2.9 Light-Indexed Deferred Rendering

DAMIAN TREBILCO

INTRODUCTION

Current rasterization-based renderers utilize one of two main techniques for lighting: forward rendering and deferred rendering [Calver07] [Shish05] [Koonce07]. However, both of these techniques have disadvantages.

Forward rendering does not scale well with complex lighting scenes, and standard deferred rendering suffers from high memory usage. Standard deferred rendering also has trouble with transparency, multi-sample anti-aliasing (MSAA), and per-material lighting schemes.

This article presents a middle ground technique that keeps the key advantages of both forward and deferred rendering: minimal memory usage while being able to handle complex lighting scenes on a per-material basis.

RENDERING CONCEPT

Typical deferred rendering stores the material properties at each fragment and renders lights by accessing the fragment's data. This implementation aims to do the reverse: Store the light properties at each fragment and access these properties when rendering the main scene using forward-pass rendering.

The most direct way of achieving this is to store light direction, color, and attenuation at each fragment. However, this approach limits the number of lights that can influence each fragment, as storing these properties consumes a large amount of buffer storage space.

Some recent work by Wolfgang Engel with "Light Pre-Pass Rendering" [Engel08] provides one solution to this problem. This is achieved by storing surface normals in a pre-pass, calculating diffuse and specular terms for each light, and adding the results to a light buffer. These lighting terms are then used in a forward material rendering pass to do the lighting. However, this technique requires that all materials be lit with the same lighting scheme and requires a normal output pre-pass.

LIGHT-INDEXED DEFERRED RENDERING

This new approach simply assigns each light a unique index and then stores this index at each fragment the light hits, rather than storing all the light or material properties per fragment. These indices can then be used in a fragment shader to look up into a lighting properties table for data to light the fragment.

This technique can be broken down into three basic render passes.

1. Render depth only pre-pass.
2. Disable depth writes (keep depth testing only) and render light volumes into a light index texture. Standard deferred lighting and shadow volume techniques can be used to find what fragments are hit by each light volume.
3. Render geometry using standard forward rendering. Lighting is done using the light index texture to access lighting properties in each shader.

The problem with steps 2 and 3 occurs when lights overlap. If light volumes could not overlap, then step 2 could simply write the light index to the texture, which can be directly accessed in step 3.

In order to support multiple light indices per fragment, it would be ideal to store the first light index in the texture's red channel, the second light index in the blue index, and so on. To do this, a light index packing scheme will be needed.

Presented here are three light index packing schemes that use differing amounts of CPU and GPU processing time. The *CPU sorting* method is purely CPU based, while the *bit shifting* method is a purely GPU-based technique. The *multi-pass max blend equation* method uses a moderate amount of GPU processing time with the option to use the CPU to offload some work.

LIGHT INDEX PACKING: CPU SORTING

An easy CPU-based solution for light index packing involves sorting the scene lights based on light volume overlap. Assuming 8-bit light indices and an RGBA8 light index texture, four overlapping light indices can be rendered using the following steps:

- On the CPU, create four arrays to hold light volume data. Then for each scene light, find the light data array it can be added to without intersecting any of the existing lights in the array (e.g., attempt to add to array one, then attempt to add to array two, etc.). If a light cannot be added, it will have to be discarded or stored to be processed in a second pass.
- Clear the light index color buffer to zero.
- Enable writing to the *red* channel only and render light volumes from light data array 1.
- Enable writing to the *green* channel only and render light volumes from light data array 2.
- Enable writing to the *blue* channel only and render light volumes from light data array 3.
- Enable writing to the *alpha* channel only and render light volumes from light data array 4.

The advantage of this method is:

- No unpacking in the fragment shader is needed.

The disadvantage to packing light indices this way is:

- Requires sorting the scene lights on the CPU.

Using this packing method, lights can be prioritized by sorting the important lights first. The light overlap count and number of total scene lights can be varied by changing the number of render targets and the bit-depth of the render targets.

If a scene is mostly made up of static lights, these lights can be presorted into light volume arrays or have an intersection tree generated for fast runtime access.

Light Index Packing: Multi-Pass Max Blend Equation

A GPU-based solution to storing multiple indices based on the fragment pass was suggested by Timothy Farrar [Farrar07], and has been modified to better support light-indexed rendering.

