Diffuse Global Illumination via Direct and Virtual Indirect Light Sources

Greg Douglas
Auran
2003

Abstract

The quest for realistic computer generated images continues. An exciting area of which is real-time rendering. Images need indirect light, soft shadows and color bleeding in order to exhibit realistic global illumination qualities. Achieving real time frame rates requires compromise as high quality methods such as view dependent ray tracing may consume hours instead of the milliseconds available. Today's applications must run on today's computers, the majority of which are still using last years hardware at best. Light-mapping or precomputed light and shadow textures are a well known technique, used in computer and video games, to enhance the realism of statically lit scenes at very low run-time cost.

This paper presents a new lighting technique capable of rendering diffuse global illumination for static scenes of high complexity built from arbitrary polygonal meshes. It draws from existing methods such as ray tracing, photon mapping and radiosity to produce worthwhile results with a controllable precomputation cost. The precomputation phase and the final real-time render phase are designed to run on systems with limited memory, and do not require any specific CPU and GPU features beyond floating point calculation and texture mapping.

Keywords
Global Illumination, Indirect Light, Photon Map, Rendering, Real Time

Contents

1 Introduction

This paper presents a new lighting technique that draws on methods and tools used in *ray tracing*, *photon mapping* (Jensen 1996) and *radiosity* (Goral et al. 1984). The lighting result that is produced approximates diffuse global illumination. The technique works by using a photon map to locate and sample concentrations of light energy that have collected in the scene. These samples are converted into a set of *virtual* lights that are rendered along with *direct* lights, in a view independent manner, via ray tracing, onto texture maps covering the scene surfaces. This technique is largely independent of scene complexity and is highly scalable in that the sampling quality and time are directly related to configuration settings that may be dramatically altered while allowing the result to remain representative of the lighting solution. Implementation details and related issues are also presented. **Appendix A** shows sample output from this technique.

This technique has some parallels with *Instant Radiosity* (Keller 1997) in that indirect light may be accurately simulated by creating more direct light sources. It also has ties to traditional radiosity that implies that every surface acts a secondary light source during a simulation phase in which energy is distributed throughout the scene.

No special pre-processing is required for the geometry. The implementation works on triangle soups, however structures such as BSP (Binary Space Partition) and Octrees are essential for accelerating ray tracing and ensuring efficient operation of the process. The scene does not need to be tessellated into a grid as many Radiosity techniques require. If the surfaces are processed in turn, very little memory is required for the whole process, as buffers may be shared and the results stored along the way. Only the scene geometry and various acceleration structures need be stored. The photon map and surface sample points need not persist throughout the process.

2 How does it work?

In this lighting solution, *direct* lights represent the first bounce of light reflecting off a surface, and the *indirect* lights represent the final bounce. The photon map becomes a tool to statistically simulate and sample energy distributed in a scene. The balanced kd-tree is a tool; to not only accelerate manipulation of the photon map, but sample energy concentrations in the scene in an efficient and well-distributed manner.

Once a set of direct and indirect lights are created from the scene description and the indirect light sampling process, the lights are applied by illuminating the texels of the lightmaps covering the scene surfaces. Various acceleration techniques are utilized to reduce the number of light samples and speed the ray testing method used to determine visibility.

These are the basic steps of the lighting process:

1) Generate global photon map. Not too many photons are required, a few hundred thousand to a few million should be adequate for *NumPhotons*.

2) Decide on the *NumIndirectLights* number of indirect lights. Several hundred, to a few thousand will produce a good result. The number of photon samples will be approximately *NumSamples = NumPhotons / NumIndirectLights*.
3) Use a balanced kd-tree (3d tree) to store the photons. Traverse this tree, sampling the first *NumIndirectLights* photons.
4) Use the photon position and surface normal to sample the nearest *NumSamples* photons that have similar surface normals.
5) Add all the photon energy for each of these indirect lights together, and create a new direct light.
6) Process these *indirect* lights together with the original direct lights, but treat them as (radiosity style) surface emitters. Bias the source area to compensate for light source bright spots, as these surface lights are approximated by point sources.
7) Process the *direct* lights using multi sampling so they have an area effect with soft shadow edges. Tens to a few hundred samples will look nice.

3 Why does it work?

This lighting technique borrows concepts and methods from several existing techniques. This section describes the details of how particular methods have been used or modified to help this technique produce the desired results.

The photon map construction is fairly standard, but the Russian roulette (Arvo James et al. 1990) rules are modified slightly. The photon map stores energy to emit *the final bounce*, not visualize on the surface before being seen by the eye. Because of this, stored diffuse photons must have the surface color applied otherwise they will reflect the incident light color instead of the reflected light color when emitted. In addition, if specular light is to be approximated (and it can be in a very dodgy manner), the specular light samples must be stored at every bounce otherwise the specular effect will be effectively lost.

The kd-tree is a three dimensional, axis aligned BSP tree. When balanced, it partitions space in a way that is consistent with the distribution of photon locations. Note that using the kd-tree in this way has been used by Per Christensen (Christensen 2001) to accelerate *final gathering* in photon map enhanced ray tracers by pre-computing irradiance. The sample positions in the kd-tree are *safe* since a photon is stored at a surface there. Light energy is conserved by accumulating energy from emitted photons and distributing it to virtual lights for emittance. Refer to **figure 1** to see how one thousand indirect lights were created from one million photons in a photon map. Notice there is a higher density of indirect light sources in areas where light is being reflected.

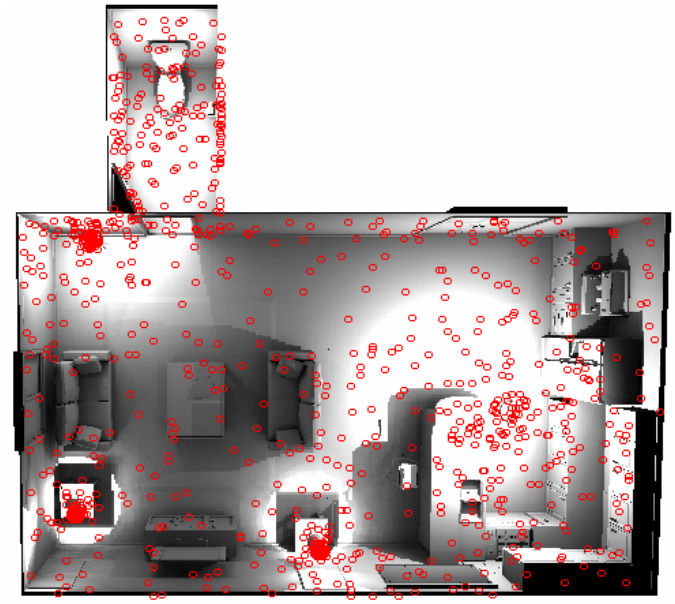


Fig. 1: Indirect lights sampled from the top levels of the kd-tree. This image shows one thousand lights from a one million-photon map. A higher density of (indirect) light sources is visible in areas where light is being reflected. The three almost solid clusters are actually inside lamp covers. The other two lights were not covered. Although hard to tell from this angle, many of the new light sources in darker areas are actually located on the ceiling.

When these virtual indirect lights are created, like traditional direct lights, they emit from an infinitely small point. This can lead to bright spot artifacts at surfaces very close to the light source. It may be thought that a solution would be to spread the light out on a surface, or across multiple samples, but this would defeat much of the benefit obtained by this technique. Besides, an alarming number of indirect lights (perhaps a thousand or so) are already used to render this light, so the number of lights should be minimized. It turns out that by biasing the source emitter area in the lighting equation, the bright spot artifact that would otherwise appear can be effectively eliminated. Refer to **figure 2** for original and corrected results. Further work needs to be done to explain exactly what the optimal bias number is, but simple experimentation produces a good value very quickly. If the bias is too small, the bright spots are evident, if it is too large, the light result is darkened, but at the moment where the spot artifacts disappear and the light begins to noticeably darken, the optimal bias is found. This bias factor is consistent and appears to relate only to the size (and shape) of the energy sample, but nothing else. A value of 10 was found to be suitable for lights sampled with a sphere and 30 to be suitable if sampled with a cube. It might appear that the difference could be pi and that 10pi or such would be more precise, but that is not correct. It must also be noted that the sphere and cube samples are only approximations. Smaller and more numerous samples approach an accurate result. Although the sample shape is a sphere or cube, the photons lie on the scene surfaces such that a spherical sample flattens to a disc and cube sample to a polygon. The size of the sample volume and the scene geometry contribute to the accuracy (or lack of) in this phase.

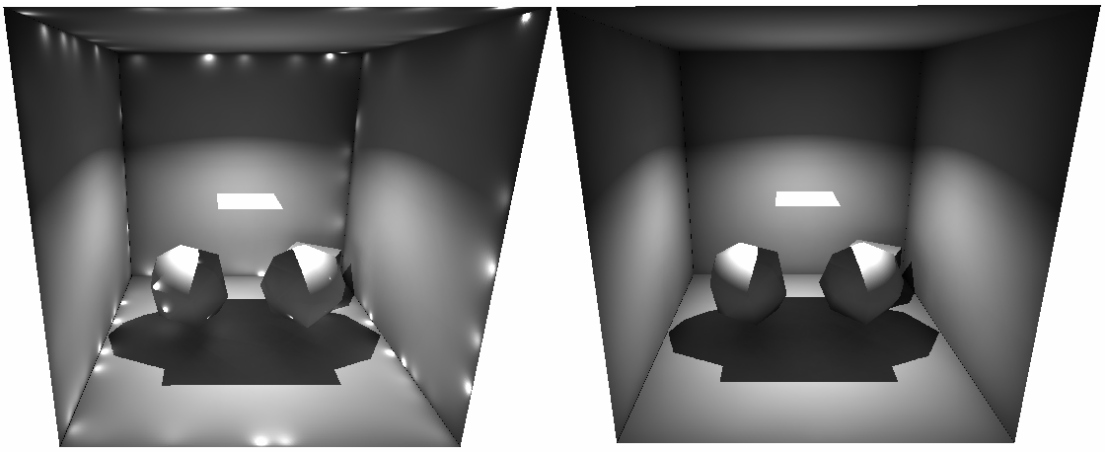


Fig. 2: Light source is above the white table. Left: Bright spot artifacts caused by point light sources on nearby surfaces. Right: Bright spot artifacts removed by biasing the source area in the lighting equation.

4 Limitations

Several limitations exist relating to this lighting technique. There are performance and quality considerations, and the lighting result is not a complete lighting solution, as it does not simulate all the ways light interacts with the scene and viewer.

The inner loop of this lighting technique requires an occlusion test, and that is traditionally the slowest part of a ray tracer. Even a hardware implementation would require extra rendering and reads from a stencil buffer or shadow buffer.

Better results are produced using higher resolution output texels, and more indirect lights. This means the process may need to run for minutes to hours to produce results of a desirable quality. A standard ray tracer should produce higher quality view dependant results much faster.

Caustics and Specular light effects are not produced by this technique, or at least not visualized correctly. Note that the photon map could produce caustics by simulating photons interacting with reflective surfaces and participating media, though the results would not be good since very few photons are used and the implementation does not visualize the photon map directly. This is actually a serious drawback in scenes with lots of reflective surfaces. For example, if a light was close to a mirror, the reflected light is not visible. It is however distributed correctly for other diffuse representation. A dodgy compromise has been implemented that stores photons on reflective surfaces, so they can be emitted from that surface and create some light even though the light is emitted diffusely, thus losing the directional focus that characterizes specular light. Note that this technique was never intended to simulate specular light. The specular light is best rendered in real time since it is largely view dependant. The specular style light that is missed is caustics, that is, highly directional indirect light seen on diffuse surfaces. The earlier example, of a light, close to a mirror reflecting a bright spot on a nearby surface, is familiar in the architectural scenes tested. Caustics are commonly noticed as the striking light patterns, seen dancing near swimming pools and water, caused by reflection and refraction.

A low quality lighting result may show artifacts relating to the number and separation of lights in the scene. The result looks a bit like an object lit by football stadium lights, with a small number of distinct and separated shadows rather than one soft shadow. **Figure 3** shows an example of this.

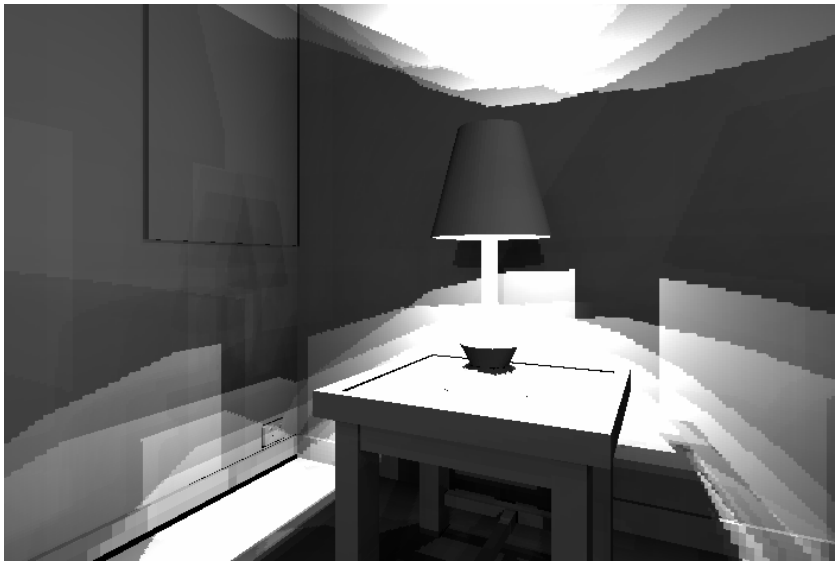


Fig 3: Too few indirect lights cause light and shadow banding.

Appendix A describes early attempts that led to the development of the lighting technique presented in this paper.

5 The Entire Process

This section overviews the entire process from importing the scene to preparing output. Several significant parts of the process are expanded as pseudo code. Future sections will describe particular parts of the processes in depth.

Import geometry and lights

Scene Geometry and lights are added from an external file or directly in code, via an interface class. Mesh consolidation is performed with shared or near-identical polygon vertices are merged and degenerate triangles are removed.

Initialize Direct lights from input lights

Direct lights are initialised from original light sources. Surface lights are subdivided into multiple patches. Omni lights and spot lights will be mutli-sampled later.

Build acceleration structures from mesh geometry

Polygon BSP trees are constructed for each polygon mesh object.

Transform local space geometry into global space

World space polygon representations are constructed for every surface in the scene.

Calculate bounding volumes for mesh objects

Various bounds and extra information is collected or calculated for each mesh object.

Create acceleration structures from mesh objects

Scene and/or region BSP trees are constructed to store mesh obj7ect references. Visibility tables may also be calculated here.

Create smoothing groups from mesh object polygons

Smoothing groups are created from polygons that share edges with near-identical positions and normals.

Create lightmaps from smoothing groups

Lightmap sample structures are created from sets of polygons in smoothing groups.

Build Photon Map *

Photon map is created by simulating light reflected and absorbed within the scene.

