Image-Based Lighting

Paul Debevec
USC Institute for Creative Technologies

This tutorial shows how image-based lighting can illuminate synthetic objects with measurements of real light, making objects appear as if they're actually in a real-world scene.

Image-based lighting (IBL) is the process of illuminating scenes and objects (real or synthetic) with images of light from the real world. It evolved from the reflection-mapping technique[1,2] in which we use panoramic images as texture maps on computer graphics models to show shiny objects reflecting real and synthetic environments. IBL is analogous to image-based modeling, in which we derive a 3D scene's geometric structure from images, and to image-based rendering, in which we produce the rendered appearance of a scene from its appearance in images. When used effectively, IBL can produce realistic rendered appearances of objects and can be an effective tool for integrating computer graphics objects into real scenes.

The basic steps in IBL are

1. capturing real-world illumination as an omnidirectional, high dynamic range image;
2. mapping the illumination onto a representation of the environment;
3. placing the 3D object inside the environment; and
4. simulating the light from the environment illuminating the computer graphics object.

1 A microscope, modeled by Gary Butcher in 3D Studio Max, rendered using Marcos Fajardo's Arnold rendering system as illuminated by light captured in a kitchen.

Figure 1 shows an example of an object illuminated entirely using IBL. Gary Butcher created the models in 3D Studio Max, and the renderer used was the Arnold global illumination system written by Marcos Fajardo. I captured the light in a kitchen so it includes light from a ceiling fixture; the blue sky from the windows; and the indirect light from the room's walls, ceiling, and cabinets. Gary mapped the light from this room onto a large sphere and placed the model of the microscope on the table in the middle of the sphere. Then, he used Arnold to simulate the object's appearance as illuminated by the light coming from the sphere of incident illumination.

In theory, the image in Figure 1 should look about how a real microscope would appear in that environment. It simulates not just the direct illumination from the ceiling light and windows but also the indirect illumination from the rest of the room's surfaces. The reflections in the smooth curved bottles reveal the kitchen's appearance, and the shadows on the table reveal the colors and spread of the area light sources. The objects also successfully reflect each other, owing to the ray-tracing-based global-illumination techniques we used.

This tutorial gives a basic IBL example using the freely available Radiance global illumination renderer to illuminate a simple scene with several different lighting environments.

Capturing light

The first step in IBL is obtaining a measurement of real-world illumination, also called a light probe image.[3] The easiest way to do this is to download one. There are several available in the Light Probe Image Gallery at http://www.debevec.org/Probes. The Web site includes the kitchen environment Gary used to render the microscope as well as lighting captured in various other interior and outdoor environments. Figure 2 shows a few of these environments.

Light probe images are photographically acquired images of the real world, with two important properties. First, they're omnidirectional—for every direction in the world, there's a pixel in the image that corresponds to that direction. Second, their pixel values are

0272-1716/02/$17.00 © 2002 IEEE

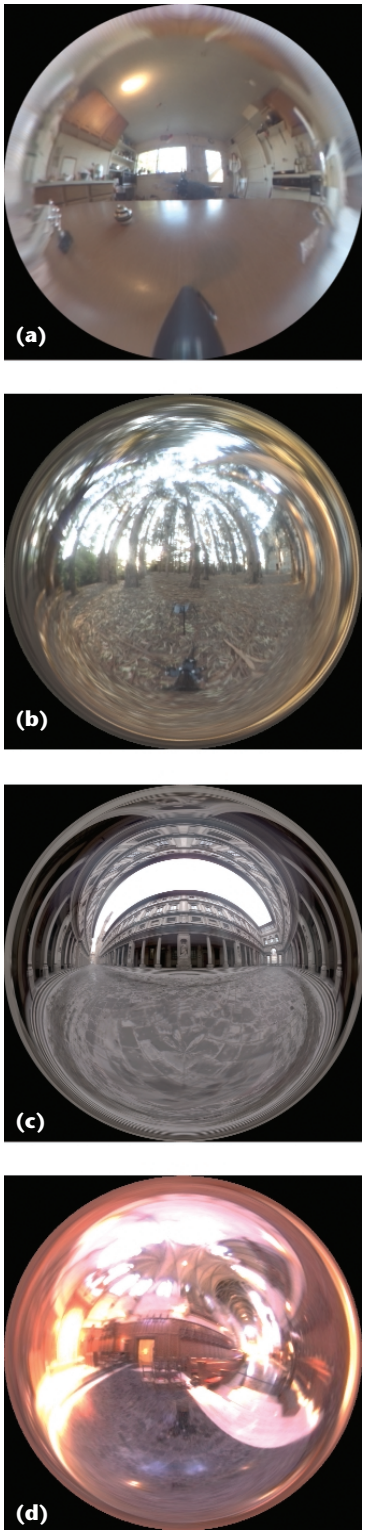
linearly proportional to the amount of light in the real world. In the rest of this section, we'll see how to take images satisfying both of these properties.

