<< RETURN

Real-time global illumination

Eskil Steenberg

6th February 2004

Abstract

In this article we discuss the problem of global illumination. We discuss various observations of the problem to explore new approaches of solving it. Then we take thees observations and derive a number of practical examples how global illumination can be emulated in real-time rendering engines that can run on current level of hardware. The article discusses everything from simple minor changes to more drastic designs, but all are designed to fit in to existing technology.

Observations

In order to reach our goal we should first make a few observations about our problem. This will form the basis for a "tool box" of techniques that can be used to improve graphic engines.

It can't be done

If we consider the vast amount of photons that leaves even the most tiny of light sources, and how each photon can in theory bounce an infinite number of times against surfaces, and then the likelihood of hitting the tiny surface that is your retina, we quickly realize that no computer is even remotely close being able to simulate reality. So we can stop looking for that next generation CPU, graphics processor or feature. There is no silver bullet.

This however gives us the right to cheat. Even people who write off-line rendering engines cheat, so why shoulden't the real-time community cheat? This also states that anything is fair game as long as the result looks good. Actually global illumination is all about getting away with cheating. This is where the fun begins.

Please define real-time

If you ask people to define real-time you get many different answers. A film person may say 24fps, while an arcade person may say 60fps, and a professional player may not be satisfied until the frame rate is way above 100fps. In reality this is because different parts of the brain have different priority. Motion detection is very fast and that is why the professional player needs above 100fps to be able to dodge that incoming rocket (the need for fast motion feedback is obviously also influenced by the 1.5 frame latency that all engines have). Detecting colors are also fast, but recognizing shapes and images are considerably slower. Although modern people are trained in picking up fast cuts, around 10fps is probably the limit.

Global illumination has a very low priority, and may not need updating very often. My experience is that many things in a game engine do not have to be updated every frame. For instance AI doesnt have to be computed every frame. (Often this can result in better behaving AI. If a grenade is thrown in to a group of enemies it looks unrealistic when all enemies simultaneously turn since they all realized the treat at the same time). My suggestion is that one build a "kernel" that can let different tasks have different priority and by doing this only update the most crucial parts of the simulation. Therefore whenever even precomputation is mentioned it may mean "computation stretched out over a longer time then one frame".

The error allowed is huge

If we consider that ambient light only contribute a small portion of light to each surface, and that each surface point is influenced by a full hemisphere, we come to the conclusion that the influence of an object in the environment is very limited. Let us consider this: If we assume that an objects surface has about 30

The ideal size of a primitive is one pixel

In the Siggraph paper "The Reyes image rendering architecture" form 1987, researchers at Pixar claimed that each primitive (in their case a quadrillion) should be proportional in size to the pixels that render them. (In their case a half a pixel due to anti aliasing). The point was that anything larger would look bad, and anything smaller would be redundant. This makes things very interesting for global illumination since the pixels are so very large. A 1920 * 1080 image rendered by a camera viewing 90 degree horizontally, gives a pixel angle of 0,046875 degrees. This means that in a distance of one meter the ideal size of a primitive is 0.8 millimeter. When rendering an environment hemisphere for the purpose of global illumination it may be equal to a 8*8 image spanning 180 degrees, is gives each pixel 22.5 degree span and a primitive at one meters distance should be as large as 39 centimeters!

This means that a 180 centimeter long (and preferably 180 centimeters wide) person at the distance of 4.6 meters can be replaced by a single primitive. This tells us something of the extrema coursness that we can use and how simple our data sets really can be.

In what direction does light travel?

Obviously from the light source via surfaces and then in to your eye, but that is in reality. Ray tracing usually goes the opposite direction. This has some obvious benefits as you can disregard all rays of light that doesn't hit your eye. However if that ray bounces around a room it is very unlikely to hit the tiny light bulb hanging from the ceiling, so you are back to the same problem again. This is why it is common for ray tracing renderers always direct the rays directly toward the light sources once thy have bounded around the geometry. The engine knows that some objects are light sources and that they are important light contributors. This makes stocastic raytracers with in the realm of production speed rendering algorithms, but it does however create some problems when dealing with effects like coustics, or an environment where many surfaces are light sources, or are not light sources but are so bright that they could be considerd as such.

So what we can see here is that there is an enormous benefit to letting the engine know more about the environment, and go in different directions in different instances. By just dividing you the environment in two categories of lightsources and surfaces, we get significant performance improvements. What we should do is not just divide our environment up in two groups, but let all geometry and light sources be sorted and stored in such a way that we easely can find the light sources and/or surfaces that influence any point in the environment the most.

