Inferred Lighting:
Fast dynamic lighting and shadows for opaque and translucent objects

Scott Kircher*
Volition, Inc.

Alan Lawrance†
Volition, Inc.

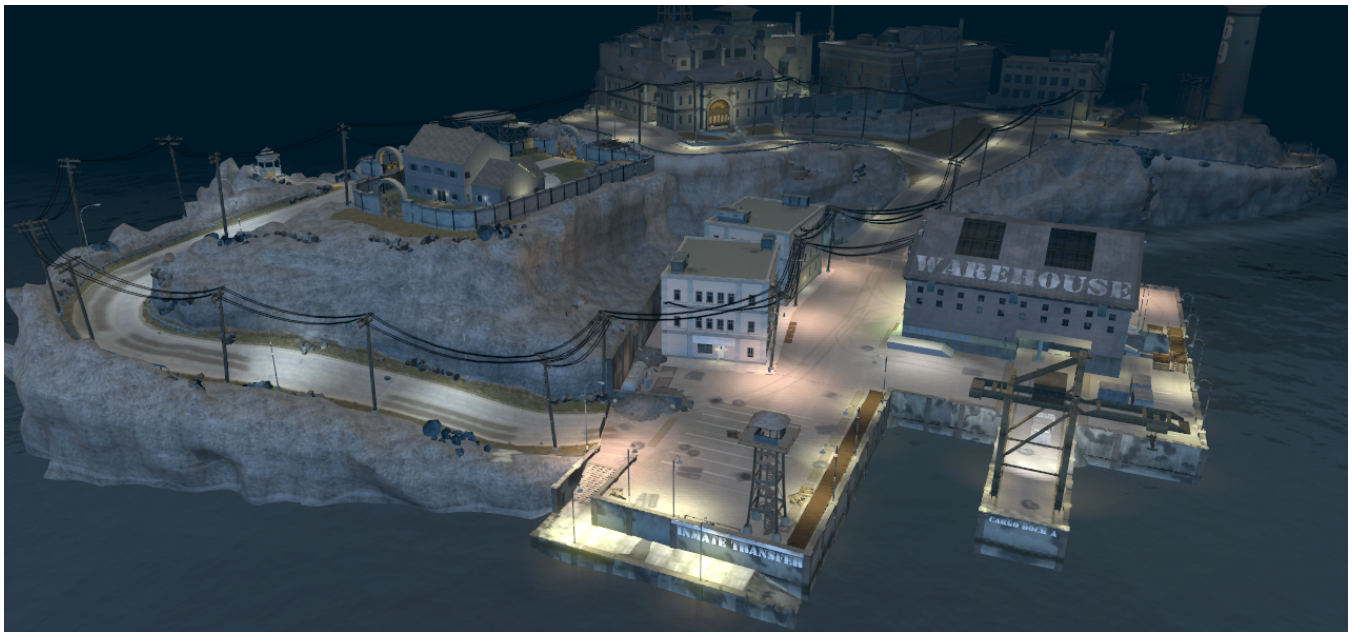

Figure 1: *Inferred lighting allows this complex scene with 213 active lights to be rendered at 1280x720 resolution, with 8x MSAA, at 23fps (43.5ms) on a GeForce 8800GTX.*

Abstract

This paper presents a three phase pipeline for real-time rendering that provides fast dynamic light calculations while enabling greater material flexibility than deferred shading. This method, called *inferred lighting*, allows lighting calculations to occur at a significantly lower resolution than the final output and is compatible with hardware multisample antialiasing (MSAA). In addition, inferred lighting introduces a novel method of computing lighting and shadows for translucent objects (alpha polygons) that unifies the pipeline for processing lit alpha polygons with that of opaque polygons. The key to our method is a discontinuity sensitive filtering algorithm that enables material shaders to "infer" their lighting values from a light buffer sampled at a different resolution. This paper also discusses specific implementation issues of inferred lighting on DirectX 10, Xbox 360, and PlayStation 3 hardware.

*e-mail: scott.kircher@volition-inc.com
†e-mail: alan.lawrance@volition-inc.com

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: real-time rendering, fast dynamic lighting, alpha-polygon shadows, deferred shading

1 Introduction

Dynamic lighting is an important graphical aspect of many entertainment software titles. In particular, the lighting situation in so-called *open world* games can change drastically between day and night scenes. During night scenes, it is often desirable to have a very large number of dynamic lights for cars, neon signs, and so on. Even seemingly static lights, such as street lights, must be able to light dynamic objects, and so their lighting cannot be solely precomputed. The method described in this paper is not restricted to open world games, however. Any game featuring a very large number of dynamic lights could potentially benefit.