We can take omnidirectional images in a number of ways. The simplest way is to use a regular camera to take a photograph of a mirrored ball placed in a scene. A mirrored ball actually reflects the entire world around it, not just the hemisphere toward the camera. Light rays reflecting off the outer circumference of the ball glance toward the camera from the back half of the environment. Another method of obtaining omnidirectional images using a regular camera is to shoot a mosaic of many pictures looking in different directions and combine them using an image stitching program such as RealViz's Stitcher. A good way to cover a particularly large area in each shot is to use a fisheye lens,[4] which lets us cover the full field in as few as two images. A final technique is to use a special scanning panoramic camera (such as the ones Panoscan and Sphereon make), which uses a vertical row of image sensors on a rotating camera head to scan across a 360-degree field of view.

In most digital images, pixel values aren't proportional to the light levels in the scene. Usually, light levels are encoded nonlinearly so they appear either more correctly or more pleasingly on nonlinear display devices such as cathode ray tubes. Furthermore, standard digital images typically represent only a small fraction of the dynamic range—the ratio between the dimmest and brightest regions accurately represented—present in most real-world lighting environments. When part of a scene is too bright, the pixels saturate to their maximum value (usually 255) no matter how bright they really are.

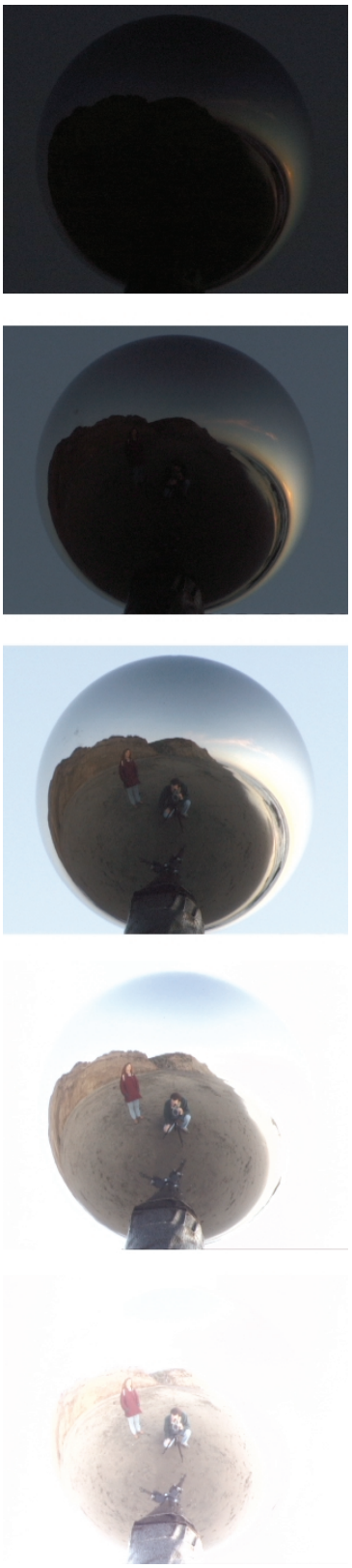
We can ensure that the pixel values in the omnidirectional images are truly proportional to quantities of light using high dynamic range (HDR) photography techniques.[5] The process typically involves taking a series of pictures of the scene with varying exposure levels and then using these images to solve for the imaging system's response curve and to form a linear-response composite image covering the entire range of illumination values in the scene. Software for assembling images in this way includes the command-line mkhdr program at http://www.debevec.org/Research/HDR and the Windows-based HDR Shop program at http://www.debevec.org/HDRShop.