Create Indirect Lights *

Photon map is sampled to create more lights, where energy is reflected in the scene.

Free Photon Map

Photon map is no longer required and its resources may be released.

Light the world *

All lights, Direct and so-called Indirect are sampled to light the lightmaps. Visibility is determined via ray testing.

Pack lightmaps

Related lightmap rectangles are packed together into larger texture pages.

Finalize lightmap texture coordinates

Polygon vertices are adjusted with final lightmap texture u,v coordinates.

Output lightmap textures

Lightmap texture pages are written to disk.

Geometry is ready for export

Data structures are ready to export or further consolidation.

* Expanded description of this part of the process follows.

Specific parts of the process in more detail.

Build Photon Map

```
Create photon emitters from Direct Lights
Calculate total light power for scene
While photon map no full
        For each light
                If light probability to emit is ready
                        Emit photon
                        While photon not absorbed or lost
                                Trace photon
        If all photons have been lost for several passes
                Exit now to prevent endless loop (scene is misconfigured)
For all photons in photon map
        Adjust power by: total power / number of lights
```

Create Indirect Lights

```
For number of desired indirect lights
        Sample photon map
        Create indirect light and add to direct light set
```

Light the world

For all lightmaps

Prepare shared memory
Calculate sample points for texels
For all lights

Find objects that potentially occlude
Sample single lightmap with single light:
For every lightmap sample point and every light sample point

Calculate lighting equation
Ray test visibility
Add result to lightmap sample color

Apply exposure correction
Apply multi-sample filtering
Fill RGB output from texel samples.
Release shared memory

6 Parts of the process

6.1 Smoothing group creation

A smoothing group is a collection of polygons that appear smooth because they share edges, with vertices having both the same position and normal. Smoothing groups are usually specified by the artist using an application to construct the scene meshes. Sometimes mesh exporting programs consolidate the mesh polygons and calculate smoothing information. In order to maximise flexibility, the implementation consolidates the mesh by merging shared vertices and creates smoothing groups by merging shared edges. Detecting and merging shared vertices and edges can be a time consuming operation if implemented in a brute force manner. Once polygon and vertex counts reach the tens of thousands, algorithms that are more efficient must be employed.

This implementation uses a dynamic kd-tree to accelerate neighbor queries to detect shared vertices and edges. As edges are described as indices into vertex pools, the edges are two dimensional integer pairs that are easily inserted and queried via the kd-tree. Refer to Appendix B for a description of balanced and dynamic kd-trees. Other possible structures such as hash tables, modified for multi dimensional range queries, and r-trees, may be suitable for similar data manipulation. Such other structures however, are often complex to implement and have significant per-node memory overhead. Performing mesh consolidation, smoothing group and lightmap creation allows the implementation to easily handle *polygon soups* from any source, with excellent performance.

6.2 Lightmap structure creation

A lightmap in this implementation is a rectangle of samples that will eventually be output as a rectangle of texture. The lightmap structure stores a list of the polygons that form a surface as part of the larger scene. A parameterization is also stored to accommodate conversion between *texture space* and *world space*.

Currently, polygons are grouped to form a lightmap as follows: A seed polygon is selected from the remaining polygons in a *smoothing group* belonging to a mesh object. The polygon group is grown outward from the seed polygon as long as various criteria are met. The potential polygons are already being selected from the same smooth group, so for a simple world-axis oriented planar projection, the face normals are compared for similarity. The signed dominant axis of the face normal determines whether the polygon can be added to the lightmap's polygon group or not. Other polygon attributes such as lightmap texture resolution must also be identical, to be a part of the polygon group. As the lightmap represents a single texture, uniquely covering the lightmap's polygons, care must be taken to prevent polygons from overlapping. A polygon strip that looks like a ribbon with ends crossed, or like the twisted surface of ice-cream from a ice-cream machine, will contain overlapping polygons that would otherwise be grouped together. Overlapping polygons are rejected by a two dimensional separating axis test, and will form part of some other lightmap.

Lightmaps are bounded by a world-axis aligned grid, rounded outward to the nearest grid position, plus one or more *border* grid units. These extra border units relate to the extra texels, added to help with texture seams, filtering and mip mapping issues discussed elsewhere. **Figure 4** shows an example lightmap diagram.

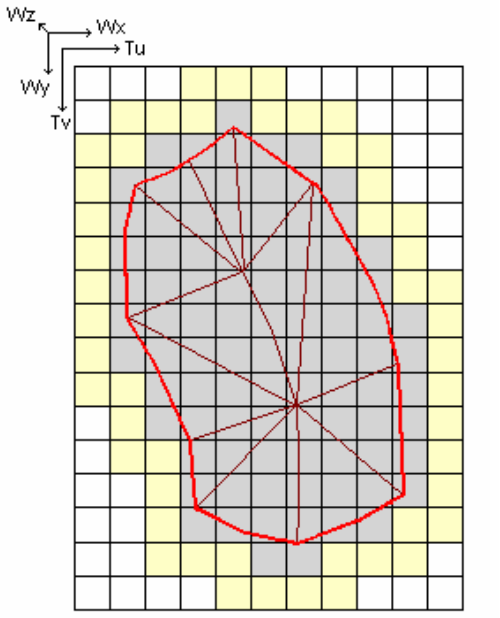


Fig. 4: Diagram of a lightmap. Wx,y,z represent world axes. Tu,v are the texture axes. Note the lightmap is at least one whole grid unit larger than the contained geometry in each direction. The grid represents texels in a texture. At least one extra texel around the geometry is sampled for use by bilinear interpolation.

Sample points representing texels or sub-texels are calculated across the face of the lightmap. The sample points usual lie in the center of the texels, but may be in other locations for antialiasing. Earlier implementations calculated the sample points for all lightmaps as a phase before lighting. The current implementation delays sample point creation for each lightmap, until they are required for use in lighting. The sample points are dumped as soon as they have been made use of. Doing this greatly reduces memory usage as each sample point is a full *float vector3* position, normal, color and some flags.

6.3 Photon Map

The photon map, as described earlier, is a tool used by this technique to simulate, store and sample light energy in a static scene. **Figure 5** shows photons stored in the photon map, rendered in the scene at their positions.

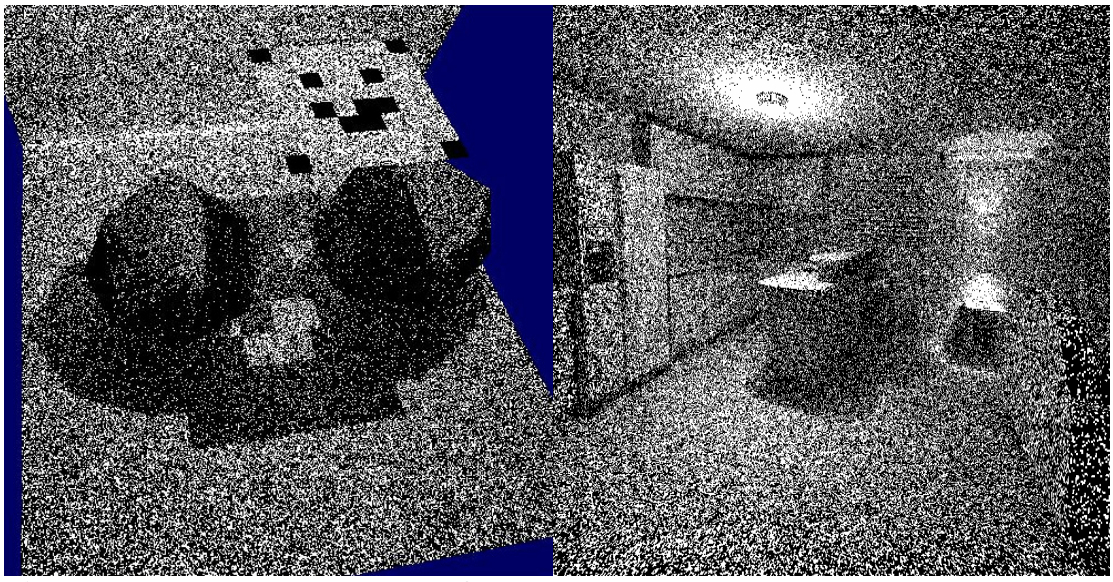


Fig 5: Photons in a photon map. **Left:** The light source is positioned above floating table. The table has some alpha transparent texels. Notice how the shadow varies in density under the spheres. **Right:** Kitchen and Lounge with ceiling light and lamp illuminating scene.

6.3.1 Photon storage

As hundreds of thousands to millions of photons may be stored in memory, the photon structure size must be minimized. The implementation uses the compression methods described by Jensen, originally developed by Greg Ward. The photon power or flux is stored in a 4byte RGBE, Red, Green, Blue, Exponent form. It is compressed and decompressed as required. The incident direction and/or surface normal of the photon are stored as two byte-size signed integers representing angles quantized with a resolution of less than 1.5 degrees. The photon position remains a full 12 byte float vector3 because it is accessed very frequently. The whole structure is about 20 bytes. Slightly more if extra components are required, or padding is added for memory address alignment. As the photon map is only used to create indirect

light sources, it can be constructed, used, and thrown away, so its memory usage is not as critical as it could otherwise be.

During kd-tree balancing, photon elements are sorted in place so no extra memory is required. Photons are allocated in blocks in a growing structure that behaves like an array.

6.3.2 Photon emitting and reflecting

It is difficult to precisely control how many photons will be emitted and stored. If one was to set a exact number of photons to be emitted, some may be lost entirely due to gaps in the scene mesh, others may be stored multiple times due to the diffuse reflectance model. The quality of the photon map and the amount of memory used is determined by the number of photons stored, so controlling this is important. The photon map is intended to statistically represent the density of energy distributed throughout the scene. Lights of different intensity need to contribute their relative share of photons to the photon map.

The photon emitting process continues until at least *n* photons are stored in the photon map. If a scene contained gaps or if lights were facing away from a non-enclosed scene, many or all of the photons may be lost. Such a configuration must be detected to prevent endless looping. Each light emitter is given a firing threshold and has an accumulator that increments each time the opportunity to fire passes. In this way, lights are repeatedly cycled and given the occasion to fire a photon until the photon map is full. The photon map will thus contain no less, and often slightly more photons than requested.

Photons are emitted from light sources in proportion to their contribution of energy relative to the combined energy of all lights. Photons are emitted in a quasi-random manner. Each light type has an emittance function appropriate for its form. Point, Spot and Surface lights are supported. Point lights are actually spheres and photons are spawned from random points on the surface of the sphere. Spot lights use a random cosine distribution, oriented in the light's direction, and scaled by the lights outer-arc. Note that when the *direct light* is calculated for a spot light, it uses inner and outer arc values for greater control, similar to the OpenGL and DirectX APIs and familiar to real-time 3D artists. Surface lights spawn photons from random locations across a polygon surface that has been tessellated into triangles, and weighted according to the relative area of the triangle to the whole surface.

Photons are all emitted with the same power, but as they are reflected, their power may vary and filtering may occur. If a photon mapping solution were to consider participating media such as translucent prisms, it may be desirable to store separate red, green and blue photon maps. If other aspects of a complete realistic lighting model were to be simulated, separate photon maps would be required to maintain efficiency. A photon map just for caustics is often used in commercial renderers. These caustic photon maps store a large number of photons and need to be directed at specular surfaces and are only constructed for portions of the scene actually visible to the (synthetic) camera. This is because caustic photon maps are visualized directly and must produce well defined output. The global photon map used by this process is not visualized directly so it's implementation and usage is relatively simple.

6.3.3 Photon sampling

To make use of a photon map, the photons stored in it must be sampled. A density estimate can be made by either sampling with a fixed size, or a fixed photon count. Both methods produce identical results when comparably configured. Using a fixed sample size for direct photon map visualization does not usually produce as good results as a fixed photon count because the fixed photon count adapts its search size to provide a better density estimate. Finding the nearest *n* photons efficiently can be challenging. Most methods work by taking a smaller fixed size sample and resizing up if the sample contained too few photons. Performing multiple samples in order to find enough photons wastes time and needs to be minimized. When photons are located, they are kept in a priority queue so a farther photon may be exchanged with a nearer one efficiently. This implementation uses a *binary heap* as a priority queue. Most operations on a binary heap call a *Heapify* function (not described here) that rearranges the structure to preserve the heap property.

The automatic sampling method used by this implementation proves to be efficient, resampling in some scenes well less than 1% of the time when tested to directly visualize the photon map. The initial Min sample sizes (ie. Sphere radius or box extents) is almost arbitrary due to the algorithms rapid estimate correction. Min should be set to a small number above zero, and Max should be set to a reasonable limit relative to the scene size (for example, two to five meters for a real size house.)

This is the basic algorithm for automatic sample sizing:

```
Sample photons with current size estimate
If collected photons < desired limit
        Adjust radius based on area density estimate
        (Eg. If 25% less photons were found than desired, increase the sample area by
        ~25% by adjusting the search radius.)
        If radius > maxRadius
                Accep the sample and continue
        Else
                Loop back and sample again
Else (If collected photons = desired limit)
        Set radius to furthest photon dist + 10% or so for conservative overestimate.
        Don't loop back, but record and use the adjusted size next time.
```

Note that the number of photons sampled will never exceed the desired limit because the nearest *n* samples are collected, and the search will terminate as soon as that limit is reached.

The shape of the photon map sample may be a sphere or a cube. Results from either are visually similar. Performance of the cube sampler may be slightly better, and may produce better results. The cube sample shape was chosen based on the logic that since the top levels of the kd-tree are used to create indirect lights, a cube sample would be more compatible and accurate with the axis aligned subdivided space of the kd-tree.

6.3.4 Kd-tree sampling optimizations

The kd-tree is used to accelerate queries on the photon map as well as locate well-distributed indirect light concentrations. Here are some optimizations used by the kd-tree traversal function to sample photons in the photon map:

- Sqrt() is avoided. The squared radius is maintained instead, and vector components (eg. Extents[axis]) are used where possible. With a sphere shaped sample, the sign check is performed on the axis distance, as the squared radius is always positive.
- Iteration is used instead of Recursion where possible. When sub trees are skipped, the code sets new *Start* and *End* range within the tree and just re-iterates instead of re-calls.
- The code is reordered such that the test on the current node can be skipped entirely if the current node is known to be located too far from the test position.
- Ordering the priority queue is delayed until it is full. The caller checks if the queue is ordered or not. If the queue hasn't been ordered yet, the caller makes a single pass through the unordered queue to find the maximum and sets this as the first element. An alternative would be to call heapify once, if this had not occurred thus far.
- When photons are popped off the resulting queue, they are not actually removed. Doing so would cause the queue to heapify on each removal. Instead, the queue is treated as an array again, and elements are processed in that order. The first element is still the least cost (farthest to sample location) as expected.

