The Path-Buffer

Bruce Walter*
Jed Lengyel*

Abstract

With a small modification the z-buffer can be made flexible enough to be used in global illumination algorithms traditionally handled only by ray casting, but still enjoy the benefits of z-buffer algorithms including predictable performance, memory coherence, and acceleration using specialized z-buffer hardware.

The fundamental difference between the path-buffer and a standard z-buffer is that the global near clipping plane is replaced with a per pixel near clipping value z_0. The success of each pixel write depends on compares with both the current z and z_0 values. We demonstrate the power of this additional test by showing how two global illumination algorithms can be adapted to use the path-buffer instead of ray casting.

First we adapt Kajiya-style path tracing for finding view-dependent global illumination solutions. Since the scene database is streamed through the path-buffer, and no complicated meshing or per-element storage is needed, very large and complex models may be illuminated. Both diffuse and non-diffuse reflectance functions can be handled as well any complex geometric primitives which can be rasterized (e.g. displacement-mapped spline surfaces). Second we show how the path-buffer can be used to do an image gather from a coarse radiosity solution in two-pass radiosity methods.

1 Introduction

Visibility testing is a fundamental problem in computer graphics and occurs as a step in most graphics algorithms. Many different methods have been proposed, but the two most common visibility tests in use today are the z-buffer and ray casting. The interactive rendering community has focused on the z-buffer because of its predictable resource requirements and because it is easy to accelerate using specialized hardware. When it was first introduced, the z-buffer[2] was considered far too "brute-force" and wasteful of memory. The z-buffer has since gained enormous popularity and is the basis for rendering engines that can scan-convert and do hidden surface removal on millons of primitives per second.[8][5] The high speed of these engines and the ability to handle millions of primitives inspired the path-buffer technique.

The global illumination community has tended to use ray casting for visibility testing. Nearly all current global illumination methods (hierarchical radiosity[7], clustering[16], path tracing[9]) use ray casting in part because the standard z-buffer lacks the flexibility needed to efficiently implement the visibility tests required by these methods.

Previous methods which combined z-buffers and global illumination include the hemi-cube[3] formulation of radiosity, and the two-pass radiosity method[18] where hemi-cubes of varying widths are used to compute shading from a previous radiosity solution. These methods order the computation by element or pixel respectively and require a rasterization step to gather the incoming light for each element or pixel in the scene. The path-buffer technique described below orders the computation by direction instead of by ray or by element and so requires only a constant number of rasterization steps to sample the directions of a sphere. Path-buffer algorithms also tend to be progressive in that they can quickly an image and then refine that image as more scan conversions are done.

Other work on illumination using scan conversion includes the item-buffer [21] in which a scan conversion is used to accelerate the inital step in ray tracing, and shadow maps [11] which scan convert from the view of the light to compute direct illumination. A survey of previous techniques combining z-buffers and global illumination is given in [17].

Other modified z-buffer and point-sampling algorithms (e.g.[13] [4]) keep more than one z value but use it for transparency and CSG (Constructive Solid Geometry) operations.

Section 2 describes the basic path-buffer algorithm. Path tracing using the path-buffer is described in Section 3 and gathers from radiosity solutions is discussed in Section 4. Some results from our software implementation are listed in Section 5. Appendix A gives some of the mathematical basis for path tracing and Appendix B gives some details of our software implementation.

2 The Path-Buffer as a Parallel Ray Caster

Ray casting and z-buffers are both solutions to the visibility problem and hence are in some sense equivalent (see Figure 1). Any ray could be replaced by a scan conversion with a one pixel z-buffer, and any z-buffer scan conversion could be replaced by a set of rays passing through each pixel.[1]

We can see some of the limitations of standard scan conversion if we think of the z-buffer as a parallel ray tracer casting a million rays (one through each pixel). The nature of the z-buffer places three constraints on the rays.

Restriction 1: All rays must either be parallel when using an orthogonal projection or pass through a single point when using a perspective projection.

Restriction 2: The rays are distributed in a regular pattern corresponding to pixels.

Restriction 3: All rays originate at the near clipping plane.

The innovation of the path buffer is that it eliminates this third restriction by splitting the near clipping plane into a near clipping value per pixel. Thus each pixel contains two

*Program of Computer Graphics, Cornell University, Ithaca, N.Y. 14853

[1] This may not be true if anti-aliasing other than supersampling is used, but we will not be consider such techniques here.

z values z and z_0, the near clipping value. The standard z-buffer tests z at each pixel and writes only if the new z is less than the previous z. The path-buffer only writes the pixel if the new z is greater than z_0 and less than the current z. If an update is successful we replace z with the new z but do not change z_0.

