Hemisphere Lighting With Radiosity Maps

By Shawn Hargreaves

This lighting model was designed for fast moving objects in outdoor environments. Its goals are to tie the moving objects in with their surroundings, to convey a sensation of speed, and to be capable of rendering large numbers of meshes at a good framerate on first generation shader hardware.

It combines a crude form of radiosity lighting with world to object shadowing using just one texture lookup and four pixel shader blend instructions. Approximation is the name of the game here, with performance being by far the most important consideration!

Hemisphere Lighting

There is a probably apocryphal story that in the early days of color television, someone pulled off a successful scam selling kits that claimed to upgrade existing black and white TVs to display a color picture. This was done using a bit of plastic that fitted over the TV screen and tinted the top third of the display blue, the middle green, and the bottom brown, on the assumption that most things in life have sky at the top, trees in the middle, and soil underneath. Not perhaps the most robust of solutions, and I suspect the people who bought this kit were not terribly happy, but it would have worked ok at least for a few carefully chosen images!

Hemisphere lighting is basically just a shader implementation of the same concept.

Most conventional lighting models evaluate some sort of equation for the more important few lights in a scene, and then add in a constant ambient term as an approximation of all the leftover bits and pieces. This works nicely for scenes with many complex light sources, but is less than ideal for outdoor environments where all the direct light is coming from the sun. With only a single light source available, fully half of every object will be in shadow, and thus will be illuminated only by the ambient term. This gets even worse in overcast or rainy weather conditions, because as the sun is obscured by clouds its direct contribution becomes less, and almost all of the light in the scene ends up being provided by the catch-all ambient constant. A constant amount of light results in a constant color, which makes things look flat and boring.

This is clearly wrong, because you only have to step outside on a foggy morning to notice that even though the sun itself may by entirely hidden, there is still enough variation in light levels that you can easily make out the contours of whatever you are looking at.

The problem with conventional lighting models is that in the real world, the majority of light does not come directly from a single source. In an outdoor setting some of it does indeed come straight from the sun, but more comes equally from all parts of the sky, and still more is reflected back from the ground and other surrounding objects. These indirect light sources are extremely important, because they will often be providing a much larger percentage of the total illumination than the sun itself.

Hemisphere lighting is a simple way of emulating the indirect light contributions found in a typical outdoor scene. Any kind of complex radiosity lighting could be modelled by encoding the surrounding light sources into an HDR (high dynamic range) cubemap, but it is impractical to update such a cubemap in realtime as large numbers of objects move around the world. So we need to approximate, cutting down the complexities of the real world into a more efficient realtime model.

The sun is easy: a per-vertex dot3 can handle that quite nicely. With this taken out of the equation, we are left with a pattern of light sources that can be roughly divided into:

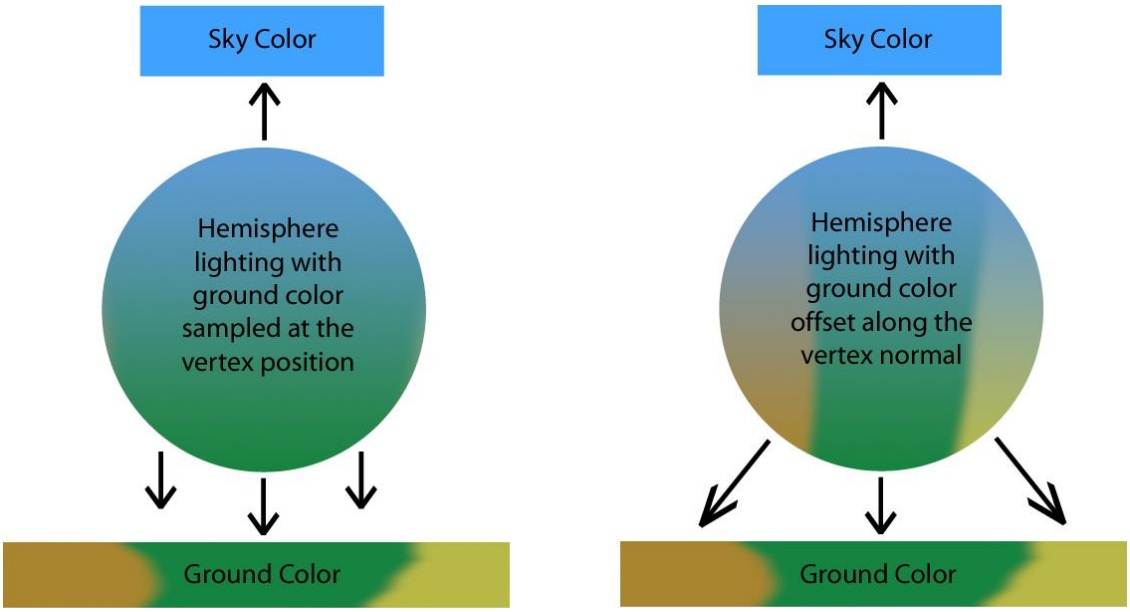
- Sky: usually blue; emits lots of light; located above your head.
- Ground: some other color; darker than the sky; located underneath you.