6.4 Original Direct Lights

The focus of this paper is on indirect diffuse light, but the direct diffuse light is usually a significant contributor to a final lighting solution. Direct lights are handled in a fairly standard manner. The contribution from each light is calculated per pixel (texture sample) and added to the lightmap. The lights are multi sampled in order to achieve softer shadows and lighting. Lights that would otherwise be point sources become surfaces. Recall that 'soft shadows are a consequence of partial visibility of an extended light source' (Hasenfratz et al. 2003). Spot light samples start at a section on the surface of a sphere. Surface lights may be used as an additional light type and are simulated like this: All polygons that describe the emitting surface are made into patches. These patches are recursively subdivided by splitting them with axis aligned planes until they reach a configured size threshold. The surface light patches then become smaller spot lights. They actually use a surface-to-surface radiosity style lighting equation rather than that of a point or spot light.

Point lights and spot lights are multi sampled by generating points on the surface of a sphere the size of the light. These points may be generated uniformly, randomly, uniform with jittering, or with a similar method. The larger the light sphere is, the more samples are required to make the shadows appear smooth. The output texture resolution may also influence the lighting result, as lighting artifacts may be more visible on higher resolution output. Lower resolution output has the effect of blurring

light samples together reducing the light sample aliasing. A quick preview with soft shadows may require 25 samples per light, but for high resolution, final quality, 150 to 300 samples may be required for smaller and larger lights respectively. **Figure 6** shows soft shadows cast by a spot light.

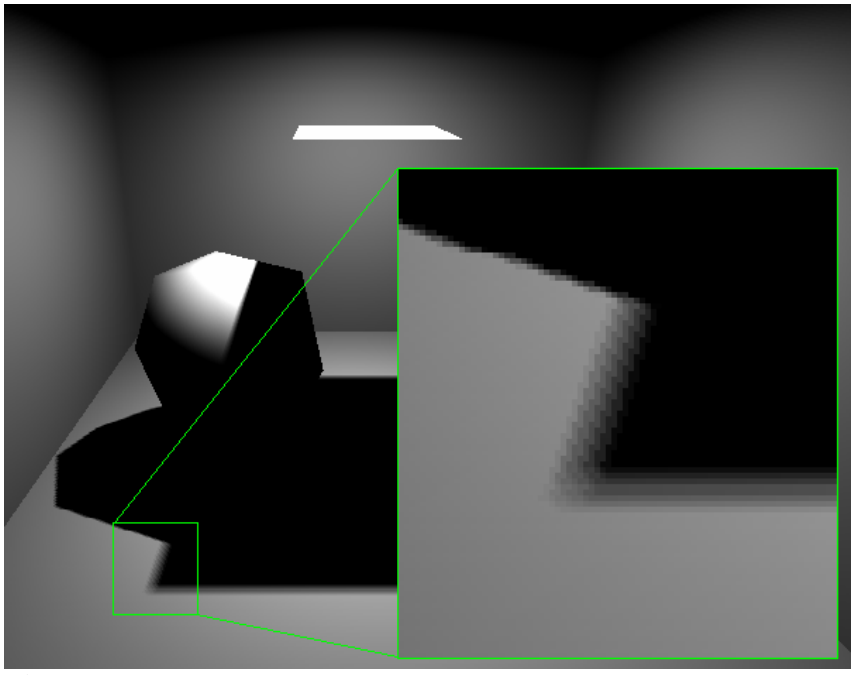


Fig. 6: A soft shadow cast from a (direct) spot light. The light source is positioned above the floating table. Notice the shadow penumbra appears wider at different locations, particularly further from the light source.

6.5 Ray tracing

When *ray tracing* is referred to in this paper, it is not about the traditional method of tracing a ray through a screen pixel to the scene objects, with shadow ray tests back to light sources. Instead, it is simply referring to testing line segments for point-to-point visibility, or *first hit* results. Although *rays* are frequently referred to, *infinite* rays are almost never used. Line segments that represent limited logical rays are used instead. Line segments limit the test scope to the scene size, or smaller local space. Additionally, precomputation can accelerate some calculations involving line segments.

Implementing ray tracing techniques in code in a brute force manner is relatively simple, though completely impractical. Acceleration methods and structures are thus essential for feasible operation. Such methods attempt to minimize the number of objects and polygons considered for testing. Structures are used to group objects, divide space, or store relationships so that tests, limited to specific locations in the scene, may be performed efficiently.

6.5.1 Potential Test Objects

A scene is usually a collection of many objects. These objects may be referenced by, or stored in high level entities such as regions, rooms or cells. The objects in this

implementation are effectively polygon soups and may contain up to 65 thousand individual polygons each. Although objects may be a polygon soups, connected surfaces and non-intersecting geometry produce efficient output with fewer artefacts.

In order to reduce the number of tests performed, only the regions, objects and polygons potentially intersecting the test are considered. To cull whole regions, various high level structures, or look up tables derived from them, may be used. To reduce the number of objects tested, an axis-orthogonal, shallow BSP tree is used to store object references. The object's bounding box is used to classify the object within this structure. As the scene or region size is known, a BSP is created to a fixed depth, simply partitioning the region into a set of equal size voxels. Objects are inserted into this structure, as if dropped in the top, and fall through to be stored in the leaves. A single object may be referenced multiple times within the tree. This BSP could have been constructed in a number of different ways, and other structures could have alternatively been used. This particular structure is simple to implement and provides excellent query performance. As described later, a convex hull is constructed that bounds all paths between a light source and surface. This hull is tested with the BSP to produce a reduced set of objects to ray-test.

A single mesh object may be stored in up to two BSPs, one for opaque polygons, the other for transparent or translucent polygons. The opaque set is always tested first as it is more efficient to do so, and if a hit occurs for an occlusion test, occlusion is guaranteed, even if other translucent objects would also have intersected.

6.5.2 Polygon BSP

The purpose of the polygon BSP is to reduce the number of line-segment versus polygon tests. These tests are used to determine point to point visibility between a light and a receiving surface. Approximately 50% of CPU time is spent performing this function.

The top levels of the BSP are split by axis-orthogonal planes, dividing the mesh space into voxels. A *voxel* is a volume element, but in this description, it is simply a three dimensional axis aligned box. The remaining nodes are split by the planes of polygons from the mesh. Polygons may be stored on any level below the voxel subdivision, not just in the leaves. A similar BSP structure is described by Jim Arvo in his paper "Linear-Time Voxel Walking for Octrees". (Arvo 1988). Refer to **figure 7** for a visualization of the axis-orthogonal portion of the BSP tree. This type of tree can be constructed quickly, and traversal proves to be faster than other related tree structures. Polygon descriptions are stored in the tree. The polygons are stored as triangles in the final implementation, but n-gons would be efficient for meshes mostly comprised of flat quads. Polygons are stored as two-dimensional edge planes, and a three-dimensional face plane, for fast point in polygon and line segment testing.

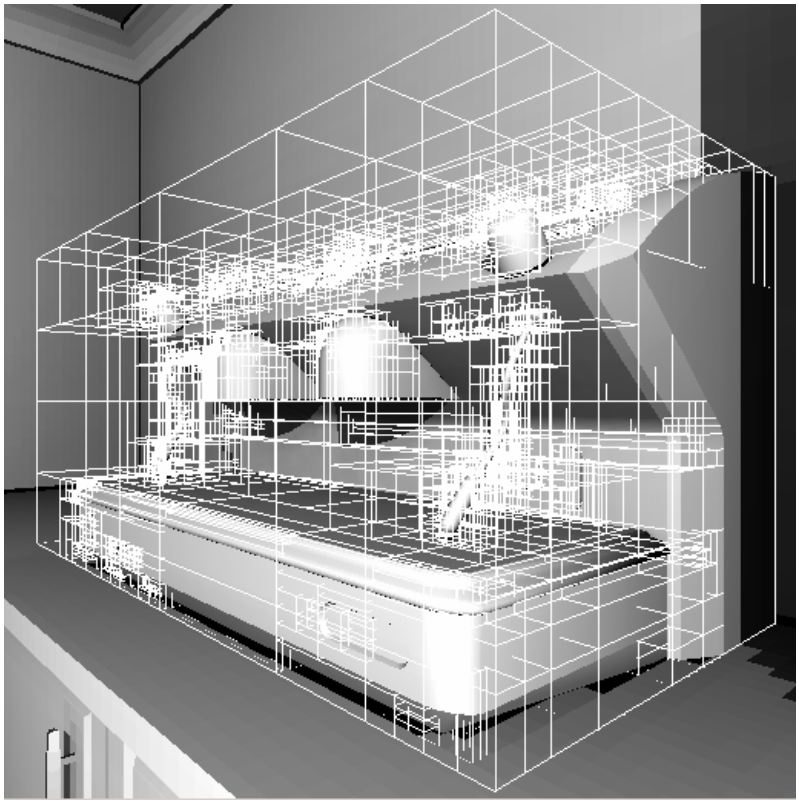


Fig. 7: BSP generated from coffee machine model. Only the top-level axis aligned (voxel style) portion of the BSP is shown. Further down the tree, the mesh polygons are used to create hyperplanes.

Here is the pseudo code for BSP construction

```
Function Initialize
        Add all polygons to a single node.
        Find the axis aligned bounding box of the node.
        Call Function SplitByAxis

Function SplitByAxis
        boxDelta  = boxMax - boxMin
        axisSet = boxDelta component indices sorted from large to small.
        If current tree depth < depthLimit
        AND number of polys in node > min polys in node
                For index 0 to 3
                        Use the next axis index from axisSet and make a splitting
                                plane and bounding boxes for each side
                        Split the node polygons into two groups by testing
                                with box bounds, not just the split plane
                        numSpanned = numFront + numBack - numOriginal
                        fraction = numSpanned / numOriginal
                        If fraction > fractionThreshold (30% works well)
                                Don't accept this split and continue
                        Else
                                Accept this split and break
                If none of the 3 axes were accepted
```

```
                Call function SplitByPolyPlane
            Create two new nodes
            Put polygon groups into the two new nodes
            Remove contents from original node
            Call function SplitByAxis
        Else
            Call function SplitByPolyPlane

Function SplitByPolyPlane
        Find best (least cost) split plane from poly set
        Split poly set using plane and reasonable size epsilon
        For each of the two potential children
            If num polys on this side of plane > 0
                Create child node
                Fill node with polys
                Call function SplitByPolyPlane
```

The function to find the best polygon splitting plane proved to be very difficult to make worthwhile. Obvious choices are to find a plane that minimizes splitting, or creates better balancing. None of these choices or combinations consistently helps performance, though they can dramatically affect tree size or tree depth. Creating an optimal BSP for traversal speed is very difficult. Not only could early, apparently good split choices prove to be bad later during construction, but the cost of traversing the tree changes at different points. For example, the first few levels of the tree cull large areas of the test space, but later levels may cost more than just testing the node contents. In addition, the configuration of polygon geometry and rays may create inefficient situations. For example, if a flat rectangular surface was tessellated into many triangles for some reason, it may be better that all the polygons be placed in one node, rather than divide the polygons into more nodes. If the node is rejected early in the test, then many polygons are culled at once. However, if the node was consistently encountered, early on, many polygons may be tested for no effect. In the latter case, it would have been better to split the node contents into more nodes. The fact that tree levels closer to the root are like voxels (forcing the mesh to be divided efficiently into chunks early on) greatly eases the responsibility for later levels to be optimal.

Traversal of the BSP tree is fairly standard. A better understanding of BSP traversal and some optimisations were learned from Havran et al. (Havran et al. 1998). A few optimizations are worth noting:

- Nodes that have axis aligned planes are handled specially. Distance to axis-plane tests are performed without the dot product. When the test line segment is split, the mid point calculation is different for axis and arbitrary planes. Reciprocal absolute line segment components are pre-calculated to eliminate a divide in that case.
- The function calls itself recursively rather than using an explicit stack. This proves to be faster on Intel based PCs but should not be considered optimal for all machines.
- The traversal uses iteration if possible and only recursion when necessary. If the line segment is not split and no new variables are required, the function

can repeat using a loop, merely swapping local values such as current node or test segment end points.
- Test information is stored in a structure and passed as one parameter, minimizing the number of parameters pushed for each function call.
- If the line segment is entirely on either side of the plane, it is passed in its entirety to the next test. This prevents the mid point from drifting outside the original line segment and compromising stability.
- Most traversal performance is gained from skipping sub trees beyond the range of the search, and by trimming the test line segment whenever it intersects a plane. This is standard behaviour for a BSP traversal routine.

Here are some other traversal optimizations that were tried, but proved inefficient:
- Instead of passing the line segment test as two end points, and calculating the mid point if split, pass the line segment as Min and Max normalized times. This sounds like a good idea because it removes the midpoint calculation, but instead, there is a higher cost for non-axis aligned planes. If the tree only held axis aligned planes, this may be the preferred method.
- A large number of configurations and reordering of code were tried including: Unrolling, using indices instead of conditions, minimizing code at the expense of increased recursion, front to back traversal of the line segment, careful and unchecked mid point calculation.

When a node containing polygons is intersected, all the polygons at that node are tested with the line segment. This test will terminate early if any of the polygons are hit. The mid-point calculated by the node plane is not useful here because the polygons stored in the node do not lie precisely on the node plane. They lie, at any orientation, within an epsilon distance of the node plane.

6.5.3 BSP Epsilons

Epsilons are relatively small numerical values used to control an error margin. Epsilons are most commonly used to preserve stability, particularly when working with floating point numbers where precision is limited and slides up and down the number range. The BSP code makes extensive use of two epsilon values, with a couple more for less critical use. Here is a list of the epsilons used:
Major:
- Polygons are classified during tree construction as being in front, behind, on or spanning a hyper plane. This classification routine uses a *construction* epsilon. This epsilon is critical for a) stability of the construction phase, b) number of nodes generated for the tree. This epsilon causes the hyperplanes to have a thickness, otherwise they would be infinitely thin.
- During tree traversal, the current line segment (or test shape) is classified as in front, behind, or spanning the current node's hyperplane. A *test* epsilon is used here. This epsilon must be larger than the *construction* epsilon as it has to accommodate for the *thick* planes use during the construction phase and any error that may be introduced by traversal calculations.

Minor:
- Axis splits cause polygons to be classified against two bounding boxes. Any polygon that is at least partially in a box is classified as intersecting, and is

stored in the node represented by that box. The classification function uses a tiny epsilon for working with floating point values.

- Polygons are described as if their edges form a set of planes. The planes are pushed outward by a tiny margin to overcome floating point error and fill any tiny gaps that may unintentionally exist.

Both the major epsilons described above are of critical importance to the efficiency of the BSP. Understanding the role played by these epsilons allows them to be tweaked to provide a significant performance increase. Here are some extreme examples: If the construction epsilon size is larger than the entire mesh, the whole mesh will fit into a single node and every single polygon will be tested with each traversal. On the other extreme, if the epsilon is almost zero, polygons will split others in the set, into a huge number of tiny fragments, repeatedly trimming tiny slivers off each other, as they reach the limits of floating point precision. This is particularly observed when working with arbitrary polygon soups. With tiny epsilons, care must be taken because a polygon may try to split itself, as its own points may not lie close enough to its own plane, as often occurs with greater than three sided polygons. Correctly sized epsilons prevent both these extremes from occurring.