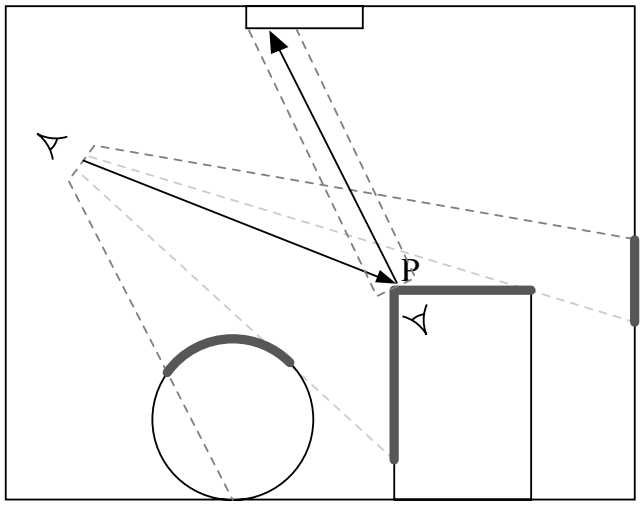

Figure 1: *A standard z-buffer rasterization from the eye point results in a set of points which are nearest to the eye-point but beyond the near-clipping plane. A second rendering from just behind P (with a near clipping plane passing through P) will find the closest surface, and in effect trace a single ray.*

2.1 Using the Path-Buffer

When adapting algorithms to use the path-buffer instead of ray casting, we still have to be aware of restrictions one and two. In this paper we will use orthographics projections for all rays except eye rays. This means that each scan conversion will effectively cast a large number of parallel rays as in Figure 2. Thus we will need to reorder the computation to use a large number of parallel rays in order to make efficient use of each scan conversion.

Given a set of parallel rays we want to trace, we will chose a orthographic projection in the direction of the rays and set the view frustum large enough to encompass all the starting points of the rays. We will then project each of the ray starting points onto the view plane to find the pixel closest corresponding pixel. The z_0 value for this pixel will be set to match the its ray's starting point. Scan converting the scene will then cast *all* of the rays *at the same time.* Afterwards we can read the results of each ray from its corresponding pixel.

Some error is introduced when matching rays to their closest pixel, but this has not caused a problem in our tests. However this will become more of a problem if ray starting points are widely separated or if the path-buffer resolution is small. This could be overcome by using sub-pixel positioning.

2.2 Advantages of Path-Buffer

Besides tracing the advantage of tracing multiple ray simultaneously, the path-buffer also has all the advantages of scan conversion and z-buffers. Scan conversion is easily parallelizable as evidenced by the multitude hardware accelerators which exist. The path-buffer does require an additional compare to z_0 on each write which is not supported by most current hardware. However the additional cost is both small and predictable and thus very suitable for hardware implementation.

There is a large class of modeling primitives which is easier to rasterize than to intersect with a ray. Imagine trying to efficiently intersect a ray with a displacement- and texture-mapped spline surface! Scan conversion also has the advantage of good memory coherence for texture and displacement map access.[4] In general ray-tracing tends to randomize memory accesses, which increases page thrashing. Whereas scan conversion can efficiently handle scenes which may be larger than main memory (e.g. procedure models) by streaming the scene through the scan converter.

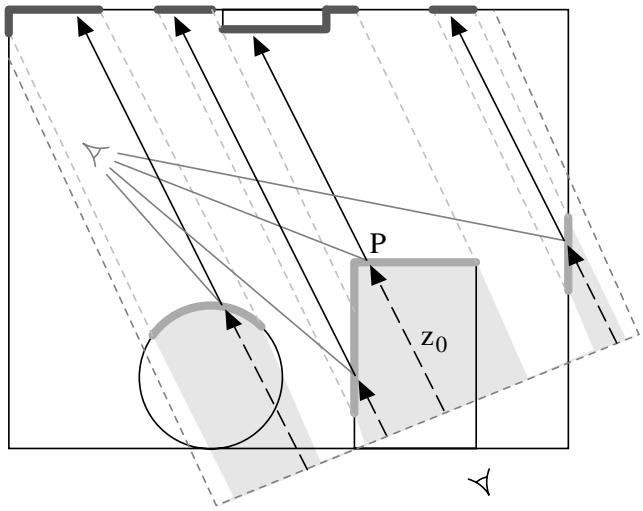

Figure 2: *The points obtained by a standard z-buffer rasterization from the eye point (as in Figure 1) are then projected onto the new viewplane. The path-buffer uses z of each projected point as the z_0 per-pixel clipping plane. The second path-buffer rendering is tracing* all *of the sample rays at once from all of the different starting points.*

3 Path Tracing

Path tracing[9] was introduced as a way to estimate the global illumination due to both direct and indirect lighting. Light transport is estimated by tracing possible light paths backward from the eye. These paths are generated randomly and by averaging over many such paths an accurate estimate of the total illumination is obtained. See Appendix A for more details on how path tracing works.

