Photorealistic Terrain Lighting in Real Time

The great outdoors: rolling hills, majestic mountains, sun-drenched plains. An increasing number of games are taking place in outdoor environments, but getting them to look like the view out your window (if you have a house with a nice view) or a scenic postcard (if you don't) is not easy. Outdoor scenes are very complex visually, which makes them hard to render realistically, especially at the high frame rates required for games.

Modern terrain engines (running on powerful graphics cards) are getting pretty good at handling the geometric complexity — all the triangles needed to render those ridges and ravines, erosion lines and canyons — but that just isn't enough. After all, when we look at something in real life, what we see is the light reflected from it. And outdoor scenes have complex lighting, which is a major contributor to the visual intricacy that we find so pleasing (and which makes that house with the nice view so expensive). Getting this right for a single time of day is hard enough, but what if the time of day changes in your game? It would be nice to capture those subtle shifts of light and shadow.

In this article, we present two different methods for achieving these effects in real time. One or the other may be a better match for your game, depending on how you construct the game environments.

NATY HOFFMAN | *In 1997, after seven years as a microprocessor architect at Intel (where he was the lead architect for the Pentium processor with MMX chip and involved in the MMX, SSE, and SSE 2 instruction set extension projects), Naty took the leap into the game industry and full-time software engineering. He has since been coding up a storm at Westwood Studios, where he has been working on EARTH AND BEYOND as a computer graphics and optimization specialist.*
KENNY MITCHELL | *Kenny is one of an increasing number of professionals entering game development with a strong academic background. His Ph.D. thesis, "3D Database Environments," introduced the use of real-time 3D graphics on consumer PCs for database visualization. He entered the game industry in 1997 at VIS Interactive plc, where he developed voxel and NURBS real-time rendering technologies. Kenny is director of 3D computer graphics software engineering at Westwood Studios, where his responsibilities include research and development of cutting-edge 3D graphics.*

Light, Radiance, and Irradiance

Light is electromagnetic energy which radiates through space in all directions — we need a specific physical quantity that describes the intensity of a single ray of light. Luckily, there is such a quantity, called radiance.

To understand how radiance is defined, let us look at a patch of surface (see Figure 1). This patch gives off light (either by glowing or reflection, it doesn't matter) from every point and in all directions, as represented by the red arrows. This light can be measured as a certain amount of energy emitted every second — in other words, as power, which is measured in watts. If we are interested in the light emitted from a specific point on the surface, we can't use power, since the power emitted from a zero-area point is zero. But we can use power per area (measured in watts per square meter), which is represented by the blue arrows. This varies from point to point on the surface. Finally, we are interested in the light emitted in a specific direction from that point. Power per area is useless for

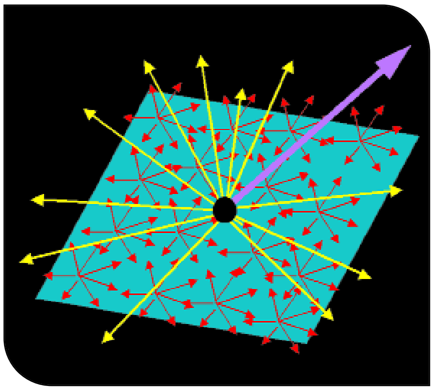
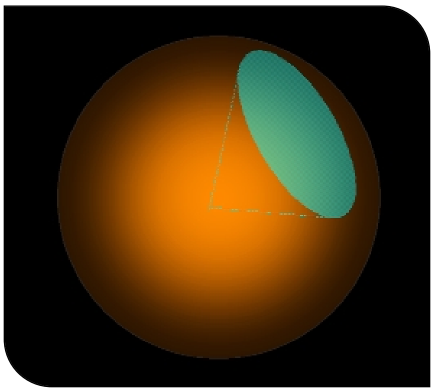
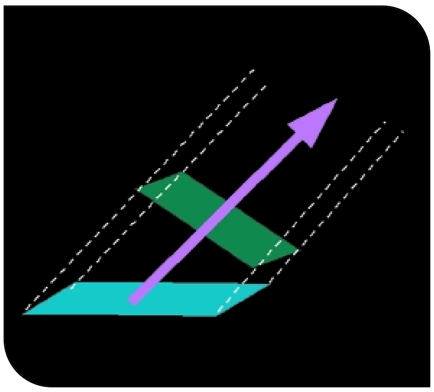




FIGURE 1 (left). Light radiating from a surface. FIGURE 2 (center). A solid angle, measured in steradians. FIGURE 3 (right). Projected area.

this, but we can use power per area per solid angle (solid angles, shown in Figure 2, are the 3D extension of angles and are measured in steradians), which is radiance, as represented by the purple arrow. To be specific, radiance is power per projected area per solid angle. The projected area is the area projected in the direction of the ray (see Figure 3), which is just the area times the dot product between the surface normal and the light ray. It is important to define radiance this way so that the intensity of the light ray is not dependent on the direction of the surface, or even whether there is a surface at all. The units of radiance are watts per meter squared per steradian.

Radiance is a spectral quantity; it can have a different value for each wavelength in the electromagnetic spectrum. For computer graphics we usually just look at three frequencies (red, green, and blue). In real life, radiance has a very large range — the radiance of the sun is more than a million times that of a dark shadow. However, in graphics we are usually limited to a small range, so we will pick a maximum radiance that we can handle, define it as 1.0, and scale everything accordingly. This scale factor is somewhat arbitrary and can vary from scene to scene.

