Spatial Global Illumination

Michael H. Auerbach, University of Maryland 2011

Spatial Global Illumination is a technique for rendering spatially approximated photo-realistic lighting of fully dynamic scenes at unusually fast speeds. Spatial Global Illumination offers an excellent compromise between quality and performance making it a highly viable solution for next generation video games.

Spatial Global Illumination (SGI) was developed by Michael Auerbach [Auerbach 2011] as a proprietary global solution to be used in an upcoming engine. Research was preformed over many years beginning in 2008. Today, Michael is working with 3d artist Jordan Gabrielle and Dexsoft Multimedia to integrate his research into a functional game engine for use in future project. In the following chapters, Michael describes the benefits and limitations of SGI as well as implementation and research.

Origin

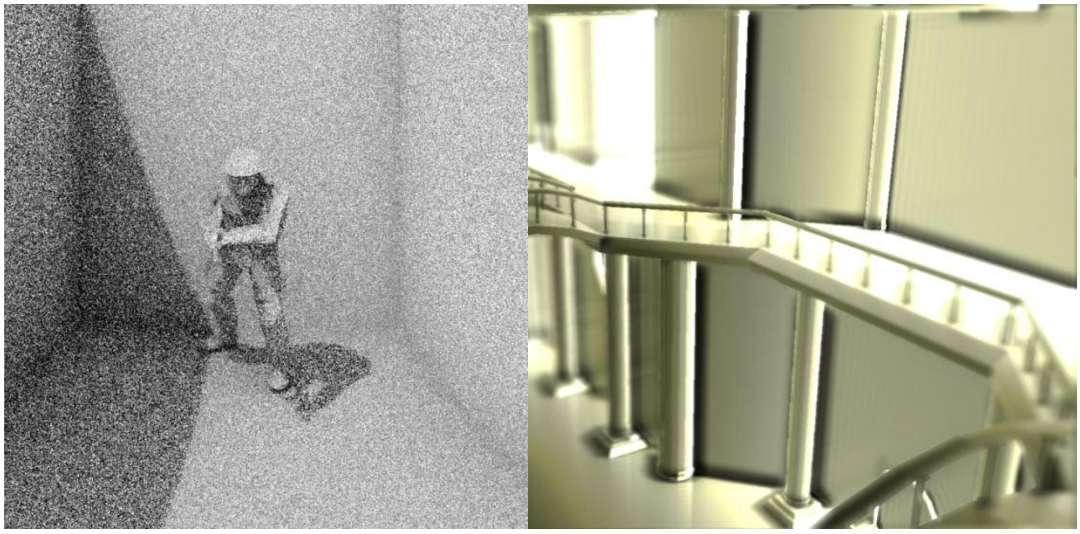
Global Illumination, specifically diffuse reflections and radiosity lighting, is an essential asset to portraying mood and spatial perception in games. However, due to the complexity of simulations creating such phenomena, many game developers omit this important aspect of scene lighting to provide faster rendering times. When we were in the design stages of our engine, we were split between using a more accurate static radiosity solution (inspired by Valve) and a fully dynamic lighting model (inspired by Crytek). We experimented with various methods over a couple of years, including ray-tracing, screen-space methods, analytical solutions, light propagation volumes, and other various techniques.

Scene from "Portal2" courtesy Valve Corporation – the radiosity lighting model used in Valve's Source engine gives their games a unique feeling and induces mood and emotion; two elements that we feel are vital to immersing the player into the environment and captivating their imagination.

Unfortunately, none of these solutions met our standards. Either the method was too slow for real-time use or the method was too limited in terms of quality and effective output.

For a while, we decided to use static global illumination, and began experimenting with some sort of PRT system. Unfortunately, we found that this system used a large amount of memory over large scenes, and did not recreate the "feeling" or radiosity lighting that we desired. Furthermore, the system was very limiting as it only supported static geometry, something that many modern games are moving away from (source, BF3, dynamic destruction). We wanted a solution that would offer true radiosity results while remaining dynamic enough to be affected by dynamic objects and procedural destruction.