Path tracing has a number of benefits. It is a view-dependent algorithm which can handle arbitrary BRDFs (bidirectional reflectance distribution functions) including both diffuse and specular surfaces. No meshing is required and no additional data needs to be stored with the primitives, thus large scenes and procedural models can be handled.

The main drawback is that typically a very large number of paths must be computed. This combined with the expensive nature of ray casting has limited use of path tracing. However the quality of the images it produces has led to

a search for faster methods which capture the same effects. Ward[20] has developed a related method which caches lighting information at surfaces to reduce the number of rays needed. This system has found widespread use in the Radiance system for global illumination[19].

3.1 Ordering by Direction

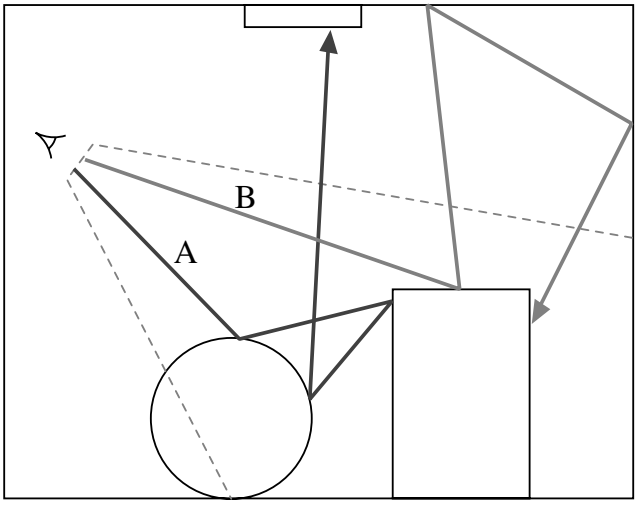

Figure 3: *Montecarlo path-tracing works by adding in the contributions of a finite set of energy-transport paths (chosen at random) to approximate the integral over all paths. A and B are two example paths.*

Standard path tracing works by computing paths in a pixel serial fashion with all paths for a single pixel being computed before the next pixel is started. The path-buffer can be used to greatly accelerate this process by simultaneously tracing one path for each pixel. However need to modify the way the paths are generated in order to get the ray coherence we need to efficiently use the path-buffer. In standard path tracing, the paths for neighboring pixel are generated independently like those in Figure 3. This causes all ray coherence to be lost after the first bounce. However we can reorder the computation by having the paths for different pixels share directions. As shown in Figure 4 this gives the parallel rays we need for efficient use of the path-buffer. This allows to shoot one path for each pixel by scan converting the scene a number of times equal to the depth to which we want to trace the paths. By repeating this process by number of paths we want per pixel, we can produce a path traced image using only z-buffer style scan conversion.

3.2 Direction Coherence

By sharing directions between the paths for different pixels, we have introduced a correlation in the error at neighboring pixels. This correlation will be visually obvious if an insufficient number of paths are traced. Visually this looks like noticable images of the scene being "pasted" on surfaces of the scene. The solution is to shoot more paths until this error is no longer objectionable which is also the solution for the error in ordinary path tracing. Each pixel will still converge to the correct value as more paths are used.

Another problem occurs at non-diffuse surfaces. Directional highlights are more difficult to correctly approximate

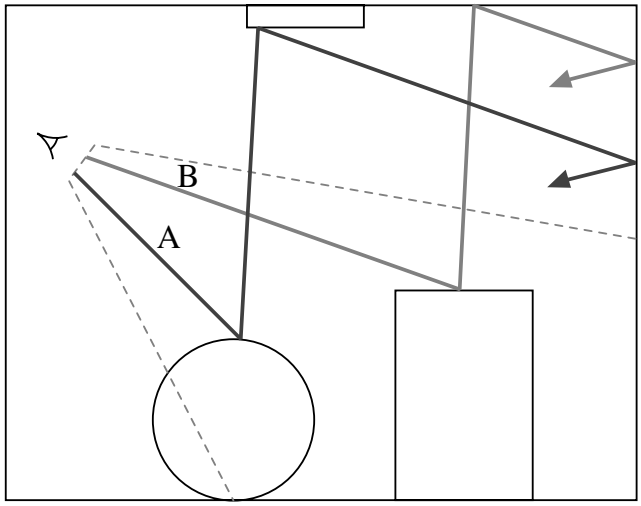

Figure 4: *Path-tracing with shared directions — path A and path B share the same direction for each bounce.*

by picking uniform random directions. Ordinary path tracing handles this by choosing the directions partially based on the local BRDF. However since our directions apply to many different paths, this is more difficult to do with the path-buffer. Thus for our implementation we simply use more paths when non-diffuse objects are present. We feel that this is more than offset by our decreased cost to trace a path as long as our BRDFs are not too highly directional.