Another important quantity for measuring light is irradiance. The irradiance at a point *p*, *E*(*p*), is simply the total of all the incoming radiance in all directions at *p*. This total is usually calculated via integration over all incoming directions in the hemisphere around the surface normal (see Figure 4). Irradiance is power per area (not projected area, since it's tied to a surface), and measured in watts per meter squared. Since radiance is defined using projected area, the integration needs to convert from one to the other, which gives us:

$$E(p) = \int_{\vec{v}\in H(p)} L_i(p,\vec{v})\vec{N}(p)\cdot\vec{v}\,d\Omega \qquad \text{Eq. 1}$$

where $\vec{N}(p)$ is the normal vector at *p*, *H*(*p*) is the hemisphere of outgoing unit vectors $\vec{v}$ centered on $\vec{N}(p)$, $L_i(p,\vec{v})$ is the incident radiance from direction $\vec{v}$ into point *p*, and *d*Ω is the infinitesimal solid angle used in integration.

Why is irradiance important? For purely diffuse, nonglowing surfaces (which we assume our terrains are), the outgoing radiance $L_o(p)$ from the point *p* is the same in all directions, and is equal to the irradiance at *p* times the surface color, C(*p*), divided by π:

$$L_o(p) = \frac{C(p)}{\pi}E(p) \qquad \text{Eq. 2}$$

So if we can find the irradiance, we can compute the radiance from it. Color, like radiance and irradiance, is measured separately for red, green, and blue. However, unlike those quantities, its scaling is not arbitrary. A diffuse color of 1.0 means that the surface reflects all the incoming energy that hits it, a diffuse color of 0.5 means that it absorbs half and reflects half, and so on. For nonglowing surfaces, the color can never be greater than 1.0.

Outdoor Lighting

Since the radiance of a surface point depends on its irradiance, let's look at the irradiance of a terrain point *p* (see Figure 5). We see several factors that contribute to the irradiance. The sun, which covers a small solid angle (it is about 0.5 degrees across)

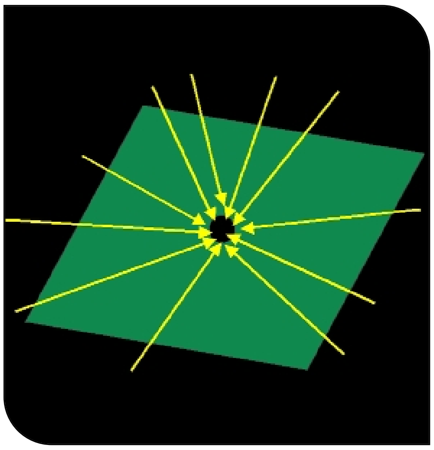
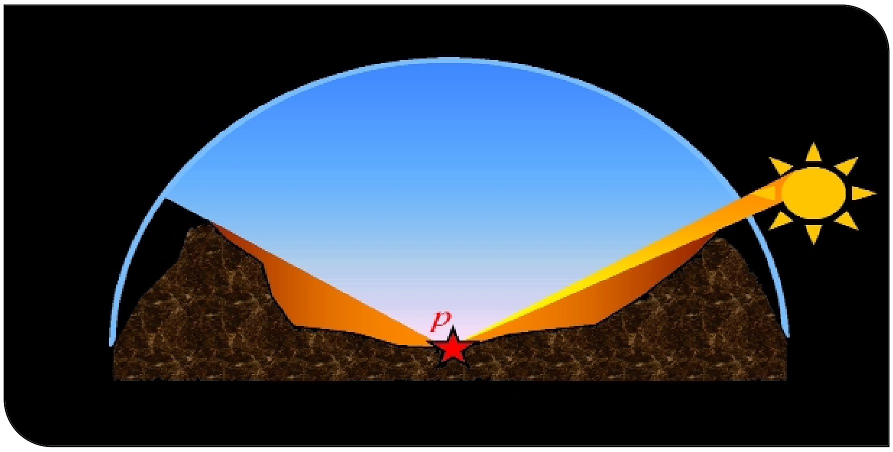



FIGURE 4 (left). Irradiance. FIGURE 5 (right). Outdoor irradiance.

but has a very high radiance, is the most important contributor. In this case, part of the sun's solid angle is hidden by other terrain points. Also important is the sky, which has a lower radiance than the sun but covers a much larger solid angle. This makes it very different from the lights we are used to in real-time graphics and will cause its contribution to look very different from that of a directional light. This contribution is most noticeable in shadows. The remaining incoming directions have interreflections, indirect light reflected from the sun and sky off other terrain points into p. This too is most noticeable in areas of shadow.

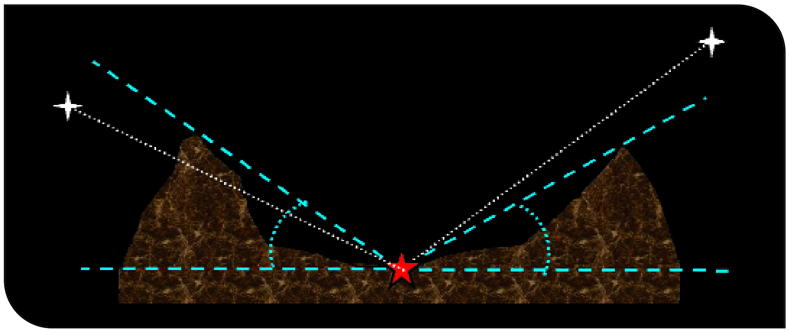
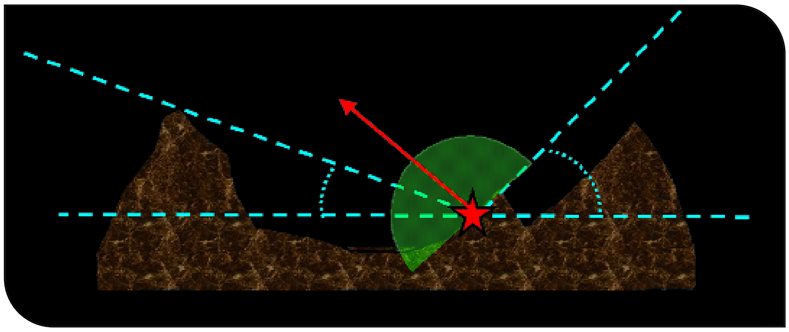



Figure 6 (top). Horizon angles. Figure 7 (bottom). Horizon angle clamping.

