

# Image-Based Diffuse Lighting Using Visibility Maps: Additional Notes

Ivan Neulander\*  
Rhythm and Hues Studios

## 1 Example

Here is an example of visibility map (VM) construction and sampling, using the face mask object shown below. This was all done on a *Pentium III* 1GHz with 1GB RAM, running Linux.



Figure 1: The face model used in our example (left) diffusely shaded with a point light (right). It consists of 37,584 triangles.

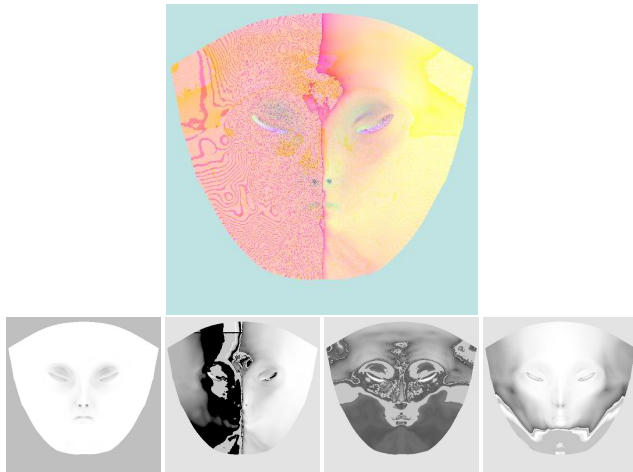


Figure 2: The VM (top) for the model in Figure 1 consists of four channels (bottom). The first channel represents the percentage ambient visibility, and the following three channels represent the  $x, y, z$  components of the unit visibility vector (the average direction of all unoccluded ray samples over the visibility hemisphere). The  $400 \times 400$  map used 100 rays per pixel, and took about 5 minutes to generate.

### 1.1 Animation

An animation sequence demonstrating our results with Visibility Map shading can be found at [www.rhythm.com/~ivan/visMaps.html](http://www.rhythm.com/~ivan/visMaps.html).

\*e-mail: [ivan@rhythm.com](mailto:ivan@rhythm.com)



Figure 3: Rows 1-4:

- 1) Environment maps.
- 2) Cosine-filtered environment maps.
- 3) Diffuse shaded maps using VM from Figure 2.
- 4) Final renders using above diffuse shaded maps.

Each diffuse shaded map took about 1 second to generate given the VM and the cosine-filtered environment map.

## 2 Quality Comparison of our Diffuse Shading Approximation

Because our VM-based diffuse shading model uses just a single lookup of the cosine-filtered environment map, it is not physically accurate. Despite this, it is visually plausible in most cases, particularly with low-frequency environments.

Here we compare the quality of our shading model to 1) a model that completely ignores occlusion, basing diffuse illumination only on a surface's shading normal; 2) the physically correct solution<sup>1</sup>, which samples the environment map over the visibility hemisphere, taking occlusion into account at each sample. We consider three environments, with different profiles of dynamic range and spatial frequency.

<sup>1</sup>Note that the physically correct model, unlike VM lighting, requires all ray tracing to be repeated given any change to the environment or orientation of the object.

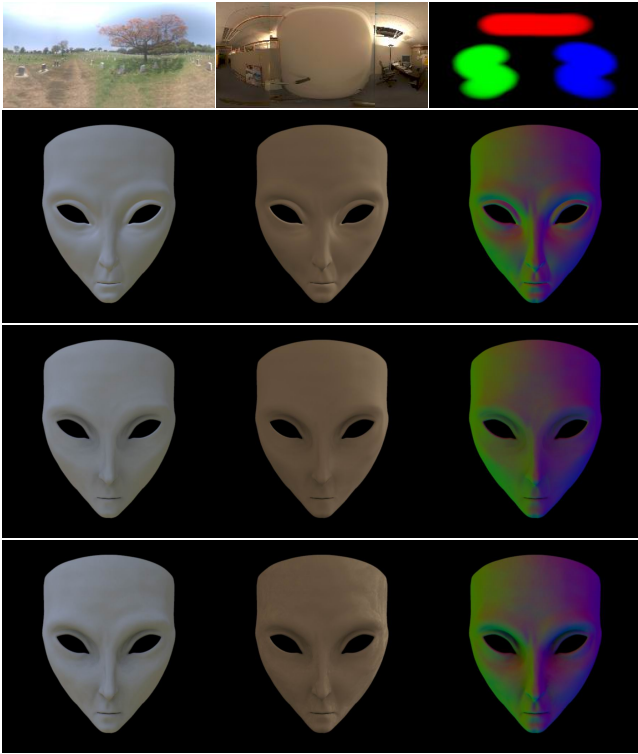


Figure 4: Rows 1-4:  
 1) Environment maps.  
 2) Diffuse shading with no occlusion testing.  
 3) Our VM diffuse shading model.  
 4) Physically accurate diffuse shading.

