Raytracing revealed

Painting a scene with light rays is one of the most fascinating parts of graphics theory. Find out everything you need to know

Rendering views of imaginary worlds on a computer screen is an intriguing pursuit. There are many techniques used depending on the requirements of the output medium (games and movies, for example, generally use different techniques). The technique that produces the most lifelike results is raytracing, since it renders the images using realistic lighting and shadows, as well as taking into account reflections and refractions of the light.

Over the past couple of decades – perhaps even longer than that – computer graphics have become ever more prevalent in movies. It seems that not a week goes by without the release of another digitally enhanced production that redefines the art of placing computer graphics centre-stage on the big screen. In 2009, we were treated to films like *Up* from masters of digital animation Pixar, and *Avatar* from James Cameron. Both of them relied on computer-generated graphics like never before. I, like many, am fascinated with the 'behind the scenes' videos you can sometimes find online, or with the interviews with the producers and animators that attempt to scratch the surface of what's required to render the graphics for these movies.

It's not just movies, either. Modern computer games – admittedly in close collaboration with the GPU – continue to refine the art of smooth 3D graphics display. But before we discuss the techniques involved in this area, it's helpful to define the term '3D graphics' in this context. I'm not referring here to graphics that need cinema-friendly glasses, but instead to the rendering of a 3D scene on a 2D window: your screen. In other words, items that are behind or are cut off by closer items in the scene won't be seen on the screen.

Raytracing is one method for rendering a 3D scene on a flat screen, and in particular a scene with lots of complicated light interactions. Where is all this light coming from? Well, there might be several light sources in the scene; there might be mirrors and other reflective surfaces (not necessarily planar); and there may also be transparent or translucent objects. Ray tracing assumes that we can follow (that is, trace) individual light rays as they bounce around the scene, hence its name.

Which rays should we trace?

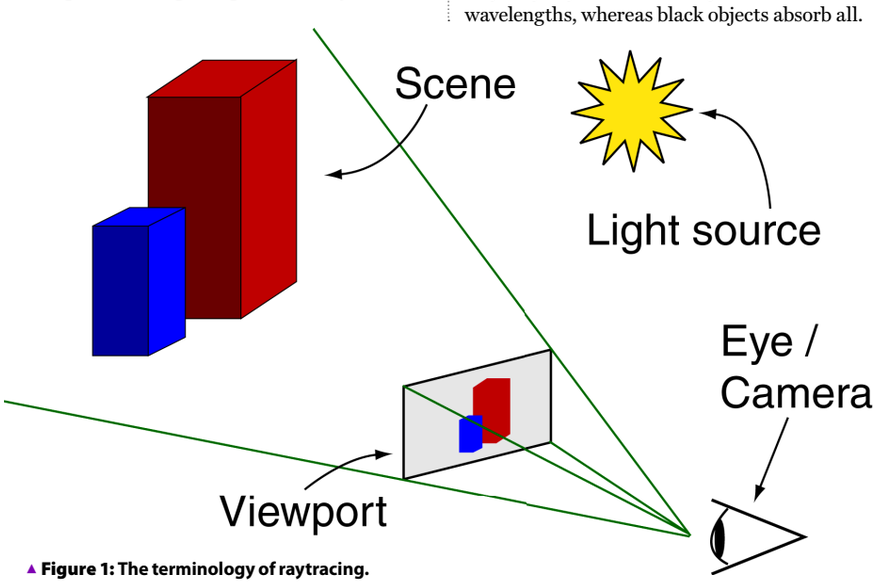
As with any kind of process that creates a computer-generated image, we must have a scene that we are trying to render. The scene will consist of a collection of objects – some simple shapes like spheres and cubes; some more complex and free-form, perhaps with textured surfaces. The objects will all be three-dimensional and have depth, as well as length and breadth. The renderer will assume a particular viewpoint looking at the scene, which we can call the 'eye' or the 'camera'. The camera will be looking at the scene through a viewport or window. The image we create will be the view of the scene at that window (Figure 1). This is a little different from how your eye or a real camera works, where the image is projected behind the lens.

In raytracing, we trace the rays of light from the scene and find out where they hit the ▸

viewport. If you think of the viewport as being a rectangle of size 1,024 x 600 pixels (the size of the screen on the netbook I'm using to type this) then our job is to work out the RGB colour value of each of the 600,000-odd pixels on the screen by tracing light rays from the scene towards the camera.

Actually, if you think about it, that's a pretty daunting task. Suppose we have a single light source behind the camera illuminating a reflective sphere hovering over an infinite chessboard (the classic raytraced image). Light rays from the source propagate in all directions. Those that shoot off behind the camera we'll never see. The only light we do see is, in effect, shooting out from the source in a cone. Much of the light in the scene bounces off the sphere away from the viewport. Trying to simulate all the light rays coming out of the light source would be computationally infeasible as well as a waste of time: the vast majority of them will hit parts of the scene that aren't visible from the camera or will be reflected elsewhere.