3.3 Optimizations

Another standard acceleration technique for path tracing is to shoot a shadow ray to a light at each path intersection point. This makes each path a much better estimate of the true lighting and greatly reduces the number of paths which need to be shot. This technique can still be used in the context of scan conversion by using shadowmaps. We randomly choose a point on the light sources and generate shadowmaps for that point. The shadow rays are replaced by look-ups into these shadowmaps. To make shadows more accurate, we pick a new light point for each set of paths.

We can also get effects like anti-aliasing, depth of field, and motion blur using the techniques of the accumulation buffer[6]. Since we are already accumulating our image over many scan conversions of the scene, these effects can be added essentially for free.

4 Local-Pass for Radiosity Post-Process

The cost of computing a radiosity solution increases with the resolution of the mesh on the scene. Thus for complex scenes is often prohibitively expensive to find a radiosity solution on a mesh which is fine enough for direct display. However Rushmeier[12] noted good images can be produced much less expensively by using a two-pass method. In the first phase a radiosity solution is computed on a coarse mesh. Then the local pass makes an image by doing a gather from the radiosity solution at each pixel. The produces images with far more detail than could be captured in the coarse mesh.

Since the introduction of the this two-pass method, great progress has been made in accelerating the radiosity solution pass. However relatively little work has been done

on speeding up the local pass. Smits[16] has shown how the radiosity pass can be accelerated to the point where he now reports that the local-pass now accounts for most of his computation.[2] The problem will be even worse for animations where the radiosity solution is computed once, but the gather must be computed at each frame. We believe the path-buffer has great potential for accelerating the local pass.

The process of doing a gather is very similiar to that for path tracing. Possible light paths are still traced backward from eye. However the paths are limited to depth two. At the second bounce instead of continuing the path, we just use the computed radiosity value from the radiosity solution. Hence it is straightforward the path-buffer path tracing algorithm to do gathers as well.

5 Results

We created a software implementation to evaluate how well the path-buffer would work in practice. All pictures are 256 by 256 pixels and all timings are done on an HP 735 workstation. The timings are included for completeness only as the real performance potential tests will come when the path-buffer is combined with hardware scan conversion.

A simple diffuse box environment cosisting of 36 triangles[3] was used to test the path tracing with the path buffer. Figures 5, 6, and 7 show solutions for the same environment with a maximum path depth of one, two, and three. A depth one solution includes only direct lighting while using greater depths includes better approximations to the indirect lighting as well. Notice how the shading in the shadows and on the front of the yellow box change as more of the indirect light is included. Each of these renderings was computed using 1024 paths/pixel. Each path/pixel required 5 scan conversions to generate the hemi-cube shadowmap plus a number of scan conversions equal to the depth to trace the the paths. Figure 7 took 35 minutes to generate. To demonstrate that the algorithm can handle non-diffuse BRDFs, we changed the yellow box into a yellow brushed metal box in Figure 10. We also increased the paths/pixel to 4096 to adequately sample the non-diffuse BRDF.

We also tested using the path-buffer to do a gather from a radiosity solution. Figure 8 shows a radiosity on the box environment using a coarse mesh. The gather using 1024 paths/pixel is shown in Figure 9. The number scan conversions is the same number of as for a depth two path trace. This figure used 210 triangles and took 32 minutes to generate. We also ran a gather on a more complicated room consisting of 14920 triangles. The radiosity solution for this environment is shown in Figure 11. We used a Monte Carlo radiosity solver which generates the right answer on average, but the result for individual polygons is often incorrect especially for small polygons. A gather using 4096 paths/pixel is shown in Figure 12. This image took 233 minutes to generate. Note the subtle effects like the secondary illumination from the ceiling above the left door and the indirect lighting shadow behind the posts next to the fireplace. To demonstrate just how important the indirect illumination is in this scene, we have included a direct lighting only solution in Figure 13.

[2] Personal communications

[3] Our implementation splits polygons into triangles.

6 Conclusions

We believe that the path-buffer is a valuable extention of the standard z-buffer. By adding z_0 value to each pixel and an additional z compare to each pixel write, scan conversion can be applied to new algorithms which were previously exclusively the domain of ray casting. Moreover this addition is simple enough that it can easily be implemented in hardware.