A Tale of Two Methods

At Westwood Studios, we ran into the need for realistic outdoor lighting in two different projects: EARTH AND BEYOND and PIRATES OF SKULL COVE. Since the two projects had different needs, we ended up with two very different systems. For both games we wanted high-quality terrain lighting without significantly impacting the run-time performance. However, for EARTH AND BEYOND it was important to have short preprocessing times, a low data footprint, and flexibility in changing the lighting parameters on the fly. For PIRATES OF SKULL COVE the paramount concern was achieving a very high quality overall lighting environment, including the terrain, clouds, and sky. These reasons led us to independently develop an analytical method for EARTH AND BEYOND and a video-based rendering method for PIRATES OF SKULL COVE. We will describe both methods in this article.

Analytical Method

Using this method, we calculate the terrain lighting dynamically as the lighting conditions change with the time of day. This requires us to solve Equations 1 and 2 for every terrain point. We are able to do this in real time by using several simplifying approximations and by doing as much offline precomputation as we can get away with.

We separate the lighting solution into two parts: sunlight and skylight. We calculate each one separately and add the two solutions to get the final lighting result. In our current implementation, we update a lightmap in software as the lighting changes. In the future, we are considering doing the lighting calculation completely in hardware.

Sunlight. Since the sun's solid angle is small, we treat it as a directional light source except for the possibility of partial occlusion (shadowing). Then the sun's direct contribution to the irradiance at each terrain point p, $E_{SunDirect}(p)$, is given by:

$$E_{SunDirect}(p) = O_{Sun}(p) L_{iSun} \vec{N}(p) \cdot \vec{v}_{Sun}$$

where $O_{Sun}(p)$ is the sun's occlusion factor at p (1 = completely visible, 0 = completely occluded), and $\vec{v}_{Sun}$ is the outgoing sunlight direction vector.

We calculate the dot product by using palette normal mapping. In a preprocessing step, we quantize all the terrain normals into a table of 256 normals and store a table index for each terrain point (a process very similar to palettizing a color texture). Then each time we recompute the lighting, we calculate $L_{iSun}\vec{N}(p) \cdot \vec{v}_{Sun}$ for each of the 256 normals. After this step, all that remains for each lightmap texel is to compute $O_{Sun}(p)$ and perform a lookup.

How do we calculate $O_{Sun}(p)$? Again, we use precomputation. We simplify the calculation by assuming that the sun travels in an arc directly over the X-axis. For each texel in the lightmap, we precompute and store horizon angles in the same plane as the sun's arc, thus creating a horizon map (first introduced by Max; see For More Information). The horizon angles (see Figure 6) are stored as 16-bit fixed-point numbers so that 0x0000 corresponds to 0 degrees and 0xFFFF corresponds to 90 degrees. To compute these horizon angles, we scan from each point along the X-axis in both directions, computing elevation angles and remembering the largest ones. We need to scan for a fairly large distance if we want mountains to be able to cast long shadows. This scanning is not expensive in terms of arithmetic calculation, but it is expensive in terms of memory accesses. For this reason it is important to store the height field with the rows along the X-axis so we can scan along rows, as this improves cache behavior.

As the sun's position and color changes, we calculate maximum and minimum angles each frame:

$$\theta_{Min} = \theta_{Center} - \frac{1}{2}\alpha \qquad \theta_{Max} = \theta_{Center} + \frac{1}{2}\alpha$$

where θ_{Center} is the elevation angle of the center of the sun's disk and α is the angular diameter of the sun (about 0.5 degrees, although to achieve pleasing soft-shadow effects, a larger quantity can be used — soft shadow edges also have a useful antialiasing effect on the low-resolution lightmap). $O_{Sun}(p)$ is then calculated by comparing θ_{Min} and θ_{Max} with the appropriate horizon angle ϕ

(measured from the horizontal):

$$\text{if}\begin{cases} \phi \le \theta_{Min} & O_{Sun}(p) = 1.0 \\ \phi \ge \theta_{Max} & O_{Sun}(p) = 0.0 \\ \theta_{Min} < \phi < \theta_{Max} & O_{Sun}(p) = \dfrac{\theta_{Max} - \phi}{\alpha} \end{cases}$$

Another way of looking at it is that $O_{Sun}(p)$ is equal to $(\theta_{Max} - \phi) / \alpha$, clamped between 0 and 1. This is simply the fraction of the sun's angle that is unoccluded by other terrain.

There is one important thing to make sure of when using horizon angles. Since irradiance is only measured over the hemisphere around the surface normal, the horizon angles need to be clamped to this hemisphere (see Figure 7). We didn't do this at first, and we had strange visual glitches appear from negative dot products and other peculiarities.

We will ignore the effect of interreflections from sunlight. This is a significant approximation, but it does not overly harm realism, because we will count interreflections from skylight, and the effects of interreflections are most noticeable where the sunlight contribution is small — in shadowed areas. In Figure 8, we can see the result of the sunlight calculation. Note that the shadows are perfectly dark. This is what you would expect to see in an airless environment such as the moon where the sky is pitch black and contributes no illumination even during the daytime. EARTH AND BEYOND (for which we developed this method) actually has such environments, so this is not totally useless, but usually this is not sufficient. Usually, one must also consider and calculate the contribution of skylight.

Skylight. The sky's color varies over its area as well as over time. For simplification, we can divide the sky up into a small number of patches based on elevation angle, on azimuth angle relative to the sun's arc, or both. A small number of patches can capture the sky's illumination nicely — in fact, for our first implementation we have treated the entire sky as one color.

Since the portion of sky "seen" by each terrain point does not change, all we need to do is to precompute the contribution of each sky patch to each terrain point. Namely, what would the illumination of this terrain point be if that patch of sky had a radiance of (1,1,1)? This can be stored in the form of a texture. Then during run time, all that needs to be done is to multiply each sky patch's texture with the current radiance of that patch, add the results together, and we're done.