Assuming 8-bit light indices and an RGBA8 light index texture, four overlapping light indices can be rendered using the following steps:

- Clear color and stencil buffers to zero.
- Set blend equation mode to MAX.
- Mask out writes to blue and alpha channels.
- Set the stencil to increment on stencil pass and set the stencil compare value to only pass on values 2 (only allow a maximum of two writes per fragment).
- Render the light volumes and output (index, 1.0–index) in the red and green channels.
- Mask out writes to *red* and *green* channels and enable *blue* and *alpha* channels.
- Set the stencil to decrement on stencil failure and set the stencil compare value to only pass on values equal to 0.
- Render the light volumes and output (index, 1.0–index) in the blue and alpha channels.

Unpacking for each light index is done as follows (a zero index is assumed to be no light):

- Index1 = red channel
- Index2 = 1.0–blue channel (ignore if equal to red channel)
- Index3 = green
- Index4 = 1.0–alpha channel (ignore if equal to green channel)

The advantage of this method is:

- Simple unpack

The disadvantages to packing light indices this way are:

- Only supports a maximum of four overlapping lights.
- Requires two passes of the light volumes for four light indices.

- Uses the stencil buffer, which may be needed by shadow volumes or the light volume passes. If only two light indices are needed, then the stencil buffer does not need to be used.

- If only one or three lights hit a fragment, light indices 1, 2 or 3, 4 will be the same index.

Using this packing method, lights can be prioritized by using the high and low indices for primary important lights and the mid-range indices for secondary lights.

This method can also be combined with the CPU sorting technique above to sort the scene lights into two light data arrays. Each array is allowed intersecting lights as long as no more than two lights share the same intersecting space. Each array can then be rendered (red-green pass and then blue-alpha pass) without the need of the stencil buffer.

Light Index Packing: Bit Shifting

Another GPU-based solution to light index packing involves bit shifting and packing.

Again, assuming 8-bit light indices and an RGBA8 light index texture, four overlapping light indices can be rendered using the following steps:

- Clear the color buffer to zero.

- Set the blend mode to `ONE, CONSTANT_COLOR` where the constant color is set to 0.25. This shifts existing color bits down two places (`>> 2 = * 0.25`) and adds the two new bits to the top of the number.

- Render the light volumes and break the 8-bit index value into four 2-bit values and output each 2-bit value into RGBA channels as high bits, for example, red channel = `(index & 0x3) 6`. This index splitting can be done offline and simply supplied as an output color to the light volume pass.

Unpacking for each light index requires a video card that can do bit logic in shaders (shader model 4) or with some floating-point emulation. The following GLSL code uses floating-point math to unpack each light index into a 0…1 range—suitable for looking up into a light index texture of 256 values:

```
#define NUM_LIGHTS 256.0

// Look up the bit planes texture
vec4 packedLight = texture2DProj(BitPlaneTexture, projectSpace);

// Unpack each lighting channel
vec4 unpackConst = vec4(4.0, 16.0, 64.0 , 256.0) / NUM_LIGHTS;

// Expand the packed light values to the 0.. 255 range
vec4 floorValues = ceil(packedLight * 254.5);

float lightIndex[4];
for(int i=0; i< 4; i++)
{
  packedLight = floorValues * 0.25; // Shift two bits down
  floorValues = floor(packedLight); // Remove shifted bits
  lightIndex[i] = dot((packedLight - floorValues), unpackConst);
}
```

Bit packing also allows a lot of different light overlap counts and scene light count combinations. Some of the possible combinations are listed in Table 2.9.1.