Traditionally, games have relied on a technique that is commonly called *forward rendering* for handling multiple dynamic lights. In forward rendering, the interactions between each light and each object in the scene must be explicitly accounted for. For example, the rendering engine typically must query the scene to determine which objects a particular light affects. In many cases, a pass over at least a portion of the scene geometry is required for each light. These queries and rendering passes can be quite expensive, and must be performed regardless of whether or not the light casts shadows.

An increasingly popular technique called *deferred shading* allows

the scene geometry to be rendered only once, and then non-shadow casting lights are applied as screen-space operations [Saito and Takahashi 1990; Shishkovtsov 2005; Filion and McNaughton 2008]. This solves the light-object interaction complexity problem, but makes it difficult to implement advanced material shaders, as effectively one material shader must be able to handle all desired shading effects. In addition deferred shading is incompatible with lighting alpha polygons. Deferred shading renderers must revert to forward rendering to light translucent objects [Shishkovtsov 2005].

This paper presents a dynamic lighting method, called *inferred lighting* that combines the strengths of forward rendering with those of deferred shading. Our method enables a very large number of non-shadow casting dynamic lights, while allowing material shaders to be almost as flexible as in forward rendering. In addition, inferred lighting introduces a new method for lighting alpha polygons, allowing the same lighting techniques to be used on translucent and opaque objects. This method even makes translucent objects compatible with stencil shadow volumes [Heidmann 1991], which has traditionally not been possible.

1.1 Related Work

The rendering technique now called deferred shading is sometimes credited to Deering *et al.* [1988], but it was more directly introduced by Saito and Takahashi [1990] for non-photo-realistic rendering. Saito and Takahashi rendered objects into a *geometry buffer* (G-buffer) storing normals, depth, and other geometric properties, and then performed 2D image space techniques to extract contours, silhouettes, and other information. Deferred shading appeared again in the PixelFlow hardware architecture proposed by Molnar *et al.* [1992].

The advent of multiple-render-target (MRT) technology in 2002 made deferred shading an effective real-time rendering technique [Thibieroz 2004; Hargreaves and Harris 2004]. Since then it has appeared in several game engines including S.T.A.L.K.E.R. [Shishkovtsov 2005], Killzone 2 [Valient 2007], and StarCraft 2 [Filion and McNaughton 2008].

Standard deferred shading turns lighting into a simple screen-space operation and thereby makes non-shadow casting lights very cheap. However, because all data needed for shading must be stored in the G-buffers, increasing material complexity means more storage space required for G-buffer data, and more memory bandwidth consumed reading and writing that data. This places severe limits on the kinds of materials that can be rendered with deferred shading on current game console hardware. In addition, deferred shading is fundamentally incompatible with lit alpha polygons. Short of the K-buffer hardware modifications proposed by Bavoil *et al.* [2007], the G-buffer cannot store more than one set of geometry data per pixel. Thus, multiple layers of lighting are not possible. Translucent geometry must be lit using a separate forward rendering pass.

So-called *light pre-pass rendering*, as proposed by Engel [2008] mitigates the material complexity problem of deferred shading. In his approach, the G-buffer contains very little information (*i.e.*, only normals and depth). Lighting calculations are then partially performed as screen-space operations into another *light buffer*, and then a second scene geometry pass is rendered using the lighting buffer as input for a full material shader. Light pre-pass rendering is also being used by Insomniac Games [Lee 2008], and is closely related to our proposed method. However, light pre-pass rendering still requires alpha polygons to be lit in a separate forward rendering pass, and requires that the light buffer be full resolution to avoid significant lighting artifacts. Inferred lighting and light pre-pass rendering were developed concurrently and independently.