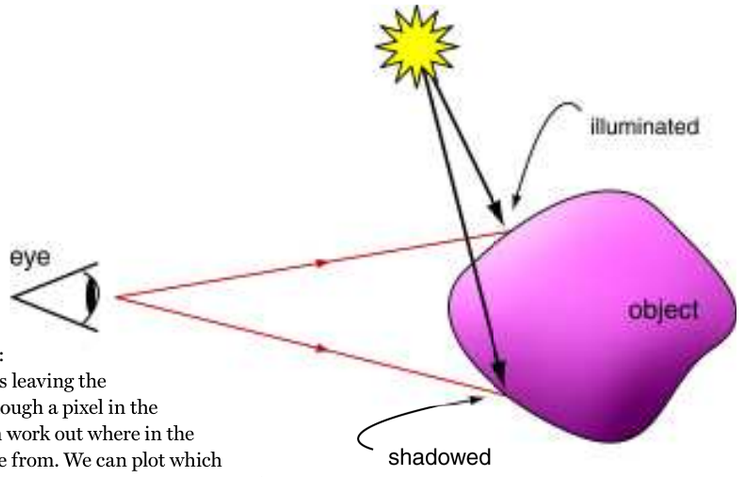
To improve our computational performance, we should, in effect, only simulate and trace those light rays that are guaranteed to hit the viewport at the right angle to actually enter the camera. But how do we know which ones those are? The answer is to look at the problem in reverse: simulate the rays as leaving the camera, passing though a pixel in the viewport, and then work out where in the scene that ray came from. We can plot which objects it hits and what the interactions are, maybe following that ray back even further in the scene. The colour of the pixel will be some combination of the objects the ray bounced off.



▲ **Figure 1: The terminology of raytracing.**

Bouncing light rays

Let's get a little more precise about all this 'bouncing'. Look at the wall wherever you are. It's a solid object and you can't see through it. It has a colour, too. Mine's a light yellow. Why do I see it as yellow? Because the sun is shining on it (we assume the sun puts out light of all wavelengths) and the wall reflects light of that particular wavelength or combination of wavelengths, and absorbs all the other ones. So, yellow objects reflect yellow light; that's why they're yellow. White objects reflect all wavelengths, whereas black objects absorb all.



▲ **Figure 2: When shadows occur with raytracing.**

So, you might ask, why are white objects white and not a perfect mirror, which of course reflects all light too? The answer is that white objects scatter the light due to imperfections in the surface of the object, whereas mirrored surfaces reflect the light evenly. We can make a simplifying assumption to begin with: that only mirrored surfaces reflect light from somewhere else in the scene (as well as from the light sources) and other objects merely 'emit' light of a particular colour – provided that they're being illuminated, of course.

We'll leave reflections for a moment and concentrate on the colour of opaque objects, since this is a good point to discuss shadows and shading. Raytracing has to take into account both of these in order to make the scene more realistic.

Shadows are fairly simple. A shadow is cast if a ray of light from the source cannot reach the area in question. Figure 2 shows two rays from the eye hitting an object. The top ray hits the object at a point from which a line can be drawn to the light source without hindrance. The bottom ray hits the object at a point from which a straight line cannot be drawn to the light source: the object itself gets in the way. The bottom point is then in shadow.

Shading, on the other hand, is more subtle. The way to think of shading is as the angle that rays from the source hit the surface of the object. If the angle is 0°, the light source is directly 'above' the object, and the point gets the full effect of the illumination. If the angle is greater than 0° – say, 45° – the point is tilted away from the light and thus doesn't get the full illumination. As the angle increases towards 90°, the brightness decreases, until at 90° exactly the rays of light just graze the surface and give no illumination.

Modelling shadows

We can model this effect pretty realistically by using the cosine function. When the light is perpendicular to the surface, the angle is 0, and the cosine of it is 1.0. As the angle increases, the cosine of the angle decreases. This is initially a fairly slow transition, but there's a rapid drop off as the angle increases past 60 degrees. When the angle is 90 degrees, the cosine is 0.0. This method of allowing for shading is known, unsurprisingly, as cosine shading.

Although this all seems very rigorous and mathematical and accurate, it actually produces some images with very harsh black shadows, as if they were produced on the moon.

