,0> }

#declare time=0.0;
#declare bx=0.0;
#declare by=5.0;
#declare vx=5;
#declare vy=0.0;
#declare g=9.8;

box { <-5,by,2>, <0,-2,-2> pigment {Blue} }
plane { <0,1,0>,0 pigment {Gray} }

#while(time <= clock)
        #declare dt=clock_delta;
        #declare bx=bx+vx*dt;
        #declare by=by+vy*dt-0.5*dt*dt;
        #declare vy=vy-g*dt;
        sphere { <bx,by,0>,0.2 pigment {Green} }
        #declare time=time+clock_delta;
#end
```

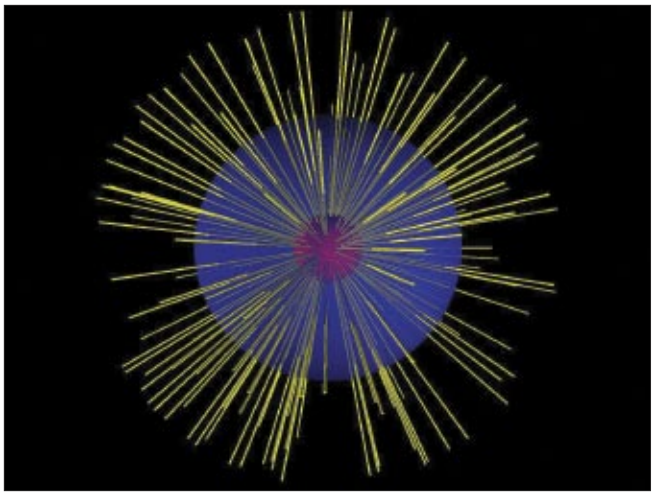

Fig. 4. Same as Fig. 3, except a semi-transparent Gaussian surface has been added (see text).

Illustrating Time-Dependent Phenomena with Povray

Illustrating time-dependent phenomena requires computer animation. This means that several images of the same scene are created where each successive frame is a version of the scene a bit later in time. When the resulting images are rapidly displayed, one after another, the animation[4] can be seen. Telling Povray to create such a series of images is straightforward. To start, Povray must be told how many separate images of the scene are to be created by defining the variable "Final_Frame" as an input parameter to the software. For example, telling Povray "Final_Frame=10" will tell it to create 10 separate images of your scene. On the Windows and Macintosh versions, there is a pull-down menu for doing this.[5] On the Linux version, this is passed in directly on the command line.

When Povray sees that the "Final_Frame" variable has been defined, it will run through the given scene definition text file the corresponding number of times. During each run through, two internal variables will be defined by Povray, "clock" and "clock_delta." "Clock" is a variable that will run between 0 and 1, while "clock_delta" will be defined as 1/Final_Frame and is the ampunt by which the variable "clock" will be incremented with each run through the scene definition. For example, if "Final_Frame = 10," then variable "clock" will be 0, .1, .2, .3, ... 1 during each successive run through the scene definition ("clock_delta" would be 0.1). The trick to performing animation with Povray is to make the positions of objects that are supposed to move a function of the "clock" variable. Povray will produce a separate output-image file for each value of the "clock" variable.

This is all illustrated with the example of a ball being pushed off a table with only a horizontal component of velocity. The Povray Code is shown in Table IV. The "#declare" block initializes variables, including the ball's initial position (*bx* and *by*) and components of velocity (*vx* and *vy*). A "#while" loop is then executed to run between variable time = 0 to the current value of "clock," integrating the equations of motion of the ball. This code produces successive image files, each containing a longer and longer trail of balls executing free-fall projectile motion. The final image produced is shown in Fig. 5. If these images were assembled into a movie (see Ref. 4), an audience could be shown a time-evolving projectile in flight. If the trail of balls is not wanted, moving the sphere statement from within the "#while–#end" block to outside (and after) it would show only a single (moving) sphere per frame. We have produced enhanced

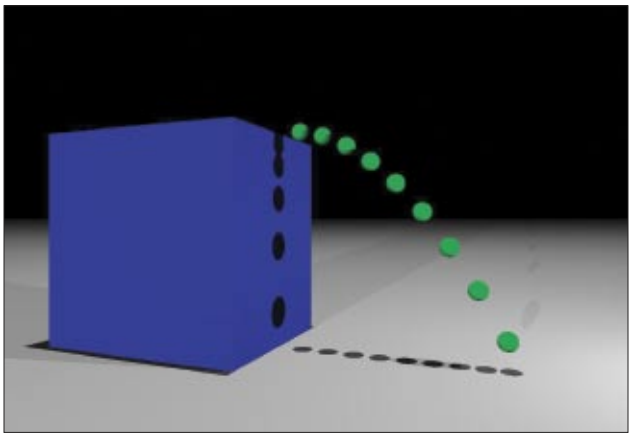

Fig. 5. A ball pushed from the edge of a table with only a horizontal component of velocity. It was produced using the code shown in Table IV.

versions of this that includes the time-evolving velocity vector and components on the ball itself. We have also run computer-lab sessions giving the students the basic code shown and having them modify and experiment with the initial parameters.

We also note two pedagogical features in Fig. 5 that ray tracing has provided. The shadows on the horizontal plane are equally spaced, indicating unaccelerated motion. The shadows on the vertical wall of the table are spaced further and further apart, indicating accelerated motion.

Conclusions

We have provided a brief tutorial on using Povray for illustrating ideas in basic physics. For each system we have developed over the years, we are constantly impressed with the stunning visual quality of the final result versus the small amount of time needed to produce it. Completed scenes are also useful time and time again. We have produced a Povray image or movie for a variety of situations in elementary physics teaching, and the reader is invited to view our collection at the web address given in Ref. 6.

The reader is also encouraged to read the documentation provided at the website http://www.povray.org. Povray is a very full-featured ray-tracing program with many options for giving objects texture, color, and indicies of refraction to name a few. There are also many options for different lighting sources and a whole array of primitive objects. As an example, when learning about "surfaces of revolution" back in college calculus, the "lathe" object would have been particularly useful. Finally, it is worth noting that there are other ways of producing figures with similar visual appeal as those produced by Povray, a few of which can be found in Ref. 7.

We would be interested in seeing your own Povray creations; please contact the author at tbensky@calpoly.edu.

References

1. R. Good, *Classical Electromagnetism,* 1st ed. (Saunders College Publishing, New York, 1999), p. vi.
2. Pixar Animation Studios, 1200 Park Ave., Emeryville, CA 94608 (http://www.pixar.com).
3. Povray's coordinate system is left-handed. This means (when looking at the computer screen) *+x* runs right, *-x* runs left, *+y* runs up, and *-y* runs down. *+z* runs into the page and perpendicular to it, while *-z* runs out of the page.
4. We highly recommend Quicktime Pro from Apple Computer Corp. (http://www.apple.com). It will import a series of images (in many possible formats) and assemble them into a variety of popular computer-movie formats.
5. For Windows: Pull down the "Render" menu, then select the "Edit Settings/Render" option. For Macintosh: With a Povray text file open, pull down the "Edit" menu, then select the "filename.pov settings" option. In both cases, you'll see a text box labeled "Command line" or "Command line options." Type "final_frame = 20" to, for example, render 20 separate frames of your scene.
6. See http://atom.physics.calpoly.edu/Graphics for more of our work.
7. See, for example, http://www.vpython.org, http://www.falstad.com/mathphysics.html, and http://physics.nad.ru/.