Instant Radiosity Implementation

Zahari Shoylev

Introduction

Rendering in Computer Graphics is the process of taking a geometric description of a scene as an input and producing an image output. Current computer system GPUs support large sets of fast two- and three- dimensional transformations and polygon visibility algorithms to provide real-time graphics rendering engines with the computing power to display virtual environments in real-time. Other, more comprehensive algorithms exist to process rendering, namely radiosity and ray-tracing algorithms, but these algorithms have been generally too complex and computationally intensive to implement on a GPU. Radiosity and ray-tracing implementations are orders of magnitude slower than algorithms that rely on simple three-dimensional transformations. However, images obtained by radiosity or ray-tracing rendering engines are more realistic.

Radiosity solves the global illumination problem of rendering images that contain both direct and indirect illumination information. Direct illumination is light that objects or surfaces in the scene receive from light sources. Indirect illumination is light that objects receive after this light bounces off or interacts with other objects in the scene. For example, if we have a white polygon bordering a red polygon at a specific angle, if we illuminate both and render them with radiosity, the white polygon may contain a shade of red near the border. This realistic effect of color bleeding appears because radiosity supports some aspects of indirect illumination – light reflected from the red polygon illuminates the white polygon.

Because radiosity is a solution to the global illumination problem, it also supports other realistic effects like shadows. However, unlike ray-tracing and specific shadow algorithms

like stencil shadows, it supports soft shadows. Hard shadows, as opposed to soft shadows, have no shadow gradient on their edges. Realistic images require soft shadows.

The original radiosity algorithm did not support specular light and reflections [Goral]. Unfortunately, reflections and specular lights can be considered an important aspect of indirect illumination. Depending on the specific radiosity algorithm, different ways to implement reflections exist. However radiosity does not support these natively as a part of the algorithm, and implementing specular light and reflections would necessarily be an additional step required to create realistic images when using a radiosity algorithm.

While creating realistic images is important, many rendering applications require real-time rendering. GPU rendering systems can create believable images in real-time. For example, a scene description may contain pre-processed radiosity and shadows data. The rendering engine then can embed this data into the final image by running specific vertex shader programs. However such methods place a lot of constraints on the dynamics of the scene, because the information needed to render such scenes is preprocessed.

Other methods focus on providing algorithms that would approximate the global illumination radiosity solution in real time. The “Instant Radiosity” algorithm [Keller] provides such an approximation. The goal of this paper is to show how to implement a version of this algorithm to take advantage of an average GPU.

Previous work

Most radiosity algorithms rely on a finite element method to approximate a solution to the global illumination equation. The algorithm divides scene polygons into smaller patches and uses the mathematical framework developed for calculating heat transfer between surfaces to determine the volumetric flow rate or flux of light in the scene. The algorithm only accounts

for diffused light – the major simplification is that it considers all polygons to be perfect lambertian surfaces [Goral].

The classic radiosity algorithm is iterative. It has to be invoked separately for each color band we are processing (i.e. for red, green, and blue). Processing each color requires multiple passes over all patches in the scene. Each pass processes all patches. For each patch, the algorithm determines how much light the patch receives from other patches (or how much light the patch emits to other patches). Each pass provides a better approximation to the rendering equation. In a progressive radiosity algorithm, eventually light flux emitted or received would fall below a cut-off value and the image can be rendered from the color values of the patches [Cohen1].

The original algorithm presented by Goral et al describes the form factor analysis, patch flux calculation, and additional implementation details. The algorithm divides quadrilateral surfaces into relatively small user-defined total number of uniform patches. A form factor is computed between every two patches. The form factor reflects how much light flux is transferred between the two patches, and is calculated geometrically. In this implementation there are no hidden surfaces or occluding patches, so only the patch orientation is important for calculating the form factors. After this pre-processing step, the algorithm feeds the form factors into a matrix form of the equation solving the total patch to patch light flux in the scene. The matrix can then be solved with any matrix solver. The original implementation used Gaussian elimination. Once the matrix is solved, the correct color intensity values (per color) can be assigned to each patch. An additional pre-rendering step to enhance the final image quality is to linearly interpolate the visible patch polygons and smooth out the intensity differences between patches [Goral].