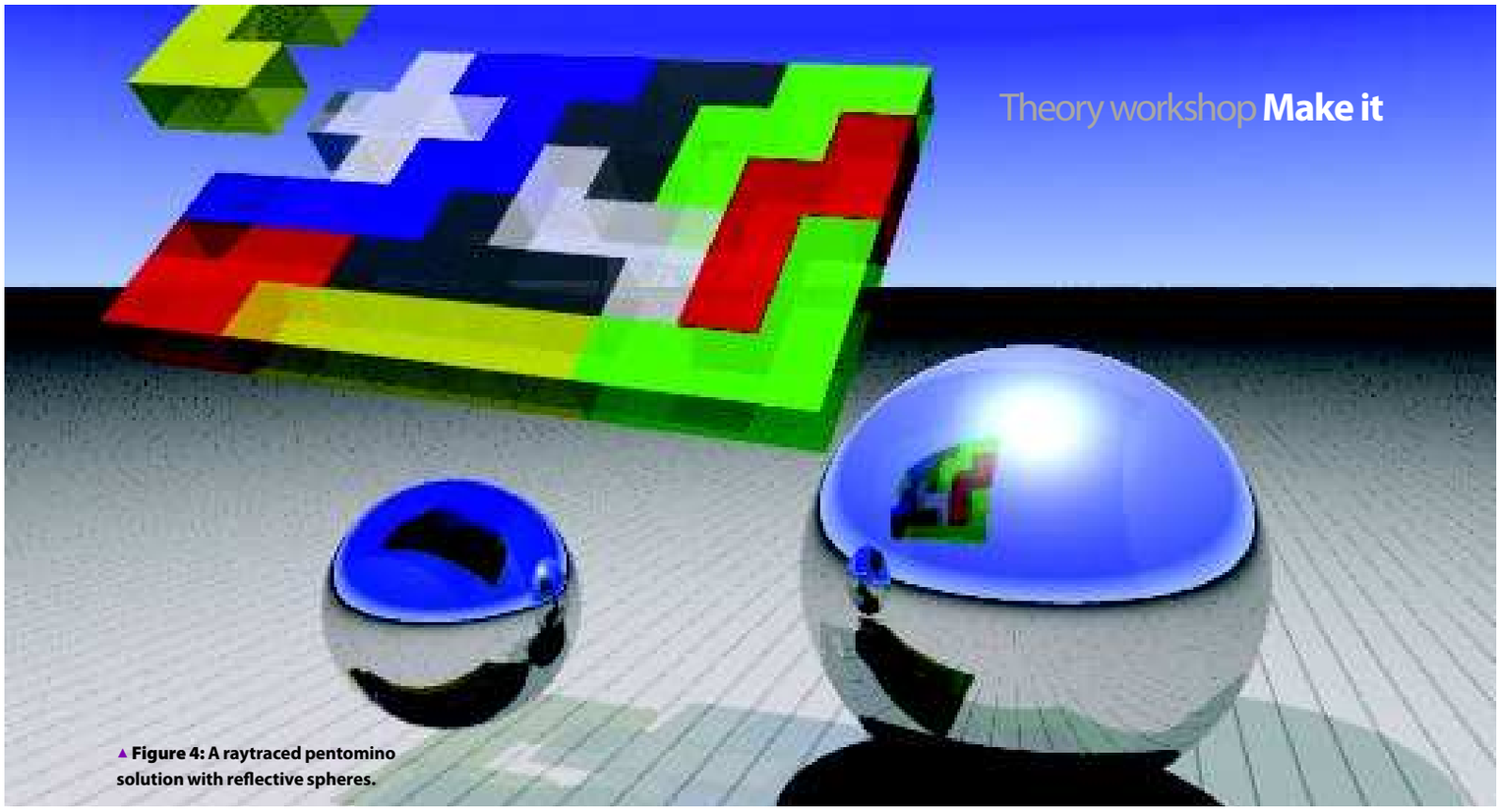
Spotlight on… POV-Ray

POV-Ray (Persistence of Vision Raytracer) is an open-source, free raytracing program. It is quite extraordinarily powerful and produces some incredible images, some of which can take several minutes (or even hours) to render.

POV-Ray comprises two parts: a complete programming language to describe a scene, the light sources, the camera position and the various solid objects that form part of the scene (as ways to merge or join them); and the renderer that takes a description written in this language to produce the final ray-traced image. It's a great way to explore what's possible with raytracing.

I played around with POV-Ray to see if I could produce an interesting image. The result is Figure 4, where I show a solution to the 6 x 10 pentomino puzzle we discussed last issue, built with translucent pieces hovering in the air – the 'U' about to be placed – and then two hovering shiny spheres nearby reflecting the solution (and each other). Notice the shadow on the ground from the main light source is coloured, since the pieces are slightly transparent.

You can download POV-Ray from **www.povray.org** or find it on your SuperDisc. You can see the source for this image at **www.tinyurl.com/RayPentomino**.



▲ **Figure 4: A raytraced pentomino solution with reflective spheres.**

The reason for this is that, in the real world, air helps to scatter light, making it softer. There are other effects at work too: light scattered from coloured objects does illuminate other surfaces, despite our simplifying assumption. As a result, raytracing programs tend to use a given value for ambient light or background illumination. In this way shadows are never plain black, but are actually illuminated by a general 'glow', allowing you to see some detail in the shadows. The ambient light is usually defined by a small positive value, say 0.1 or 0.2, and the light for shading (diffuse light) is defined as 1.0 minus the ambient light value.