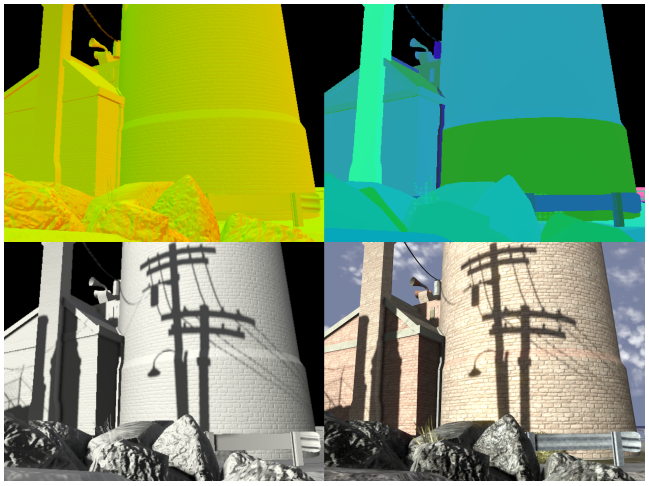

Figure 2: *(Top Left:) The 800x450 normals G-buffer. (Top Right:) The 800x450 DSF data G-buffer. (Bottom Left:) The 800x450 L-buffer after rendering lighting. (Bottom Right:) The final 1280x720 8xMSAA output framebuffer.*

2 The Inferred Lighting Pipeline

The pipeline for inferred lighting consists of three distinct stages, a *geometry pass*, *light pass* and *material pass*. There are no additional passes necessary for translucent geometry.

2.1 Geometry Pass

The first pass stores geometric properties to screen-space buffers called G-buffers. The information stored in these buffers will depend on the needs of the light pass, but at a minimum it will consist of normals, depth, and discontinuity sensitive filter (DSF) data. Visualization of the view-space normals and DSF G-buffers are shown at the top left and right of Figure 2, respectively.

The normals use 16 bits per component, but only the X and Y components are stored since the Z component can be reconstructed in the shader. A mapping from full sphere to half sphere is used to avoid storing a Z sign bit. For a right-handed coordinate system, the mapping for a view-space normal n is

$$n' = \frac{n + (0,0,1)}{||n + (0,0,1)||} \tag{1}$$

where only the X and Y components of n' are stored in the normal buffer. The reverse mapping, used when the normal buffer is read, is

$$z = \sqrt{1 - ||(n'_x, n'_y)||^2}, \tag{2}$$

$$n = (2zn'_x, 2zn'_y, 2z^2 - 1). \tag{3}$$

The mapping has a singularity at $n = (0,0,-1)$, but this would be a normal facing directly away from the viewer, and should never need to be encoded.

DSF data is stored as two 16 bit values, linear depth of the pixel and an ID value. Details on how the DSF data is used in the material pass is explained in §3.

Other data can be stored in the G-buffer, such as specular power, motion vectors (for per-object motion blur), or a material ID to allow for BRDF lighting. Details on using additional G-buffer data

is beyond the scope of this paper, but it is expected that users of inferred lighting will tailor G-buffer usage to suit their specific needs. Creating additional G-buffers should be done with care, as they are memory intensive and incur a performance cost.

2.2 Light Pass

The second pass calculates the contribution of the ambient and dynamic lights that affect the scene. Lighting calculations are performed in screen-space using the G-buffer as input, and the results are written to the light buffer (L-buffer) for use in the material pass. Since lighting is done entirely in screen-space, there are no geometry passes necessary for non-shadow casting lights. As a consequence of the fact we use a lower resolution G-buffer and L-buffer than our final output, the viewpoint will have to get closer to a surface before normal map detail will become apparent in the lighting. Implementations can trade off between speed and quality by adjusting the G-buffer/L-buffer resolution.

The L-buffer consists of four 16 bit channels. The diffuse lighting in stored in the RGB channels, and specular lighting in the alpha channel. The specular lighting value is encoded as the accumulated intensity of the specular highlight. The specular color is reconstructed in the material pass by scaling the stored intensity by the color of the diffuse lighting.

For each light processed in the light pass, a full screen quadrilateral polygon is rendered that executes the lighting shader for every screen pixel. Since a given light may only affect a portion of the screen, there are two optimizations available to skip pixels not affected by light. The first is to apply a scissor rectangle that bounds the screen-space influence of the light. The second is to utilize the stencil buffer to determine where the light volume intersects visible scene geometry. We use the depth-fail stencil volume technique, as described in [Kwoon 2004]. These optimizations should be used together, but it can be faster to skip the stencil optimization for lights that take up a very small amount of screen space.

2.3 Material Pass

The final pass renders the scene using full material shaders but samples lighting values from the L-buffer rather than doing its own light calculations. Alpha objects must be sorted and rendered after opaque objects, but are lit in the same manner.

The material pass is rendered at the framebuffer resolution, which is higher than the resolution used for the L-buffer. This requires the material pass to up-sample from the L-Buffer and perform special filtering, which is described in §3.