Cohen et al proposed multiple improvements to how the original radiosity algorithm handled form factors by introducing the hemi-cube. In essence, each patch is represented by its center

point. The "view" each patch has on the scene is a hemi-cube placed on top of the patch in the direction of its normal. The hemi-cube's surface is covered with discrete pixels with pre-determined form factors. Every other patch in the scene is then projected on the hemi-cube's pixels using a z-buffer algorithm to determine visibility. After determining that a patch is visible from the cube, the algorithm will sum all the pre-computed form factors for the pixels this patch projection covers. This determines the form factor between the center point of the patch and every other patch. This summation has to be done for every other patch. Also, the hemi-cube and form factor calculation has to be done for every patch in the scene. This is very intensive computationally, but can handle occluded patches, shadows, and very complex environments [Cohen2].

Coombe et al suggested that the hemi-cube form factor computation can be done almost exclusively on the GPU. When used on modern systems, this GPU-accelerated radiosity would be able to render realistic images in real time. Progressive radiosity on graphics hardware simulates the conventional uniform patch subdivision algorithm. However, instead of using geometrically subdivided patches, the algorithm uses texels as the patch elements, similar to what Heckbert [Ward] proposed. Unfortunately, using textures to compute the outgoing light flux would not be possible in graphics hardware – it would require the GPU to write data to arbitrary locations in arbitrary textures. Instead, the algorithm implements a "gather" approach. The system renders the scene from the point of view of each texel (texture and patch element). A modified hemi-cube method uses a vertex program to calculate the projection of remote patches. To compute visibility and form factors, the algorithm only has to render the scene once from the point of view of each patch. The algorithm iterates over all patches [Coombe].

Keller proposed an alternative method to create realistic images with color bleeding effects and soft shadows, using a simpler algorithm to approximate a radiosity solution in real time.

Keller's method [Keller] generates a discrete particle approximation of the radiance in the scene with a quasi-Monte Carlo integration [Morokoff][Niederreiter][Keller2]. The method produces a number of attenuated light particles that can be directly rendered in hardware as point light sources. This results in a very robust algorithm that can produce realistic images even with a small number of light points – usually 30 to 500. Since most of the computationally intensive part of the algorithm is computed in hardware, the algorithm heavily relies on the GPU speed. Also, unlike the original radiosity algorithm, which has quadratic run time, Keller's instant radiosity does not depend on form factor calculation and does not require solving the radiosity equation, thus making the algorithm's run time linearly proportional to the number of light particles in the scene – a value that can be adjusted by the user. While this algorithm generates believable images in real-time, it is not as physically correct as other, slower radiosity algorithms. Keller's algorithm is best used for interactive walkthroughs, dynamic scenes, and as a fast radiosity preview.

Instant Radiosity

This paper discusses implementing the Instant Radiosity algorithm [Keller] on a PC system utilizing the GPU's processing power for most of the algorithm's computation. The paper covers specific implementation details and important speed-up considerations. This implementation uses the Ogre Eihort SDK, as well as other open-source architecture-independent tools to maximize the optimization gains, such as Tiny XML.

Radiosity algorithms operate with the assumption that surface elements within the scene are ideal lambertian surfaces – perfect diffuse reflectors and emitters. This means that light energy incident to a surface is scattered equally in all directions. Thus, regardless of positioning, a surface would seem to emit a flux of light. Let Bj be the energy leaving a

surface. Then Bj = Ej + Pj * Hj [Goral] where Ej is the rate of energy emission, Pj is surface reflectivity, and Hj is incoming light energy. Also, Hj = sum(i=1..N, Bi*Fij) where Fij is the form factor between surfaces i and j. Note that the human eye sees intensity, so in a radiosity algorithm we have to divide the light energies by Pi and work with the modified equations [Goral].

By using a Monte Carlo simulation, the instant radiosity algorithm uses a quasi-random integration scheme to solve the radiance equation described by Goral et al [Keller]. Thus the algorithm avoids solving the matrix form of the equation. The quasi-random integration uses low discrepancy sampling as introduced in [Keller2]. The instant radiosity algorithm uses deterministic light particle simulation to avoid intermediate solutions of the radiosity equation and to immediately render an image in hardware. The algorithm simulates the behavior of diffused light particles. It projects a user-determined amount of light particles in a scene by using low discrepancy sampling. The particles represent light flux intensity. Once the algorithm places all light particles within the scene, it renders the scene with shadows once per light particle, using the particle as a point light source. Each image represents the light flux emitted from the point on the surface where the particle is attached. All images are accumulated in hardware to produce a single final image. Increasing the number of particles provides a better radiosity resolution and physically correct images, but slows down the algorithm. Usually a range of 50 to 300 light particles will produce realistic images [Keller].