### 3 Ray Sampling Details

Our ray generator uses the following method to compute a quasirandom set of rays that are uniformly distributed over the unit hemisphere centered about some vector  $\vec{N}$ .

For each ray, we first compute a pair of values  $h_1, h_2$  from a two-dimensional Halton sequence generator. These form a sample in the unit square  $[0, 1] \times [0, 1]$ . We add to these a pair of random offset values in  $[0, 1]$ , which are chosen once per pixel (at each pixel, the Halton generator is reset and a new pair of random offsets is computed). The resulting sums are computed modulo 1, producing  $\xi_1, \xi_2$ . We apply  $\xi_1$  and  $\xi_2$  to the following formula<sup>2</sup> to obtain a canonical hemisphere ray direction:

$$\begin{aligned} x &= \cos(2\pi\xi_1)\sqrt{1-\xi_2^2} \\ y &= \sin(2\pi\xi_1)\sqrt{1-\xi_2^2} \\ z &= \xi_2 \end{aligned}$$

Finally, we rotate the canonical ray using a previously computed matrix based on  $\vec{N}$ .

#### 3.1 Halton Implementation

The following is a basic C++ implementation of a two-dimensional Halton sampler suitable for our purposes:

```
class Halton {
    float _invBases[2];
    float _prev[2];

    float halton(float baseRec, float prev) const {
        float r = 1 - prev - 1e-10;
        if (baseRec < r) return prev + baseRec;
        float h = baseRec;
        float hh;
        do {
            hh = h;
            h *= baseRec;
        } while (h >= r);

        return prev + hh + h - 1;
    }

public:
    void Reset() {
        _prev[0] = _prev[1] = 0;
    }

    Halton() {
        _invBases[0] = 1./2;
        _invBases[1] = 1./3;
        Reset();
    }

    void GetNext(float out[2]) {
        out[0] = halton(_invBases[0], _prev[0]);
        out[1] = halton(_invBases[1], _prev[1]);
        _prev[0] = out[0];
        _prev[1] = out[1];
    }
};
```

#### 3.2 Ray Direction Examples

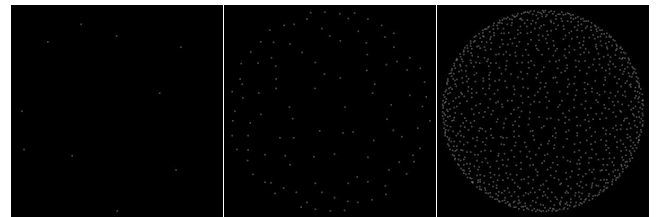


Figure 5: Distribution of 10, 110, 1110 hemispherical ray samples.

### 4 Speed Advantages and Optimizations

There are several advantages to our two-dimensional representation of the VM. Assuming a continuous texture mapping is used, a gaussian blur can be trivially applied to the VM in order to efficiently reduce sampling noise without blurring the entire rendered image. Also, the VM can be effectively mip-mapped to provide fast anti-aliasing. Finally, the balance between speed and quality during VM computation is easily controlled by adjusting the resolution of the VM.

In addition, we use an adaptive sampling technique to expedite VM creation for geometry with variable occlusion complexity. In this case, some areas of the object's surface require many more ray samples than others. We take advantage of the variance-reducing

<sup>2</sup>Obtained from Phillip Dutre's Global Illumination Compendium

properties of the Halton sequence, which allows us to continue drawing quasi-random samples while maintaining an even overall distribution.

Our approach is to take multiple iterations over the pixel, casting a fixed number of rays each time. Each iteration uses a different distribution of rays, as generated by repeated evaluations of the Halton sequence. If several consecutive iterations fail to produce a significant difference in the computed ambient visibility or average direction, no further iterations occur and the pixel is considered fully shaded.

## 5 Ambient Occlusion Maps

Our technique is a generalization of ambient occlusion mapping. Applying a VM to a solid white environment map produces a view-independent ambient occlusion texture, as demonstrated in the following rendered images.



Figure 6: Top: Ambient occlusion map applied to white surface. Bottom: Final frame without (left) and with (right) ambient occlusion texturing.

Ambient occlusion maps are useful in simulating self-shadowing in a variety of traditional (non-image-based) lighting environments. Our method is optimized to generate them without performing any environment map lookups or creating an intermediate VM. Because they are view-independent, ambient occlusion maps can be applied essentially for free by premultiplying them with existing textures. For this reason, they are particularly valuable in real-time rendering applications.