Note that the epsilon values relate to a) the units of measure and dimensions of the mesh polygons, and b) the size of mesh features. For example, a mesh that is 0.023 units wide will have an epsilon, perhaps 1000 times smaller than a mesh that is 23.0 units wide. Also, if the mesh was 20 units wide, but had many features at 0.001 units, it may require smaller epsilons for both construction and traversal, to efficiently partition the mesh.

6.5.3.1 Construction Epsilon

The *construction* epsilon is necessary so that polygons can be classified as expected. Consider a cube, having six faces, with two triangles per face. This mesh should produce a BSP that has six nodes that form a perfectly *left* or *right* weighted tree. This type of tree or sub tree represents a convex hull. During construction, each face of the cube becomes a hyperplane for a node. A node will hold the two triangles that form one face of the cube, however the future of the other eight triangles touching this plane is not as obvious. These triangles must be classified as if they were entirely in front or behind, even though their points or edges are touching the plane. So a polygons is classified like this:

In front	→	Vertices are > +epsilon.
Behind	→	Vertices are < -epsilon.
On	→	Vertices are > -epsilon and < +epsilon.
Spanned	→	Vertices are < -epsilon and > +epsilon.

Notice that only the polygon vertices are considered during classification, not the polygon plane. Because the plane is *thick*, a polygon may have any orientation and still be classified as *on* the plane.

6.5.3.2 Test Epsilon

If the BSP test epsilon is calibrated to be slightly smaller than the distance that sample points are offset from the surface beneath them, a beneficial effect occurs. Recall that most time is spent in point to point visibility testing. Since both start and end locations are not on, but in front of their respective surfaces, and the test epsilon is smaller, the source and destination surfaces are never tested. Only potential occluders

in between are tested. If the test epsilon were larger than the sample offset distance, the source and destination surfaces would always be tested even though they could not be occluders. As surfaces may be highly tessellated, this configuration can significantly increase performance.

6.6 Polygon testing

If the large numbers of scene polygons represent flat convex surfaces, they may be stored as *n-gons* (greater than three sided polygons) and tested with greater efficiency. Care needs to be taken when mesh data contains n-gons, but not all the polygon points lie on the polygon plane (within a reasonable epsilon). In this case, the polygon should be tessellated into triangles to ensure stability, otherwise tree construction and tree traversal epsilon values will require enlargement that may adversely affect performance.

6.6.1 Ray vs. Poly

Determining the intersection of a ray and polygon is often a two-phase operation. First, the ray intersects the plane of the polygon, then that intersection point is tested to see if it lies inside the polygon. If no extra information such as the polygon normal is precomputed, the fastest ray versus triangle test is reportedly the barycentric style test described by Möller and Trumbore (Möller et al. 1997). Even compared to methods with precomputed components this method performs well. It has the side effect of also computing the barycentric coordinates useful for mapping into the triangle for things like texture or normal sampling. A slightly faster method for computing the intersection of a line segment and a polygon, uses the distance of the two line segment endpoints from the plane, to calculate the normalized time of intersection. This test can early-out if both distances are on one side of the plane. The normalized time is then used to project a point onto the plane and a *point in polygon* test is performed.

All kinds of point in polygon tests exist, but one of the simplest and fastest is to walk around the edges of the polygon in a clockwise manner and test if the point in question is to the left or right. If the point is not to the right of any edge, then it is not inside the polygon and the test can exit immediately. This method works for all convex n-gons. Since the point lies on the plane of the polygon, this method need not occur in three dimensions, as only the two dimensions other than the dominant axis of the face normal are required. A similar method turns the edges into planes. A dot product test determines if the point lies in front or behind the plane. Once again, this test need not be performed in 3D. So, instead of storing three points and a normal to describe a triangle, two axes (2 bytes) and three edges are used to describe the triangle. Each edge is a 2D normal and distance (total 3 floats). The new polygon description is almost the same size as the original polygon vertices version but now the test if a point is inside a polygon is just:

```
if ((edge[i].x * point[ix] + edges[i].y * point[iy]) < edges[i].d) { return false; }
```

Repeated for each edge of the polygon.

As noted by Steve Worley, the only way to simplify further is if the edges were described as lines with the equation Cx+y=d. The edges would need to be in homogenous space and horizontal and vertical lines would need to be special cased.

6.6.2 Edge/Point reordering to improve point-in-poly early-out

Both triangles and n-gons can have their edges reordered to accelerate point in poly testing (Green et al. 1993). The idea is to bound the polygon with a rectangle then test how each edge would cut away the remaining space outside the polygon. The method for n-gons and triangles is a little different, though conceptually the same. Imagine if the polygon was an octagon. It would be best to test opposite edges first, rather than just work around the winding. After the first four opposite sides had been tested, most (space) culling has occurred and an early-out likely achieved. Experiments with triangle edge reordering showed 0-1.5% speed increase, for the process using heavy point in poly testing. As the precomputation time is low and the run time improvement always positive or negligible, this optimization may always be worth adding. The implementation described in this paper does not use explicit bounding rectangles on individual triangles, but the voxel-like subdivision within the BSP is believed to create a similarity, causing this method to work.

6.6.3 Translucent textures

When testing polygons that have a translucent or transparent texture, care must be taken to prevent neighboring polygons that share and edge from being redundantly tested, causing extra filtering along the shared edge. This problem can be overcome by recording the intersection time. If another hit occurs on a translucent polygon at the exact same time, then it is most likely to be a shared edge and can be ignored (assuming polygons are not layered and coplanar).

6.6.4 Coplanar polygons

Polygons on the same plane with opposite normals are sometimes used to simulated double sided, thin objects such as plant leaves. The back facing polygon to a ray test can be biased so it is not detected before the front facing polygon. If this is not done, coplanar polygons may interfere when sampling normals on a surface.

6.7 Lighting

A simple lighting model is used by this implementation. *Direct* lights use a lighting equation similar to that of the DirectX and OpenGL fixed function pipelines. Surface lights and *indirect* lights use approximations of the radiosity form factor. The intention is to rapidly produce a nice looking and representative, though not necessarily accurate lighting result. Other more accurate lighting equations may be used for both direct and so called indirect lights. The output of this implementation is a 24bit texture that is modulated or 2x modulated with the base diffuse texture(s). As the result is a diffuse texture with shadow and light color information, specular and other effect lighting is added afterward for improved realism.

6.7.1 Approximate surface light equation

Surface lights and Indirect lights both use equations that approximate the energy transfer between surfaces and are similar to the radiosity form factor. Refer to **figure 8** and for a simple diagram of the lighting configuration.

$$\frac{\cos A \cos B}{\pi r^2 + biasAreaA} \times visibility$$

Eq 1: *Surface* and *Indirect* Light

Visibility is determined by opaque occluders and translucent media. It is typically just 0 or 1, but may be a RGB filter in the range 0-1 for each color channel.

Equation 1 represents light attenuation. The source light color and intensity is already taking into account the source emitter area effect. As the lighting is calculated per texel, energy calculations and visibility testing are approximated by points and ray tests with relative areas introduced for consistency.

The radiosity form factor represents the proportion of energy leaving one patch and received by another patch. **Figure 9** Shows a point to area form factor thats has an interesting geometric property. The form factor is the area of the projection of patch k, projected onto a hemisphere centered at j, then projected orthogonally down onto a circle.

Surface area light diagram.

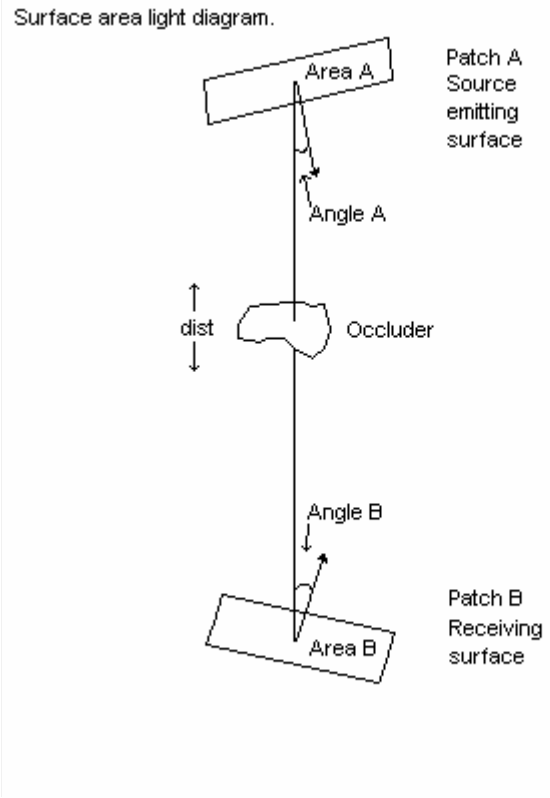


Fig. 8: This diagram shows the configuration of emitting and receiving patches. In the described implementation, the emitter may be a *Direct Surface Light*, or a *Virtual Indirect Light*.

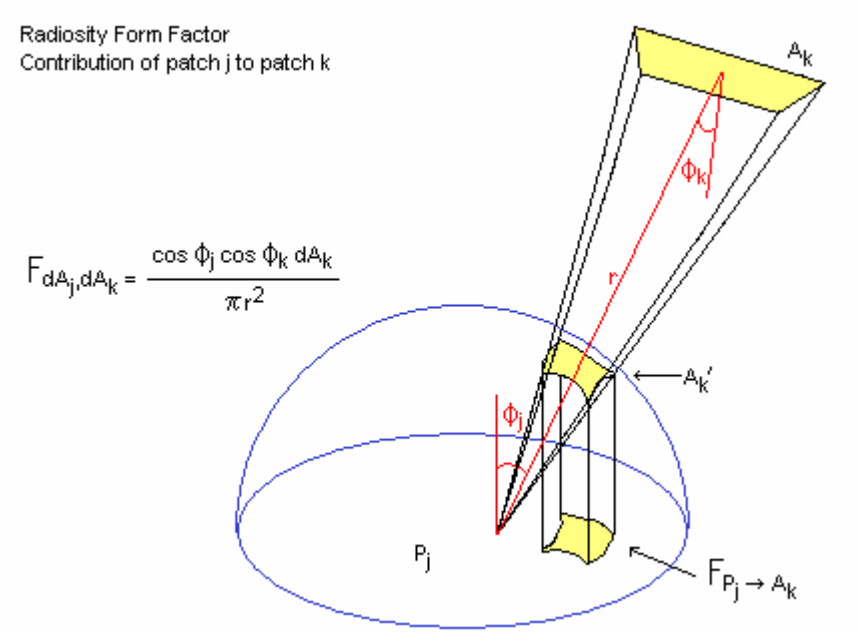


Fig. 9: This diagram shows how surface areas influence each other, as if a patch is projected through a hemisphere and dropped onto a disc.

Equation 1 uses an *area bias* to compensate for the fact that area light sources are simulated by point sources. As described earlier, bright spot artifacts will appear close to point sources unless corrected. **Figure 10** demonstrates the numerical effect of the area bias in the lighting equation. **Figure 2** showed the visual effect.

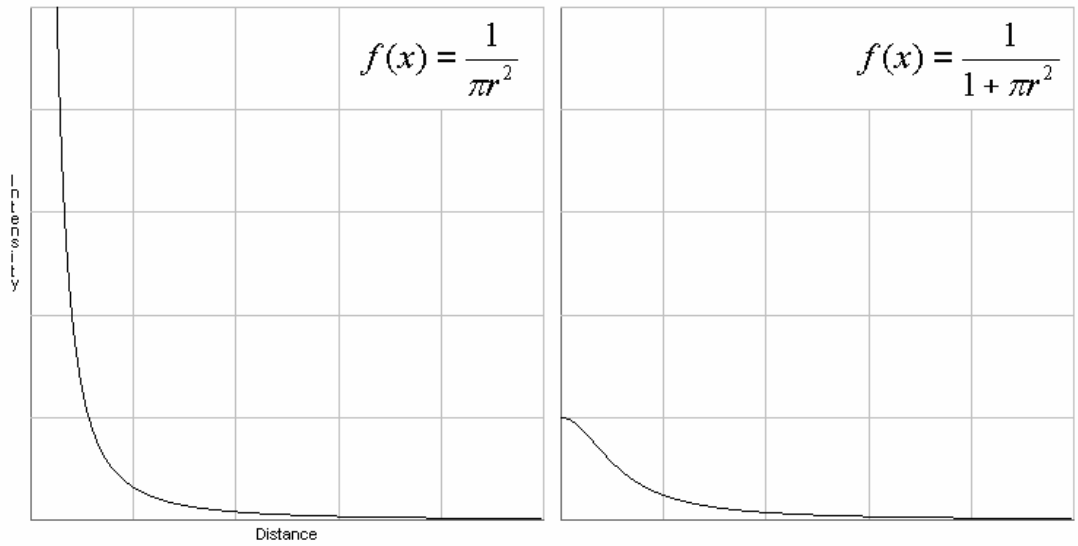


Fig. 10: The effect of the *area bias*. The extreme intensity near to the light source is effectively clipped back with little effect elsewhere. A bias of 1 is shown in this example.

6.8 Color Correction and Output

When light sampling is complete, the color samples must be converted into a format suitable for rendering via texture mapping and viewing on a monitor. The color samples are 32bit float per color channel, they will be converted to 8bit unsigned integer per color channel. Converting number ranges in this way is often called quantizing.

If the sample values were between zero and one, the conversion would simply be a matter of scaling to the new number range and losing precision. The samples actually range from zero to infinity, or at least some large positive number. Because of this unknown upper bound, the values must be clipped and optionally adjusted in some way.

The human eye and brain combination is capable of recognizing millions of levels of brightness, from the darkness of an overcast night, to the blinding light of the sun on a clear day. Computer video cards typically allow a color value between 0 and 255. Monitors often have an analog color range, but can only achieve the darkness of a blank screen and the brightness of a fully lit pixel due to the limitation of the CRT, LCD or other display technology being used.

If two lights shone on a single surface, a very bright magenta (red, blue) and an extremely bright cyan (green, blue), the surface would be saturated in color across all components. Since the cyan light was brighter, it would be preferable if the final color was not clipped to white (full bright), but showed the hue of that intense color. The following algorithms describe the color clipping via normalization and color clipping via saturation:

Clip Normalize – Preserves Hue
```
max = largest color component
if max > 1.0
        for each color component
                color = color / max
```

Clip Saturate – Just clips components, often resulting in white
```
for each color component
        if color > 1.0
                color = 1.0
```

Lights vary in intensity and the human eye automatically adjusts to the lighting situation in a short period of time. Photographs and images cannot capture exactly what the eye would see, but the controls on cameras allow adjustment to make the image at least look representative. One such control is *exposure* (Elias 2003). Exposure adjustment has the effect of changing the intensity of bright and dark areas in the image. **Equation 2** describes the adjustment operation. **Figure 11** shows a comparison between the original and two settings of exposure adjustment.