HDR images typically use a single-precision floating-point number for red, green, and blue, allowing the full range of light from thousandths to billions to be represented. We can store HDR image data in a various file formats, including the floating-point version of the TIFF file format or the Portable Floatmap variant of Jef Postsanzer's Portable Pixmap format. Several other representations that use less storage are available, including Greg Ward's Red-Green-Blue Exponent (RGBE) format[6] (which uses one byte each for red, green, blue and a common 8-bit exponent) and his new 24-bit and 32-bit LogLuv formats recently included in the TIFF standard. The light probe images in the light probe image gallery are in the RGBE format, which lets us easily use them in Ward's Radiance global illumination renderer. (We'll see how to do precisely that in the next section.)



2 Several light probe images from the Light Probe Image Gallery at http://www.debevec.org/Probes. The light is from (a) a residential kitchen, (b) the eucalyptus grove at UC Berkeley, (c) the Uffizi gallery in Florence, Italy, and (d) Grace Cathedral in San Francisco.

Figure 3 (next page) shows a series of images used in creating a light probe image. To acquire these images, we placed a three-inch polished ball bearing on top of a tripod at Funston Beach near San Francisco and used a digital camera with a telephoto zoom lens to take a series of exposures of the ball. Being careful not to disturb the camera, we took pictures at shutter speeds ranging from 1/4 second to 1/10000 second, allowing



3 A series of differently exposed images of a mirrored ball photographed at Funston Beach near San Francisco. I merged the exposures, ranging in shutter speed from 1/4 second to 1/1000 second, into a high dynamic range image so we can use it as an IBL environment.

the camera to properly image everything from the dark cliffs to the bright sky and the setting sun. We assembled the images using code similar to that now found in mkhdr and HDR Shop, yielding a high dynamic range, linear-response image.

Illuminating synthetic objects with real light

IBL is now supported by several commercial renderers, including LightWave 3D, Entropy, and Blender. For this tutorial, we'll use the freely downloadable Radiance lighting simulation package written by Greg Ward at Lawrence Berkeley Laboratories. Radiance is a Unix package, which means that to use it you'll need to use a computer running Linux or an SGI or Sun workstation. In this tutorial, we'll show how to perform IBL to illuminate synthetic objects in Radiance in just seven steps.

1. Download and install Radiance

First, test to see if you already have Radiance installed by typing `which rpict` at a Unix command prompt. If the shell returns "Command not found," you'll need to install Radiance. To do this, visit the Radiance Web site at http://radsite.lbl.gov/radiance and click on the download option. As of this writing, the current version is 3.1.8, and it's precompiled for SGI and Sun workstations. For other operating systems, such as Linux, you can download the source files and then compile the executable programs using the `makeall` script. Once installed, make sure that the Radiance binary directory is in your `$PATH` and that your `$RAYPATH` environment variable includes the Radiance library directory. Your system administrator should be able to help you if you're not familiar with installing software packages on Unix.

2. Create the scene

The first thing we'll do is create a Radiance scene file. Radiance scene files contain the specifications for your scene's geometry, reflectance properties, and lighting. We'll create a simple scene with a few spheres sitting on a platform. First, let's specify the material properties we'll use for the spheres. Create a new directory and then call up your favorite text editor to type in the following material specifications to the file scene.rad:

```
# Materials

void plastic red_plastic
0
0
5 .7 .1 .1 .06 .1

void metal steel
0
0
5 0.6 0.62 0.68 1 0

void metal gold
0
0
5 0.75 0.55 0.25 0.85 0.2

void plastic white_matte
0
0
5 .8 .8 .8 0 0
```

```
rview -vtv -vp 8 2.5 -1.5 -vd -8 -2.5 1.5 -vu 0 1 0 -vh 60 -vv 40
```

4 Use your text editor to create the file camera.vp with the camera parameters as the file's first and only line.

```
void dielectric crystal
0
0
5 .5 .5 .5 1.5 0

void plastic black_matte
0
0
5 .02 .02 .02 .00 .00

void plastic gray_plastic
0
0
5 0.25 0.25 0.25 0.06 0.0
```

These lines specify the diffuse and specular characteristics of the materials we'll use in our scene, including crystal, steel, and red plastic. In the case of the red plastic, the diffuse RGB color is (.7, .1, .1), the proportion of light reflected specularly is .06, and the specular roughness is .1. The two zeros and the five on the second through fourth lines are there to tell Radiance how many alphanumeric, integer, and floating-point parameters to expect.

