Approximate Radiosity Using Stochastic Depth Buffering

Andreas Thomsen & Kasper Høy Nielsen

Vol. 15, No. 4: 225–234

Approximate Radiosity Using Stochastic Depth Buffering

Andreas Thomsen
Reto-Moto

Kasper Høy Nielsen
IO Interactive

Abstract. This paper presents a simple technique for computing approximate radiosity by using a novel approach to visibility determination called *stochastic depth buffering*. By rendering random depth buffer values and using a simple sampling scheme, we gather light from parallel, global directions using the GPU. The technique makes it possible to quickly update indirect light on commodity graphics hardware. It is easy to implement and offers good trade-offs between performance and visual quality.

1. Introduction

Today, commodity graphics hardware is so fast that approximate global illumination is possible in real-time applications such as computer games. This paper presents a fast and simple technique for computing approximate, diffuse interreflections (*radiosity*) in a lightmapped, opaque scene by utilizing the GPU. It is suitable for updating dynamic lightmaps in applications with slowly changing light conditions and environments. In environments with point-light sources, traditional shadow mapping can be used to compute shadows from direct light, and our technique can be used for indirect light only.

ISSN: 2151-237X print/2151-2272 online

A general radiosity method is *gathering*, wherein the illumination received from all visible light sources and diffuse surfaces seen from a particular point is integrated over its hemisphere. Doing this for all surface points, multiplied by the surface reflectance, yields a representation of the outgoing light in the scene. Repeating the process for several iterations allows light to bounce off surfaces, and to eventually reach a steady state. Mathematically, it is best described by the rendering equation for diffuse surfaces:

$$L_{\mathrm{o}}(x,n) = L_{\mathrm{e}}(x) + \frac{R(x)}{\pi}\int_{\Omega} L_{\mathrm{i}}(x,\omega)\max(<\omega,n>,0)d\omega, \qquad (1)$$

where L_{o} $(x,\ n)$ is the outgoing radiance at a surface point x with normal n, L_{e} (x) is the emitted radiance, L_{i} $(x',\ \omega)$ is the radiance received at point x from the direction ω, and Ω is the unit sphere. $R(x)$ is the reflectance of the material at x. Because light is reflected diffusely, L_{o} $(x,\ n)$ is the same for all outgoing directions.

2. Gathering on the GPU

As with most global illumination methods, the costly part of gathering is the determination of visible surfaces seen from a point. In traditional ray tracing this is done by shooting rays in random hemispherical directions. A more GPU-friendly approach is to focus on a single global direction at a time, and compute the visibility for all points in the scene in parallel [Sbert 96, Hachisuka 05]. See Figure 1(a). This can be done by rendering a snapshot of the scene with culling from the direction of interest using parallel projection, similar to rendering a shadow map from a directional light source.

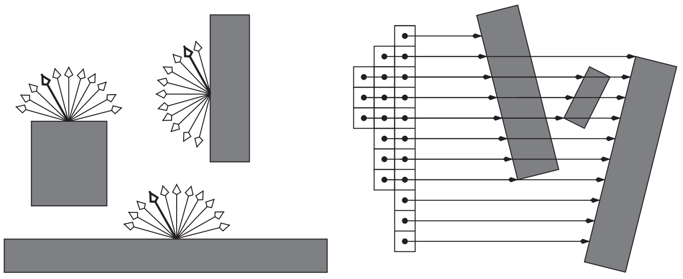

Figure 1. (a) Clustering rays by global directions (after [Hachisuka 05]). (b) Several surface layers (left) need to be accounted for in order to represent the scene (right).

We are interested in the illumination contribution in that direction, so both depth and illumination RGB values are stored. We call the resulting map the *illumination map*.

The received light is distributed to all surface points by projecting the points into the illumination map and using the resulting lookup to add the incident light contribution from the global direction. The process is then repeated for subsequent global directions to accumulate incident light.