This is trivial to evaluate in a vertex shader: just set your sky and ground colors as constants, and use the y component of the vertex normal to interpolate between them!

Radiosity Maps

Hemisphere lighting avoids the flatness that can result from a constant ambient term, but it also poses a question: what should you use for the ground color? Bearing in mind our goal of making moving objects fit in with their surroundings, it would be good if this could change appropriately depending on your location in the world.

The solution is obvious: encode the ground color into a texture as a large, top down image of the landscape. This map can then be sampled at a position corresponding to the location of the object, or even better, offset some distance (a metre or so works well) along the vertex normal. Adding this offset stretches out the sample area to include a larger region of the ground image, and introduces some horizontal lighting variation in addition to the vertical ground to sky transition.



The results may not be exactly what a high end renderer would describe as radiosity lighting, but it can be a remarkably good approximation. The underside of an object picks up color from the ground directly beneath it, while the sides are influenced by the scenery slightly off to each side, and the top is affected entirely by the sky color.

Making The Map

Ground color maps can easily be generated by taking a screenshot of your level, viewed from above with an orthographic projection. The results can be improved if you preprocess the mesh by removing polygons that are too high above the ground surface, and rotating vertical polygons to face upwards so elements like the sides of fences will contribute to the radiosity colors.

I also found it useful to add about 10% of random noise to the resulting texture, as this introduces a subtle speed-dependent flicker that gives an effective sense of motion as you move around the world.

A 1024x1024 texture (only half a megabyte when encoded in DXT1 format) is sufficient to represent a couple of square miles of landscape with enough precision to make out details such as alternating colors along the rumble strip at the edge of a racetrack, or dappled patterns of light and shadow in a forest scene.

Shadowing

Once you have the ground color encoded in a texture, it seems like static environment to object shadows ought to pretty much “just work” if you put a dark patch in the relevant portion of the radiosity map. Compared to other shadowing techniques, this is highly approximate but also incredibly cheap, and it can be very effective especially for complex shadow patterns such as a forest floor.

Unfortunately, it doesn’t “just work”. The problem with using the radiosity map to encode shadows is that even if you darken down the ground color, the sky color is still a constant and so will not be affected.

There are several possible solutions:

- Use one texture to encode the ground color, and another to encode shadows. This is the highest quality and most controllable approach, but it burns two texture units and requires double the amount of storage for the two textures.

- You could encode the shadow amount into the alpha channel of the radiosity texture. In this case your ground color would be (radiosity.rgb * radiosity.a), while the sky color would be (sky_color_constant * radiosity.a). This works well, but using alpha in the radiosity map requires at least an 8 bit texture format such as DXT5. For such a large image, storage space is a serious concern.

- At the risk of excessive approximation, it is possible to collapse the ground color and shadow data into a single RGB texture, thus allowing it to be stored in 4 bit per texel DXT1 format. The process is:

    1. Convert your radiosity map into an HSV color space.

    2. Find the average V (brightness) value.

    3. Normalize the map so that all texels have this same constant brightness level, except for areas shadowed by static environment geometry, for which the brightness is left at a lower level. Hue and saturation are not affected by this process.

    4. Convert back into RGB format.

    5. Work out a scaling factor that will turn the average radiosity brightness into your desired sky color, and set this as a shader constant.

    At runtime, the ground color can be looked up directly from the modified radiosity map. Except for areas of shadow, this will now be lacking any variation in brightness, but the changes in hue are enough for the technique to remain effective.

    To calculate the sky color, in your pixel shader dot the ground color with (1/3, 1/3, 1/3), thus converting it to greyscale. Because of the brightness normalisation this will produce a constant value for all non-shadowed areas, or a darker shade of grey if you are in shadow. Multiplying this value by the sky color scaling constant gives a correctly shadowed version of the sky term.

Combining the ground color and shadow information into a single texture creates one final dilemma: where should this texture be sampled? The radiosity lighting works best if the sample position is offset along the vertex normal, but that is blatantly incorrect for shadowing, where the map should be sampled directly at the vertex position.