Now let's add some objects with these material properties to our scene. The objects we'll choose will be some spheres sitting on a pedestal. Add the following lines to the end of scene.rad:

```
# Objects

red_plastic sphere ball0
0
0
4 2 0.5 2 0.5

steel sphere ball1
0
0
4 2 0.5 -2 0.5

gold sphere ball2
0
0
4 -2 0.5 -2 0.5

white_matte sphere ball3
0
0
4 -2 0.5 2 0.5

crystal sphere ball4
0
0
4 0 1 0 1

!genworm black_matte twist \
 "cos(2*PI*t)*(1+0.1*cos(30*PI*t))" \
 "0.06+0.1+0.1*sin(30*PI*t)" \
 "sin(2*PI*t)*(1+0.1*cos(30*PI*t))" \
 "0.06" 200 | xform -s 1.1 -t 2 0 2 \
 -a 4 -ry 90 -i 1

!genbox gray_plastic pedestal_top 8 \
 0.5 8 -r 0.08 | xform -t -4 -0.5 \
 -4
!genbox gray_plastic pedestal_shaft \
 6 16 6 | xform -t -3 -16.5 -3
```

These lines specify five spheres made from various materials sitting in an arrangement on the pedestal. The first sphere, **ball0**, is made of the **red_plastic** material and located in the scene at (2,0.5,2) with a radius of 0.5. The pedestal itself is composed of two beveled boxes made with the Radiance **genbox** generator program. In addition, we invoke the **genworm** program to create some curly iron rings around the spheres. (You can leave the **genworm** line out if you want to skip some typing; also, the backslashes indicate line continuations which you can omit if you type everything on one line.)

3. Add a traditional light source

Next, let's add a traditional light source to the scene to get our first illuminated glimpse —without IBL—of what the scene looks like. Add the following lines to scene.rad to specify a traditional light source:

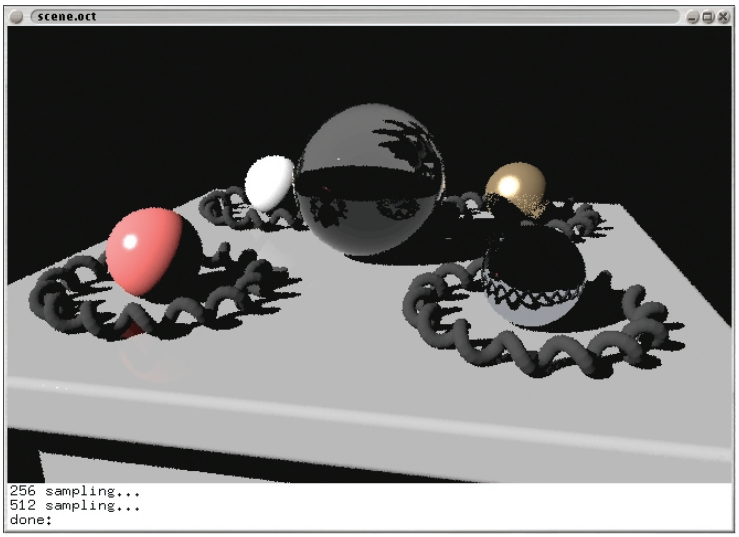
```
# Traditional Light Source

void light lightcolor
0
0
3 10000 10000 10000

lightcolor source lightsource
0
0
4 1 1 1 2
```

4. Render the scene with traditional lighting

In this step, we'll create an image of the scene. First, we need to use the **oconv** program to process the scene file into an octree file for Radiance to render. Type the following command at the Unix command prompt:

```
# oconv scene.rad > scene.oct
```

The **#** indicates the prompt, so you don't need to type it. This will create an octree file scene.oct that can be rendered in Radiance's interactive renderer **rview**. Next, we need to specify a camera position. This can be done as command arguments to **rview**, but to make things



5 The Radiance rview interactive renderer viewing the scene as illuminated by a traditional light source.

simpler, let's store our camera parameters in a file. Use your text editor to create the file camera.vp with the camera parameters as the file's first and only line (see Figure 4). In the file, this should be typed as a single line.