3 Discontinuity Sensitive Filtering

The inferred lighting pipeline as described in §2 allows the geometry pass and light pass to occur at a lower resolution than the material pass. This saves both memory and pixel shading costs. However, the lower lighting resolution will be visible along the edges of objects in the scene, resulting in unacceptable aliasing. This is especially noticeable when the foreground and background objects receive drastically different lighting values as in Figure 3.

Our solution to this problem is to perform *discontinuity sensitive filtering* (DSF) of the L-buffer as it is read during the material pass. During the geometry pass, one 16 bit channel of the DSF buffer is filled with the linear depth of the pixel, the other 16 bit channel is filled with an ID value that semi-uniquely identifies continuous regions. This ID is described in more detail in §3.1.

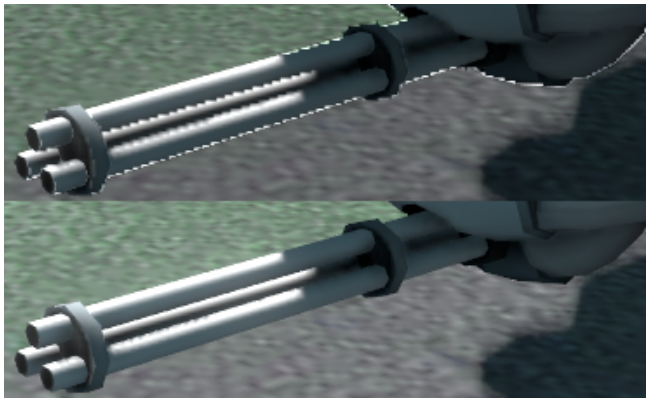

Figure 3: *(Top:) Using an 800x450 single-sampled L-buffer with a 1280x720 8xMSAA framebuffer, with no DSF, results in badly aliased lighting. (Bottom:) DSF solves this problem.*

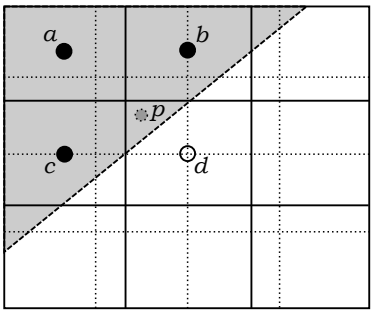

Figure 4: *DSF sampling uses only L-buffer samples that match the surface being rendered in the material pass. In this example, the dotted grid represents the high-resolution framebuffer. The solid grid represents the low resolution L-buffer. When material shading pixel p, which is part of the gray surface, sample d is thrown out, and only samples a, b, and c are used to light p.*

During the material pass, the pixel shader computes the locations of the four L-buffer texels that would normally be accessed if regular bilinear filtering were used. These four locations are point sampled from the DSF buffer. The depth and ID values retrieved from the DSF buffer are compared against the depth and ID of the object being rendered. The results of this comparison are used to bias the usual bilinear filtering weights so as to discard samples that do not belong to the surface currently rendering (see Figure 4). These biased weights are then used in custom bilinear filtering of the L-buffer. Since the filter only uses the L-buffer samples that belong to the object being rendered, the resulting lighting gives the illusion of being at full resolution. This same method works even when the framebuffer is multisampled (hardware MSAA), however sub-pixel artifacts can occur, due to the pixel shader only being run once per pixel, rather than once per sample. Such sub-pixel artifacts are typically not noticeable.

In the rare case where all four L-buffer samples are unusable the pixel shader falls back on regular bilinear filtering, possibly resulting in a small lighting artifact.

Our method of computing lighting at a lower resolution and then up-sampling with the aide of a discontinuity sensitive filter is similar in spirit to the interleaved sampling and discontinuity filtering methods proposed by Segovia, *et al.* [2006]. However, we feel our

DSF method is simpler, more efficient (does not require walking the buffer to find discontinuities), and more suited to game rendering.

3.1 Object and Normal-Group IDs

The ID value stored in the second channel of the DSF buffer consists of two parts. The upper 8 bits are an object ID, assigned per object (renderable instance) in the scene. Since 8 bits only allows 256 unique object IDs, scenes with more than this number of objects will have some objects sharing the same ID. In theory this can lead to low probability DSF artifacts. In practice, we have never observed such artifacts.

The lower 8 bits of the channel contain a normal-group ID. This ID is pre-computed and assigned to each face of the mesh. Anywhere the mesh has continuous normals, the ID is also continuous. In our implementation, a normal is continuous across an edge if and only if the two triangles that abut the edge share the same normals at both vertices of the edge.