At the time, we were also developing a new algorithm for rendering shadow maps which involved grouping objects spatially to extract penumbra delta information. This technique required us to create a very crude representation of the scene in order to accelerate shadow map generation. After a lot of testing, we found that creating a dynamic voxel map containing the scene geometry was the most efficient way to represent the scene in this case. We now had an engine that was creating a dynamic voxel map representation of the scene every frame. We realized that we could take advantage of this to create world space ambient occlusion and real-time reflections, and later, simulate radiosity lighting.



Original concepts such as real-time ray tracing (left) and LPV (right) were eliminated as efficient solutions due to poor performance and/or quality.

Scene Voxelization

Because scene voxelization has already been well optimized and discussed in (source), we will omit the general details of scene voxelization in general. However, we would like to present some "best practices" that work to improve SGI quality and performance.

First, we must consider the IA stage and the geometry we feed to the GPU. Although highly detailed geometry can be efficiently voxelized on modern hardware, any details smaller than one voxel will not be noticed. Therefore, we found it to be most efficient to only voxelize the LOD models of large objects and the bounding boxes of small objects. Effectively we save a large amount of bandwidth between the CPU and GPU and reduce both rasterization time and GPU VRAM usage.

Another optimization is to voxelize the scene without textures and store the averaged object colors per-vertex or even per object when possible. This reduces "flickering" caused by rasterization of objects smaller than a texel/voxel and extends to the concepts of texture filtering.

We must also consider voxel map size. Since we keep our voxel map centered around the player camera. We found that, for most scenes, we did not need a very large voxel map to recreate quality results. For a voxel size of 1 meter cubed, we found that a 128x128x32, 128x128x64, or even 64x64x32 voxel map worked perfectly, with results outside this range falling back to SSAO/SSGI (which at that distance can actually cover quite large areas). We also found that using cascaded voxel maps works quite well and we are currently putting more time into researching this.

Finally, we found that by snapping vertices to the regular grid of voxelization, we can further stabilize the rasterization process. In order to deal with fine or non-evenly distributed models within a voxel, we can simply store the polygons moment via the geometry shader.

Figure 1A – voxel map w/out optimization - 8ms

Figure 1B – Voxel map w/ optimizations - 3ms

Figure 1 shows how important these optimizations are in terms of quality and stability. Notice that the wholes present in 1A are eliminated by snapping the polygon vertices to the regular grid. Additional geometry, such as polygons perpendicular to the voxelization camera are also captured. Finally, we see that the use of LOD geometry in 1B presents more stable results at a fraction of the time required to voxelize full detail models.

Light Injection

Once we have obtained a 3d texture filled with a voxelized representation of the scene, we must consider how to use this data effectively to produce high quality global illumination.

We propose a method that takes parts from Light Propoagation Volumes (Crytek 2011), classical photon-mapping (Jensen 96), PRT (source), and general mathematical recursive convergence of signals based on spherical harmonic light direction and ray-marched occlusion. In general, we generate we create a second 3d texture to represent radiance values distributed on our regular voxel grid throughout the scene. Usually we found that a 3d texture with the same dimensions as the voxel map works best as voxel sizes smaller than the scene voxel map proved to add little quality gain for such a

performance hit. On the contrary, using larger light voxel sizes compared to the scene voxel size created results that were too low quality for our goal.