The algorithm uses the Halton sequence for generating the low discrepancy sampling points [Halton]. This method can generate quasi-random numbers as fast as a regular random number generator can generate random (or pseudorandom) numbers. While less random, the points will have low discrepancy and will be uniformly distributed [Halton]. Thus the quasi-Monte Carlo integration implemented in the algorithm would provide smoother convergence at a slightly faster rate.

The Halton sequence for an n-dimensional point Xm = (H(2,m), H(3,m), H(5,m), … , H(Pn, m)) where Pn is the n-th prime number. H(p, m) is the radical inverse function of m to the base p. The value is obtained by reflecting the digits of m written in base p around the decimal point. The function returns quasi-random numbers in the range [0..1] [Halton]. The instant radiosity algorithm works with quasi-random 2-dimensional points, so we will be using H(2,m) and H(3,m). Since we work with a number of points (N>30) much larger than the dimensionality of the sequence, the sequence is guaranteed to provide uniformly distributed quasi-random 2-dimensional points [Niederreiter].

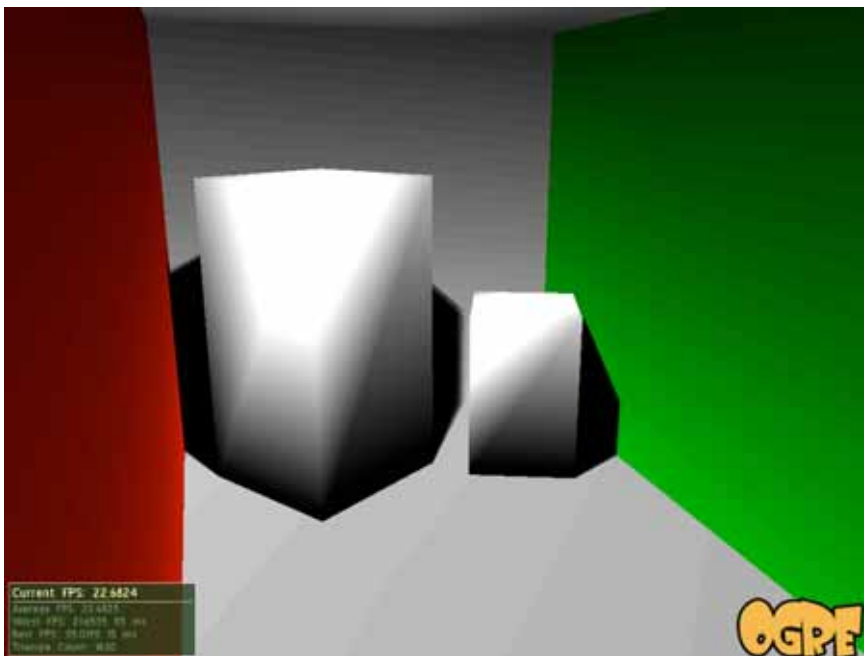
The sequence is relatively easy to implement in C++. The core of the algorithm is as follows:

```
double H(int p, int m) {
double x = 0.0, f = 1/p;
        while(m)
        {
                x += f * (double) (m % p);
                m /= p;
                f *= 1/p ;
        }
return x;
}
```

This provides a direct calculation of the radical inverse function. Incremental calculation is also possible [Keller].

To implement the instant radiosity algorithm, the rendering system has to support a scene description language and loader. My implementation uses the Ogre 3d SDK for rendering

purposes and scene graph management. Ogre has extensions for almost any commercial design and modeling product. The Cornell Box scene the program currently uses was created as a 3d Studio Max file, and then exported to an Ogre XML file using the Ogre exporter tool for 3d Studio Max. Ogre SDK then provides tools to compile the XML file into a binary that can be directly loaded into memory, using the SDK's data structures and API. My implementation relies on being able to access both the compiled scene binary and the scene XML file. Using Tiny XML, the Instant Radiosity Implementation extracts all relevant data, such as surface geometry, color, and light source location. It uses this information to update the scene representation in memory that the Ogre SDK has created from the binary scene file.