A hacky compromise is to apply an offset along the left to right component of the normal, but not in the front/back direction, so polygons facing forward or backwards will sample the radiosity map at their exact position, while side facing polygons use an offset sample point. Since objects usually travel roughly along their forward axis, this maintains a nice solid transition as they move in and out of shadow, while still allowing radiosity tints to be picked up from either side of their exact location.

The Shaders

The complete lighting model combines four elements:

- Base texture.
- Radiosity texture combining ground color and shadow information. The vertex shader calculates the sample location and ground to sky tweening factor, while the pixel shader generates a shadowed version of the sky color based on the greyscale of the ground color, and performs the hemisphere tween.
- Environment cubemap containing a static image of a typical area of the level along with a specular highlight in the alpha channel. The envmap intensity is calculated in the vertex shader, combining a per-vertex reflectivity amount (mapped by artists) with a 1-cos Fresnel approximation.
- The direct sun contribution is calculated per vertex using a straightforward infinitely distant dot3 light.

```
vs.1.1


// vertex inputs:
#define iPos       v0                  // vertex position
#define iNormal    v1                  // vertex normal
#define iDiffuse   v2                  // reflectivity amount
#define iTex0      v3                  // base texture coordinates

dcl_position       iPos
dcl_normal         iNormal
dcl_color0         iDiffuse
dcl_texcoord0      iTex0


// constants:
def c0, 0, 0, 0, 1

#define VS_CONST_0              c[0].x
#define VS_CONST_1              c[0].w

#define VS_EYEPOS               1       // object space eye position

#define VS_CAMERA1              2       // 4x4 object to screen matrix
#define VS_CAMERA2              3
#define VS_CAMERA3              4
#define VS_CAMERA4              5

#define VS_ENVMAP1              6       // 3x3 object to world matrix
#define VS_ENVMAP2              7
#define VS_ENVMAP3              8

#define VS_FOG                  9       // fog transform vector

#define VS_AMBIENT              10      // ambient light color
#define VS_LIGHT_COLOR          11      // diffuse light color
#define VS_LIGHT_DIR            12      // object space light direction

#define VS_RADIOSITY_U          13      // radiosity U mapping
#define VS_RADIOSITY_V          14      // radiosity V mapping

#define VS_RADIOSITY_SIDE       15      // object sideways offset

#define VS_RADIOSITY_SAT        16      // ground vs. sky vector
```

```
// outputs:
//
//  oPos    = position
//  oFog    = fogging
//
//  oT0     = base texture coordinates
//  oT1     = radiosity map sample location
//  oT2     = environment cubemap coordinates
//
//  oD0.xyz = dot3 sunlight
//  oD1.xyz = radiosity ground to sky tween factor
//  oD0.w   = fresnel term
//  oD1.w   = specular intensity



// transform the vertex position
mul r0, c[VS_CAMERA1], iPos.x
mad r0, c[VS_CAMERA2], iPos.y, r0
mad r0, c[VS_CAMERA3], iPos.z, r0
add oPos, c[VS_CAMERA4], r0

// calculate the fog amount
dp4 oFog, iPos, c[VS_FOG]

// output the base texture coords
mov oT0.xy, iTex0



// ***************** RADIOSITY HEMISPHERE ****************

// stretch the radiosity lookup area to either side of the model
dp3 r0.x, iNormal, c[VS_RADIOSITY_SIDE]
mad r0.xyz, r0.x, c[VS_RADIOSITY_SIDE], iPos

// planar map the radiosity texture
mov r0.w, VS_CONST_1

dp4 oT1.x, r0, c[VS_RADIOSITY_U]
dp4 oT1.y, r0, c[VS_RADIOSITY_V]

// calculate the ground to sky radiosity tween factor
dp4 oD1.xyz, iNormal, c[VS_RADIOSITY_SAT]



// ***************** FRESNEL / SPECULAR CUBEMAP ****************

// calculate and normalize the eye->vertex vector
sub r0.xyz, iPos, c[VS_EYEPOS]
dp3 r0.w, r0, r0
rsq r0.w, r0.w
mul r0.xyz, r0, r0.w

// dot the vertex normal with eye->vert
dp3 r1.x, r0, iNormal

// fresnel term = (1 - r1.x) * reflectivity amount
mad oD0.w, r1.x, iDiffuse.x, iDiffuse.x

// also output a non-fresnel version of the reflectivity amount
mov oD1.w, iDiffuse.x

// reflect the view direction through the vertex normal
add r1.x, r1.x, r1.x
mad r0.xyz, iNormal, -r1.x, r0

// transform the environment map sample location into worldspace
dp3 oT2.x, r0, c[VS_ENVMAP1]
dp3 oT2.y, r0, c[VS_ENVMAP2]
dp3 oT2.z, r0, c[VS_ENVMAP3]
```

```
// *************** DOT3 SUNLIGHT ***************

// let's do a boring old school per vertex diffuse light, too...
dp3 r0.x, iNormal, c[VS_LIGHT_DIR]
max r0.x, r0.x, VS_CONST_0
mul r0.xyz, r0.x, c[VS_LIGHT_COLOR]
add oD0.xyz, r0, c[VS_AMBIENT]
```

```
ps.1.1

// inputs:
//
//  v0.rgb = dot3 sunlight
//  v1.rgb = radiosity ground to sky tween factor
//  v0.a   = fresnel term
//  v1.a   = specular intensity
//
//  c1     = sky color

def c0, 0.3333, 0.3333, 0.3333, 0.3333

tex t0                          // base texture
tex t1                          // radiosity texture
tex t2                          // environment cubemap

// envmap + specular
lrp r0.rgb, v0.a, t2, t0        // fresnel tween between envmap and base
mad r0.rgb, t2.a, v1.a, r0      // add the specular component

// radiosity hemisphere
dp3 r1.rgb, t1, c0              // greyscale version of the ground color
mul r1.rgb, r1, c1              // calculate sky color
lrp r1.rgb, v1, t1, r1          // tween between ground and sky
mul_x2 r0.rgb, r0, r1           // apply the radiosity color

// per vertex sunlight
mul_x2 r0.rgb, r0, v0           // output color * diffuse
+ mov r0.a, t0.a                // output base texture alpha
```