Precalculating the contribution of direct skylight is fairly simple. Since the sky radiance is constant over each patch, we just solve the integral from Equation 1 to compute the irradiance. If we define the patch simply enough, the integral will have a nice analytical solution that we can calculate directly. But what about the interreflections? We skipped doing them for the sunlight; however, we would really like to have them for the skylight, since this will affect the realism of our shadows. The problem is that such interreflections are a global illumination problem, where the radiance of each point depends on those of many other points, so we need to perform a very slow iterative process (like a radiosity solve). If the terrains are static and you can afford long preprocessing times in your game, this is a very good solution. Just set up the appropriate patch of sky as an area light source in some tool such as Lightscape, and let it crunch away overnight for a solve. Repeat for each patch and you're done. However, for EARTH AND BEYOND we required much shorter preprocessing times.

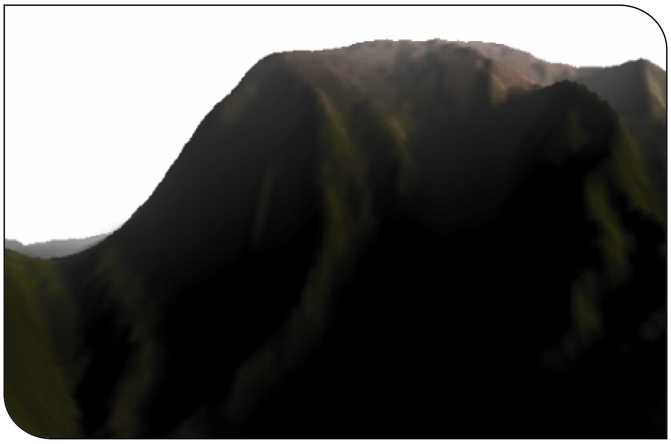


FIGURE 8. Terrain lit by sunlight only.

Can we solve this directly? If we separate $H(p)$ (the hemisphere of directions around the normal) into $D(p)$ (the subset of directions in which the sky is visible) and $H(p) - D(p)$ (the remaining directions, in which the sky is occluded by other terrain points) and use this separation to expand Equation 1, we get:

$$E_{Sky}(p) = \int_{\vec{v}\in D(p)} L_{iSky}(\vec{v})\vec{N}(p)\cdot\vec{v}\,d\Omega + \int_{\vec{v}\in H(p)-D(p)} L_{oSky}\left(x(p,\vec{v})\right)\vec{N}(p)\cdot\vec{v}\,d\Omega$$

where $x(p,\vec{v})$ is the terrain point visible from point p in direction $\vec{v}$.

The $L_{oSky}\left(x(p,\vec{v})\right)$ factor is dependent on the lighting solution for other terrain points, which is the problem. Luckily, we ran into a paper by Stewart and Langer (see For More Information) which gives a nice approximation for $L_{oSky}\left(x(p,\vec{v})\right)$ and also shows that under diffuse lighting conditions, such as skylight, this approximation introduces very small errors. The approximation is amazingly simple: just assume that $L_{oSky}\left(x(p,\vec{v})\right) = L_{oSky}(p,\vec{v})$ for all $\vec{v}$ in $H(p) - D(p)$. This means that the lighting on all terrain points visible from p is the same as the lighting of p itself. Why does this give good results? More details are available in Stewart and Langer's paper, but the basic idea is that for a surface such as a terrain under diffuse lighting, each point tends to "see" points which have lighting similar to itself. The terrain points visible from a point in a dark valley tend also to be in a dark valley, points visible from a bright mountain peak tend also to be bright mountain peaks, and so on.

Applying this approximation results in:

$$E_{Sky}(p) = \int_{\vec{v}\in D(p)} L_{iSky}(\vec{v})\vec{N}(p)\cdot\vec{v}\,d\Omega + \int_{\vec{v}\in H(p)-D(p)} L_{oSky}(p)\vec{N}(p)\cdot\vec{v}\,d\Omega$$

Applying Equation 2 gives us:

$$E_{Sky}(p) = \int_{\vec{v}\in D(p)} L_{iSky}(\vec{v})\vec{N}(p)\cdot\vec{v}\,d\Omega + \int_{\vec{v}\in H(p)-D(p)} \frac{C(p)}{\pi} E_{Sky}(p)\vec{N}(p)\cdot\vec{v}\,d\Omega \qquad \text{Eq. 3}$$

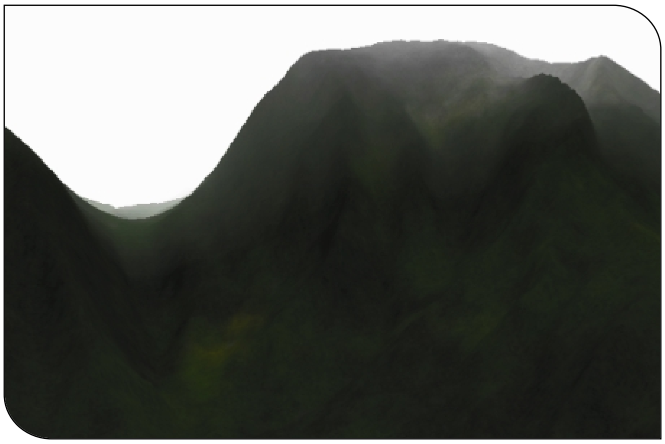
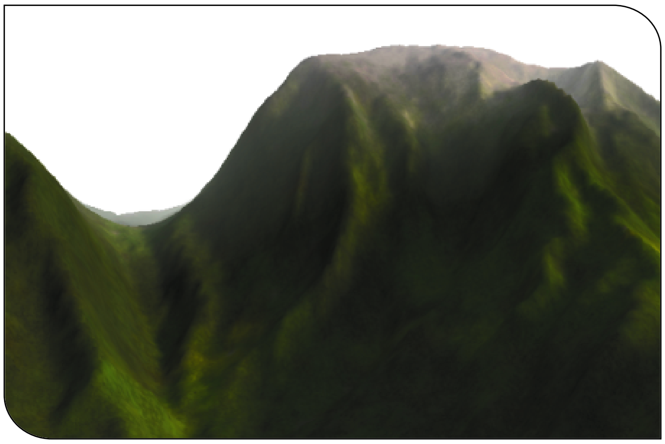



FIGURE 9 (left). Terrain lit by skylight only. FIGURE 10 (right). Terrain lit by both sunlight and skylight.