The Cornell Box rendered in Ogre Eihort, using Instant Radiosity; Soft shadows visible on the back walls

The rendering system in my implementation supports two predefined variables. N is the number of point lights attached to light surfaces in the scene. p is be the average scene diffuse reflectivity. In realistic scenes, p is be on the order of 12% to 60%, 35% in the Cornell Box scene. While it is possible to dynamically calculate the value of p, the system applies a user-defined value. N, by definition, is also a user-defined value. Note that N indirectly represents

the number of sampling light particles the system uses to do carry out the quasi-Monte Carlo integration of the radiosity equation. A larger N results in a more physically accurate model, but requires more rendering time, as it increases the number of point lights in the scene.

First, the Instant Radiosity Implementation loads the binary scene in memory. The system does not modify the scene geometry as represented in this file. The system only adds a number of point light sources in the scene in such a way that the scene is illuminated with a radiosity global illumination lighting model. After the scene is loaded, it can be directly rendered in hardware. Since we are using the Ogre SDK tools to compile the binary scene file, we know that the binary file is optimized for rendering.

To be able to determine the location of the point lights, the system has to load the XML file representing the scene geometry. This is the XML version of the binary scene file. We obtain the XML file by running an Ogre converter tool on a 3d Studio Max scene file. The binary file Ogre uses for rendering is the compiled version of this XML file.

3d Studio Max Scene File -> Ogre XML -> Ogre Mesh

The Instant Radiosity Implementation uses Tiny XML to load the XML file. Once loaded, the system identifies the location and geometry of the first light source, and consequently, every light source in the scene. This is possible because light sources in the scene have specific material names. The system can successfully enumerate all light sources in the scene.

Then the system runs the quasi-random walk preprocessing part of the algorithm [Keller]. It places N point lights on the light source surfaces. For example, in the case of a single rectangular light source surface, the Instant Radiosity Implementation will use an isometric projection of 2-dimensional quasi-random Halton points from the unit square to the rectangular light surface [Keller]. Thus, the system attaches N point lights on the surface of the rectangular light source. The point light color values are then the same as the light source on which the point light is placed.

From the initial N point lights, the system emits p*N rays in quasi-random directions. The rays originate from the first p*N point lights on the light source [Keller]. The Instant Radiosity Implementation ray-traces these rays to find the p*N points in the scene where the rays intersect the scene. A simple ray-tracing algorithm can be implemented to calculate triangle-ray intersection. The algorithm can benefit from common ray-tracing optimizations such as an octree bounding box hierarchy algorithm. However, since we are currently only interested in performing the ray-tracing operation only once, as a pre-loading step, such optimizations are not necessary. The implementation simply uses the XML scene description to find the intersections between the triangular polygons in the scene and the emitted rays.

After the system identifies the p*N intersection points, the system attaches p*N point lights at these intersection points. The system needs to obtain the color and diffused reflection values of the surfaces on which these points of intersection are located, which is readily available from the XML scene description. The color of the surface is assigned as the light color value. The intensity of the point light is the intensity of the originating point light reduced by the diffused color reflection value of the surface.

From these new p*N point lights, the system emits p*p*N rays in quasi-random directions outwards from their respective surfaces. Then the system assigns more point lights in the scene at the new intersection points. This step is repeated until p*p*…*p*N<1. The implementation thus creates a total of M point lights in the scene. For example, the value of M with p=60% and N=60 would be 120 total point lights.

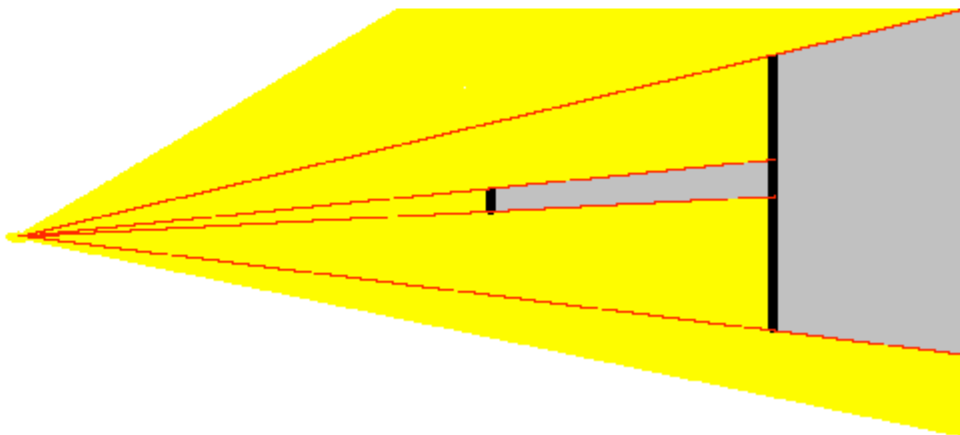
Notice that since we assume perfect lambertian surfaces, the outgoing light flux from any point on a surface is be the same regardless of direction. Our model approximates this well, as the point lights emit the same flux intensity in any direction. Also, this model is a good approximation of the particle behavior of diffused light [Keller]. By using point lights, the algorithm provides a quasi-Monte Carlo solution to the integral radiosity equation by directly