All surfaces rendered into the illumination map need to be accounted for, because they can potentially reflect or block light. Ideally we would need to store a list of fragments per pixel. See Figure 1(b). However, storing this much information in a single pixel is neither possible on a DirectX 9-class GPU nor on current generation game consoles. One solution is to use *depth peeling* [Everitt 01], where the depth buffer is used to "peel away" depths in successive passes [Hachisuka 05]. Standard depth peeling is expensive on current GPUs because of its multipass nature. It requires n passes to account for n depths. Faster variations exist [Myers and Bavoil 07, Liu et al. 09, Bavoil and Myers 08, Enderton et al. 10], some of which require more exotic hardware features. However, although depth peeling is accurate, it is not practical as an approximation because it considers all surface layers in each iteration. Because there is no upper bound on the number of layers, this can lead to many texture reads, multiple passes, and high memory usage—especially for scenes with complex regions and many layers.

3. Stochastic Depth Buffering

Instead of using depth peeling, we employ a more approximate technique: after clearing the depth buffer with 1s (1 being the maximum depth), we render all scene surfaces with per-fragment, uniformly distributed, random z values in the range [0, 1]. Because of the depth rejection, this creates a noisy image, where surface pixels are sometimes rendered as visible, and sometimes as occluded. See Figure 2. The visible surface pixels are shuffled by randomizing their depths, so a single illumination map lookup will return a sample from a random surface along the projection ray this pixel represents. If there is only one surface along the projection ray, there will be no shuffling. Taking more samples from neighboring pixels will return a random selection of surfaces in the vicinity of the ray.

From this observation, we make the key assumption that n nearby samples will be a good representation for the set of actual surface points that the ray intersects. The quality of this approximation depends on the number of samples, the resolution of the illumination map, and the amount of overdraw in the scene. Essentially, image resolution is traded for capturing additional

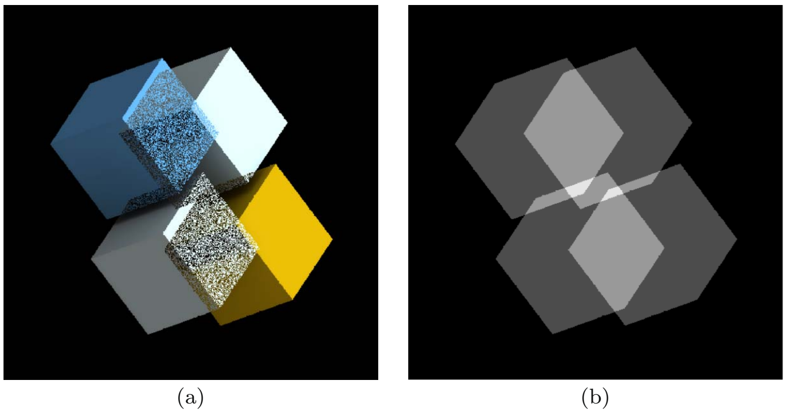

Figure 2. (a) The illumination map for one global direction from a scene containing four cubes. (b) Image visualizing the amount of overdraw in the illumination map.

surfaces hidden along a ray. We can use this approximation as a single-pass replacement for depth peeling.

4. Calculating the Light at a Sample Point

To calculate the indirect light on a surface element at position x with normal n, we want to solve Equation (1). Using Monte Carlo integration with a uniform distribution over the surface of the sphere Ω we get

$$L_{\mathrm{o}}(x,n) = L_{\mathrm{e}}(x) + \frac{4R(x)}{N}\sum_{s=1}^{N} L_{\mathrm{i}}(x,d_s)\max(<d_s,n>,0), \tag{2}$$

where d_s is the set of global sampling directions. The illumination map is used to compute the incoming radiance L_{i} $(x,\ d)$ at x from a given direction d. By projecting x into the illumination map, we can sample the incident illumination in the neighborhood of x'. See Figure 3. Samples that lie below the normal plane $(x,\ n)$ are rejected, and the one closest to x along the direction d of the remaining is chosen. If no sample is found, the color of the background (e.g., a sky-dome) is used. For choosing the surface nearest the sample point, the depth along the ray direction for surface position p, $dot(p,\ d)$, is stored together with each illumination value.