The world is a hierarchical BRDF

A Bidirectional Reflectance Distribution Function is basicly a function that computes how incoming light reflects of a particular surface. You could argue that it is an idea of a universal way of describing materials. However one of the reasons why different surfaces behave so drastically different is because of micro geometry. Micro geometry is usually defined as geometry that is smaller then one pixel. This is geometry that is so small that modelling the world including this property would give us gigantic data sets, but they still matter a great deal to how the surfaces appear. Torrence cook where early in defining shaders that where influenced by micro geometry, and in 2000 the Siggraph paper "Illuminating Micro Geometry Based on Precomputed Visibility" took the next step of actually computing a BRDF from geometry.

So going back to the question of what micro geometry is, we can say that anything is micro geometry if viewed from a proper distance. A 180 centimeter long person rendered at 4.6 meters distance can be micro geometry, and therefor be defined as a BRDF. So we can use a BRDF as a primitive! A primitive that contains geometry and materials. We add size and position in the world and we have all we need. However we want the size of our primitive to be one pixel and therefore we make it hierarchical so that once we get closer to an object we can render it as multiple BRDFs. For games we can simply define the BRDFs computation as a dot product, modulated by color.

Rembrandt could do it

So we have concluded that this is a problem that no hardware can simulate properly. How is it that people like Rembrandt, Leonardo da Vinci, and Raphael could do it so well? They knew substantially less about how light works than we can teach computers, yet they are so good. Perhaps we should not try to copy nature and physics but rather try to copy the great masters. The truth is that we are not looking for realism, we are trying to make our brains fire the right synapses. Our visual system is constantly looking for clues to what the data our retinas collect really means and there are some special things that we look for and recognize.

To prove this, render a sphere with a plain gray ambient color and place it in front of a plain white color. This looks totally unrealistic while it in fact is very realistic. In a completely evenly lit environment the ambient light of the sphere should be plain. But our brain refuses to believe that there are any evenly lit environments, because it have never encountered one before so it seems kind of unlikely. If we now add a slight low frequency perlin noise it looks a lot better. Obviously the noise isn't an accurate simulation of light but our brain recognizes the unevenness caused by environment lighting and thinks this image is more realistic. The brain can't exactly determine what causes this and the exact influence of the environment, but it does recognize some effects such as contact shadows, shadows in cavities, and how environments tend to get uniformed colorings. Thees effects can be copied to fool the brain that the light is behaving realistically.

Pleasing the mind

Finally and perhaps slightly off topic. We have to decide what our images are created for. As I am a strong believer in that computer games are an art form, artistic freedom must be given. So called Non photo realistic renderers or toon shaders have been used in some games, but them aside, I think its time for the "realistic" games to look into the possibilities. In todays film, commercial, and music video market a lot of tweaking is done in the post production. I would advise any graphics programmer to visit a local post production facility and see what they do in terms of retouch, color correction and finishing. They highlight areas of interest, remove unwanted distractions and composite multiple shots. The importance of this step should not be underestimated, some even claim that the key to becoming a successful photographer is to bribe his/her copier with love and alcohol. While the photographer decides the subject the post decides what you think of it.

Practicalities

So that is some theory. Let us kick of some simple applications of these ideas, without making any major changes to the way an engine operates. One obvious way one can do global illumination is to precompute the global illumination per vertex or in a light map, this however has been done before and it requires all the geometry and lights to be static, so we are going to skip that and go straight to the next step of actually considering solutions where both the light sources and geometry are dynamic. However I would like to point out that it may be a path worth exploring depending on the engine you are developing. Perhaps a hybrid engine where some parts of the light is precomputed and others are computed by real time algorithms may be a good middle ground for your engine.

Cavity shading

The first we can do is to precompute the visibility of each point on a surface (either per vertex or texel) to see how large part of the surfaces hemisphere is obstructed. This simple technique is great for shading down cavities and are often done by texture artists by hand. To make this a little more sophisticated you can let the color of the obstructing geometry colorize the surface and weight the influence of the obstruction geometry by distance.

Bended normals

To light a surface we use normals. In a mathematical sense a 3d normal is a vector that is perpendicular to surface, but in computer graphics the normal usually represents the ideal direction of incoming light and that may not necessarily be the one and same. So what we can do is to bend the normals away from any obstructing geometry. This makes it possible to light the surface from a point below the plane of the surface. This is obviously impossible if we consider only directional lighting, but if we consider indirect lighting it becomes possible because the incoming light bounces off the surrounding surfaces. If you have a flat surface and you tweak the normals just slightly away from the environment the surface will get a slightly uneven lighting and it will look a lot better.

The great thing about these two approaches is that they fit very well with the way engines are traditionally written. All we need to do is change our input data.

Pre-determined surface samples

For static geometry like indoor environments with dynamic lights, there is an other good way of simulating global illumination. An offline global illumination renderer, based on stochastic raytracing sends out a number of rays to sample the environment, but in a static environment we can precompute and store surfaces in the environment to be used to light the surface in real time.