sampling the light flux intensity in the scene [Keller2], bypassing the geometry-derived form factor equation.

The system adds all the point lights in the Ogre SDK scene description in memory. Since all the mesh data is already loaded, the implementation only has to add the lighting information before it can start rendering the scene. Once the point lights are added to the scene description in memory, the Ogre SDK lighting model is set to use additive stencil shadows. Stencil shadows gives the most correct physically point light shadows [Crow], as it projects the shadow volumes for every light point. Using an additive lighting model allows the system to combine multiple point lights in hardware with an accumulation buffer. Since all the rendering is done in hardware, even old hardware can manage over 5 frames per second with M=60. Also, many modern GPUs support volume shadows as a hardware stencil buffer implementation [Everitt].

The system must be capable to render physically correct shadows, preferably in hardware. The shadow algorithm this implementation uses is additive stencil volume shadows.

Every shadowed scene has occluders and shadow receivers:



Volume shadows as extruded silhouettes; Point light illuminating the scene from the left

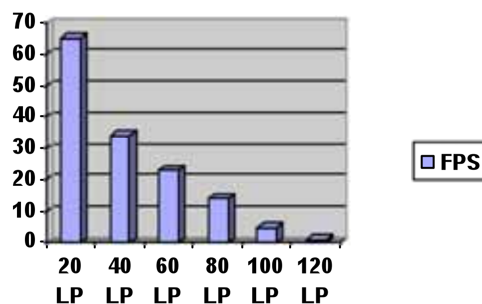
The essence of the algorithm is to draw shadow volumes – projections of the contours of the occluders from the light source onto infinity. The rendering system uses a vertex program to extrude the contour's vertices to a finite or infinite distance [Everitt]. The vertices are

extruded away from the light source, in the direction of the light rays. After the vertex program constructs the shadow volumes, the system renders them on the stencil buffer only (by disabling the color and depth buffers) with back-face culling turned on. Also, the stencil operation for the stencil buffer is set to increment on depth-pass. Since the GPU thus only renders front faces, this operation counts the shadow volume front faces per pixel in the stencil buffer. After that, the system switches to front-face culling, and sets the stencil operation to decrement on depth-pass. The system renders the shadow volumes again, counting back faces and decrementing. After these operations, a 0 pixel in the stencil buffer will correspond to a lit surface. This specific algorithm is known as depth-pass or z-pass. This algorithm will work with a multiple number of shadow volumes [Everitt]. With the use of an accumulation buffer, this operation can be extended to multiple light sources as well.

Additive stencil volume shadows as implemented in the Ogre SDK

Unlike other systems, the Ogre SDK will allow any number of point light sources within a scene when the options “iteration once_per_light point” and “scene_blend add” are used in the material script in conjunction with stencil shadows. Also, Ogre can optimize rendering to match the GPU capabilities. By using an additive lighting model and options mentioned above, the instant radiosity rendering system only has to specify the correct light intensity and color values per point light, as described above, and then add the point lights to the scene description. Since the scene description is a precompiled binary already in memory, the only processing step when loading the scene is to calculate the point light values.

Analytical results



Frames per second per point light on an ATI Radeon 9200se using the Cornell Box scene

The algorithm’s run time is linearly dependent on the number of point lights in the scene N, the user-defined quasi-Monte Carlo sampling resolution. Increasing the number of light points by n will require no more than n additional images to be rendered, before the final image is ready. However, runtime also depends on scene complexity. Since extruding contour vertices for stencil volume shadows is a time-consuming operation that has to be repeated multiple times for each point light, and since the runtime of this operation depends solely on the geometric complexity of the scene, increasing the number of polygons in the scene while keeping N constant will linearly increase the run-time. If we increase the number of polygons

by l, potentially N*l more shadow volumes will have to be constructed to render an image. Notice that this linear correlation to scene complexity is much better than the quadratic relationship between scene complexity and runtime in classic radiosity algorithms [Keller].