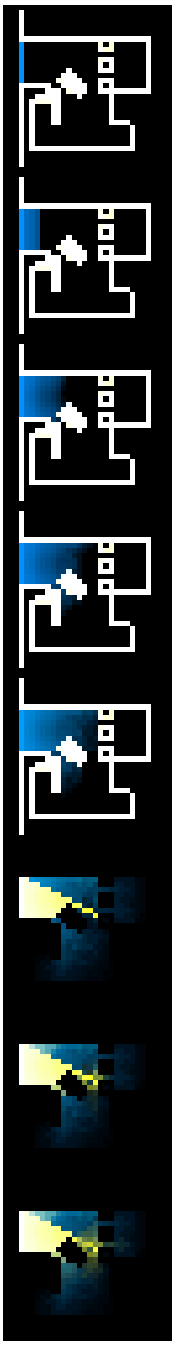
To make SGI independent between scenes (to improve SLI/crossfire performance (source) and for highly volatile scene geometry), we recreate the entire lighting simulation per-frame. At the start of each frame, we clear the 3d texture to black (0,0,0) and inject lighting data into the volumes. However, unlike LPV, we do not need reflective light maps for this process, thus making light injection significantly faster. Also, to makes sure that only geometry radiates light back into the scene (and not empty space), we only inject light into voxels that have their correlated position in the scene voxel map occupied. Simply, we cull each light per voxel and for each light we sample n texels from the shadow map and compare to sampling offset position. This way we can reduce "flickering" caused by shadow map aliasing and dynamic scenes. Often 3x3x3 samples is adequate for our use, however, this greatly varies on the voxel dimensions and shadow map granularity used. For shape and area lights, we simply voxelize their meshes and add their light color to the 3d light voxel map. For faster performance, you can also skip the shadow map sampling and inject the light source as a point into the 3d light voxel map. For lights that influence the scene over greater distances, such as the sun, sampling the shadow map is our best option, however, using a coarser shadow map is best (ideal if you are using cascaded shadow maps for the sun). Also, for skylight we can skip light injection altogether and simply add an ambient occlusion value via backwards ray-marching. Alternatively, you could render an isometric shadow map perpendicular to the ground plane as an estimate to skylight to be used in light injection.

Basically, this process is a simplified photon mapping scheme that uses both forward and backwards ray-tracing/ray-marching to converge global illumination results. Note that the light injection step is collecting radiance information directly from light sources and storing it spatially into a voxel

map. To complete the simulation, we then preform voxel-to-voxel tracing around the scene, and finally a forward ray-trace via rasterization to gather the lighting data.

Radiosity Simulation

While the light injection stage is somewhat similar to that proposed b Crytek (LPV Crytek 2011) less the reflective shadow maps, the Radiosity Simulation differs greatly from radiance propagation and is more similar to classic radiosity simulations and photon mapping.



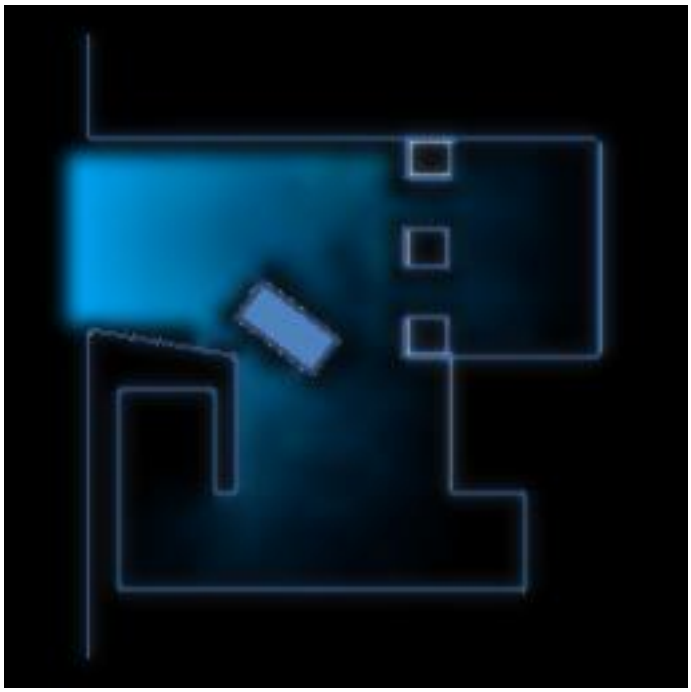
At each voxel, we are interested in gathering the light emitted from visible surrounding geometry by ray marching through the scene voxel map. If the ray collides with a occupied voxel, we add the corresponding radiance value in the light voxel map weighed by the trace distance. If we solve this problem recursively, we can achieve multiple bounce radiosity. However, at the time of this writing, this requires way too many texture fetches (in the order of thousands per voxel) and is unacceptable for current hardware.