The resulting pseudocode shader for rendering the illumination map is shown in Figure 4. The `CalculateIndirectLight` function for computing one part of the indirect illumination sum in Equation 2 is shown in Figure 5.

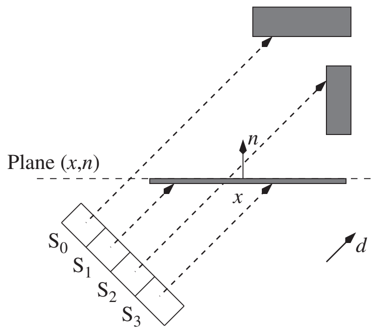

Figure 3. Selecting a sample: the illumination buffer is sampled near position x with normal n, and the sample closest to x along direction d and above the normal plane (x, n) is chosen, where S_i is the ith sample.

5. Gathering with Lightmapping

With lightmapping, every surface is given a unique set of UV-coordinates pointing into a tightly packed texture atlas containing the scene's illumination. In order to use a pixel shader to update the lightmap, we first precompute two texture maps, one containing the texels' world-space center and one containing their normals.

For performing one iteration of light propagation, we render the scene into the illumination map (using the pixel shader in Figure 4) for one randomly selected global direction. The lightmap is then updated by rendering a "fullscreen" quad, using the code in Figure 6 that samples the illumination map, that uses the code in Figure 5 for each texel in the lightmap. Thus, one iteration updates the lightmap with indirect illumination from one global direction.

Several iterations have to be run to accumulate the lighting contributions, averaged by the number of samples taken. To see the result instantly, we use leaky integration (i.e., value = value * 0.99 + newvalue * 0.01). The resulting shader code is shown in Figure 6.

```
IllumPixelShaderOut IlluminationBufferPixelShader(In vertexOut)
{
  IllumPixelShaderOut out;
  out.Sample.rgb = CalculateLight(vertexOut);
  out.Sample.w = dot(vertexOut.WorldPosition, d);
  out.depth = random();
  return out;
}
```

Figure 4. Illumination map pixel shader for global direction d.

```
float3 CalculateIndirectLight(float3 p, float3 n)
{
  float weight = dot(n, d);
  if (weight <= 0) return float3(0,0,0);
  float4 bestSample = float4(SkyColorInDirection(d), infinity);
  float2 illumPos = ProjectToIlluminationMap(p);
  for (int i=0;i<neighborHoodSamples;i++)
  {
    float2 illumTexCoord = illumPos + sampleOffset[i];
    float4 illumSample = tex2D(illumMap, illumTexCoord);
    float3 samplePos = GetWorldPos(illumTexCoord, illumSample.w);
    if (dot(samplePos-p, n) > 0 && illumSample.w < bestSample.w)
    {
      bestSample = illumSample;
    }
  }
  return 4 * bestSample.rgb * weight;
}
```

Figure 5. Calculate indirect light at position p and normal n for global direction d. The factor 4 in the last statement comes from Equation 2.

```
float3 LightMapUpdatePixelShader(In vertexOut)
{
  float3 pos = tex2D(LightMapPosition, vertexOut.lightMapTexCoord);
  float3 normal = tex2D(LightMapNormal, vertexOut.lightMapTexCoord);
  float3 newColor = CalculateIndirectLight(pos, normal);
  float4 prevColor = tex2D(LightMapColor, vertexOut.lightMapTexCoord);
  return newColor * lightMapBlendNew + prevColor * lightMapBlendPrev;
}
```

Figure 6. Shader for updating the lightmap.