Future work

At low sampling rates images experience aliasing issues. To solve this, while also slightly improving convergence rate, Keller introduces the concept of jittered low discrepancy sampling for Instant Radiosity rendering. By looking at the two dimensional radical inverse function, it is obvious that the points generated by it lie within a grid. While the grid guarantees a uniform distribution, it is prone to aliasing problems [Keller]. If we approximate the grid resolution to 1/N, we can "jitter" each quasi-random point by a small amount. We can replace H(2,m) by H(2,m) + e/N where e is a random variable. This assures the sample will remain within its square of the grid, maintaining the property of uniform distribution, but will also randomize the sampling to reduce aliasing problems. In this specific implementation, different quasi-random points have to be jittered differently. The first N initial points attached to the light source are jittered by e/N. The next p*N points are jittered by e/(p*N). Aliasing is reduced, while convergence rate slightly increases. [Keller]

It is relatively easy to introduce specular highlights as an extension to the Instant Radiosity rendering system. First, we turn back on the full BRDF when rendering, enabling specular highlights. Then we introduce an extra step in the pre-processing phase, when we create the point lights and attach them to the scene. By a random decision, each surface hit by a ray is determined to be either diffuse or specular. If we have decided on a specular reflection, the ray is mirrored, and a spot light is created. This light only illuminates the specular surface that mirrored the ray. Specular light particles increase the total number of lights required to render

the final image. The Instant Radiosity rendering system would rely on hardware spot lights to render these “specular” lights. While this approach would create realistic specular highlights, it is insufficient to create specular reflection of objects. An additional ray-tracing pass would be required [Keller].

In the current implementation, the algorithm does not support real-time walkthroughs. However, modifying the system to handle dynamic scenes is relatively easy. In a dynamic scene, the system will limit the quasi-Monte Carlo sampling rate to a fixed number of N sampling paths. We accumulate the last N paths sampled into N images with generation time recorded [Keller]. The newest path to be traced and accumulated in an image will then replace the oldest image in the set of N images. Then, the N images are accumulated and averaged.

References

[Goral] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg and Bennett Battaile, *Modeling the Interaction of Light Between Diffuse Surfaces*, Cornell University, Ithaca, New York 14853

[Cohen2] M. Cohen and J. Wallace, *Radiosity and Realistic Image Synthesis* Academic Press Professional, Cambridge, 1993

[Cohen1] Michael F. Cohen, Shenchang Eric Chen, John R. Wallace, Donald P. Greenberg, *A Progressive Refinement Approach to Fast Radiosity Image Generation*, Program of Computer Graphics, Cornell University, Computer Graphics, Volume 22, Number 4, August 1988

[Ward] Gregory J. Ward, Paul S. Heckbert, *Irradiance Gradients*, Third Eurographics Workshop on Rendering, 1992

[Coombe] Greg Coombe Mark J. Harris Anselmo *Lastra, Radiosity on Graphics Hardware*, Department of Computer Science University of North Carolina

[Keller] Alexander Keller, *Instant Radiosity*, Universit¨at Kaiserslautern (SIGGRAPH 97)

[Keller2] Alexander Keller, *Quasi-Monte Carlo Radiosity*, Rendering Techniques ’96, Proc. 7th Eurographics Workshop on Rendering, pages 101–110

[Halton] John Halton, GB Smith, *Algorithm 247: Radical-Inverse Quasi-Random Point Sequence*, Communications of the ACM, Volume 7, 1964, pages 701-702

[Crow] Franklin C. Crow, *Shadow Algorithms for Computer Graphics*, University of Texas at Austin, Austin, Texas (SIGGRAPH 77)

[Everitt] Cass Everitt and Mark J. Kilgard, *Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering*, March 12, 2002, NVIDIA Corporation, Copyright 2002, Austin, Texas

[Niederreiter] Harald Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods,* Society for Industrial and Applied Mathematics, 1992

[Morokoff] William J. Morokoff, Russel E. Caflisch, *Quasi-Monte Carlo Integration*, Journal of Computational Physics, Volume 122 , Issue 2 (December 1995), Pages: 218 - 230