To simplify the derivation, we will assume for now that the sky radiance is constant over the entire sky. In this case, $L_{iSky}(\vec{v})$ becomes L_{iSky} and we can take it out of the integral. It is not difficult to extend the derivation for a sky that is divided into a number of patches, each with its own radiance, if the patches are parameterized carefully.

Note that:

$$\int_{\vec{v}\in H(p)-D(p)} \vec{N}(p)\cdot\vec{v}d\Omega = \int_{\vec{v}\in H(p)} \vec{N}(p)\cdot\vec{v}d\Omega - \int_{\vec{v}\in D(p)} \vec{N}(p)\cdot\vec{v}d\Omega = \pi - \int_{\vec{v}\in D(p)} \vec{N}(p)\cdot\vec{v}d\Omega$$

Applying this to Equation 3 and using some algebra gives us:

$$E_{Sky}(p) = \frac{L_{iSky}\int_{\vec{v}\in D(p)} \vec{N}(p)\cdot\vec{v}d\Omega}{1-C(p)\left(1-\frac{1}{\pi}\int_{\vec{v}\in D(p)} \vec{N}(p)\cdot\vec{v}d\Omega\right)}$$

This is a nice closed-form expression which takes interreflections as well as direct skylight into account. We can further simplify it by substituting:

$$I(p) = \int_{\vec{v}\in D(p)} \vec{N}(p)\cdot\vec{v}d\Omega$$

which gives us:

$$E_{Sky}(p) = \frac{L_{iSky}I(p)}{1-C(p)\left(1-\frac{1}{\pi}I(p)\right)}$$

Now we need to calculate $I(p)$ — Stewart and Langer also describe how to do this in their paper. This is based on dividing the sky into a number of sectors and using horizon angles (remember those?) to represent $D(p)$. We will use eight horizon angles in the eight compass directions. We already have two horizon angles for the sunlight shadows, so we need to compute just six more in the preprocessing stage. We will not store those extra angles, we will just use them to calculate the skylight texture. We do not need to scan very far for these angles to get good results, which is a good thing because now we need to scan along columns and diagonals of the height field, which is not the best memory access pattern.

Given these eight horizon angles ϕ_i (measured from the vertical this time), the equation for $I(p)$ is:

$$I(p) = \frac{1}{2}\vec{N}(p)\cdot\sum_{i=0}^{7}\left(\left(\phi_i - \frac{\sin 2\phi_i}{2}\right)\Delta\sin_i \quad \left(\varphi_i - \frac{\sin 2\phi_i}{2}\right)\Delta\cos_i \quad \frac{\pi}{4}\sin^2\phi_i\right)$$

Eq. 4

$$\Delta\sin_i = \sin\left(\frac{\pi}{4}(i+1)\right) - \sin\left(\frac{\pi}{4}i\right)$$

$$\Delta\cos_i = \cos\left(\frac{\pi}{4}i\right) - \cos\left(\frac{\pi}{4}(i+1)\right)$$

Note that the three expressions inside the sum in Equation 4 are the components of a 3-vector. The sum will produce a vector, then the dot product between this vector and $\vec{N}(p)$ is calculated and the result divided by two. Note also that $\Delta\sin_i$ and $\Delta\cos_i$ are constants and can be computed once and reused.

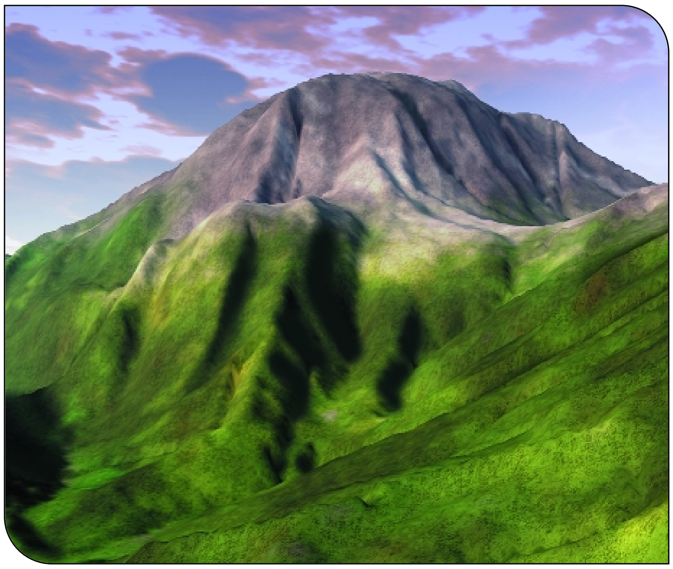
Since each vector being summed depends only on ϕ_i, we can precalculate this vector for each sector and for 256 values of ϕ_i, resulting in a 256×8 table. Note that better accuracy is achieved if we use the tangent of the angle for the table lookup (the tangent of the angle is calculated easily when computing horizon angles).

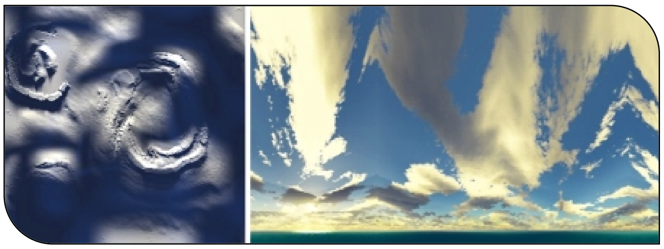
If we want more than one sky patch, we can assign different sectors to different patches to get two, four, or even eight patches (eight is overkill, though). In this case, a skylight texture needs to be computed and stored for each patch. You don't really need many patches to get good results — currently we are using one, and we intend to try two.