6. Limitations

Because depth rejection returns the fragment with the lowest random depth value, the illumination map drawing order is not important. Each surface fragment will have exactly one chance per pixel of having the lowest depth value, resulting in a uniform distribution. However, stochastic depth buffering gives accurate results only when the amount of overdraw is small compared with the number of samples used. When our key assumption is violated (i.e., a surface is not consistently represented in the sampling neighborhood), there is a chance that points will sample light from nonvisible surfaces. Intuitively, less overdraw and more samples mean less risk of sampling from the wrong surface.

The chance of a surface being visible at a given pixel in the illumination map is $\frac{1}{K}$, where K is the amount of overdraw in the illumination map at that pixel. If we assume that the overdraw is constant in a local neighborhood of

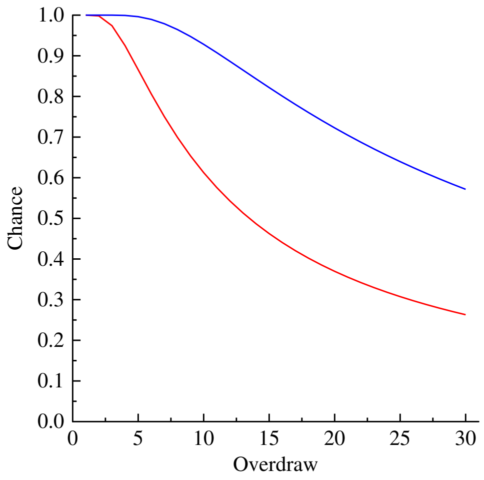

Figure 7. Chance of hitting a surface with 9 (red) and 25 (blue) lookups.

the illumination map with n samples, the chance of the neighborhood containing a sample from a surface is $C_n(K) = 1 - (1 - \frac{1}{K})^n$. Figure 7 shows how the chance of a surface being represented drops as the amount of overdraw increases for common neighborhood sizes of 9 (3×3) and 25 (5×5).

Too much overdraw introduces noise caused by missed surfaces in the illumination map on a per-pixel level. However, the resulting error in the low-frequency illumination is hard for the eye to see, especially when it is accumulated over many iterations.

In order to reduce overdraw, surfaces are rendered with culling enabled. This is valid if all objects are closed. If this is not the case, culling can be disabled and backfaces rendered with color 0 to allow backfacing casters to block light. Another way to reduce overdraw is to run the method on smaller regions of the scene. Within a region, the random depth is used; outside normal depth, comparison is used.

As generation of uniform random numbers is not directly supported on the GPU, we compute random depths by using a scaled world-space position plus a random CPU-generated offset (per iteration) as an index into a tiled 3D texture containing uniform random values.

Our implementation uses alpha to store depth in the illumination buffer, which precludes use of transparent surfaces. The technique could be extended to support transparency by rendering alpha into the illumination buffer as well (in a secondary texture); and then we would alpha blend the closest transparent surfaces with the closest opaque. However, this would require depth sorting of the sampled fragments, resulting in a more complex pixel shader, and we have chosen to leave out this extension in order to keep the implementation simple.

7. Examples

For evaluating the error imposed by our key assumption, we have constructed a scene containing a stack of 5 red and 2 white two-sided planes. Lighting the scene by a white skydome enables us to visually assess the amount of leaked red light onto the isolated top plane, caused by the (up to 7 times) overdraw in the illumination map. Running the scene with 5000 iterations and a (worst-case) 1×1 sampling filter (Figure 8(a)) shows a visible artifact, where the top white plane receives red color because of the overdraw. In comparison, the ground truth top plane has no red color bleeding. Figures 8(b) and 8(c) show the scene lit using 3×3 and 5×5 samples, respectively, together with the relative error computed as the measured red level in relation to the gray level. Thus, increasing the size of the filter reduces visual artifacts but also makes the resulting pixel shader more expensive and less precise unless the resolution of the illumination map is increased accordingly.