By comparing normal-group IDs the discontinuity sensitive filter can detect normal discontinuities without actually having to reconstruct and compare normals. Both the object ID and normal-group ID must exactly match the material pass polygon being rendered before the L-buffer sample can be used (depth must also match within an adjustable threshold).

We pre-compute normal-group IDs by first constructing the dual graph of the mesh in question. We then remove any edges in the dual graph across which the mesh has discontinuous normals, and randomly assign normal-group IDs to each connected component of the resulting graph. These IDs are assigned to the corresponding faces of the original mesh. In our implementation, we pack the normal-group IDs into the fourth component of the mesh tangent vectors.

4 Lighting Alpha Polygons

In addition to allowing the L-buffer resolution to be less than the output framebuffer resolution, DSF also enables a novel method of lighting alpha polygons. The main idea is that alpha polygons are rendered during the geometry pass using a stipple pattern, so that their G-buffer samples are interleaved with opaque polygon samples. The light pass will automatically light those stippled pixels. No special case processing during the light pass is necessary, as long as the lighting operations are one to one (*i.e.*, do not involve blurring of the L-buffer). In the material pass the DSF for opaque polygons will automatically reject stippled alpha pixels, and alpha polygons are handled by finding the four closest L-buffer samples in the same stipple pattern, again using DSF to make sure the samples were not overwritten by some other geometry.

Figure 5 shows (left) an example of how the stipple pattern for a translucent object (the cockpit window glass) interleaves with the opaque object samples in the L-buffer. It also shows (right) the final result after the material pass. Since the stipple pattern is a 2×2 regular pattern, the effect is that the alpha polygon gets lit at half the resolution of opaque objects. Opaque objects covered by one layer of alpha have only slightly reduced lighting resolution (one out of every four samples cannot be used).

For alpha polygons it is possible to simply use a discontinuity sensitive bilinear filter (as in §3), with the sample locations taking into account the lower resolution sampling pattern. However, we use a small radius cone filter on the nearest four stipple samples, as we found it gave slightly higher quality results. DSF data should be used to throw out samples that don't belong to the surface in question, regardless of filtering method.

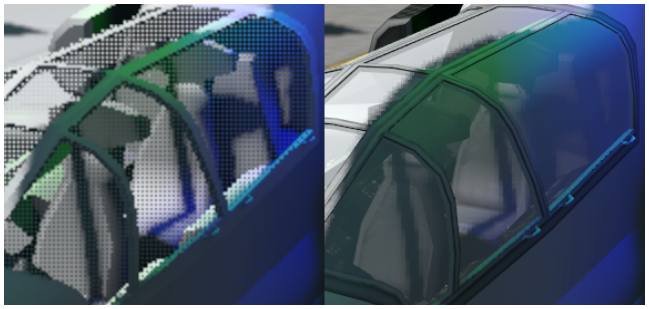

Figure 5: *(Left) In the L-buffer, alpha polygon samples are interleaved with opaque polygon samples. (Right) During the material pass, DSF and knowledge of the stipple pattern resolves appropriate lighting values.*

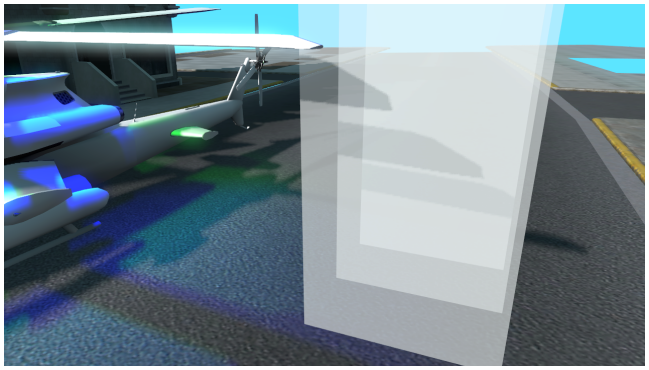

Figure 6: *A helicopter blade casting a shadow on 3 overlapping layers of alpha polygons. Inferred lighting allows alpha polygons to be lit in essentially the same way as opaque polygons.*

Multiple layers of lit alpha objects can be achieved by assigning a different stipple pattern to each layer. We use a 2×2 regular pattern, which means there are four available patterns. Thus, up to four layers of overlapped lighting are possible (including the opaque background). Typically, by this point the background is so obscured by layers of alpha that loss of lighting information is not catastrophic. Different sampling patterns can be used. For example, a 4×4 pattern would allow up to 16 layers, but at greatly reduced resolution. Figure 6 shows an example of multiple layers of alpha objects being lit by inferred lighting.