We have shown two sample applications of the path-buffer: path tracing and gathering from radiosity solutions. To test the practicality of these path-buffer algorithms, we created a software implementation of the path-buffer. The results of this implementation have been very encouraging. However we think that the real value of the real power of the path-buffer will be realized when its combined with hardware support. Thus is our hope that hardware designers will consider adding path-buffer support to their future scan conversion and z-buffer hardware designs.

6.1 Future Work

We are actively seeking to find an available hardware scan conversion design which can support the path-buffer. Nearly all of the additional cost in handling bigger models will come in the scan conversion of the scene. Thus with hardware scan conversion we expect be able to handle models which are orders of magnitude bigger than the ones presented here.

There are also many improvements which can be made in our implementation of path tracing. The number of scan conversions required per path can reduced by making better use of the shadow maps and by reusing scan conversions from the eye over multiple paths.

Better ways need to be found to handle very specular surfaces. This might be done by altering the directional sampling based on the BRDFs which the paths hit. Perhaps by letting each path "vote" on important directions to sample or by special casing specular intersections for separate calculation (similiar to two-pass techniques[18].)

Also for future work is the inclusion of levels-of-detail. Objects could be use different levels of detail depending on the distance from the eye or the current depth of the paths. This could greatly reduce the cost of very complex scenes.

A Appendix: Mathematical Foundation

A.1 Rendering Equation

The rendering equation was introduced by Kajiya[9] to give a mathematical basis for the design and evaluation of realistic rendering algorithms. Kajiya also introduced a method called path tracing which is capable of producing good approximate solutions to the rendering equation using Monte Carlo techniques. In this appendix, we will give a quick summary of the equations which govern path tracing. For more detail see [9] [15]

The rendering equation can be formulated in many different ways. It can be formulated as an integration over directions as follows:

$$\mathrm{L}(\vec{x},\omega) = \mathrm{L_e}(\vec{x},\omega) + \int_\Omega \rho(\vec{x},\omega,\omega')\mathrm{L}(\vec{x}',\omega')\cos\theta\, d\omega' \qquad (1)$$

where θ is the angle between ω' and the surface normal at $\vec{x}$ and $\vec{x}'$ is the point on the first surface visible from $\vec{x}$ in direction ω'. Alternatively it can be formulated as an integral

$\vec{x}$,$\vec{x}'$,...	points on surfaces in scene
ω,ω',...	directions
L	radiance (light power per unit area per steradian)
L_e	emitted radiance (light power per unit area per steradian)
L_r	reflected radiance defined by L - L_e
ρ	bidirectional reflectance distribution function
g	geometry or visibility function. Equals one if two points are mutually visible, zero otherwise.
S	set of surfaces in scene
S_e	set of light emitting surfaces
A_e	area of all light emitting surfaces
Ω	sphere of all possible direction
$\|\vec{x}' - \vec{x}\|$	distance between $\vec{x}'$and $\vec{x}$

Table 1: Symbols and Terms

over all points on all the surfaces in our scene.

$$L(\vec{x},\omega) = L_e(\vec{x},\omega) + \int_S \rho(\vec{x},\omega,\omega')L(\vec{x}',\omega')g(\vec{x},\vec{x}')\frac{\cos\theta\cos\theta' d\vec{x}'}{\|\vec{x}'-\vec{x}\|^2} \quad (2)$$

where ω' is the direction from $\vec{x}$ to $\vec{x}'$, θ' is the angle between ω' and the surface normal at $\vec{x}'$, and $g(\vec{x},\vec{x}')$ is 1 if $\vec{x}'$ is visible from $\vec{x}$ and 0 otherwise.

Exact solutions to these equations are only known for a few very simple cases, so we use an approximation method known as Monte Carlo integration. The integral of a function $f(x)$ over the range X is approximately equal to $f(x_i)/p(x_i)$ where $x_i \in X$ is a randomly chosen value according to the probability density function $p(x)$.

$$I = \int_{x\in X} f(x)d\mu_x \approx \frac{f(x_i)}{p(x_i)} \quad (3)$$

This is a valid estimate in the sense that although any individual value may not be correct, on average we will get the right answer. Thus we can always get a better estimate by averaging many such estimates. How quickly such an average will converge to the correct solution depends on the function, f, and probability density, p, used. In many cases it is worthwhile to carefully design $p(x)$ based on knowledge of $f(x)$, however in this paper we will simply use uniform probability density functions where $p = 1/(\int_X d\mu_x)$. For more information on Monte Carlo integration see [14] [10].