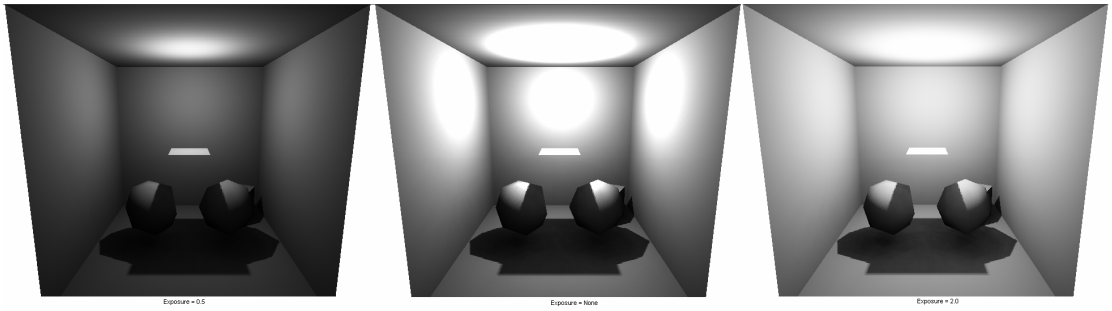


Fig 11: Exposure adjustment. Left set to K=0.5, Right set to K=2.0, and Middle original image without adjustment. Notice the obvious hot spots in the non-adjusted middle image where intensity is clipped.

$$brightness = 1 - e^{-light \times K}$$

Eq 2: Exposure equation. K is a correction constant.

As the exposure adjustment works on the original high precision color values, it must be performed before color quantizing. The final step before outputting the color data is to combine multiple texture samples that were taken for anti-aliasing. This step is performed after quantizing in order to soften high contrast edges as part of the anti-aliasing effect. For example, if two over-bright samples were combined, the result would still be over-bright, and will be clipped for no tangible effect. Samples that were already clipped will merge to produce better in between values.

6.9 Lightmap Texture packing

Texture pages typically represent power of two sized, possibly square textures that are efficient or are the required dimensions for video hardware. Sizes such as 256x256 or 512x512 are common. Many such texture pages may be required to store the lighting result of a scene.

Lightmaps may be generated from individual polygons, projected neighboring polygons, or complex parameterised *charts* of polygons. Whichever way lightmaps were created, they will be numerous, and will likely have complex shapes. The lightmaps must be packed into the smallest number of texture pages to make efficient use of limited memory.

Mip mapping has the effect of blurring nearby texels together at every subsequent level, eventually leading to a 1x1 texture that often represents the average color of the whole texture. When different textures are packed together into the one texture page, the average, and even the neighboring texel colors are often unrelated. Artifacts that show discoloration become visible, with increasing frequency as lower level mip maps are selected. In an effort to reduce this problem, unused texture space may be filled with texels relating to the lightmap or extra border texels may be added. Many software programs that use lightmaps only allow or generate the top few mip levels.

Textures need to be optimized for cache efficiency. Video memory is limited, and the cost of swapping textures in and out of video memory, or chip memory is high.

Textures near each other, that will likely be rendered together, should be packed together. The classification of what *near* is can be implemented in various ways. Lightmaps that are classified as near each other will be assigned a grouping identifier that will cause them to be packed together if possible.
Examples of grouping classifications are:

- The object identifier or index in the scene set.
- A room or *cell* that may define parts of the scene.
- A octree division of the scene into blocks.

6.9.1 Box sorting

Optimally sorting boxes and other shapes is known as a hard problem because depending on early sort operations, that may appear to be good, later ones are effected, perhaps negatively. This is true for BSP construction. Luckily there are simple alternatives to find the ultimate solution, and these simple approximations yield excellent results.

Boxes are stored in groups, so the sort process works on a group of boxes that should be packed together. Boxes are sorted from large to small. Height and width form the major and minor sort keys, so that same height boxes are still ordered based on width. Identical size boxes have no defined order. The idea is to create a set of *free* boxes, that are sorted small to large, and place the lightmap boxes into these *free* boxes. When unused space remains, it is split into more boxes and each of these is added to the *free* list. If there are no free boxes available, a new full size box is created, logically representing a new texture page. The following algorithm describes the process.

Add lightmap bounding boxes to a *wait* list, where they wait to be placed.
Sort all *wait* boxes from large to small, by height then width.
While boxes remain in the *wait* list
    Find a *free* box that is just big enough to store this *wait* box.
        (Free boxes are kept sorted from small to large.)
        If there are no *free* boxes, a new texture *page* is created at the maximum texture size. This new page has one full size *free* box, and it is added to the *free* list.
        The first box in the free list that is large enough to hold a waiting box is used.
        Calculate up to two smaller boxes from unused (usually L shaped) space around the placed box, and add these to a *free* list'
    Call the empty texture space reclaim function.
Free unused *free* boxes
Call the unused texture space reduction function.

6.9.2 Unused texture space reduction

When texture pages are less than half full, an optimization can be attempted that will create two, quarter size textures and pack them. The process may continue recursively for several iterations before limits such as the minimum texture size is met, or lightmaps use most of the texture page space. A texture page is considered half empty when less than half the space measured vertically is used. When this occurs, it is already known that the height of the tallest box is less than half the page

Height. If the width of widest box is also less than half the width of the page, then the page is a candidate for reduction. Two quarter size pages are created and added to a *free* list. The contents of the page are run through the same algorithm described above so that the boxes are placed once again.

5.9.3 Empty texture space reclamation

When a lightmap is placed, it uses up a rectangle equal to its size, but the lightmap image may be *thin diagonal*, *hollow*, or a *complex convex* shape, causing much texture space to be wasted. At this point in the process, a space reclaiming function may be called in an attempt to locate (rectangles of) free space and add them to the *free* list so they may be used. Many complex algorithms could be developed to locate this space. Many other methods could have been used to reduce this situation, such as:

- Splitting convex lightmaps during their construction phase. This would create smaller, convex shapes, but increase the number of seams and discontinuities in the lightmaps.
- Rotating lightmaps from diagonal to horizontal or vertical alignment, or to fit arbitrarily shaped free space. Although diagonal shapes may be *corrected*, and other shapes may benefit from improved fitting, rotation causes variation in the texture axes, highlighting seams.

Both these methods add complexity, extra work and generate new artifacts, even if they may increase texture space efficiency. It is debatable whether these methods are even worthwhile, as discussed later.

The chosen method simply samples the lightmap at regular intervals, such as up to 8 times in width and 8 times in height, and attempts to flood fill rectangles. If a seed position was not flagged as *empty* (by flagging with a NaN, or other Id), a rectangle is grown, one row or column at a time in each of four directions until it is blocked. If the rectangle is larger than some threshold, it is added to the *free* list and available for use by other lightmaps.

Tests showed that increasing and decreasing the configuration parameters, though reclaiming more or less space, did not reduce the number of texture pages used. See **figure 12** for some examples. Attempting to reclaim more space takes lots more time, but pays rapidly diminishing returns. The reason for this is that number of texture pages is largely determined by the number of *large* lightmaps. The smaller lightmaps merely fill out the wasted space left by the large ones. The large lightmaps may be efficient, with most of their space filled with valid color samples, or inefficient, with large quantities of flagged invalid samples. The inefficient cases are usually *hollow*, *diagonal* or *concave*. Since the flood fill method predominantly reclaims square shapes, this is not a problem, because horizontal and vertical rectangular shaped lightmaps are already packed well by the original method, and contribute little to the number of texture pages used. The inefficient cases are readily filled with boxes and their space reclaimed. Note that this method works best when lightmap shapes are mostly homeomorphic to discs. If they were all, for example, identical size triangles, they would not pack at all, and reclaimed space would remain unused. There are potential cases that would hinder the regular sample flood fill method. If a lightmap looked like a grid that just happens to lie in perfect alignment with the sample pattern,

it may contain plenty of free space, but none if it would be discovered by sampling. This pathological case is unlikely to happen, and if it did occur regularly, a possible solution is to simply jitter the regular sample points by up to half their step size.

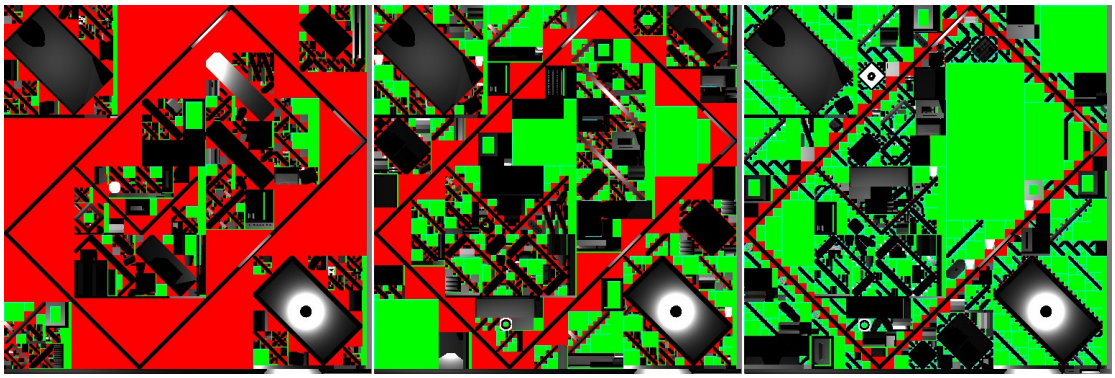


Fig. 12: Left, Middle and Right show small to large amounts of reclaimed space respectively. In these images, Green (Light Gray in B&W) is reclaimed space, Red (Dark Gray in B&W) is unused space and lightmaps fill the used space. Note that although lightmaps are shuffled around, no more or less texture pages were used, even in these three extreme examples.

6.9.4 Box sorting performance optimizations

When a naïve box packing process is run on tens to hundreds of thousands of lightmaps, it may start to slow down. The QuickSort algorithm was used to sort *wait* boxes. A fast insertion sort maintains the *free* box list. A block memory allocator rapidly feeds the process with new *free* boxes. These tools significantly reduce the packing time that would otherwise occur with a brute force implementation.

6.10 Image Improvement Techniques

6.10.1 Mutli Sampling

Mutli sampling lights by turning point sources into surfaces has some anti aliasing effect, but is only part of the solution. Various parts of a scene will not be classified such that a soft shadow will render there.

6.10.2 Anti-aliasing

The two main types of anti-aliasing experimented with in this implementation are multi sampling and super sampling. Multi sampling uses a table of offsets such as the corner and center points of a texel. These extra samples are then averaged, possibly with weighting. Super sampling merely renders a higher resolution output, typically 2x or 4x some size, then filters the result down. A quick and effective alternative is to sample texel corners, as this logically only adds one extra row and column of unique sample points. With this method, the samples at two edges of each texel are shared with its neighbours so they really don't need to be computed at all. The processing cost is tiny and the result is adequate for our purposes.

Unfortunately all types of anti-aliasing tend to reduce the effectiveness of correcting texel samples located inside solid space (discussed later in this section). This occurs

simply because multiple correct samples are merged to produce an incorrect result. It is not possible to assume that lighter or darker sub-samples are more correct than each other, as it may be light or shadow that is bleeding into the wrong place.

6.10.3 Detection and correction or samples inside *solid* space

Often, near walls, or places where two surfaces meet, texel samples are generated such that they bleed the wrong color. For example, a wall made from two opposite facing polygons, with no thickness, may divide a floor that is lit on one side, but not on the other. Depending on where the texel sample is, light or darkness may bleed from the other side of the wall. To correct this, walls, as in real life should be modeled with volume. The sample texel size should relate to the feature size of the model geometry. If the model is tiny or has intricate details such as thin pipes, a higher resolution texel is required.

Another solution is to use CSG (Computational Solid Geometry) methods and turn the scene meshes into a *skin*. All overlapping and intersecting polygons are resolved into a skin around the solid space with hidden and junk parts of the mesh removed. This is a difficult process to make robust, introduces many new mesh faces, modifies the model geometry, and still doesn't help in cases where objects are stacked on each other without merging.

A generic solution was found that works with polygon soups. The method is stable even when meshes contain triangles that intersect, are poorly shaped and do not form a manifold style surface. However meshes that contain closed manifold style surfaces that do not obviously intersect produce much better results. This method tries to determine if a texel sample point is inside solid space, and if so, correct it by shifting the sample point up to one texel width away. First the sample point is elevated slightly away from the surface, via its normal, or the polygon face normal. This is a good practice because it can prevent accidental collision with the owning face, as well as facilitate other optimizations.

A set of tiny rays are cast from this point. Only 4 to 10 rays are necessary for good results. If any of the rays hit a back face, then the texel sample is inside solid space, otherwise it is okay. If the texel sample point was in solid space, a set of position offsets, ordered from near to far relative the original sample point, are tested. When none of the shift attempts succeed, the sample point is left unmodified, because it is likely to be well inside solid space and remain unnoticed. See **figure 13** for before and after results. This method runs reasonably fast as it exits early when the sample point is already valid, which is most of the time as the output resolution is increased. The tiny ray tests are not very expensive as only a small portion of the scene is considered for each test. It may be best if objects that rest on top of each other are made to intersect each other slightly, in order to ensure that sample points between them are detected as *in-solid*.

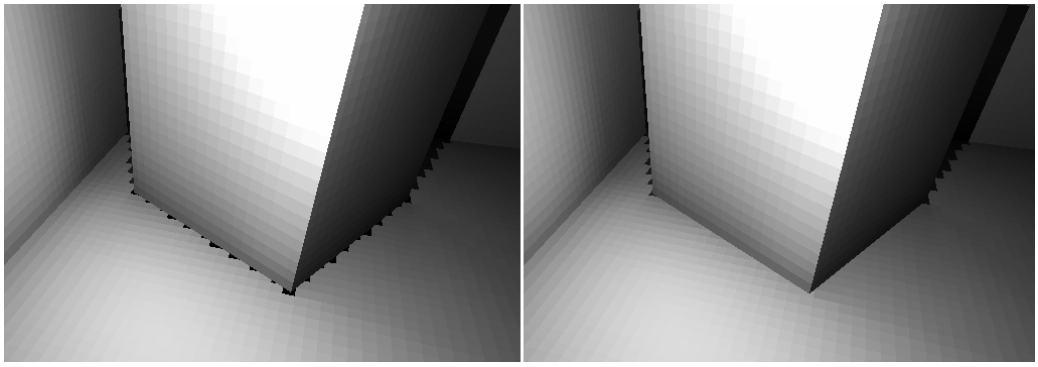


Fig. 13: Texel sample points inside solid space. Left: Color bleeding from samples located in solid space. Right: *Bad* sample points detected and corrected, by shifting to a valid nearby location.

6.10.4 Texture seam reduction

Whether textures are generated from individual triangles, axis projections, or a minimal distortion texture atlas (Lévy et al. 2002), when the textures meet up along a polygon edge, they may display an artifact called a texture seam. Refer to **figure 13** for examples of this.