To solve this issue, we propose using intermediate results that are represented spatially throughout the voxel map. Instead of completing each ray march in a single pass, we can complete each ray march over numerous passes, thus lowering the ray-march distance while maintaining recursive radiosity lighting results. However, care must be taken to manage the per-pass results as to not bleed radiance values from invalid sources into our selected ray. If all rays that intersect are allowed to exchange radiance information, then we simply have an overall effect that looks very similar to LPV and does not accurately portray secondary shadows and other important global illumination effects. In other words, we cannot simply blur the light volumes (even selectively) as in LPV. To the right we see the results of each radiosity simulation pass.

The solution to this problem is to store, at each selected voxel, the endpoints of each active ray as well as the collected radiance data. This concept is similar to cone mapping (source) as it can continue each succeeding pass's ray-march at the closest voxel in a direction that is visible to the selected voxel. It should be noted that, as LPV converges outward from the source voxel, we are in fact converging inward, but using a backwards ray-marching to arrive at the radiating voxels' positions. The benefit of this is that, unlike LPV, only light volumes that are within geometry occupied voxels may contribute radiance information to the source voxel. This creates a more physically correct algorithm and a much higher quality result. For example, in SGI, voxel located right in front of an area that is in the sun does not necessarily gain any radiance value from the sun. In LPV, this is not the case, and the voxel would become brighter as a result of being close to the nearby sunlit area.

Images (left to right): slice of the light voxel map using LPV, slice of the light voxel map using SGI, difference between the two where red is a larger difference. Notice that using LPV, light does not converge throughout the entire scene and does not cast secondary shadows.

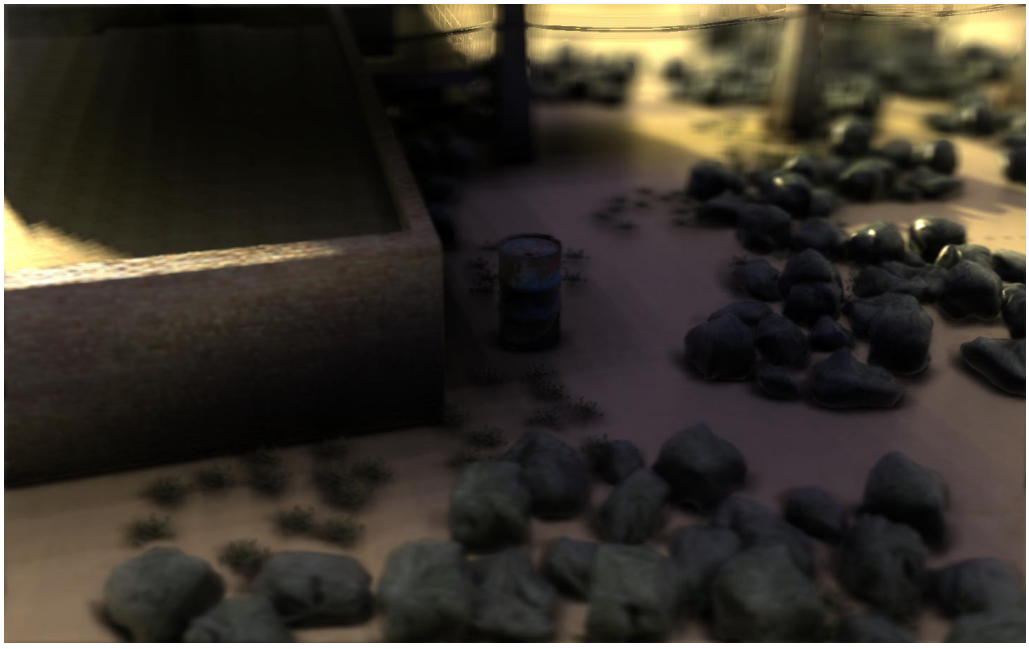
To the right we see a light voxel map slice showing the final results of a parking garage scene lit with only skylight. The upper left corner of the slice is connected to the outside and is the only source of light for the entire scene. Notice that the radiosity simulation bounces the light across a very large area. Secondary shadows and other global illumination phenomena are created procedurally by the spatial radiosity simulation.