These parameters specify a perspective camera (**-vtv**) with a given viewing position (**-vp**), direction (**-vd**), and up vector (**-vu**) and with horizontal (**-vh**) and vertical (**-vv**) fields of view of 60 and 40 degrees, respectively. (The **rview** text at the beginning of the line is a standard placeholder in Radiance camera files, not an invocation of the **rview** executable.)

Now let's render the scene in **rview**. Type:

```
# rview -vf camera.vp scene.oct
```

In a few seconds, you should get an image window similar to the one in Figure 5. The image shows the spheres on the platform, surrounded by the curly rings, and illuminated by the traditional light source. The image might or might not be pleasing, but it certainly looks computer generated. Now let's see if we can make it more realistic by lighting the scene with IBL.

5. Download a light probe image

Visit the Light Probe Image Gallery at http://www.debevec.org/Probes and choose a light probe image to download. The light probe images without concentrated light sources tend to produce good-quality renders more quickly, so I'd recommend starting with the beach, uffizi, or kitchen probes. Here we'll choose the beach probe for the first example. Download the beach_probe.hdr file by shift-clicking or right-clicking "Save Target As..." or "Save Link As..." and then view it using the Radiance image viewer **ximage**:

```
# ximage beach_probe.hdr
```

If the probe downloaded properly, a window should pop up displaying the beach probe image. While the window is up, you can click and drag the mouse pointer over a region of the image and then press "=" to re-expose the image to properly display the region of interest. If the image didn't download properly, try downloading and expanding the all_probes.zip or all_probes.tar.gz archive from the same Web page, which will download all the light probe images and preserve their binary format. When you're done examining the light probe image, press the "q" key in the **ximage** window to dismiss the window.

6. Map the light probe image onto the environment

Let's now add the light probe image to our scene by mapping it onto an environment surrounding our objects. First, we need to create a new file that will specify the mathematical formula for mapping the light probe image onto the environment. Use your text editor to create the file angmap.cal with the following content (the text between the curly braces is a comment that you can skip typing if you wish):

```
{
angmap.cal

Convert from directions in the world \
 (Dx, Dy, Dz) into (u,v) \
 coordinates on the light probe \
 image

-z is forward (outer edge of sphere)
+z is backward (center of sphere)
+y is up (toward top of sphere)
}

d = sqrt(Dx*Dx + Dy*Dy);

r = if(d, 0.159154943*acos(Dz)/d,0);

u = 0.5 + Dx * r;
v = 0.5 + Dy * r;
```

This file will tell Radiance how to map direction vectors in the world (Dx, Dy, Dz) into light probe image coordinates (u, v). Fortunately, Radiance accepts these coordinates in the range of zero to one (for square images) no matter the image size, making it easy to try out light probe images of different resolutions. The formula converts from the angular map version of the light probe images in the light probe image gallery, which differs from the mapping a mirrored ball produces. If you need to convert a mirrored-ball image to this format, HDR Shop has a Panoramic Transformations function for this purpose.

Next, comment out (by adding **#**'s at the beginning of the lines) the traditional light source in scene.rad that we added in step 3:

```
#lightcolor source lightsource
#0
#0
#4 1 1 1 2
```

Note that these aren't new lines to add to the file but lines to modify from what you've already entered. Now, add the following to the end of scene.rad to include the IBL environment:

```
# Image-Based Lighting Environment

void colorpict hdr_probe_image
7 red green blue beach_probe.hdr
 angmap.cal u v
0
0

hdr_probe_image glow light_probe
0
0
4 1 1 1 0

light_probe source ibl_environment
0
0
4 0 1 0 360
```

The **colorpict** sequence indicates the name of the light probe image and the calculations file to use to map directions to image coordinates. The **glow** sequence specifies a material property comprising the light probe image treated as an emissive glow. Finally, the **source** specifies the geometry of an infinite sphere mapped with the emissive glow of the light probe. When Radiance's rays hit this surface, their illumination contribution will be taken to be the light specified for the corresponding direction in the light probe image.