In Figure 9, we can see the result of the skylight calculation. The lighting is very different from the strong directional lighting in Figure 8 and is very good for an overcast day where the sun is completely hidden by clouds and the only light comes from the sky. In this case a gray value for sky radiance would produce good results.

Summary. In the general case, we add the two solutions together, giving the result seen in Figure 10. We get strong sunlight, soft-edged shadows, and subtle variations of light and dark in the shadowed regions resulting from skylight. A time-lapse movie of a day/night cycle using this technique is also available on the *Game Developer* web site at www.gdmag.com.

The equations from the previous sections enable us to calculate the irradiance. To get a lighting (radiance) solution, we need to multiply by the diffuse color and divide by π. One possible way





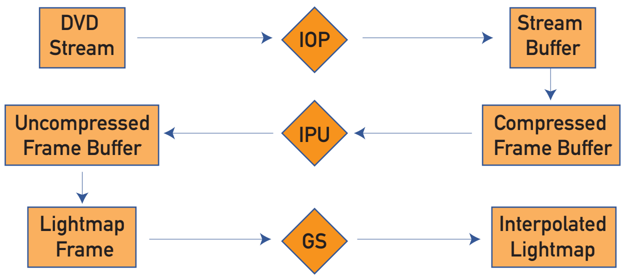


FIGURE 11 (left). Scene from EARTH AND BEYOND showing the analytical terrain lighting. FIGURE 12 (top right). Lightmap + skybox — a single frame from a video-based illumination map (VBIM) sequence. FIGURE 13 (bottom right). PS2 video-streaming implementation.

of doing this is to calculate the irradiance divided by π (the division can be put into the various tables so it adds almost no extra cost) and then store the result into a lightmap. This lightmap can be modulated with a color texture by using multi-texture or multi-pass methods. Our terrain engine uses a diffuse color texture that is the same resolution as the lightmap, combined with a repeating detail texture to add noise.

This requires two passes, unless you have a card with three or more texture units. We ended up using a slightly different solution — since we calculate the lightmap in software anyway, we multiply it with the diffuse color texture in software. Then we can draw the terrain with a single pass (light * color map in one texture stage and detail texture in the other).

This makes the lightmap calculation a little more expensive, but we don't care much. The reason is that the lighting changes slowly over time, so we don't need to recalculate the lightmap that often. If we amortize the lightmap calculation over many frames, the performance hit is low. We do this by running the lightmap calculation in a separate thread, but it is possible to do it without multi-threading. We do plan eventually to optimize the lightmap calculations (the inner loop is a very good fit for MMX or the new 128-bit MMX instructions in the Pentium 4), but at the moment this is on the back burner due to the low performance impact.

The precalculation is fast (about six seconds for a 512×512 lightmap on a 450MHz Pentium III) and doesn't require excessive storage (two 16-bit horizon angles and a 24-bit RGB value for each lightmap texel, for a total of 1.75MB for a 512×512 lightmap).

Future directions. Currently, we calculate and upload lightmaps in software. It is tempting to use hardware to generate the lightmaps instead, using texture-blending techniques (such as Direct3D 8 pixel shaders) and multiple passes. The skylight contribution is fairly simple. Each patch's factors can be stored as an RGB texture, and we can just render each texture (modulated with the current patch color) additively to add them up. The sunlight contribution is a little more complicated. Dot-product blending can do the diffuse lighting, and if we drop soft shadows then a simple alpha test can handle the shadowing. If we want soft shadows, we need to perform a subtraction, multiply by a constant, and clamp per pixel — this should be doable in a pixel shader, and we plan to investigate this possibility.

Local cloud shadows can be simulated by having an additional texture, projected from above, which modulates the sunlight contribution. It would be nice to be able to simulate the sunlight interreflections. Polynomial texture maps (see Malzbender, Gelb, and Wolters under For More Information) appear very promising for achieving this. This addition should increase lighting realism even more.

Video-Based Rendering Method

Think for a minute how long it would take to compute a truly photorealistic terrain rendering, with every detail and nuance represented faithfully. Hours? Days? Accurately simulating photons of light as they interact with particles in the atmosphere without loss of detail could take . . . (insert long duration joke here).

One promising way to approach this level of realism is through the use of image-based rendering techniques (see Debevec in For More Information). In the case of terrain lighting, all the calculations may be either captured from real photographs or computed offline with sophisticated terrain-rendering tools. The results can then be stored as illumination maps and applied as textures in real time with no CPU processing cost.

Figure 12 shows a single frame from a video sequence applied as a single lightmap texture pass to the terrain. In this color image we can see the effects of self-shadowing, cloud shadows, sunlight, skylight, atmospheric blue effects, atmospheric scattering, and haze. All the calculations for these effects are precomputed in the game's asset creation process from raw height-field data using a high-quality terrain-rendering application, Terragen. In addition to streaming lightmaps, a matching sky dome video texture is streamed.