Scene convergence

After the radiosity simulation, we add the indirect illumination approximation to the scene. We apply all indirect illumination terms per pixel. For diffuse light reflection, we simply lookup the light voxel map at the location of the texel being shaded with an offset in the texel's normal direction. We scale this offset inversely with the material's "softness" parameter to simulate subsurface scattering. Effectively, soft materials gain more uniform indirect illumination in all directions as harder materials will appear to have more defined normals simulating a low level of sub-surface light scattering.

To simulate dynamic reflections at each texel, we calculate a reflection ray about the texel's normal from the camera. For high quality settings, we actually preform a ray-march in this direction, and add the light voxel map's color at the collision point to the texel's specular term. For lower settings, we omit the ray-march and simply scale the reflection ray by a artist defined length and add the light voxel map's color at that location to the texel's specular term. For sharper reflections, you can raise the resulting color value to the nth power. For more blurry reflections you can average the specular term with neighboring light voxel map values around the traced (or approximated for lower settings) location.



Scene using SGI – the light voxel map is augmented with the scene

Performance:

Below we see a chart containing the average maximum allotted frame rendering time in our upcoming engine. Note that the left column is set to extreme quality and the right column is set to minimum quality.

Single Pass G-Buffer w/ tessellation	Single Pass G-Buffer w/
Shadow Maps 10 ms	Shadow Maps 4 ms
SSAO 2 ms	SSAO 1 ms
Scene Voxelization 3 ms	Scene Voxelization 2 ms
Spatial Radiosity 5 ms	Spatial Radiosity 1 ms
SGI <1ms	SGI <1ms
Deferred Lighting 4 ms	Deferred Lighting 2 ms
SSR + Spectral 5 ms	SSR + Spectral 2 ms
Post Processing 5 ms	Post Processing 1 ms
23 FPS	60 FPS

We can see that SGI is actually quite efficient, only requiring less than 9ms per frame to render extremely high quality results. SGI is slower than SSAO and other static techniques in current deployment, such as Enlighten. However, the ability to render photo-realistic lighting in fully dynamic scenes with no pre-computation is very powerful. Artists can update the scene in real-time cutting down on development time. Additionally, by removing the limitations of static geometry, SGI can be used in applications that require procedural destruction and complex dynamic worlds. Additional uses include accelerating CAD to provide more realistic feedback to architects and artists in real-time. Further research should be placed into comparing the results of SGI with ray-traced CAD results.

SGI requires a significant amount of Video RAM to store the voxel maps and uses a large amount of internal GPU bandwidth. However, In my last paper, “Optimizing Screen-Space Ambient Occlusion:

DFSSAO," I show that modern GPUs have an excessive amount of internal bandwidth compared to previous generations and that bandwidth is increasing exponentially. Large numbers of texture fetches, on the order of 250 per pixel, are readily available in less than 1 millisecond on current generation hardware [Auerbach 2011]. Our implementation of SGI does not exceed even half of this limit, and thus, the large number of texture fetches used to render SGI does not cause any performance loss on modern hardware.

Conclusion

SGI is an efficient and powerful rendering technique that solves the indirect lighting equation spatially to increase performance for real-time applications. SGI works with completely dynamic scenes which decreases development time and streamlines content creation by eliminating pre-processing. SGI enables complex physical interactions between objects and procedural destruction to be rendered with photo-realistic lighting previously only available for static geometry. As a result, SGI is a highly viable global illumination solution for game developers, combining full dynamic lighting with an immersive photo-realistic feel.

A typical scene rendered with minimum settings in our engine (top). Note the world space ambient occlusion, radiosity lighting, and real-time reflections created using SGI. The scene also feels ambient and delivers emotion and mood to the player. Scene rendered at 140 fps on a gtx570 w/ only sunlight and skylight. Bottom is a different scene at medium settings rendered at 75 fps at 1200p on a gtx570.