The technique has been tested on six scenes, ranging from 48 triangles to 135 K triangles. The results are shown in Figure 9. The tests were run on a 2.66-GHz CPU with an NVIDIA 8800 GPU, using DirectX 9. The scenes in Figures 9(a), 9(b), 9(c), 9(d), and 9(f) are lit using high dynamic range (HDR) light probes, whereas 9(e) is lit with a local area light. The illumination map was sampled using a 3×3 filter, which we found gave an adequate visual quality for the majority of the scenes. For the more complex Sponza scene (Figure 9(d)), a 5×5 filter was used. Except for 9(e), the total computation time was between 2 seconds and 17 seconds, using a float-16 illumination map of 512×512, and a 512×512 lightmap (1024×1024 for the Sponza scene). Hence, one global direction iteration takes 0.5 to 3 ms on average, making interactive illumination updates possible. Because 9(e) contains a bright local light source, a larger number of iterations is needed to capture the direct light

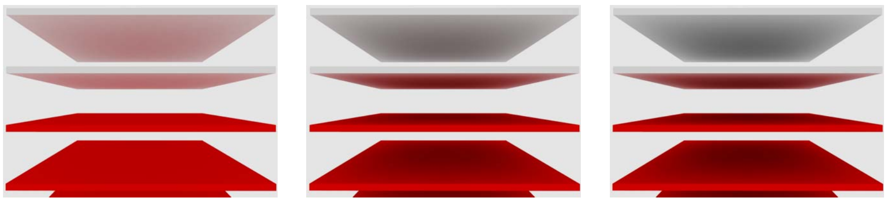
(a) (b) (c)

Figure 8. Error introduced by overdraw using three different filter sizes: (a) 1×1: 41% error. (b) 3×3: 8% error. (c) 5×5: 1% error.