Finally, because we changed the scene file, we need to update the octree file. Run **oconv** once more to do this:

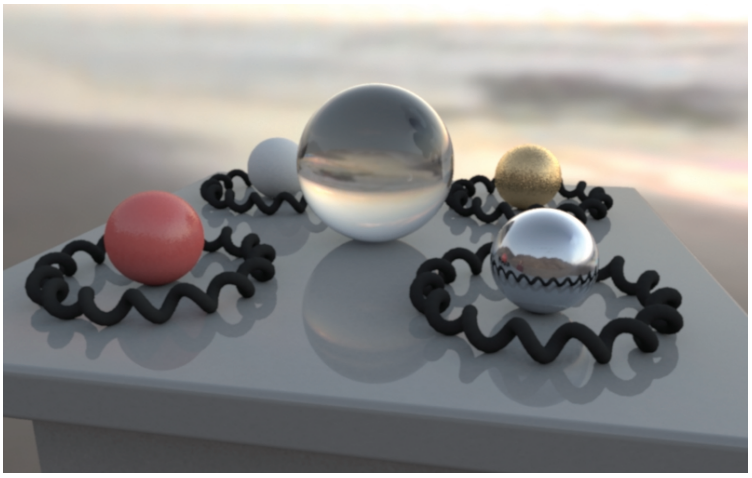
```
# oconv scene.rad > scene.oct
```

7. Render the scene with IBL

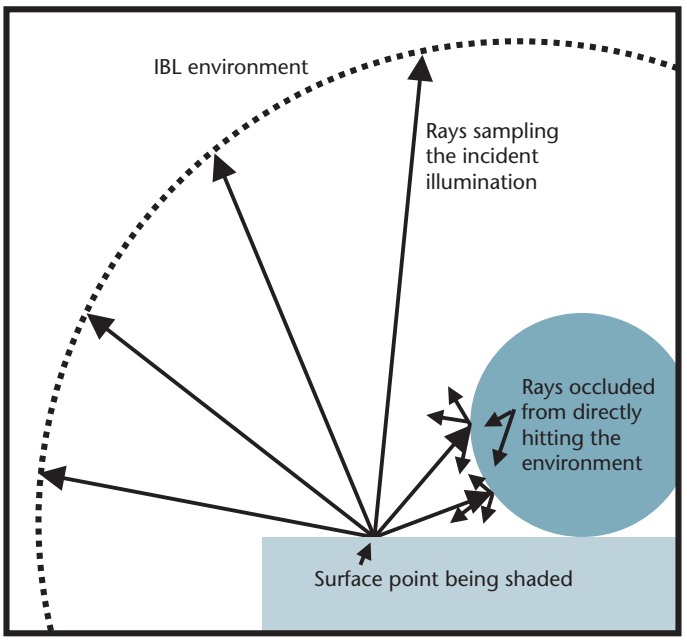
Let's now render the scene using IBL. Enter the following command to bring up a rendering in **rview**:

```
# rview -ab 1 -ar 5000 -aa 0.08 -ad \
 128 -as 64 -st 0 -sj 1 -lw 0 -lr \
 8 -vf camera.vp scene.oct
```

Again, you can omit the backslashes if you type the whole command as one line. In a few moments, you should see the image in Figure 6 begin to take shape. Radiance is tracing rays from the camera into the scene, as Figure 7 illustrates. When a ray hits the environment, it takes as its pixel value the corresponding value in the light probe image. When a ray hits a particular point on an object, Radiance calculates the color and intensity of the incident illumination (also known as irradiance) on that point by sending out a multitude of rays (in this case 192 of them) in random directions to quantify the light arriving at that point on the object. Some of these rays will hit the environment, and others will hit other parts of the object, causing Radiance to recurse into computing the light coming from this new part of the object. After Radiance computes the illumination on the object point, it calculates the light reflected toward the camera based on the object's material properties and this becomes the pixel value of that point of the object. Images calculated in this way can take a while to render, but they produce a faithful rendition of how the captured illumination would illuminate the objects.