A.2 Naive Path Tracing

Path tracing in its most naive form works by integrating Equation 1 using a uniform probability over the sphere of directions so that $p(\omega') = \frac{1}{4\pi}$. We randomly choose direction ω' to get the estimate:

$$L(\vec{x},\omega) \approx L_e(\vec{x},\omega) + 4\pi\cos\theta\rho(\vec{x},\omega,\omega')L(\vec{x}',\omega') \quad (4)$$

We can then evaluate L($\vec{x}'$,ω') by recursively applying Equation 1 and choosing another random direction ω'' to get:

$$\begin{aligned} L(\vec{x},\omega) \approx\ & L_e(\vec{x},\omega) + 4\pi\cos\theta\rho(\vec{x},\omega,\omega') \\ & \left[L_e(\vec{x}',\omega') + 4\pi\cos\theta'\rho(\vec{x}',\omega',\omega'')L(\vec{x}'',\omega'')\right] \end{aligned}$$

We can repeat this procedure as many times as we like, but we need some way to keep our computation is finite. Typically a maximum depth is set for this recursion and then the remaining L is assumed to be a constant often called the ambient value. This introduces some bias to our solution but the bias decreases with increasing depth of the recursion. The choice of a good ambient will both decrease the bias and increase the accuracy of our estimates. However there is currently no good method for automatically choosing an ambient value other than trial and error or comparison to a known solution. In this paper will use an ambient value of zero unless otherwise noted.

A.3 Path Tracing

We can reduce the number of paths needed by being a little less naive in our sampling. In a typical scene the set of light emitting surfaces, S_e, is only a very small portion of the set of surfaces, S. Only paths which hit a surface in $S_e$will result in a non-zero estimate, but the chance of our randomly hitting a light source may be very small. All these zero estimates mean that our convergence of our approximation is slow and many paths are required. We can improve the convergence by reformulating the equation that we are estimating.

Let us define $L_r = L - L_e$. Then our radiosity equation will be roughly of the form:

$$L = L_e + \int_\Omega \rho L_e\cos\theta\, d\omega' + \int_\Omega \rho L_r\cos\theta\, d\omega'$$

If the light sources constitute a small portion of our scene will be better off by reformulating the second term as an integration over the light surfaces in the style of Equation 2 to get:

$$\begin{aligned} L(\vec{x},\omega) &= L_e(\vec{x},\omega) + L_r(\vec{x},\omega) \\ L_r(\vec{x},\omega) &= \int_{S_e} \rho(\vec{x},\omega,\omega')L_e(\vec{x}',\omega')g(\vec{x},\vec{x}')\frac{\cos\theta\cos\theta' d\vec{x}'}{\|\vec{x}'-\vec{x}\|^2} \\ &\quad + \int_\Omega \rho(\vec{x},\omega,\omega'')L_r(\vec{x}'',\omega'')\cos\theta''\, d\omega'' \end{aligned}$$

We can estimate the surface integral by using the Monte Carlo integration. The uniform probability density for this integral is $p(\vec{x}') = 1/A_e$ where $A_e$is the total area of all light sources. Choosing random values for $\vec{x}''$ and ω'' we get:

$$\begin{aligned} L_r(\vec{x},\omega) &= \rho(\vec{x},\omega,\omega')L_e(\vec{x}',\omega')g(\vec{x},\vec{x}')\frac{\cos\theta\cos\theta' A_e}{\|\vec{x}'-\vec{x}\|^2} \\ &\quad + 4\pi\cos\theta''\rho(\vec{x},\omega,\omega'')L_r(\vec{x}'',\omega'') \end{aligned} \quad (5)$$

As before we can evaluate $L_r(\vec{x}'',\omega'')$ by using recursive applications of these equations up to some maximum depth.

There path tracing shown here is still somewhat naive. There are number of improvement which have been made in path tracing, but are not used in our current implementation. Arvo[1] has shown that the bias due to the depth limit can be avoided by using probabilitic termination of paths. Also much work has been done on finding non-uniform probility density functions which give more accurate estimates. These improve the convergence of the estimate and so reduce the number of paths which need to be traced.

A.4 Radiosity Gather

A gather from a radiosity solution can be done using the same basic equations as for path tracing. For a gather we will use Equation 5, but instead of recursively applying it, we use the pre-computed radiosity solution to evaluate $L_r(\vec{x}'',\omega'')$. This is essentially the same as doing depth 2 path tracing with spatially varying ambient values

Image Buffer

color	Color	12 bytes
total_attn	Color	12 bytes
pos	Vector	12 bytes

Path Buffer

z	Float	4 bytes
z_0	Float	4 bytes
image_pixel	Pointer	4 bytes
light	Color	12 bytes
attn	Color	12 bytes

Table 2: Per Pixel Data Structures

PathBufferTrace (database, image_buf, path_buf)

```
InitializeImagePositions(image_buf,database.eye);
transform = PerspectiveProjection(database.eye);
for depth := 1 to maxdepth
    next_dir = RandomDirection();
    ScanConvertPositions (path_buf,transform,image_buf);
    ScanConvertScene (path_buf,transform,database);
    UpdateImage(image_buf,path_buf,transform);
    transform = OrthographicProjection (next_dir,image_buf);
```