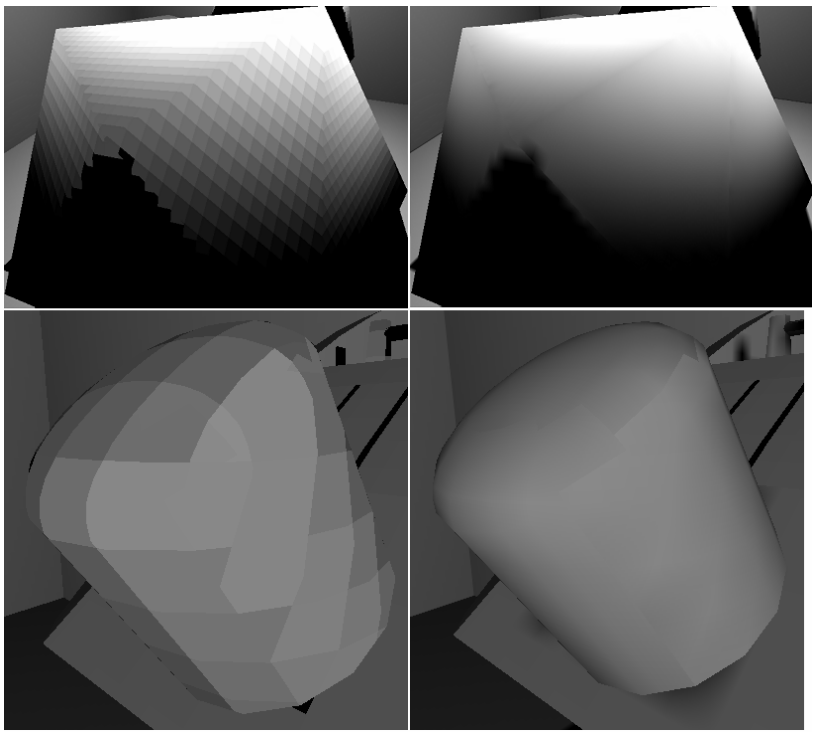


Fig. 14: Texture seams. Several seams are visible. Left is unfiltered, right is bilinear filtered. Both sets show curved surfaces with different dominant axes mapped with low-resolution texture and rendered with bilinear filtering

The following are some factors that contribute to the visibility of seams:

- *Texel orientation*. The two dimensional U and V directions may not be identical between polygons, particularly if the polygon normals have different dominant axes (such as one major in +X and the other major in –Z).

- *Texel Size and shape*. Texels are often aligned with two of the world axis and look like Lego™, but because they are mapped to sloped polygons, the final shape of the texel is distorted. On a single face, it may be skewed into a diamond shape. Across several smoothed faces, it may look quite unusual. If polygons use different texture resolutions, then the texels on either side of the edge will be of different size.
- *Texel origin relative to vertex position*. Unless texel origins are snapped to vertex positions (introducing a bunch of other issues), the texture axes don't line up, or line up along one shared axis only.
- *Texel color*. If the texture color along a seam is of high contrast, the seam will be much more visible.
- *Texel neighbors*. When bilinear filtering is used, four texels are sampled in vertical and horizontal directions. If the neighboring set of colors is not matched across the edge boundary, and does not extend at least one texel, seams will appear.
- *Mip maps*. When mip mapping is used, neighboring texels are recursively filtered down. Mip mapping of packed lightmaps causes completely unrelated colors to be merged and displayed. Mip mapping within a single lightmap merges colors such that they may no longer match their edge neighbors.
- *Bilinear filtering*. Bilinear filtering exaggerates the seams initiated by most of the above factors.

Texture distortion is inevitable when a curved surface is projected onto a plane, or flattened in some way. This is the classic map makers problem. The texture seam issue is very much related, as curved surfaces, when flattened must join up, even if the entire surface is flattened into a single chart.

Surfaces that are not curved do not pose a problem, as samples along polygon edge boundaries may simply be sampled beyond the boundary onto the next shared surface. If there is no shared surface, the sample point can be pushed back inside the original surface polygons. Extra *border* samples created to improve bilinear filtering or mip mapping may be handled in a similar manner, shifting or sampling outside, rather than by extrapolating existing samples.

7 Optimizations

7.1 Caching

7.1.1 Last hit poly/object caching

It is possible to take advantage of coherence for visibility testing. After all, if a visibility test for one sample intersected an occluder, there is a reasonable chance that the next sample may also be occluded by the same object or polygon. Both these items are cached, the last polygon that was hit, and the object owning that polygon. The visibility ray test simply checks the cached polygon first. If that polygon was not hit, the cached object is tested first. If there is still no hit, the cache is invalidated and the test continues without the benefit of the cache. Experiments showed that allowing the cache to have a lifetime of up to three misses could increase performance, but

more experiments would be required to check if there is consistent benefit. Caching can provide a significant performance increase. An estimate of 30% is not unreasonable, with better performance at higher resolution and when occluders are regularly close to light sources.

7.1.2 Potential occluder object test sets

As a lightmap may consist of multiple polygons, and a light may consist of multiple samples, it can be worthwhile bounding these higher-level objects and caching some information about their relationship. If both the light source and the destination surface have bounding boxes, a convex hull can be computed that bounds all light rays that could possibly be considered between the two entities. This hull can be used to collect a list of objects that are potential occluders, and this reduced set of objects may be reused for all ray tests with between the light and the destination object or surface. The cost of computing the bounding hull is small and amortized over the (usually) numerous ray tests. Since the two objects are boxes, the bounding hull may be computed directly, without the need for generic algorithms like QuickHull (Eddy 1977).

7.1.3 Redundant poly tests in the BSP

Polygon references are stored in the BSP multiple times. This occurs because the polygons are not physically split, but are logically split, so a hyperplane dividing a single polygon merely causes two references to the same polygon to be stored. This means that both *parts* of the polygon may be tested at different times. It is possible to flag the polygons in a way that prevents redundant testing. Enabling or disabling this cache proved to provide a tiny speed increase or decrease. The benefit is only realized when lots of redundant testing occurs. The effectiveness is highly dependant on the BSP tree construction method, configuration, and the mesh geometry. With highly optimized *point in poly* testing and the relatively expensive cost of random memory access to a flag or cache table, this potential speed-up should be attempted with care.

7.1.4 Precalculated lighting values and constants

Some parts of the lighting equations may be pre-calculated. Anything that is constant should be calculated and stored. Some of the non-constant parts of the lighting equation that are reused should be stored locally to *baby sit* the compiler. If high precision is not required, storing reciprocals and multiplying can reduce the number of divisions. Along that line, modern PC CPUs have low precision floating point instructions that can speed up reciprocal divides and square roots. Some equations can be merely reordered to make better use of precomputed constants. The spot light specific code benefited from this, with divisions completely removed.

7.2 Early outs

The fastest code is the code not executed. There are various tests, and parts of the process where the result is logically determined before it is exhaustively computed. Here are some:

- *Light behind surface*. Simply test if the incident light would strike the front face of the texel sample. This is a dot product test that is already part of the lighting equation.

- *Insignificant light*. Like other types of radiation, light decays or is attenuated as the distance from the source increases. The tricky part here is determining when the light energy is no longer significant and can be ignored without being noticed. An effective way to determine if the light is significant, is to compare with a *minimum energy threshold* and a *percentage of emitted light threshold*, which ever is larger.
  This early out occurs before the (relatively slow) visibility test and can fire up to 40% of the time (with 5-30% more common), but is highly dependant on the number, size, and position of lights. Remember that this lighting method relies on using large numbers of (often tiny) lights, so the conditions for culling are ideal.
  The values found to be reasonable in the described implementation were:
  Fixed Light Threshold = 0.04% of saturation, or 0.1 / 255 for 8bit final color components.
  Emitted Light Threshold = 0.1% of source energy, or 0.1 * LightIntensity.
  These values can be adjusted for increased speed, or output accuracy. In the described implementation, the visual difference in output was indistinguishable through a range of extreme tests, while the performance increased significantly.
  If the thresholds are set too high, a scene may suddenly become dark as lights are culled too early. If the thresholds are set too low, no performance increase will be realized. The combination of the fixed and relative threshold, as described, greatly reduces the chance of sudden visual artifacts due to changed lighting conditions.
- *Light source too far from destination object or lightmap*. Lightmaps and objects have simple bounding volumes such as boxes or spheres. If the closest point on the bounding volume is further from the light than a threshold, then that light's contribution is insignificant, and should be ignored. The threshold value is calculated in a similar manner to the *insignificant light* test described earlier. The threshold should be conservative, only exiting early if the light is definitely too far away to consider.
- *High level scene information.* High level scene information such as Cell & Portal connections, Rooms, Object hierarchies and visibility tables should be used whenever they are available, and if their knowledge could be coded as a early out test. For example, a scene with Cell & Portal descriptions may have a Cell to Cell visibility table. That table can be simply accessed to determine, for example, that a light inside Cell A could not possibly be seen in Cell B.

Here are some non-early outs:

- Back face of polygon. It may seem intuitive to back face cull as you would with normal rendering, but if the scene represents solid objects and the test is for occlusion only, then checking for back faces is a waste of time. It is best to treat polygons as double sided as long as they are opaque.
- BSP Traversal. When traversing a BSP for the sole purpose of visibility testing (shadow testing), there is no need to logically traverse a ray or line segment from start to end. There is also no need to delay testing polygon contents if there is a possibility the polygons might be intersected by the test. The idea is to test the polygon contents stored in the BSP tree as efficiently as possibly, not using naive or generic ray testing traversal methods which would

provide otherwise useful things such as *first hit* results. The intention is to exit as soon as any occluder is detected.

8 Future extensions

1. The technique presented in this paper could be implemented in hardware for both view dependant and view independent rendering. Today's programmable GPUs could handle the most time consuming part of the process: lighting and visibility testing. A pixel shader could compute per pixel lighting values and add them to a destination buffer. A stencil or shadow buffer could be used to perform visibility testing between lights and scene geometry.

2. The movie 'Final Fantasy: The Spirits Within' used many laboriously hand placed fake lights to simulate global illumination (though it did use a photon map for caustics) (Christensen 2001). The part of this process used to identify and create potential indirect light sources may easily be implemented as a software plug-in for other renderers. This could assist artists, or relieve them entirely of the task of placing many fake lights to simulate realistic indirect light. First, a (relatively small) photon map is constructed from the scene geometry and light sources. An arbitrary number of fake lights are then quickly created and passed back to the application for acceptance or modification by an artist.

3. Currently the texture seam issue is not resolved adequately, so this area needs more work. Recent research on automatic texture atlas construction is yet to show an efficient, robust solution to texture seam removal, beyond the significant reductions achieved by the various methods. Many of the parameterization algorithms still introduce significant texture distortion making them unsuitable for our use.

References

Arvo, Jim. “Linear-Time Voxel Walking for Octrees”. Ray Tracing News 1(12), March 1988

Arvo, James; Kirk, David. “Particle Transport and Image Synthesis”. In Computer Graphics SIGGRAPH 90 Conference Proceedings, pp 63 – 66. 1990.

Christensen, Per H. “Photon Maps At Square USA”. Part of “Faster Photon Map Global Illumination”. From SIGGRAPH Course#38 slides. 2001

Eddy, W. F. “A new convex hull algorithm for planar sets”. ACM Transactions on Mathematical Software. pp398-403, 411-412. 1977

Elias, Hugo. “Exposure” http://freespace.virgin.net/hugo.elias/graphics/x_posure.htm. 2003

Goral, Cindy M; Torrance, Kenneth E; Greenberg, Donald P; Battaile, Bennett. “Modelling the Interaction of Light Between Diffuse Surfaces. In Computer Graphics” In Proceedings SIGGRAPH 84, volume 18, pp 212--222, ACM Press. July 1984

Green, Chris; Worley, Steve. “Simple, Fast Triangle Intersection”. From Ray Tracing News V6N1, January 1993

Hasenfratz, Jean-Marc; Lapierre, Marc; Holzschuch, Nicolas; Sillion, Francois. “A survey of Real-Time Soft Shadows Algorithms". In Eurographics State of the Art Report. Eurographics. 2003.

Havran,V; Kopal, T; Bittner, J; Zara, J. “Fast robust BSP tree traversal algorithm for ray tracing”, Journal of Graphics Tools , December 1998

Jensen, Henrik Wann; Christensen, Per H; Suykens, Frank. “A Practical Guide to Global Illumination using Photon Mapping”. Siggraph 2001 Course 38, Los Angeles Aug 2001.

Jensen, Henrik Wann. “Global Illumination Using Photon Maps”. From Rendering Techniques, Proceedings of 7th EuroGraphics Workshop on Rendering. pp 21-30. Springer-Verlag/Wien. New York. 1996

Keller, Alexander. “Instant Radiosity”. In SIGGRAPH 97 Conference Proceedings, Annula Conference Series, pp 49-56. 1997

Kopp, Nathan. “Simulating Reflective and Refractive Caustics in POV-Ray Using a Photon Map”. http://www.nathan.kopp.com. May 1999.