TABLE 2.9.1 Bit-packing combinations

Lights Per Fragment	Scene Light Count	Details
2	15	1×8-bit channel Light index written directly out (4 bits)
4	15	2×8-bit channels Light index split into 2×2-bit values
8	15	1×RGBA8 surface Light index split into 4×1-bit values
16	15	2×RGBA8 surfaces (2 render targets) Light index split into 4×1-bit values
		Output to one render target at a time and use the stencil buffer like in multi-pass max blend equation when eight overlaps have been reached (requires two passes of light volume geometry)
1	255	1×8-bit channel Light index written directly out (8 bits)
2	255	2×8-bit channels Light index split into 2×4-bit values
4	255	1×RGBA8 surface Light index split into 4×2-bit values
8	255	2×RGBA8 surfaces (two render targets) Light index split into 8×1-bit values
16	255	4×RGBA8 surfaces (four render targets) Light index split into 8×1-bit values
		Output to two render targets at a time and use the stencil buffer like in multi-pass max blend equation when eight overlaps have been reached (requires two passes of light volume geometry)
1	65,535*	2×8-bit channels Light index split into 2×8-bit values
2	65,535	1×RGBA8 surface Light index split into 4×4-bit values
4	65,535	2×RGBA8 surfaces (two render targets) Light index split into 8×2-bit values
8	65,535	4×RGBA8 surfaces (four render targets) Light index split into 16×1-bit values

** Note that when 65,535 lights are used, it is advisable to split the light index into two indexes and look up into a 256´256 light data table.

The advantages of this bit packing method are:

- It scales based on the number of scene lights and the overlap requirements.
- Typically, it renders in a single pass without using additional buffers (e.g., stencil).

The disadvantages of packing light indices this way are:

- Complex unpacking.
- Requires hardware to be bit-precise in blending and floating-point math.

Using this bit packing method, lights can be prioritized by rendering the lowest-priority lights first. If more lights than the pack limit are reached, older lights will be discarded when blending.

Light Index Geometry Lighting

Once a light index packing technique is chosen, the next step is to update each standard forward-rendered shader to use the light indices.

If using multiple light indices, it is recommended that all lighting calculations be done in world or view space. This is because lighting data can be supplied in world or view space, and surface data is typically in tangent or model space. It is typically more efficient to transform the surface data once into the lighting space than to transform each light's data into the surface's space. However, using world space for lighting can cause precision issues for large worlds.

The next step is to decide how to supply a light data lookup table to the fragment shader and what data should be contained within it.

The obvious choice for supplying light data is one or more point-sampled textures. Textures have the advantage of being widely supported, but care has to be used to ensure that when updating dynamic data in the textures, the GPU pipeline is not stalled. Multiple textures can be used in alternate frames to help ensure that an update is not attempted on a texture currently being used by the GPU. If a scene is made up of static lights in world space, no per-frame updates are required, and single textures can be used. Another choice to supply lighting data is to use constant buffers (as available in Direct3D 10), but this article will focus on the texture approach.

The type of data needed depends on the lighting types you want to support. We will focus on point lights, as you can emulate a parallel light with a distant point light and spotlights can be partially emulated by rendering a cone volume. Point lights require position, attenuation, and light color data. All this data could be supplied in one 2D texture—using the *x* axis as the light

index and the *y* axis as the light property. However, it is more practical to split the light properties into different textures based on the update frequency and format requirements. Experiment with different formats and splits to determine what is fastest for the target hardware.

For the test application provided on the accompanying DVD-ROM (see Figure 2.9.1), we decided that colors would be updated infrequently and only needed low precision. Therefore, colors are supplied in a 1D RGBA8 texture. Position and attenuation lighting data need more precision and are supplied in a 1D 32-bit per component RGBA floating-point texture. The RGB values represent the light position in view space, with the alpha component containing (1/light radius) for attenuation. Using 8-bit light indices (255 lights), the total size of these light data textures is a tiny 5 KB.

Note that light index zero represents the "none" or "NULL" light, and the light buffers for this index should be filled with values that cannot affect the final rendering (e.g., black light color).

Once each light property has been looked up using a light index, standard lighting equations can be used. On modern hardware it may be beneficial to "early out" of processing when a light index of zero is reached.

Combining with Other Rendering Techniques

Many lighting techniques work well in isolation, but fail when combined with other techniques used in an application. This section will discuss ways of combining light-indexed deferred rendering (LIDR) with other common rendering methods.

Multi-Sample Anti-Aliasing

Typically, the biggest disadvantage with deferred rendering has been supporting Multi-sample anti-aliasing (MSAA). Fortunately, there are several solutions for LIDR.