Animating sequences of illumination maps presents us with considerable storage and bandwidth challenges. For example, a typical day/night cycle of 1,000 frames at 512×512×32-bit resolution would require 1GB of data. This replaces the task of calculating sophisticated lighting models in real time with the task of performing efficient playback of streaming compressed video

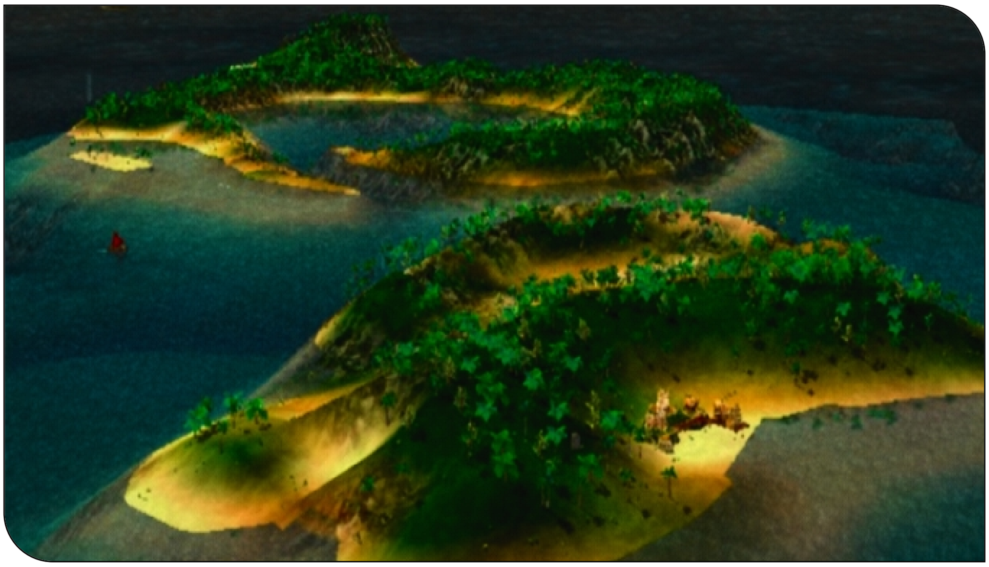


FIGURE 14. Video-based illumination maps in the PS2 game PIRATES OF SKULL COVE.

onto textures in 3D. Let's proceed with how this can be achieved on current hardware.

Video-streaming hardware implementation on Playstation 2. For our upcoming PS2 game, PIRATES OF SKULL COVE, we have the luxury of an image processing unit (IPU), which is a processor dedicated to accelerating the decompression of video data. Importantly, this relieves the CPU entirely of the burden of terrain-lighting calculations, freeing it up to concentrate on game simulation.

Figure 13 depicts the data flow of a single frame of the animation from a compressed video stream stored on DVD to the final rendered lightmap.

Concurrently with the application, compressed frames are streamed into the stream buffer in system memory, using the IO processor (IOP), in much the same way as sound or DVD video is transferred. Once a frame has been fully loaded into the stream buffer, it is copied into the compressed frame buffer, and the next frame begins to load immediately. Compression is necessary to reduce the data rate to within the required limits of DVD media.

The first decompression stage occurs in the IPU, where it processes each frame to produce an uncompressed frame in system memory suitable for upload to video memory. Blending the new frame with the previous one provides the second stage of decompression. This frame interpolation method occurs through the use of the Graphics Synthesizer (GS) and is performed in place by using a frame buffer motion-blur technique to reduce VRAM use (see "Real-Time Full Scene Anti-Aliasing for PCs and Consoles" under For More Information). This second decompression stage is important for a number of reasons:

- It acts as a form of temporal antialiasing between compressed frames, which reduces the number of full frames required for smooth animation.
- The interpolation of frames avoids sudden changes when the looping video jumps to the beginning of the sequence.
- If DVD streaming is held up for any reason, the frame interpolation process will continue unhindered. When the next frame is finally loaded, the same interpolation process will produce smooth "catch up" frames and resume the video sequence as normal.

We can see the results of the PS2 implementation for PIRATES OF SKULL COVE in Figure 14.

Video-streaming implementations on PC. On the PC, we have an implementation which decompresses the video stream using publicly available software codecs through the DirectShow API. In addition to the performance and quality trade-offs of the various software codecs, the main consideration here is to perform decompression into a system memory buffer and perform double-buffered uploads to video memory to avoid stalls. Two frames from this implementation can be seen in Figure 15, and a time-lapse video is also available on the *Game Developer* web site at www.gdmag.com.

Hardware-accelerated playback of compressed videos also exists on current PC graphics cards. However, issues with exposure in APIs and hardware conflicts with 3D acceleration are currently blocking an attractive low-bandwidth solution where video data is decompressed directly in video memory. Another possibility is uploading compressed textures. This will reduce bandwidth to the card but not as much as a video stream would.

Illumination map generation vs. real image capture. Ideally, we would capture video-based illumination maps from real video camera footage. One idea for capturing the light field of a real terrain is to place light sensors at regular intervals in a grid at ground level. Another is to extract this information from geostationary satellite image data. Realistically, the logistical problems of setting up these situations make the real image capture method entirely impractical

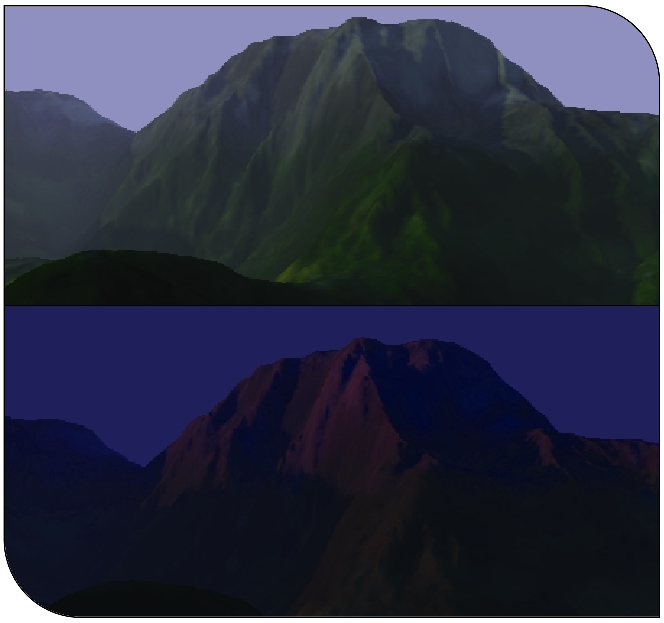


FIGURE 15. Video-based illumination maps on PC.

for game production. Although sky dome/box capture is less problematic, it is just too time consuming to wait for the perfect sunset or the perfect storm. An artist-generated illumination map and sky box could produce the same results in minutes given sufficiently sophisticated rendering tools.