Lévy, Bruno; Petitjean, Sylvain; Ray, Nicolas; Maillot, Jérome. “Least squares conformal maps for automatic texture atlas generation”. 2002

Möller, Tomas; Trumbore, Ben. “Fast, minimum storage ray-triangle intersection” Journal of graphics tools, 2(1) pp 21-28. 1997

Sillion, F. X; Arvo, J. R.; Westin S. H.; Greenberg, D. P. “A Global Illumination Solution for General Reflectance Distributions”. In Proceedings of SIGGRAPH 91, in Computer Graphics. pp 187-196. July 1991

Appendix A: Sample output

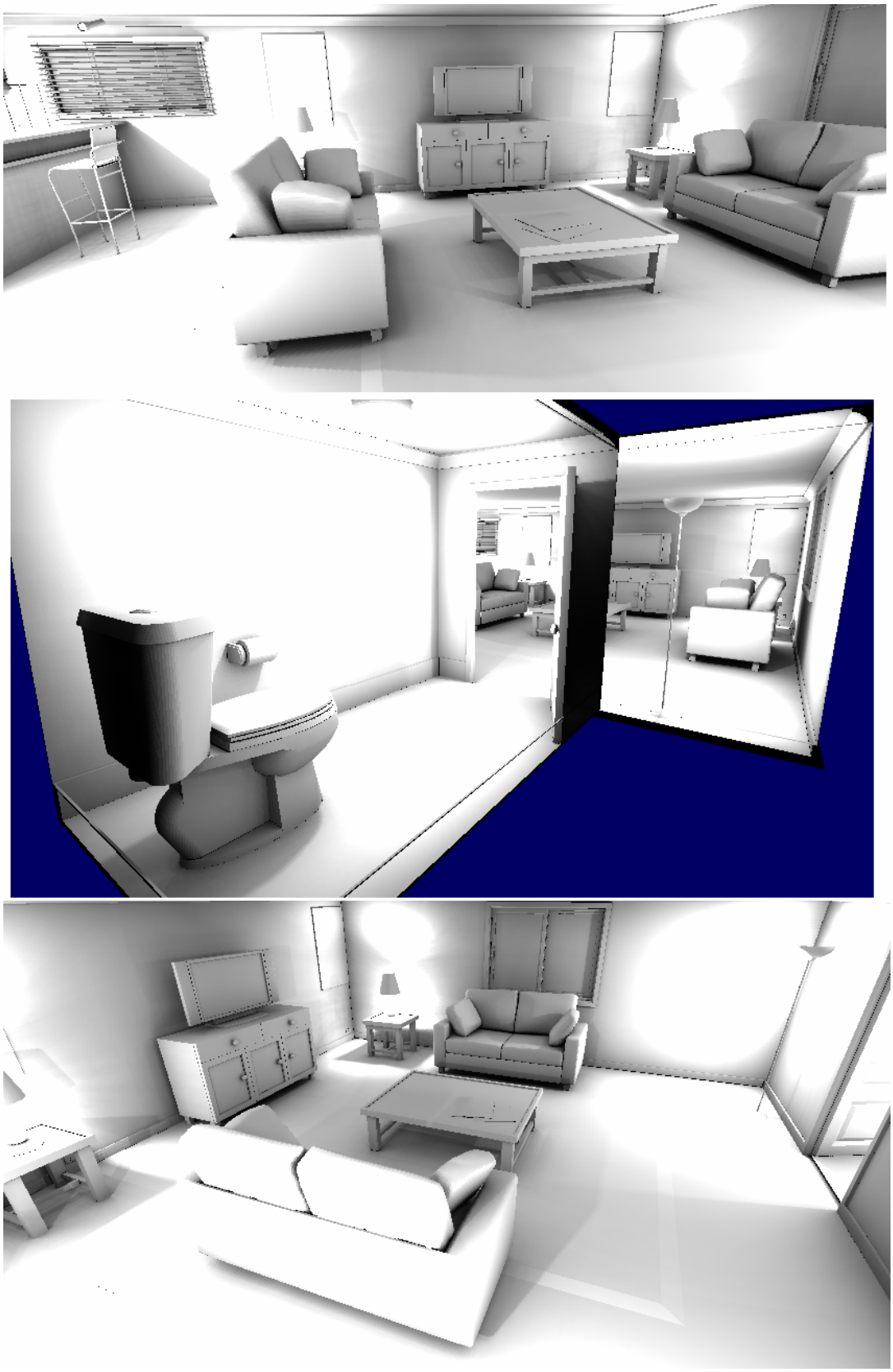


Fig A.1: A test room lit and rendered in real-time by the described technique. Only the lightmaps are rendered.

Appendix B: Early attempts - Described by the author

In the past, I had written light mapping tools for use with computer games, that produced results similar to those seen in the Unreal™ and Quake™ games. I used hybrid mixes of ray tracing and radiosity, favoring a method of blending high resolution ray traced *direct* light with low resolution radiosity style *indirect* light. Although the results looked impressive (at least compared to the competition) I was unhappy with many parts of the process. The radiosity phase required passing the scene geometry through a *potato chip cutter* to form patches of reasonable size. Form factor calculation could have been stored in memory, had I had 4-10gb of it. The whole process really needed several CPUs working in parallel to reduce the process time.

Now, some years later, and even with this new technique, not all problems are solved, and some new ones are created. Expectations are higher, geometry poly count is up, and smooth surfaces are frequent. I started out investigating Photon Maps and pretty soon became quite excited by what I was reading. I hurriedly implemented a simple photon mapper and it worked nicely, just as the papers said. But, and there's always a but. I didn't at first realize why no one was doing what I was trying to do, that is, produce a final result by directly visualizing the photon map. The two major problems with the photon map are:

1) The photons are sampled using random and statistical means and look blotchy. Filtering is required at the very least in order for the results to look acceptable.
2) The edges of polygons, the edge of the scene, and overlapping polygons cause *border artifacts* where the photon energy density is sampled incorrectly producing either extra bright or extra dark output depending on the implementation. Refer to **figure B.1** for a basic border case.

Refer to **figure B.2** shows output illustrating both these issues. Both of these problems do have solutions, such as using more photons or clipping the geometry to the sample shape, or using a z-buffer to calculate surface area. All of these solutions however, cause a dramatic slowdown and detract from the elegance of the photon map technique. Modern ray tracers often use the photon map to direct some other indirect lighting technique, or build an irradiance map which is sampled as part of an indirect light contribution process.

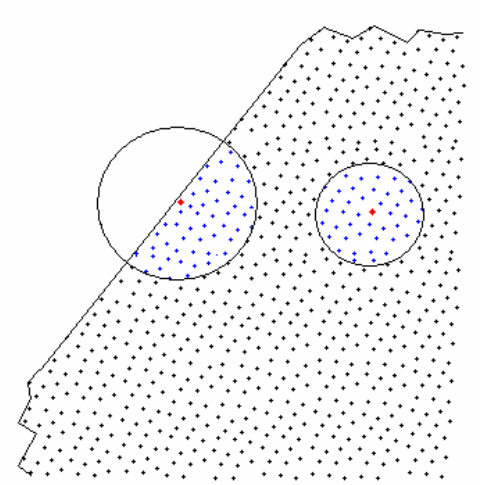


Fig. B.1: Two photon samples. The photon density sampled on a border is incorrectly estimated. The sample size has grown too large while finding *n* samples. The geometry independent photon map does not contain knowledge of borders and density correction methods are inelegant.

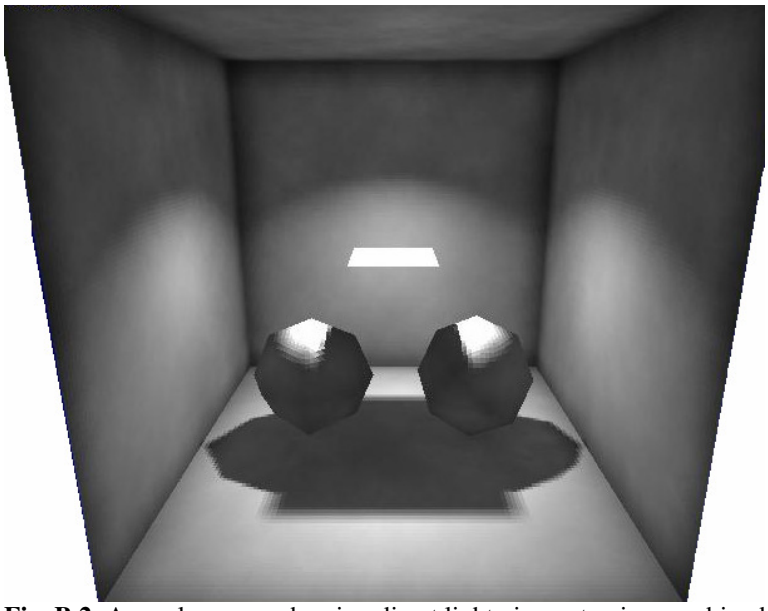


Fig. B.2: An early screen showing direct light via ray tracing combined with indirect light via direct visualization of the photon map. Notice the extra dark edges and corners caused by incorrect density estimates. Also notice the splotchy indirect light cause by randomness.

Quite disheartened, I feared I may have to give up the photon map technique completely, and looked at various other methods. I briefly investigated spherical harmonics (Sillion et al. 1991), but it did not appear to be useful to this specific task. I consider spherical harmonics to be an energy reflectance representation compressed on the surface of a sphere, and may be useful for some other, perhaps even real time, approximate lighting solution.

After browsing a paper on Instant Radiosty (Keller 1997), I recalled an idea that I had been thinking about for years and experimented with before: Turning indirect light into direct light sources. My earlier attempts had all failed, but I realized I now had the tools I needed; a photon map to collect emitted and reflected energy, and a balanced kd-tree to sample it back into light sources.

I initially used an accumulation buffer (as Instant Radiosity does) to test this new technique, but found that although it produced a artifact free result, the lighting was bland and low in contrast. Hot spots near lights disappeared and areas with little light became even darker. I called this the 'chemo' method since both the good and bad parts of the solution were removed; the high contrast lights, and the artifacts associated with the virtual light technique.

I tried merely adding light contributions to the output, and although this produced excellent lighting results, the bright spot artifacts near the light sources were unacceptable. Biasing the light source area in the lighting equation proved to effectively remove these artifacts while preserving the light integrity.

Many more challenges remained. The new technique turns the few lights in a scene into hundreds or thousands of lights. Accelerating the quantity of light samples required carefully choosing, implementing and configuring various acceleration structures.

B.1 Optimization

Something that the author has learned about low level optimizations, is that they are highly machine/system dependant, and the rules change over generations of hardware. For example, current CPUs have high arithmetic performance and relatively slow (non-cached) memory access. Thus techniques that involve precomputation or look up tables for acceleration, should be used with great care.

Unfortunately most current CPUs (and GPUs) realize their full potential only when performing a large homogenous process. This low level parallelism is difficult to take advantage of without restructuring code (often into an ugly or non intuitive state), or spending large amounts of time writing and testing assembly code. The best experience is when the compiler identifies code that could be optimized with vector operations and automatically does so. Vector and Matrix manipulation are obvious candidates for such optimization, but are not always part of a bottle-neck inner loop. As always, operations like *square root* and *divide* consume disproportionate amounts of clock cycles. Ordering conditional statements to aid branch prediction is useful. Packing data into minimal structures, aligned in memory at contiguous addresses, is critical for efficient memory access. Object Oriented methodology should be used with care in performance critical areas lest they introduce hidden, unnecessary cost. Imagine if each photon structure was derived from a base object class, allocated with *new* and implemented virtual function access to member data. Thought that may sound amusing, during the authors working life, he has seen commercial code, usually (but not always) written by enthusiastic junior programmers doing things exactly like that.

B.2 Strange Discoveries

During development, some unusual discoveries were made relating to performance. Further investigation revealed these issues are known, though rarely highlighted. Calculations involving INF (infinite) and denormal (extremely small) numbers can approach 1000 clocks of CPU time, or take several hundred times longer than expected. NANs will also cause massive slowdowns, but will not occur unless there is an error. The penalty times differ greatly between CPU brands and modules such as the FPU and SSE found on the Pentium 3 and 4 CPUs.

Appendix C: The kd-tree

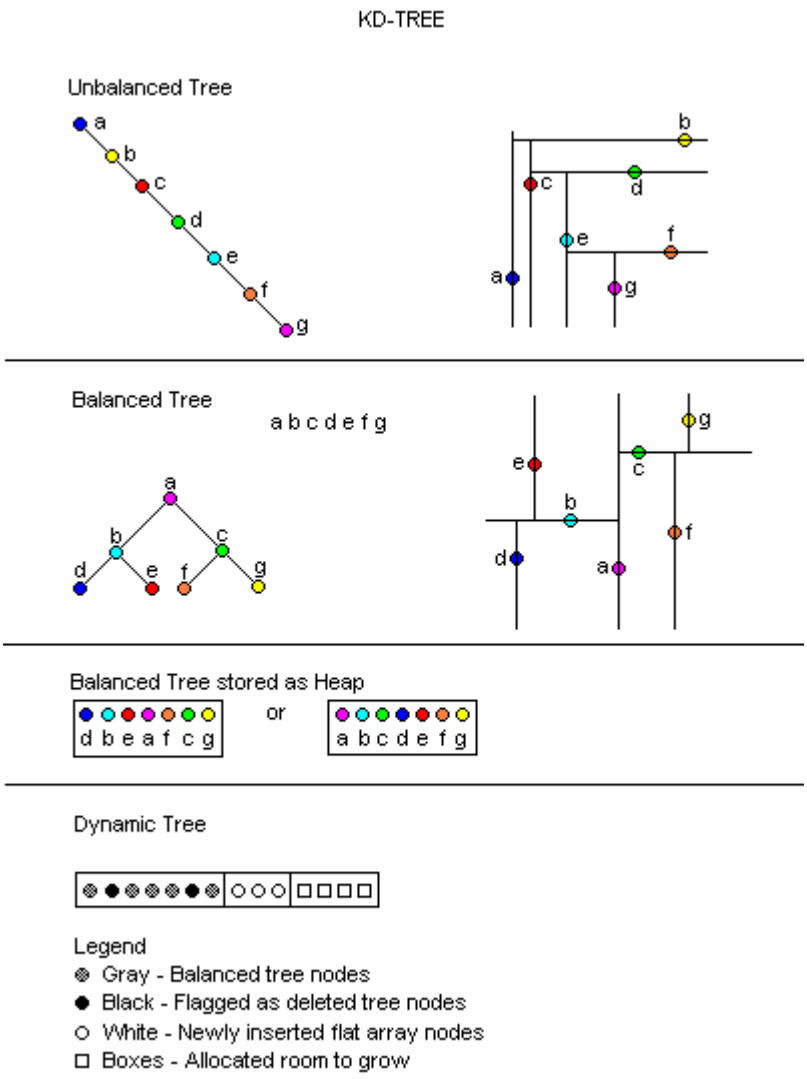


Fig C.1: kd-trees. Unbalanced, Balanced, Stored as Heap and Dynamized versions.

The kd-tree used in this implementation is similar to the one described by Jensen. The structure is a balanced three-dimensional binary tree that is stored like an array and similar to a heap. The index of each element implies its location in the tree without the need for child pointers (such as Left & Right or Front & Back.) Because it is balanced, if the last branch in a sub tree has a single child, that child must be the Left child as the Right child index is out of range. (That is why trees constructed this way are often called '*left balanced*' binary trees.) The tree is constructed using a *median split* approach. The algorithm is a '*divide and conquer'* method that is very similar to the QuickSort algorithm.

The kd-tree data can be constructed with the root node located at [0] or at [mid]. If the root is at [0] then the children are located at curIndex*2+[0,1]. This method could possibly provide better cache and stack performance. If the root is at [mid], then children are located at (endIndex+startIndex)/2+[-1,1]. This method allows all children to be found between the *start* and *end* indices and is the method used by this implementation.

Dynamic kd-tree

Kd-trees are rarely dynamic because they realize their speed from careful construction, something that cannot occur with arbitrary dynamic data. Exported mesh data is often near worst-case for a dynamic kd-tree as it does not appear in a spatially random order at all, thus causing a dynamic tree to grow in a very unbalanced fashion. Testing for uniqueness while adding vertices is also a worst-case scenario because the *Insert* and *Find* operations are interleaved. Kd-trees are most efficient when all data is added, and then the tree is balanced. Search operations may then be performed in a highly efficient manner. The kd-tree was dynamized by adding new data to an array that could be accessed as a second phase within the *Find* operation. If data is removed, deleted nodes are flagged as removed, but not actually removed from the structure. Periodically the tree is rebalanced, with the deleted nodes removed and recently inserted nodes distributed appropriately.

Appendix D: Notes on the Polygon BSP

The BSP nodes
The BSP nodes were originally homogenous and referenced by indices rather than pointers. Indices have many benefits such as simplifying serialization of the structure. Indices can be shorter than the pointer word size and indices can store other flags in unused bits. Since the BSP contains some nodes with axis aligned planes and others with arbitrary planes, the axis planes stored unnecessary data causing memory to be used less efficiently. Two different node structures were created, with different sizes. Although stored in contigous memory, the nodes are now referenced with pointers. An average of 20% of the nodes (range 15-30%) are axis planes, and tests showed that a significant amount of time was lost due to Level1 cache misses. Mixing smaller and larger nodes improved memory efficiency and increase process speed by 9%.