Additional Considerations

This form of lighting can easily be simplified, using cheaper versions to implement shader LOD on distant objects. Most significantly, the per-pixel radiosity lookups and sky color calculations can be replaced by a single CPU texture lookup at the center of the object, with the resulting sky and ground colors set as vertex shader constants, the hemisphere tween evaluated per vertex, and no work at all required in the pixel shader.

Doing single-texel CPU lookups into the radiosity map is extremely fast, and this data can be useful in many places. For instance, a particle system might do a ground color lookup when spawning a new dust particle, to see if it should be in shadow and also so it can be tinted to match the hue of its surroundings.

The radiosity maps can easily become very large, but they are also highly compressible. Large areas of the map will typically contain either flat color or smooth gradients, so good results can be obtained by splitting it into a grid of tiles and adjusting the resolution of each tile according to how much detail it contains. At runtime, a quick render to texture can expand the area around the camera back out into a continuous full resolution map.

Because the radiosity map is only a two dimensional image, there will obviously be problems with environments that include multiple vertical levels. Such cases can be handled by splitting the world into layers, with a different radiosity map for each, but this lighting model is not well suited to landscapes with a great deal of vertical complexity!

References

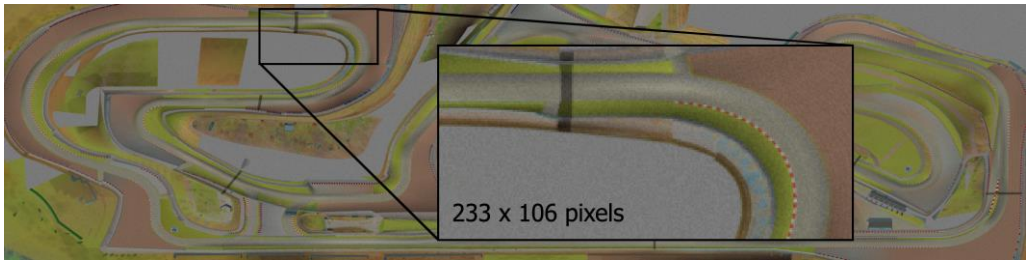
Philip Taylor (Microsoft Corporation) discusses hemisphere lighting:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndrive/html/directx11192001.asp

The non-approximate version: Image Based Lighting, Cunjie Zhu, University of Delaware:
http://www.eecis.udel.edu/~czhu/IBL.pdf

Videos demonstrating various elements of the shaders presented above can be found on the accompanying CD.

This lighting model is used in the game MotoGP 2, on PC and Xbox, developed by Climax and published by THQ. MotoGP 1 used a similar radiosity technique, but without the Fresnel term on the environment cubemap.

Color Plates



The images below were created along the zoomed in section of this 2048x512 radiosity map

Diffuse (dot3) sunlight plus radiosity hemisphere lookup

With the addition of specular and Fresnel contributions, using a static cubemap holding an image of typical surroundings

These images show the hemisphere lighting on its own, using a single DXT1 format radiosity map that encodes both shadow and ground color information

The complete lighting model, combining a base texture, radiosity hemisphere, Fresnel cubemap, and dot3 sunlight