If we want to compute how one surface influence an other we need to know a few things about it. We need to know the position, the normal, and BRDF of the reflecting surface and the distance and angle to the reserving surface. For games the BRDF can be computed as a dot product and a diffuse color. The distance and angle determines the brightness and can therefore be removed by modulating the color. So we can therefore optimize one surface influence of an other as nine floating point values of data (position, normal and color). For a particular point on a surface we can precalculate this influence of a set number of surfaces that are static in relation to the receiving surface. A light source that is introduced in this environment can now be lit directly, and by a number of surrounding surfaces that influences it. A benefit to this approach is that the added light computations can be done in hardware. How many environment sample that can be taken in to consideration can dynamically be determined by the capabilities of the hardware and, how much computing time is taken up by other things one might want to do in hardware such as advanced shaders and animation. Once this framework has been implemented it can also be used to simulate coustics and sub-surface scattering where also lighting from other directions then the surface normal influence the surface lighting.

This algorithm requires some precomputation to determine the most influential surfaces in the environment but it is still fairly straight forward. However it has one major problem. If your dynamic light source gets very close to the point of influence strange things may happen, because the actual influence doesn't come from a point but from a larger surface. This can be fixed by adding a tenth value that determine a culling plane that is in front of the surface. By moving the position of the sample in the opposite direction of the normal in to the object, it will look good even if the light source comes very close to the surface. However, then the sample may even be lit if the light is on the other side of the surface. Therefore you need to store the distance the sample got moved back in order to cull out any light sources that are within the surface. The larger the surface is the further in you move the position of the sample back. This does require some extra computation but it will remove some strange effects that you may otherwise experience.

Neighbor lists

In a complex world it can be good to be able to determine the most important neighbors. One algorithm that i frequently use is to store a list of neighbors in each object. To avoid using much CPU we use a inexact sorter. for each frame compare the furthest neighbor to a randomly selected object. If the object is closer, the furthest neighbor is replaced by the selected object. To make it even more efficient we can also compare it to a randomly selected neighbor of a neighbor. The list can have different weighting, that takes in to account, object size, brightness and other properties. Perhaps it it can be useful to have more then one list per object to store different types of neighboring objects. Having access to such list can be useful not just for graphics but for things like AI too.

Once we have this data we can convert all surrounding objects to light sources. To compute how a neighbor lights an object, we start by lighting the neighbor by the light sources. in order to do this we need a normal and there are a few different ways one can be computed. The easiest one is to take the vector form the position of the neighbor and the position of the object. If each object has a precomputed BRDF that contains information on how it reflects light in all directions this is fine, but if we use a simple dot product and a color it may look bad. In this case it is better to use a normal that is the half angle between the vector from the neighbor position to the object position and the vector from the neighbor position to the lightsource position.

Now that we know how much light and what color each neighbor reflects toward the object. All we need to do is to place one light source with the computed color and brightness in each neighbor.

Environment maps

Most modern graphics hardware supports cubic environment maps and they can be very usefully for global illumination purposes. What we can do is to paint in objects into a small scale environment cube and use it as a directional lookup table of the incoming light. If we paint in objects just as pixels, according to my tests the ideal size of such a environment cube 6x6x6 pixels large. However most hardware tends to like the size 4x4x6 or 8x8x6 pixels.

The first thing you need is a temporary cube buffer where each pixel is represented as red, green, blue and z-depth. It is advisable to use floating point values for all these values to obtain high dynamic range (this will be discussed further later). First we fill this buffer with a background image. It can be a plain color, an image or a range that simulates a sky dome. This sky dome will greatly influence the look of the finished rendering so take your time tweaking it. It is actually so influential that you may dispense with the painting in of objects, but that would be a bit to boring for this article so we are going to go ahead and make it a bit more complicated.

To paint in an object we just compute the incoming light of an object and draw a pixel of the color in the cube map pixel that represents the direction where the object can be found. With z-buffering we can create occlusion, by comparing the z-value to the previous z-value.

The environment map we get here is not yet ready to be sent to the graphics card, because a surface in one direction doesn't just get lit by the object that is in that direction but also by all other objects in that hemisphere. Therefore wee need to "smooth" this environment map, and to smooth a pixel we add the sum of all pixels in the hemisphere modulated by a dot product. This makes the colors furthest away less influential. Finally we must divide the sum color with the sum of all dot products in the hemisphere.