Table 3:

ScanConvertPositions (path_buf,transform,image_buf)

```
foreach entry ∈ path_buf
    entry.image_pixel = NULL;
foreach pixel ∈ image_buf
    if pixel.total_attn == 0 then
        continue;   skip this pixel
    pt = Transform (transform,pixel.pos);
    entry = path_buf[pt.y][pt.x];
    entry.image_pixel = pixel;
    entry.z0 = pt.z;
    entry.z = infinity;
```

Table 4:

WritePixel(x, y, z, attn, light)

```
entry = pathbuf[y][x];
if entry.z0 < z and entry.z > z then
    entry.z = z;
    entry.attn = attn;
    entry.light = light;
```

Table 5:

B Implementation

As proof of concept, we have implemented a software version of the path-buffer. Several versions of path tracing and radiosity gathering have been combined in a single implementation and use slightly different code. Thus just a general overview of the implementation will be given here. The remaining details for a particular algorithm can filled in using the equations in Appendix A.

The main data structures used are the image buffer and the path buffer shown in Table 2. The image buffer holds the current estimate of our image which we are accumulating as well as the states of all the paths currently being traced. The path buffer is used for scan conversions to find the next intersection points for the paths. We use an additional 3 bytes per pixel for shadowmaps for a total of 75 bytes per pixel. Our colors are implemented as three floats for the red, green, and blue color channels.

All the algorithms work by repeated computing sets of paths where each set of paths has one path per pixel. The results for each pixel are averaged over many such paths to produce an accurate image. Pseudocode for tracing a set of paths is shown in Table 3.

The InitializeImagePositions function sets the starting position for each path in the image buffer to the value corresponding to its pixel as determined by the view point. ScanConvertPositions maps each path in the image buffer to the corresponding location on the path buffer as determined by the projection for the current shot. We then use ScanConvertGeometry to scan convert the scene into the path buffer to determine the next intersection point for each path. UpdateImage is used to update the path and image data in the image buffer using the results of the scan conversion. If we have not yet reached the maximum depth then we repeat the process by continuing the paths in a randomly direction.

Pseudocode for ScanConvertPositions is listed in Table 4. One complication not shown is that we must also deal with the fact that more than one image pixel may map onto the same location in the path buffer. One way to handle this is to give the entry to first pixel who tries to grab it. Unfortunately this would add an additional unwanted bias to our solution. We could repeat the shot until all pixels have been handled, but this is expensive. Instead we chose to use Russian Roulette[1] to probabilistically choose between competing pixels and to weight the survivor by the correct amount to avoid introducing additional bias.

The ScanConvertGeometry function uses standard scan conversion techniques except that the WritePixel routine has been modified to add the additional compare with z_0 as shown in Table 5. The two other values we store in the path buffer are: attn $= 4\pi \cos\theta \rho(\vec{x}, \omega, \omega')$ and *light* which is the emitted and/or reflected light at the surface depending on which exact algorithm we are implementing. See Appendix A for more details. Note that in the PathBufferTrace function we pre-picked the direction for the next shot before we doing the current shot. This enables us able to immediately evaluate the bidirectional reflectance distribution function $\rho(\vec{x},\omega,\omega')$. An alternative would be to store enough information per pixel to be able to do defered evaluation.

The function UpdateImage shown in Table 6 serves two functions. It adds the collected light into our estimate for the image and it updates the status of the paths to prepare for the next scan conversion. This consists of mapping the new endpoints for the paths back into world space, storing them in the pos field of the image buffer, and accumulating the total attenuation along the path. The total attenuation gives of the relative contribution of any light we collect on the scan conversion.

References

[1] J. Arvo and D. B. Kirk. Particle transport and image synthesis. In F. Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 63–66, Aug. 1990.

UpdateImage(image_buf,path_buf,transform)

```
for y := 1 to IMAGE_HEIGHT
    for x := 1 to IMAGE_WIDTH
        entry = path_buf[y][x];
        pixel = entry.image_pixel;
        if pixel == NULL then
            continue;   Skip this entry
        pt = Point(x,y,entry.z);
        pixel.pos = InverseTransform(transform,pt);
        pixel.color += pixel.total_attn * entry.light;
        pixel.total_attn *= entry.attn;
```

Table 6:

[2] E. E. Catmull. Computer display of curved surfaces. In *Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structure*, pages 11–17, May 1975.

[3] M. F. Cohen and D. P. Greenberg. The Hemi-Cube: A radiosity solution for complex environments. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 31–40, Aug. 1985.