FIGURE 2.9.1 Screenshot from the provided test application on the DVD-ROM. The scene consists of ~40,000 triangles with 255 lights. Running on an NVIDIA Geforce 6800 GT at 1024×768 resolution, the application compares light-indexed deferred rendering (LIDR) with multi-pass forward rendering. Frame rates measured: 82 frames per second (FPS) with 1 overlap LIDR, 62 FPS with 2 overlap LIDR, 37 FPS with 4 overlap LIDR, and 12 FPS with multi-pass forward rendering.

MSAA Technique 1: MSAA Texture Surface

Render all targets (depth or stencil, light index texture, main scene) to MSAA surfaces. Then, when doing the forward render pass, sample the light index texture using the screen *X,Y* and current sample index. Sampling an MSAA texture is possible with Direct3D 10 and current console hardware.

While traditional deferred rendering can also use this technique for MSAA, it suffers from two major disadvantages:

- Already large "fat buffers" are made 2/4/8 times the size.
- Lighting calculations have to be done for all samples (2/4/8 times the fragment work), or samples have to be interpolated that only give approximate lighting. LIDR only needs to perform lighting on fragments actually generated on edges.

MSAA Technique 2: Light Volume Front Faces

Typically, when rendering light volumes in deferred rendering, only surfaces that intersect the light volume are marked and lit. This is generally accomplished by a “shadow–volume-like” technique of rendering back faces—incrementing stencil where depth is greater than zero—and then rendering front faces and only accepting when depth is less than zero and stencil is not zero. By only rendering front faces where depth is less than, all future lookups by fragments in the forward rendering pass will get all possible lights that could hit the fragment.

This front face method has the advantage of using a standard non-MSAA texture, but has a major problem with light intersections. Previously, only lights that hit a surface were counted in the packing of light indices. By only rendering front faces, all light volumes that intersect the ray from the eye to the surface will be counted in the packing count. This wastes fragment shader cycles in processing lights that may not hit the surface and can easily saturate the number of lights a surface supports.

Using this technique, it is also possible to leave out the depth pre-pass (in very vertex-limited scenes) at the cost of saturating the surface light index count faster.

Transparency

Transparency has also been a major problem with deferred rendering. However, by using the light volume front faces technique as discussed in the multi-sample anti-aliasing section, semi-transparent objects can be rendered after opaque objects using the same light index texture. This is because all light volumes that intersect the ray from the eye to an opaque surface are included in the light index texture. Therefore, in the rendering of semi-transparent objects, the forward render pass will have access to all lights that could possibly hit each fragment.

Shadows

There are several ways of combining LIDR with shadows, depending on what shadowing technique is used.

No Combined Shadows

The easiest way to integrate LIDR into an application with shadows is to only use it with non-shadowing lights. This is possible, as LIDR uses the standard forward rendering when applying lights.

For example, an application may render shadows only from one primary directional light and have lots of non-shadowing point or PFX lights. Each shader that handles the directional light shadowing simply needs to be updated to access the LIDR lights. Another option is to do the LIDR lights as a separate pass after the shadow pass.

Shadow Volumes

Shadow volumes work easily with LIDR lights. Once the shadow volume has marked the stencil buffer to indicate where the shadowed areas are, the light volumes can be rendered to ignore these areas. However, this will not work if you are using the light volume front faces technique for MSAA and transparency support.

Shadow Maps

Shadow maps can be supported in a few ways depending on the requirements.

- **Pre-pass:** By accessing the buffer depth value when rendering the light volume pass, a shadow map texture can be compared to determine if a fragment is in shadow. If hard shadows are acceptable, the shader can simply call “discard” when a fragment is in shadow. If soft shadows are desired, a “shadow intensity” value can be output in a separate render target. These shadow intensity values need to be bit-packed and unpacked in a way similar to the light index values and accessed in the forward pass to attenuate the lighting.

- **Final pass:** By packing the shadow maps into a 3D texture or a 2D texture array (Direct3D 10), the light index can be used to access a shadow map and do the shadow calculation in the forward rendering pass. This requires additional light table data for the shadow map matrices and may be difficult to support for shadowed point lights

(possibly use six consecutive shadow map indices). If using this technique, it is recommended that only a limited light index range has shadows, for example, light indices 0 to 3 have shadows.