The colour value, then, for a point on a coloured object is the actual colour multiplied by the sum of the ambient light and cosine shading using the diffuse light.

Getting reflections right

Let's now talk about reflection. I'm sure you remember the rules for reflection from school: the reflected ray has the same angle from the perpendicular (the normal vector, if you want to be precise) as does the incoming ray. And that's it – pretty easy (Figure 3, part 1).

Refraction is a little more complicated, but not by much. Here we assume that the incident ray hits the boundary of an object that has some transparency (for example, glass or water). Because light has a different speed in different materials, rays will change direction slightly (Figure 3, part 2). We've all seen this effect when looking at a straw in a glass of water: it appears that the straw is bent at the surface of the water.

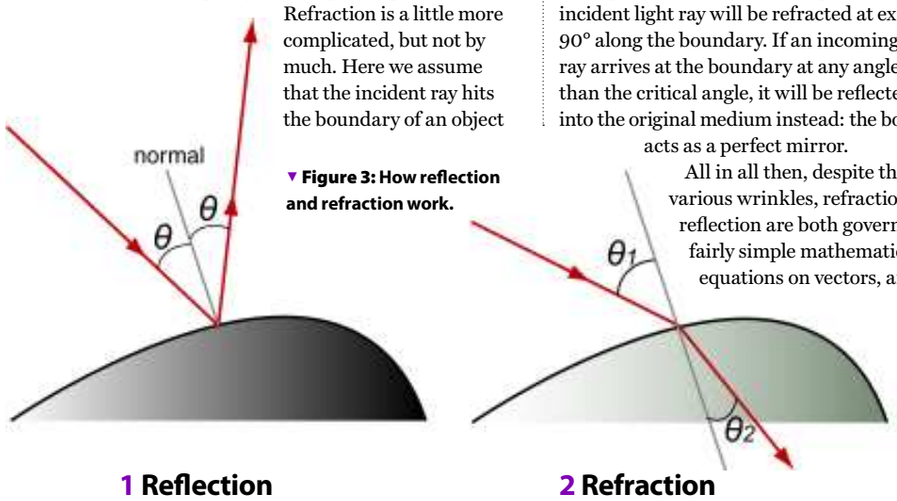
The amount of bending is a function of the refractive indices of the two materials in contact. Air, for example, has a refractive index of about 1.0003, whereas water has a refractive index of 1.33. Snell's Law states that the ratio of the sines of the angle of incidence and refraction is equal to the reciprocal of the ratio of the respective refractive indices. What this means in practice is that light is bent towards the perpendicular if the ray is travelling from the material with the lower refractive index to that with the higher. The situation is true in reverse when light travels from the higher to the lower refractive index.

That brings up another wrinkle to refraction that also must be taken into account. When light travels from, say, water to air, it bends away from the perpendicular. There will be an angle, called the critical angle, at which the incident light ray will be refracted at exactly 90° along the boundary. If an incoming light ray arrives at the boundary at any angle greater than the critical angle, it will be reflected back into the original medium instead: the boundary acts as a perfect mirror.

All in all then, despite the various wrinkles, refraction and reflection are both governed by fairly simple mathematical equations on vectors, and we model the light rays by vectors. Converting vector maths to code is quite straightforward.

Of course, there's something more to light and transparent objects that makes it even more interesting from a photorealistic viewpoint and also harder to compute the path a light ray travels (and hence the colour of the pixel in the viewport). The issue is that a light ray hitting a transparent object that also has a certain amount of reflectivity will be split. Part of the light ray will be reflected (and we know how to deal with that) and part of the light ray will be refracted into the object (and we know how to deal with that). In other words, our computations will be doubled for each of these types of objects. If the two parts of that split ray go on to meet other partially-reflective, partially refractive objects, the rays will get split again. Getting this effect right gives raytraced pictures the extraordinary photo-realism we see (and also the long rendering times). PCP

▼ **Figure 3: How reflection and refraction work.**



1 Reflection **2** Refraction

Mapping textures

Instead of viewing the surface of an object such as a sphere as being all one colour (except as modified by the shading algorithm), we could instead map a texture onto the object. The texture could be something like stone, wood, marble, or water ripples – any semi-random look that evokes a type of material. It could be a single image that's repeated across the surface of the object or it could be a randomised algorithm that mimics the texture desired. The shading algorithm works in exactly the same way, but the colour of the point intersecting the light ray will be determined by the texture itself and how it's mapped onto the object. Modern GPUs have special texture rendering hardware that performs this kind of mapping extremely quickly. ■

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pcplus.co.uk