[4] R. L. Cook, L. Carpenter, and E. Catmull. The Reyes image rendering architecture. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 95–102, July 1987.

[5] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In J. Lane, editor, *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 79–88, July 1989.

[6] P. E. Haeberli and K. Akeley. The accumulation buffer: Hardware support for high-quality rendering. In F. Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 309–318, Aug. 1990.

[7] P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. In T. W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 197–206, July 1991.

[8] C. B. Harrell and F. Fouladi. Graphics rendering architecture for a high performance desktop workstation. In J. T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 93–100, Aug. 1993.

[9] J. T. Kajiya. The rendering equation. In D. C. Evans and R. J. Athay, editors, *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20, pages 143–150, Aug. 1986.

[10] M. H. Kalos and P. A. Whitlock. *Monte Carlo Methods*. John Wiley and Sons, New York, N.Y., 1986.

[11] W. T. Reeves, D. H. Salesin, and R. L. Cook. Rendering antialiased shadows with depth maps. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 283–291, July 1987.

[12] H. E. Rushmeier. *Realistic Image Synthesis for Scenes with Radiatively Participating Media*. PhD thesis, Cornell University, May 1988.

[13] D. Salesin and J. Stolfi. The ZZ-buffer: a simple and efficient rendering algorithm with reliable antialiasing. In *Proceedings of the PIXIM '89*, pages 451–466, 1989.

[14] P. Shirley. Hybrid radiosity/monte carlo methods. *Advanced Topics in Radiosity*, 1994. ACM Siggraph '94 Course 28 Notes.

[15] P. Shirley and C. Wang. Distribution ray tracing: Theory and practice. In *Proceedings of Third Eurographics Workshop on Rendering*, pages 33–43, 1992.

[16] B. Smits, J. Arvo, and D. Greenberg. A clustering algorithm for radiosity in complex environments. *Computer Graphics*, 28(3), July 1994. ACM Siggraph '94 Conference Proceedings.

[17] K. Sung. Area sampling buffer: Tracing rays with z-buffer hardware. In *Proceedings of Eurographics 92*, volume 11, pages 299–310, 1992.

[18] J. R. Wallace, M. F. Cohen, and D. P. Greenberg. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. In M. C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 311–320, July 1987.

[19] G. J. Ward. The radiance lighting simulation and rendering system. *Computer Graphics*, 28(3), July 1994. ACM Siggraph '94 Conference Proceedings.

[20] G. J. Ward, F. M. Rubinstein, and R. D. Clear. A ray tracing solution for diffuse interreflection. In J. Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 85–92, Aug. 1988.

[21] H. Weghorst, G. Hooper, and D. P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics*, 3(1):52–69, Jan. 1984.

Figure 5: *Diffuse scene path-buffer traced with depth 1 (direct illumination only), 1024 paths/pixel.*

Figure 6: *Path-buffer traced with depth 2, 1024 paths/pixel.*

Figure 7: *Path-buffer traced with depth 3, 1024 paths/pixel.*

Figure 8: *Monte Carlo Radiosity solution on the diffuse box environment.*

Figure 9: *Path-buffer gather from Figure 8, 1024 paths/pixel.*

Figure 10: *Scene with brushed yellow metal box. Path-buffer traced with depth 3, 4096 paths/pixel.*

Figure 11: *Room with a Monte Carlo Radiosity solution.*

Figure 12: *Path-buffer gather from Figure 11, 4096 paths/pixel*

Figure 13: *Room with direct lighting only.*

The Path-Buffer

Bruce Walter*
Jed Lengyel*

PCG-95-4 May 1995

With a small modification the z-buffer can be made flexible enough to be used in global illumination algorithms traditionally handled only by ray casting, but still enjoy the benefits of z-buffer algorithms including predictable performance, memory coherence, and acceleration using specialized z-buffer hardware.

The fundamental difference between the path-buffer and a standard z-buffer is that the global near clipping plane is replaced with a per pixel near clipping value z_0. The success of each pixel write depends on compares with both the current z and z_0 values. We demonstrate the power of this additional test by showing how two global illumination algorithms can be adapted to use the path-buffer instead of ray casting.

First we adapt Kajiya-style path tracing for finding view-dependent global illumination solutions. Since the scene database is streamed through the path-buffer, and no complicated meshing or per-element storage is needed, very large and complex models may be illuminated. Both diffuse and non-diffuse reflectance functions can be handled as well any complex geometric primitives which can be rasterized (e.g. displacement-mapped spline surfaces). Second we show how the path-buffer can be used to do an image gather from a coarse radiosity solution in two-pass radiosity methods.

*Program of Computer Graphics, Cornell University, Ithaca, N.Y. 14853