When using multiple stipple patterns, it must be decided what pattern to use for each layer. Currently we either statically assign the patterns with user input, or assign them dynamically based on the depth of the objects. Neither method is optimal, but both suffice for scenes without too many dynamic, overlapping alpha objects.

Our method of lighting alpha polygons can accommodate any lighting/shadow algorithm that is one to one in terms of L-buffer pixels. Thus, it can even be used with stencil shadow volumes, which are otherwise incompatible with translucent geometry.

5 Platform Specific Implementation Issues

We have implemented inferred lighting under DirectX 10 as well as two current generation game consoles, the Xbox 360 and the PlayStation 3. The technique works equally well on all three platforms, but there are platform specific implementation issues, especially on the game consoles. The only issue on DirectX 10 is that the depth surface must be resolved to an R24X8 texture before it

can be read. All other surfaces can be read from directly.

5.1 Xbox 360

The Xbox 360 has 10MB of high performance video memory known as EDRAM. Special care must be taken when setting up the G-buffers and L-buffer in video memory, as none of the G-buffers can overlap. The L-buffer can share EDRAM with G-buffers, but ensure it doesn't overlap with the depth G-buffer as this can be used in the light pass if using the stencil optimization. It is not possible to read directly from EDRAM, so screen buffers must be resolved to textures.

The format of the DSF buffer (two 16 bit channels) in EDRAM is fixed point with a range of -32 to 32. The corresponding texture format is fixed point with a range of 0 to 1. This requires the shader to scale the shader output of 0 to 1 to -32 to 32. To maintain 16 bit precision, the texture used for the resolve needs to be created with a custom format that has a range of -1 to 1. When sampling from this texture in a shader, the results must be scaled to a 0 to 1 range.

The Xbox 360 uses the same pixel center convention as DirectX 1 though 9, which requires special handling in shaders. This can be avoided by setting the render state D3DRS_HALFPIXELOFFSET to true. This will ensure the Xbox 360 uses the same pixel center convention as DirectX 10 and the PlayStation 3.

5.2 PlayStation 3

The PlayStation 3 does not support a surface format with two 16 bit channels. It is possible to work around this limitation by using an ARGB8 surface format and packing two 16 bit values into four 8 bit values. When binding the surface to a sampler, a texture format remap must be used to ensure the texture is read correctly as CELL_GCM_TEXTURE_Y16_X16.

Although this solves the problem of storing the normals and DSF data as 16 bits per component, it does cause an issue with blending normals. An example is alpha blending a normal map decal onto a surface. Since all four channels of the normals G-Buffer are used to contain the packed 16 bit values, there is no channel available to write the alpha value.

One solution to this problem is to treat the decal object as an alpha object. This will ensure lighting is computed for both the decal and the underlying surface, and the two will be blended together in the material pass. The downsides are that we consume an alpha layer, and the lighting on the decal will be at a lower resolution.

Multiple render target support on the PlayStation 3 requires that all color targets are the same number of bits per pixel. This isn't a problem for the standard G-buffer set-up of 32 bits for normals and DSF data, but it's a restriction to keep in mind when designing G-buffers for cross-platform compatibility.

Finally, reading from the depth buffer on the PS3 requires a texture format remap of BARG and use of the texture sampling function texDepth2D_precise().

6 Additional Results

We have demonstrated several inferred lighting results throughout the above sections. Additional results and timing information follow.

Figure 7 shows a scene with 287 lights and 287 objects running with inferred lighting (but without MSAA) on a PlayStation 3. The GPU time is 5.03ms (199fps). Such tightly packed scenes are nearly

Figure 7: *A simple scene with 287 lights and 287 objects running at 199fps (5.03ms) on a PlayStation 3.*

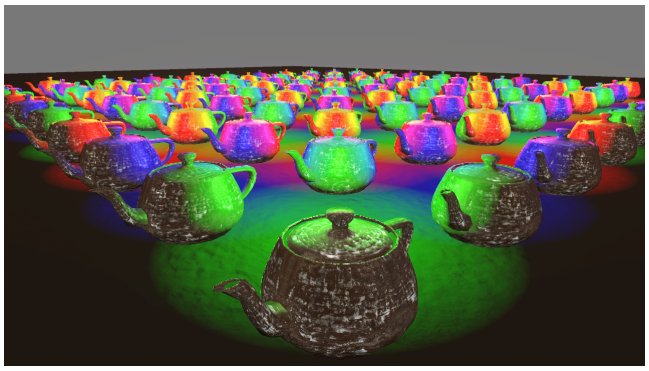

Figure 8: *A scene with 101 objects and 81 omni lights running at 34fps (29.4ms) with inferred lighting on the Xbox 360. This same scene runs at 6fps (167ms) with forward rendering.*

the worst case for forward rendering. For such scenes, forward rendering complexity would be $O(nl)$, where n is the number of objects and l is the number of lights. This is significantly worse than inferred lighting's complexity of $O(n + l)$ (which is the same as the complexity of deferred shading and light-prepass rendering).