6 The spheres on the pedestal illuminated by the beach light probe image from Figure 3.



7 How Radiance traces rays to determine the incident illumination on a surface from an IBL environment.

The command-line arguments to **rview** tell Radiance how to perform the lighting calculations. The **-ab 1** indicates that Radiance should produce only one ambient-bounce recursion in computing the object's illumination—more accurate simulations could be produced with a value of 2 or higher. The **-ar** and **-aa** set the resolution and accuracy of the surface illumination calculations, and the **-ad** and **-as** set the number of rays traced out from a surface point to compute its illumina-

8 The objects illuminated by the kitchen, eucalyptus grove, Uffizi Gallery, and Grace Cathedral light probe images in Figure 2.

tion. The `-st`, `-sj`, `-lw`, and `-lr` specify how the rays should be traced for glossy and shiny reflections. For more information on these and more Radiance parameters, see the reference guide on the Radiance Web site.

When the render completes, you should see an image of the objects as illuminated by the beach lighting environment. The synthetic steel ball reflects the environment and the other objects directly. The glass ball both reflects and refracts the environment, and the diffuse white ball shows subtle shading, which is lighter toward the sunset and darkest where the ball contacts the pedestal. The rough specular reflections in the red and gold balls appear somewhat speckled in this medium-resolution rendering; the reason is that Radiance sends out just one ray for each specular sample (regardless of surface roughness) rather than the much greater number it sends out to compute the diffuse illumination. Rendering at a higher resolution and filtering the image down can alleviate this effect.

We might want to create a particularly high-quality rendering using the command-line `rpict` renderer, which outputs the rendered image to a file. Run the following `rpict` command:

```
# rpict -x 800 -y 800 -t 30 -ab 1 - \
 ar 5000 -aa 0.08 -ad 128 -as 64 - \
 st 0 -sj 1 -lw 0 -lr 8 -vf \
 camera.vp scene.oct > render.hdr
```

The command-line arguments to `rpict` are identical to `rview` except that one also specifies the maximum x and y resolutions for the image (here, 800×800 pixels) as well as how often to report back on the rendering progress (here, every 30 seconds.) On an 800-MHz computer, this should take approximately 10 minutes. When it completes, we can only view the rendered output image with the ximage program. To produce high-quality renderings, you can increase the x and y resolutions to high numbers, such as $3{,}000 \times 3{,}000$ pixels and then filter the image down to produce an antialiased rendering. We can perform this filtering down by using either Radiance's `pfilt` command or the HDR Shop. To filter a $3{,}000 \times 3{,}000$ pixel image down to $1{,}000 \times 1{,}000$ pixels using `pfilt`, enter:

```
# pfilt -1 -x /3 -y /3 -r 1 \
 render.hdr > filtered.hdr
```

I used this method for the high-quality renderings in this article. To render the scene with different lighting environments, download a new probe image, change the beach_probe.hdr reference in the scene.rad file, and call `rview` or `rpict` once again. Light probe images with concentrated light sources such as grace and stpeters will require increasing the `-ad` and `-as` sampling parameters to the renderer to avoid mottled renderings. Figure 8 shows renderings of the objects illuminated by the light probes in Figure 2. Each rendering shows different effects of the lighting, from the particularly soft shadows under the spheres in the overcast Uffizi environment to the focused pools of light from the stained glass windows under the glass ball in the Grace Cathedral environment.

Advanced IBL

This tutorial has shown how to illuminate synthetic objects with measurements of real light, which can help the objects appear as if they're actually in a real-world scene. We can also use the technique to light large-scale environments with captured illumination from real-world skies. Figure 9 shows a computer

9 A computer model of the ruins of the Parthenon as illuminated just after sunset by a sky captured in Marina del Rey, California. Modeled by Brian Emerson and Yikuong Chen and rendered using the Arnold global illumination system.

10 A rendering from the Siggraph 99 film *Fiat Lux*, which combined image-based modeling, rendering, and lighting to place monoliths and spheres into a photorealistic reconstruction of St. Peter's Basilica.

model of a virtual environment of the Parthenon illuminated by a real-world sky captured with high dynamic range photography.