Future directions. With the increased realism of terrain lighting, objects that are placed in this environment must also match the local lighting conditions. Potential solutions range from sampling the terrain illumination map directly beneath the object to streaming irradiance environment maps along with the illumination maps (see Ramamoorthi and Hanrahan under For More Information for more on irradiance environment maps).

Controlled changes in weather conditions could be simulated by branching to alternative video streams. The frame interpolation method would permit smooth transitions between these states.

Using video-based illumination maps in terrain rendering is just one application of video-based rendering in games. With sufficient future hardware support a range of applications will open up for games, such as video-based impostors (see Wilson under For More Information).

In some cases, the sequence of lightmaps as a function of time can be represented by a polynomial texture map (see Malzbender, Gelb, and Wolters under For More Information). This is a promising direction, since the entire sequence can be stored as a small number of coefficients per texel and calculated quickly every frame. This could be useful for other applications of video-based illumination maps as well.

New Directions

Both methods presented here enable outdoor scenes to come closer to the ideal of photorealism by capturing the complexity of the outdoor lighting environment. The analytical method comes close to the quality of global illumination methods while the quality of the video-based rendering method can go as high as you want it to by using whatever offline renderer you like — raytracing, radiosity, photon maps, even real captured movies.

The preprocessing requirements differ between the two methods. The analytical method can perform preprocessing in seconds on a PC and takes up a few megabytes of data. The video-based rendering method can take hours or (in theory) days based on the quality desired, though the use of rendering farms can definitely cut down on the time needed. With the video-based method, the preprocessed data is a video stream, the size of which will vary based on the resolution, frame rate, and codec used.

Since both methods rely heavily on preprocessing, they work best with static scenes. The analytical method can support local changes in geometry by redoing the preprocessing for the terrain region affected by the change. For this to be practical, the affected region should be a very small part of the total scene.

Both methods are examples of new directions in real-time rendering. Increased programmability in hardware will enable us to perform sophisticated global illumination calculations analytically. Video-based rendering and lighting can be used to produce movie-quality effects in real time, and we will be able to use the results of movie production methods in our games.

ACKNOWLEDGEMENTS

We would like to thank ATI and Nvidia for supplying us with hardware for experimentation and demos, and Hector Yee for his help on the analytical method implementation.

FOR MORE INFORMATION

SOFTWARE

Terragen
www.planetside.co.uk/terragen
Free to download for noncommercial use. Commercial use is permitted for registered users.

REFERENCES

Debevec, P. "Pursuing Reality with Image-Based Modeling, Rendering, and Lighting." Keynote presentation at the Second Workshop on 3D Structure from Multiple Images of Large-Scale Environments and Applications to Virtual and Augmented Reality (SMILE2), Dublin, Ireland, June 2000. www.debevec.org/Publications

Heidrich, W., K. Daubert, J. Kautz, and H.-P. Seidel. "Illuminating Micro Geometry Based on Precomputed Visibility." *Computer Graphics (Proceedings of SIGGRAPH 2000)*, July 2000: 455–464. www.cs.ubc.ca/~heidrich/Papers/index.html

Hoffman, N., and K. Mitchell. "Real-Time Photorealistic Terrain Lighting." *2001 Game Developers Conference Proceedings*, March 2001: 357–367. www.gdconf.com/archives/proceedings/2001/prog_papers.html

Max, N. L. "Horizon Mapping: Shadows for Bump-Mapped Surfaces." *The Visual Computer* Vol. 4, No. 2 (July 1988): 109–177.

Malzbender, T., D. Gelb, and H. Wolters. "Polynomial Texture Maps." To appear in *Computer Graphics (Proceedings of SIGGRAPH 2001)*, August 2001. www.hpl.hp.com/ptm

Mitchell, K. "Real-Time Full Scene Anti-Aliasing for PCs and Consoles." 2001 *Game Developers Conference Proceedings*, March 2001: 537–543. www.gdconf.com/archives/proceedings/2001/prog_papers.html

Ramamoorthi, R., and P. Hanrahan. "An Efficient Representation for Irradiance Environment Maps." To appear in *Computer Graphics (Proceedings of SIGGRAPH 2001)*, August 2001. http://graphics.stanford.edu/papers/envmap

Schödl, A., R. Szeliski, D. Salesin, and I. Essa. "Video Textures." *Computer Graphics (Proceedings of SIGGRAPH 2000)*, July 2000: 489–498. www.gvu.gatech.edu/perception/projects/videotexture

Sloan, P.-P., and M. F. Cohen. "Interactive Horizon Mapping." *Rendering Techniques 2000 (Proceedings of the Eleventh Eurographics Workshop on Rendering Workshop)*, June 2000: 281–298. www.research.microsoft.com/~cohen

Stewart, A. J. "Fast Horizon Computation at All Points of a Terrain with Visibility and Shading Applications." *IEEE Transactions on Visualization and Computer Graphics* Vol 4, No. 1 (March 1998): 82–93. www.dgp.toronto.edu/people/JamesStewart/papers/tvcg97.html

Stewart, A. J., and M. S. Langer. "Towards Accurate Recovery of Shape from Shading under Diffuse Lighting." *IEEE Transactions on Pattern Analysis and Machine Intelligence* Vol. 19, No. 9 (Sept. 1997): 1020–1025. www.dgp.toronto.edu/people/JamesStewart/papers/pami97.html

Wilson, A., M. C. Lin, D. Manocha, B.-L. Yeo, and M. Young. "A Video-Based Rendering Acceleration Algorithm for Interactive Walkthroughs." *Proceedings of ACM Multimedia*, October 2000. http://woodworm.cs.uml.edu/~rprice/ep/wilson