Figure 8 shows a scene with 101 objects and 81 omni lights running at 34fps (29.4ms) with inferred lighting on the Xbox 360. The same test scene using an optimized forward renderer, which has been in development for several years, runs at 6fps (167ms). Although this test is a nearly worst case example for forward rendering, it clearly shows how well inferred lighting handles a large number of non-shadow casting lights.

Figure 9 shows the usefulness of using a lower-resolution L-buffer with DSF in the material pass. There are 122 non-shadowing lights in the scene. Both images use a 1280x720 framebuffer with 8x MSAA. The top picture uses a low resolution (800x450) L-buffer with DSF, and renders at 40 frames per second (25ms) on a GeForce 8800GTX. The bottom picture uses a full resolution (1280x720) L-buffer without DSF, and renders at 31 frames per second (32ms). In addition to the lower frame rate, notice that the bottom picture exhibits more aliasing on, for example, the handrails. This is due to the fact that the G-buffer was not multisampled. Having a multisampled G-buffer is not even possible on most platforms (since the buffer must be resolved before it is read), and even if it were, it would further reduce performance. Also, since the bottom picture does not use DSF, many full-alpha objects had to be replaced with alpha-tested objects, resulting in yet more aliasing.

As just alluded to, our alpha lighting technique can be applied to full-alpha chain link fences and other traditionally alpha-tested objects like barbed wire and foliage (see Figure 10). This is useful because alpha-testing is currently incompatible with hardware

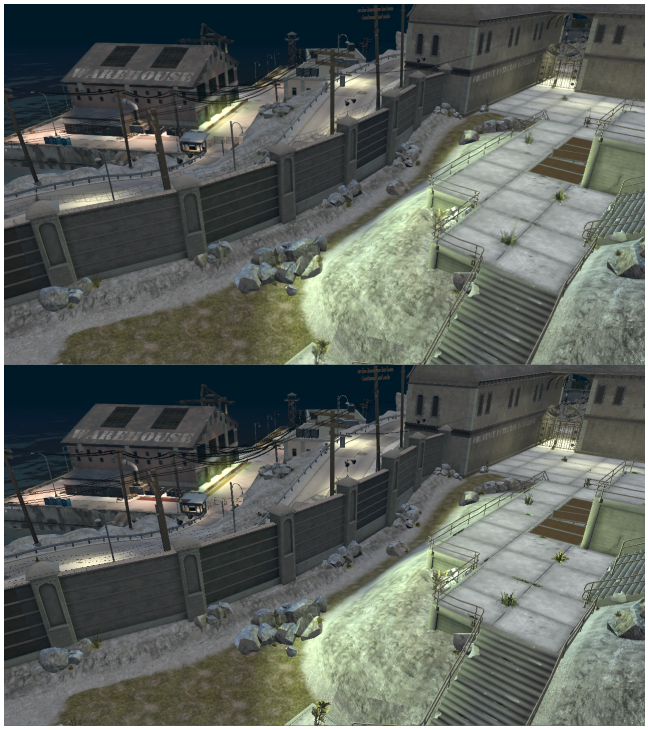

Figure 9: *A scene with 122 dynamic lights. Using an 800x450 L-buffer with a 1280x720 8x MSAA framebuffer and DSF (Top) results in both better quality anti-aliasing and faster framerate (40fps vs 31fps) than using a 1280x720 L-buffer with no DSF (Bottom).*

MSAA. To achieve the best results, we recommend doing an alpha-test in the geometry pass, in addition to the stipple pattern, to ensure that no stippled lighting samples are written in regions of complete transparency.

7 Conclusion

Inferred lighting allows for a very large number of non-shadow casting dynamic lights while supporting complex material shaders, a unified pipeline for lighting translucent and opaque objects, and hardware MSAA compatibility.