We can use extensions of the basic IBL technique in this article to model illumination emanating from a geometric model of the environment rather than from an infinite sphere of illumination and to have the objects cast shadows and appear in reflections in the environment. We used these techniques[3] to render various animated synthetic objects into an image-based model of St. Peter's Basilica for the Siggraph 99 film *Fiat Lux*, (see Figure 10). (You can view the full animation at http://www.debevec.org/FiatLux/.)

Some more recent work[7] has shown how to use IBL to illuminate real-world objects with captured illumination. The key to doing this is to acquire a large set of images of the object as illuminated by all possible lighting directions. Then, by taking linear combinations of the color channels of these images, images can be produced showing the objects under arbitrary colors and intensities of illumination coming simultaneously from all possible directions. By choosing the colors and intensities of the incident illumination to correspond to those in a light probe image, we can show the objects as they would be illuminated by the captured lighting environment, with no need to model the objects' geometry or reflectance properties. Figure 11 shows a collection of real objects illuminated by two of the light probe images from Figure 2. In these renderings, we used the additional image-based technique of environment matting[8] to compute high-resolution refractions and reflections of the background image through the objects.

11 Real objects illuminated by the Eucalyptus grove and Grace Cathedral lighting environments from Figure 2.

Conclusion

IBL lets us integrate computer-generated models into real-world environments according to the principles of global illumination. It requires a few special practices for us to apply it, including taking omnidirectional photographs, recording images in high dynamic range, and including measurements of incident illumination as sources of illumination in computer-generated scenes. After some experimentation and consulting the Radiance reference manual, you should be able to adapt these examples to your own scenes and applications. With a mirrored ball and a digital camera, you should be able to acquire your own lighting environments as well. For more information, please explore the course notes for the Siggraph 2001 IBL course at http://www.debevec.org/IBL2001. Source files and more image-based lighting examples are available at http://www.debevec.org/CGAIBL. ■

References

1. J.F. Blinn, "Texture and Reflection in Computer Generated Images," *Comm. ACM*, vol. 19, no. 10, Oct. 1976, pp. 542-547.
2. G.S. Miller and C.R. Hoffman, "Illumination and Reflection Maps: Simulated Objects in Simulated and Real Environments," *Proc. Siggraph 84*, Course Notes for Advanced Computer Graphics Animation, ACM Press, New York, 1984.
3. P. Debevec, "Rendering Synthetic Objects Into Real Scenes: Bridging Traditional and Image-Based Graphics with Global Illumination and High Dynamic Range Photography," *Computer Graphics* (Proc. Siggraph 98), ACM Press, New York, 1998, pp. 189-198.
4. N. Greene, "Environment Mapping and Other Applications of World Projections," *IEEE Computer Graphics and Applications*, vol. 6, no. 11, Nov. 1986, pp. 21-29.
5. P.E. Debevec and J. Malik, "Recovering High Dynamic Range Radiance Maps from Photographs," *Computer Graphics* (Proc. Siggraph 97), ACM Press, New York, 1997, pp. 369-378.
6. G. Ward, "Real Pixels," *Graphics Gems II*, J. Arvo, ed., Academic Press, Boston, 1991, pp. 80-83.
7. P. Debevec et. al, "Acquiring the Reflectance Field of a Human Face," *Computer Graphics* (Proc. Siggraph 2000), ACM Press, New York, 2000, pp. 145-156.
8. D.E. Zongker et. al, "Environment Matting and Compositing," *Computer Graphics* (Proc. Siggraph 99), ACM Press, New York, 1999, pp. 205-214.

Paul Debevec *is an executive producer at the University of Southern California's Institute for Creative Technologies, where he directs research in virtual actors, virtual environments, and applying computer graphics to creative projects. For the past five years, he has worked on techniques for capturing real-world illumination and illuminating synthetic objects with real light, facilitating the realistic integration of real and computer-generated imagery. He has a BS in math and a BSE in computer engineering from the University of Michigan and a PhD in computer science from University of California, Berkeley. In August 2001, he received the Significant New Researcher Award from ACM Siggraph for his innovative work in the image-based modeling and rendering field. He is a member of ACM Siggraph and the Visual Effects Society, and is a program cochair for the 2002 Eurographics Workshop on Rendering.*

Readers may contact Paul Debevec at USC/ICT, 13274 Fiji Way, 5th Floor, Marina Del Rey, CA 90292, email paul@debevec.org.