As you understand this final step is by far the slowest and that is why the choice of cube map size is such a crucial one. But once you have computed the finished environment map you can use it as many times as you want. You can split up the computation over a number of frames, and update one object at the time to get the real time performance. Since all this computation is done in software, using floating point values is fine. But once you get to the hardware you might not want to use HDRI since not all hardware supports it and it takes up more memory. But since you add the smoothing step any spikes of very bright spots are going to be flattened, so you will get away just fine with normal fixed point texture data.

There is a rumor that says that radians doesn't decay with distance and it is in part true. If you photograph a white sphere in a black room the sphere wont become darker the further away you are. However the sphere will become smaller, so average brightness of the image will diminish the further away you move. If you scatter random rays in an environment the radiance will be constant since we will assume that more rays will hit the object if it is closer. However if there are very few rays, or if we draw just one or very few pixels in an environment, this wont be good enough. In this case we actually do modulate for distance and divide the brightness with the number of times the object is sampled. With this in mind it can be a good idea to draw an object as more than one pixel in an environment map to get better occlusion.

Final thoughts

In conclusion i would like to encourage developers and researchers to be less afraid to explore new possibilities, even if they appear to be "hacks". By lowering the demand for quality, we can explore alternative techniques that may in the end benefit even high quality renderings.

In my opinion the key to solving the problem of global illumination (both in real-time and for off-line renderers) is to create alternative geometry representations for the light reflecting geometry. We need a new field of research to explore how to generate and store reflective properties of geometry. Hierarchical BRDFs is one such representation but there are probably many more. There have been much research in geometry complexity reduction while preserving visual features such as edges and silhouettes. The same type of algorithms should be developed where the reflective properties are preserved.

References

- S. Chandrasekar. Radiative Transfer. Oxford Univ. Press, 1950.
- Michael F. Cohen, Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. A progressive refinement approach to fast radiosity image generation. Proceedings SIGGRAPH '88, pages 75–84.
- H. Hoppe. Progressive meshes. Proceedings SIGGRAPH '96, pages 99–108, 1996.
- Heidrich, W., Daubert, K., Kautz, J., and Seidel, H.-P. Illuminating micro geometry based on precomputed visibility. Proceedings SIGGRAPH 2000.
- LINDHOLM, E., KILGARD, M., AND MORETON, H. A user-programmable vertex engine. Proceedings SIGGRAPH 2001.
- R. L. Cook and K. E. Torrance. A reAEectance model for computer graphics. ACM Transaction on Graphics,
- S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen, "The lumigraph," in Proc. SIGGRAPH'96, pp. 43–54, 1996.
- Marc Levoy and Pat Hanrahan. Light field rendering. In Proceedings of SIGGRAPH '96 (, 1996.
- C. Loop, Smooth subdivision surfaces based on triangles, Master's thesis, University of Utah, 1987.
- P. Debevec et al., editors. Image-Based Modeling, Rendering, and Lighting, SIGGRAPH'99 Course 39, August 1999..
- DEBEVEC, P. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In SIGGRAPH 98 (July 1998).
- Cook, R. L., L. Carpenter and E. Catmull, "The Reyes Image Rendering Architecture", SIGGRAPH 87, pp. 95102.
- E. Catmull, J. Clark, Recursively generated B-spline surfaces on arbitrary topological meshes, CAD 10, No 6 (1978): 350–355.
- B. T. Phong. Illumination for computer generated pictures. Communications of the ACM, 18(6):311–317, 1975.
- Blin, J.F. and Newell, M.E. 1976. Texture and reflection in computer generated images. Communication of the ACM. 19(10): 542-547. October 1976.

- J. Fourier, Th'eorie analytique de la chaleur, Didot, Paris, 1822.
- Kajiya, J. T. The rendering equation. Comp. Graph. 20 (1986).
- Reuven Y. Rubinstein. Simulation and the Monte Carlo Method. John Wiley and Sons, 1981.
- P. Shirley. "A Ray Tracing Method for Illumination Calculation in Diffuse-Specular Scenes," in Proceedings of Graphics Interface '90, May 1990, pages 205–212.
- R. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. In Computer Graphics (SIGGRAPH '84 Proceedings), pages 137–145, 1984. c fl Institute of Computer Graphics 28 Szirmay-Kalos / Stochastic Methods in Global Illumination
- Cook, R. L., "Stochastic Sampling in Computer Graphics", ACM Transaction on Graphics, 5, 1, 51-72, (January 1986).
- R.L. Cook. Stochastic sampling and distributed ray tracing. In A.S. Glassner, editor, An Introduction to Ray Tracing, pages 161–199. Academic Press, 1989.
- Blinn, J. F., "Simulation of Wrinkled Surfaces", SIGGRAPH 78, pp. 286-292.
- Hugues Hoppe. View-dependent refinement of progressive meshes. In Turner Whitted, editor, SIGGRAPH 97 Conference Proceedings, Annual Conference Series, pages 189–198