The distinguishing feature of inferred lighting is the DSF algorithm which allows lighting to occur at a lower resolution than the final output and provides a novel approach to lighting alpha polygons without additional scene processing.

7.1 Limitations and Future Work

While inferred lighting works very well in the case where there is a large number of lights or other operations occurring at the L-buffer resolution, it does incur a higher base cost (when there are no, or very few, lights) than deferred shading or light-prepass rendering. Further optimizations to the DSF algorithm will help ameliorate this. For example, we have plans to reduce the number of samples taken from the L-buffer, by leveraging the bilinear filter available in hardware.

Also, while the alpha lighting method solves the traditional problem of how to apply all lights and shadows in a scene to translucent objects, it is limited to a small number of lighting layers. Moreover, assigning stipple patterns in an optimal way (either dynamically or statically) is an open problem worthy of further research.

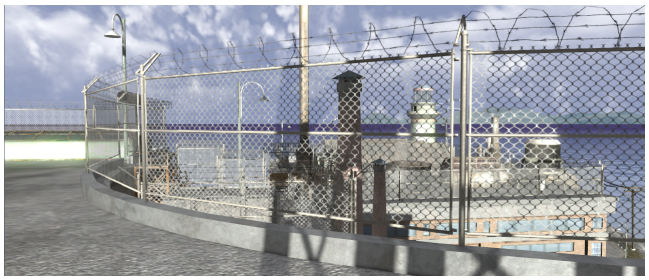

Figure 10: *Inferred alpha polygon lighting can be used to light full-alpha chain link fences and other objects usually handled by alpha-testing.*

Acknowledgements Special thanks to Tomas Arce who was instrumental in the initial design of inferred lighting, and to Jason Lowe who implemented the first PS3 version of the algorithm. Thanks also to Mike Flavin, Bart Wyatt, Andy Cunningham, and John Buckley for their feedback about the method and paper. Thanks also to Adam Pletcher, Jason Childress, and Chris Claflin for putting together many of the test scenes that appear in this paper.

References

BAVOIL, L., CALLAHAN, S. P., LEFOHN, A., COMBA, J. A. L. D., AND SILVA, C. T. 2007. Multi-fragment effects on the gpu using the k-buffer. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 97–104.

DEERING, M., WINNER, S., SCHEDIWY, B., DUFFY, C., AND HUNT, N. 1988. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *SIGGRAPH 1988*, ACM, New York, NY, USA, 21–30.

ENGEL, W., 2008. Diary of a graphics programmer: Light pre-pass renderer. Online, accessed Jan. 27th, 2009. `http://diaryofagraphicsprogrammer.blogspot.com/2008/03/light-pre-pass-renderer.html`.

FILION, D., AND MCNAUGHTON, R. 2008. Effects & techniques. In *ACM SIGGRAPH 2008 classes*, ACM, New York, NY, USA, 133–164.

HARGREAVES, S., AND HARRIS, M., 2004. Deferred shading. Online, accessed Jan. 26th, 2009. `http://developer.nvidia.com/object/6800_leagues_deferred_shading.html`.

HEIDMANN, T. 1991. Real shadows, real time. *Iris Universe 18*, 23–31.

KWOON, H. Y. 2004. The theory of stencil shadow volumes. In *ShaderX2: Introductions and Tutorials with DirectX 9*. Wordware Publishing, Inc., 197–278.

LEE, M., 2008. Prelighting. Online, accessed Feb. 10th, 2009. `http://www.insomniacgames.com/tech/articles/0209/files/prelighting.pdf`.

MOLNAR, S., EYLES, J., AND POULTON, J. 1992. Pixelflow: high-speed rendering using image composition. In *SIGGRAPH 1992*, ACM, New York, NY, USA, 231–240.

SAITO, T., AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph. 24*, 4, 197–206.

SEGOVIA, B., IEHL, J. C., MITANCHEY, R., AND PÉROCHE, B. 2006. Non-interleaved deferred shading of interleaved sample patterns. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, ACM, New York, NY, USA, 53–60.

SHISHKOVTSOV, O. 2005. Deferred shading in s.t.a.l.k.e.r. In *GPU Gems 2*. Addison-Wesley, ch. 9, 143–166.

THIBIEROZ, N. 2004. Deferred shading with multiple render targets. In *ShaderX2: Shader programming Tips and Tricks with DirectX 9*. Wordware Publishing, Inc., 251–269.

VALIENT, M., 2007. Deferred rendering in killzone 2. Online, accessed Jan. 26th, 2009. Develop Conference `http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf`.