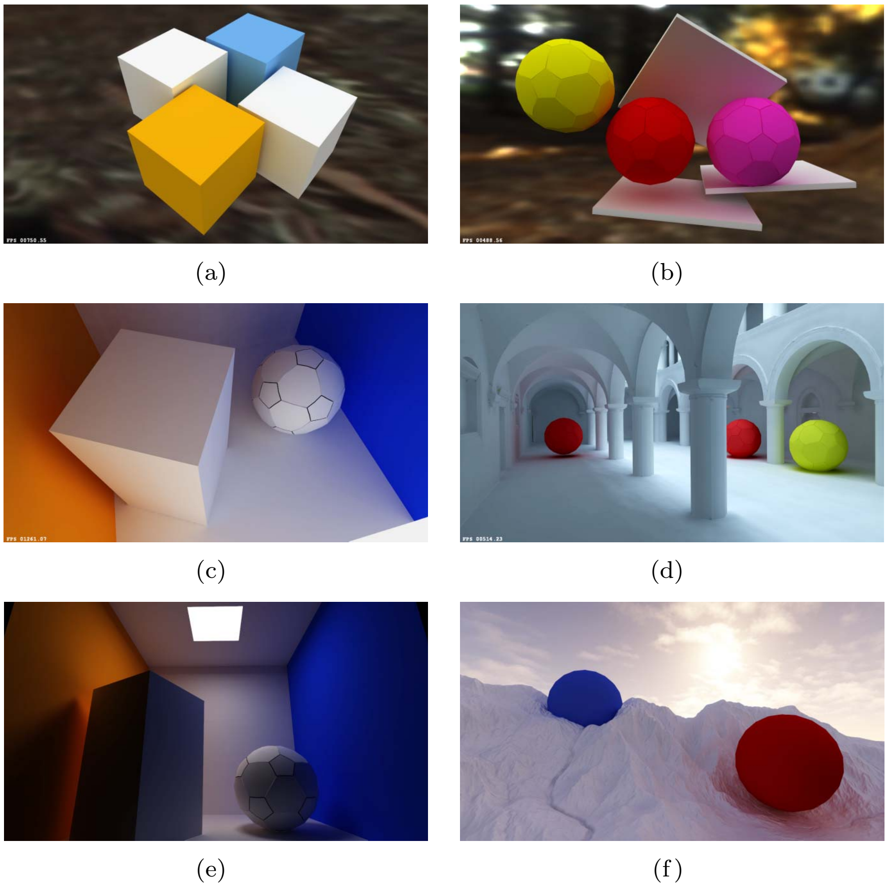

Figure 9. Example scenes: (a) 4 Cubes (48 triangles):5120 iterations, 5 seconds. (b) 3 Balls (3276 triangles):5120 iterations, 5 seconds. (c) Cornell scene (1102 triangles):4068 iterations, 2 seconds. (d) Sponza (79K triangles):5604 iterations, 17 seconds. (e) Cornell scene with local light (1104 triangles):42624 iterations, 30 seconds. (f) Landscape (135K triangles):3764 iterations, 10 seconds. (f) Landscape (135K triangles):1696 iterations, 5 seconds.

without visible banding. However, this problem relates to gathering in general, and is not a limitation imposed by our technique. One solution is to sample the direct light using shadow mapping. Moreover, a straightforward extension would be to split direct and indirect light, so direct light is computed using conventional hardware with standard light models, and indirect illumination is updated using our technique. This would also make it suitable for more dynamic worlds, where indirect illumination is updated at a lower rate.

The results are visually comparable to images produced using traditional radiosity methods. Our approximation works well in practice because of the low-frequency nature of diffusely reflected light. The technique has an upper bound for the number of texture reads, and uses only a single texture, resulting in good cache utilization and a small memory footprint. Thus, we believe that the low computation cost combined with the relatively low hardware requirements make the technique well suited for games and other real-time applications, for which computational accuracy is less important.

Acknowledgments. The authors would like to thank Morten Mikkelsen, Manny Ko, Jon Rocatis and Kasper Engelstoft for their helpful comments. Also thanks to Paul Debevec for his light probes. Finally, thanks to the anonymous reviewers for their constructive criticism.

References

[Bavoil and Myers 08] L. Bavoil, and K. Myers. “Order Independent Transparency with Dual Depth Peeling.” Technical Report, NVIDIA Open GL SDK, 2008.

[Enderton et al. 10] E. Enderton, E. Sintorn, P. Shirley, and D. Luebke. “Stochastic Transparency.” In *I3D ’10: Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2010).

[Everitt 01] C. Everitt. “Interactive Order-Independent Transparency.” Technical Report, NVIDIA Corporation, 2001.

[Hachisuka 05] T. Hachisuka. “High-Quality Global Illumination Rendering Using Rasterization.” In *GPU Gems 2*, pp. 615–634. Boston: Addison Wesley, 2005.

[Liu et al. 09] F. Liu, M. Huang, X. Liu, and E. Wu. “Efficient Depth Peeling via Bucket Sort.” In *HPG ’09: Proc. Conference on High Performance Graphics* (2009).

[Myers and Bavoil 07] K. Myers, and L. Bavoil. “Stencil Routed A-Buffer.” *Proc. SIGGRAPH ’07, ACM SIGGRAPH 2007 Sketches* 21 (2007).

[Sbert 96] M. Sbert. “The Use of Global Directions to Compute Radiosity.” PhD thesis, Catalan Technical University, Barcelona, 1996.

Web Information:

Paul Debevec light probes available at http://www.debevec.org

Andreas Thomsen, Reto-Moto, Vestergade 18E, 5th floor, 1456 Copenhagen K, Denmark (at@reto.dk)

Kasper Høy Nielsen, IO Interactive, Kalvebod Brygge 35–37, DK-1560 Copenhagen V, Denmark (khn@ioi.dk)

Received April 22, 2010; accepted in revised form August 29, 2011.