Some other failed BSP optimization attempts were:

- Balanced binary tree. The intention was to control tree size and make traversal times more consistent. The result was consistently large trees and consistently longer traversal times. Balanced branches do not help in practice. Dividing space is more important.
- Least cost polygon split planes determined by favoring node balance, split polygons, or polygons on the plane. None of these choices or combinations consistently produces better than ~2% improved traversal times than *random* or *first polygon* choices. However, rejecting expected *bad* polygons like > 30% splits may help.
- Arbitrary split planes constructed using geometry data such as edges and vertices rather than polygon faces. The result showed that objects like a sphere could be split into more pieces than the single sided convex hull that would otherwise be expected. The traversal cost also proved to be slower on average. Adding more split planes can increase the traversal cost and must be done with care.
- Tree constructed with only axis aligned planes, or only planes from the contained polygons. Neither of these methods are optimal when used alone.

Some other potential, but not implemented BSP optimizations could be:

- Try to add top level nodes to the tree that are effectively simple bounding hulls around (particularly disassociated) bunches of geometry. At least one paper describes a similar method, though the results showed that it had a positive or negative effect depending on the mesh/scene. Perhaps manually splitting a complex object into multiple parts with simple bounds, to be queried as an earlier test, would be better than attempting to integrated such bounds into the BSP.
- Test fire rays from outside and inside the mesh and use the resulting statistical information, in some way, when deciding how to construct the tree.
- When the top levels of the BSP are created in this implementation, bounding boxes are known, representing the voxel dimensions. It would be nice to pass convex hulls to polygon-plane splits further down the tree. As polygon-planes divide space, the hull must be clipped. Using this information, better splits could be found, to divide the *space* rather than naively divide the *geometry*.

This method may have the greatest potential, if it could be done efficiently. In addition, polygon points/edges could be reordered optimally for early-outs based on this knowledge.

- Actually split the polygons, build the tree, then throw the split polygons away. This method was tried in an earlier implementation. Tree construction may be slowed by the resulting huge numbers of polygon fragments and inaccuracy may be introduced with tiny, sliver polygons.

Appendix E: Process performance statistics.

Process test results
These test results were gathered at one point during the development of the process. They may not accurately represent the current or potential performance of the process. In addition, some debugging and statistics gathering code was enabled in the test build. The test results are provided for the following reasons:

- To compare image quality with different settings.
- To compare process time with different settings
- To provide rough figures of process efficiency. Eg. Does this technique require minutes, hours or days to produce usable results?

Notes on this scene and the test output:

- The tests use the same texture resolution throughout the scene. Each object, or part of an object could have used a different resolution for a more efficient result. For example, the door knobs and tap handles are small features that are lit poorly. These would greatly benefit from higher resolution, while the ceiling and some bench surfaces could have used a lower resolution without compromising the quality.
- The lamp shades covering three of the lights present a difficult case for the indirect lights. Many virtual lights are clustered around the lamps light source, on the inside surface of the lamp shade. A large number of indirect lights are required to reduce banding caused by the high intensity light from the lights in the lamp shade.
- The table of statistics shows that the total time is determined by the number of texels (the surface area and the lightmap resolution) and the number of lights (a combination of multi-sampled direct lights and virtual indirect lights).
- Test1 is darker than the other tests. If too few virtual lights are used, the indirect light in the scene is not approximated accurately enough and may look noticeably different. The minimum number of indirect lights to produce a consistent approximate result does vary from scene to scene. A rough figure is 500-1000 indirect lights, but as little as 200 may be adequate for simple scenes. Test3 used 1000 indirect lights and performs well. Only the *hard* cases of the lamp shades highlight inadequate numbers of indirect lights in those areas.
- Test4 uses the highest resolution, the largest number of lights, and more photons than earlier tests. Notice that the overall brightness is relatively similar to Test2 and Test3 that use one half to one tenth as many indirect lights. Also note that the light distribution in Test4 is the same as Test6 that took much less time by generating a lower resolution output with the identical light settings.
- Observation of the time and quality statistics shows that varying the number of light samples or the lightmap texture resolution can produce similar lighting results with dramatic differences in processing time. Quick previews can be performed in minutes, useable results in minutes to hours and high quality results in many hours.

System Configuration:
Intel Pentium 4 2ghz
WindowsXP Pro
512mb PC2100 DDR RAM
(Up to 50mb RAM actually used by process, when photon map, or output textures exist in memory.)

Scene Statistics:
Number of Polys: 67131
Number of Lightmaps: 5237
Number of Mesh objects: 35
Number of Smooth groups: 3227
Number of Light sources: 5
Scene Dimensions: 11.4m x 10.0m x 2.5m

Process Options:
Anti aliasing - off
Solid space sample correction - off for Test1,2,3, on for Test 4,5,6
Reclaim texture space - on
Reduce texture pages - off
Max photon bounces – 16

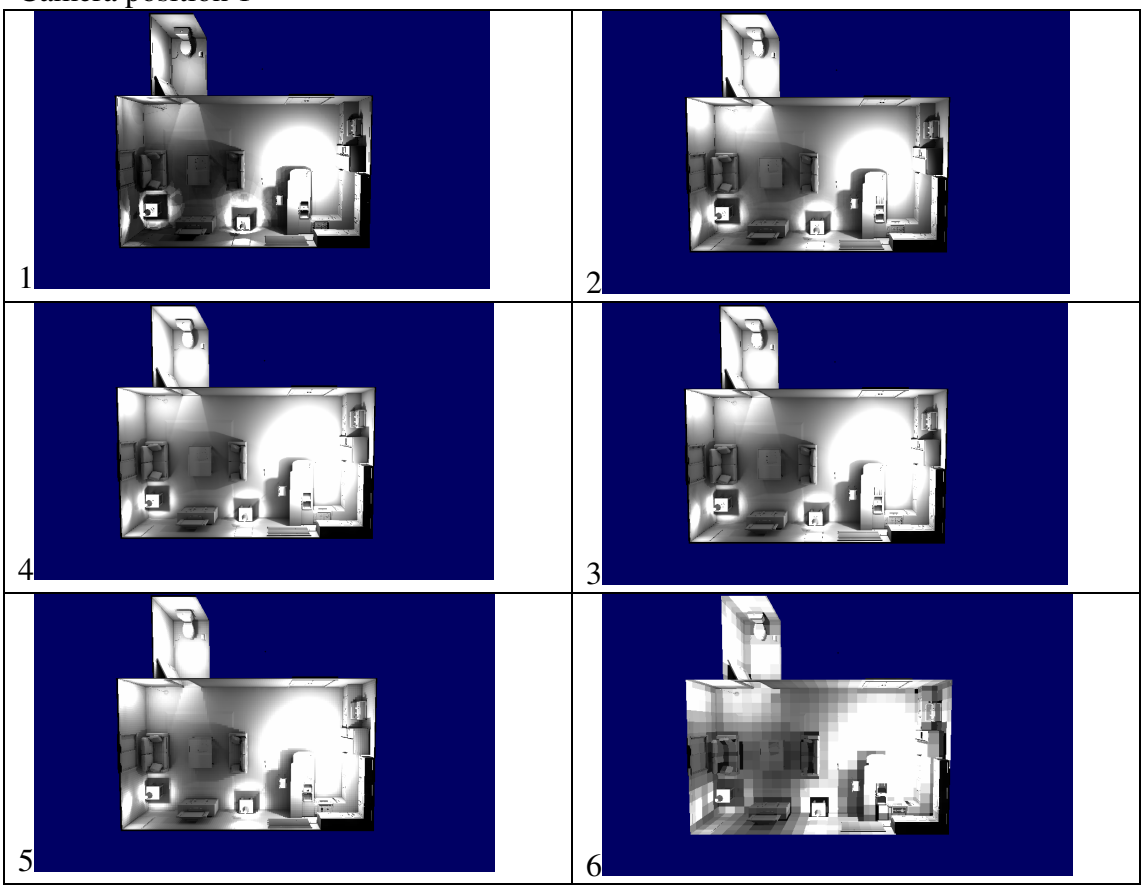
Operation	**Test 1**	**Test 2**	**Test 3**	**Test 4**	**Test 5**	**Test6**
Num Photons	100,000	1,000,000	1,000,000	2,000,000	2,0000,000	2,000,000
Lightmap resolution	0.05m	0.025m	0.025m	0.01m	0.1m	0.5m
Multisamples per direct light	20	50	50	200	200	200
Number Indirect lights	100	1,000	5,000	10,000	10,000	10,000
Total texels	685,682	1,892,375	1,892,375	7,317,169	327,842	158,387
Total Tex sample points	**569,454**	**1,392,430**	**1,392,430**	**5,987,230**	**299,660**	**157,030**
Total Light sample points	**200**	**1,250**	**5,250**	**11,000**	**11,000**	**11,000**
Total process time	2m 8.5s	19m 26.7s	1h 18m 57s	10h 40m 18s	40m 5s	24m 43s
Prepare Geometry	17s	17s	17s	17s	17s	17s
Create Smoothgroups	722ms	722ms	729ms	725ms	722ms	723ms
Create Lightmaps	4.9s	4.9s	4.9s	5.15s	4.9s	4.9s
Build Photon Map	2.8s	20.66s	20.71s	41.7s	41.8s	41.25s
Balance kd-tree	152ms	1.73s	1.72s	3.56s	3.58s	3.59s
Create indirect lights	104ms	1.16s	1.15s	2.13s	2.12s	2.17s
Build Polygon BSPs	11.3s	11.3s	11.3s	11.3s	11.3s	11.3s
Light World (& Calc Sample Points)	**1m 48.3s**	**18m 46.4s**	**1h 18m 17s**	**10h 39m 15s**	**39m 3s**	**23m 40s**
Pack Lightmap textures	78ms	522ms	523ms	714ms	31ms	13ms

The images have been ordered in a snaking pattern so they may be compared to neighbors of similar configuration.

Order of images:

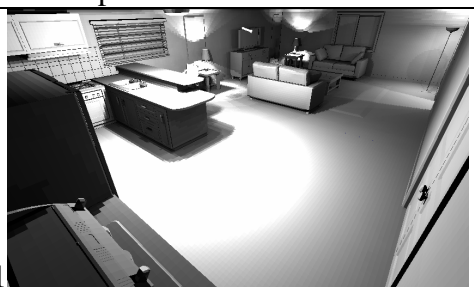
1 → 2
↓
4 ← 3
↓
5 → 6

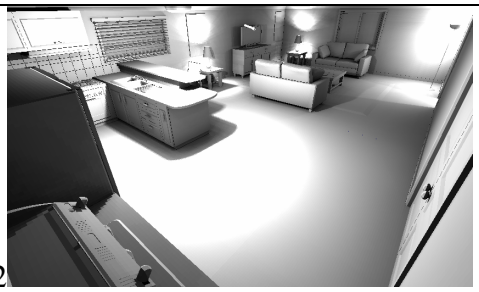
Camera position 1

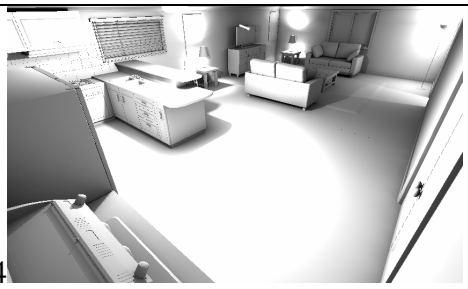


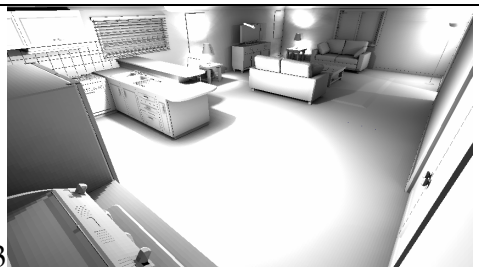
Camera position 2

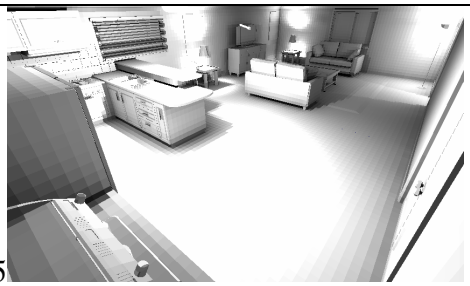
Camera position 3

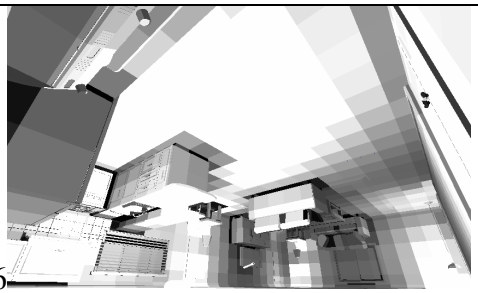

1


2


4


3

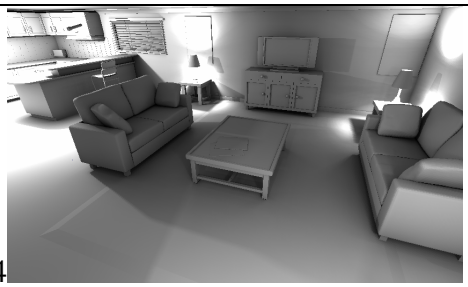

5

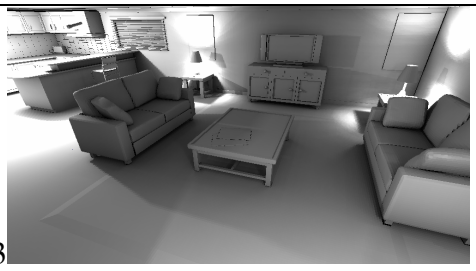

6

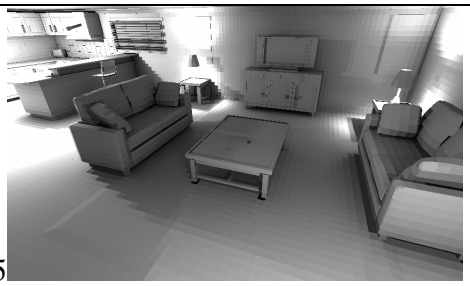
Camera position 4

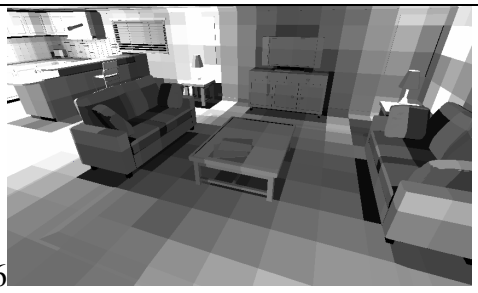

1


2


4


3


5


6