Constraining Lights to Surfaces

One of deferred rendering's greatest strengths is that all surfaces are lit equally. Unfortunately, this can also be a problem. Artists sometimes like to light scenes with lights that only affect some surfaces. This can be solved with traditional deferred rendering by rendering out another index to indicate light surface interaction, but this requires yet more buffer space and more logic in the scene lighting passes.

With LIDR, however, all that is needed is to supply different light lookup tables for the forward render pass. Lights that should not hit a surface can be nulled out by using a black color or adjusting the attenuation.

If no LIDR lighting is to hit a surface, the surface shader can simply have the LIDR calculations taken out. A common case of this might be a scene with static lights baked into a light map for static geometry. These static lights can then be rendered at runtime with LIDR for access by dynamic surfaces. Static geometry surfaces can ignore all LIDR lights and simply use the light map for lighting.

Multi-Light Type Support

Most scenes are made up of a combination of different light types, from simple directional lights to spot lights with projected textures. Unfortunately, multiple light types per scene are not easily handled with LIDR. Some possible solutions are:

- Only use LIDR for a single light type. For example, if a scene is made up of one directional light and multiple point lights, the point lights can use LIDR while the directional light is rendered using standard forward rendering. The directional light could also be "faked" as a distant point light.
- Store a flag with the light data indicating the light type. This can involve complicated shader logic and may not be practical for many different lighting types.
- Use different light index buffers for each light type. This may waste index buffer space if light types are not evenly distributed. This technique also involves complicated shader logic or multiple passes of the scene geometry for each light type.

Lighting Technique Comparison

As with all rendering techniques, LIDR has both advantages and disadvantages. To highlight these, a comparison of LIDR with other common rendering techniques is given.

When comparing LIDR with standard deferred and light-pre-pass rendering, LIDR uses smaller buffer sizes, lighting techniques can vary per material, and transparency can be supported. MSAA is supported by all techniques but uses less memory with LIDR. In addition, LIDR can be layered on an existing forward renderer (assuming a depth pre-pass) and only needs lighting reflection vectors to be calculated once.

In comparing LIDR with multi-pass or multi-light forward rendering, LIDR scales well based on the number of lights, geometry does not have to broken up for individual lighting, and object-light interactions do not have to be calculated on the CPU. Depending on the scene, significant overdraw can also be saved when using LIDR (e.g., small lights on a terrain). LIDR can also support lights that are "mesh" shaped.

However, there are significant disadvantages to using LIDR. The main one is the limit of how many lights can hit a fragment at once (current implementation has a maximum of 16). Scenes with few objects and lights would be faster with other techniques, as each material shader has a fixed lighting overhead. Shadows and "exotic" light types (e.g., projected textures) are also more difficult to support with LIDR.

Future Work

Fake Radiosity

Using LIDR it may be possible to “fake” one-bounce radiosity. This may be accomplished by:

- Rendering the scene from light projection and storing depth and color values.
- Rendering hundreds of small point light spheres into the scene using LIDR. Inside the vertex shader, look up the depth texture from the previous light projection pass to position each light in the scene. Look up the color buffer from the same light projection pass to get the light color.

Using this technique it may be possible to emulate the first bounce in a radiosity lighting pass or implement splatting indirect illumination [Dach06].

Buffer Sharing

Most implementations of LIDR need a separate buffer to store the light indices. However, on some hardware it may be possible to reuse the final color buffer as the light index texture. This is because accesses to the index texture are only ever for the current fragment position.

Conclusion

The technique presented in this article, light-indexed deferred rendering (LIDR), has been shown to be an efficient middle ground between standard forward rendering and deferred rendering techniques. LIDR handles many lights with linear scaling and with only one or two passes of the scene vertex data. LIDR also has only minimal memory requirements with one screen-sized texture for index data and small lookup tables for lighting data. With some minor changes, LIDR can also be made to handle the difficult deferred rendering cases of MSAA and transparency.

Existing applications that use forward rendering can also easily be modified to layer LIDR on top of existing lighting solutions.

Future developments on this technique will be made available on the projects Web site at http://code.google.com/p/lightindexed-deferredrender/.

References

[Calver07] Calver, Dean, “Photo-realistic Deferred Lighting,” available online at http://www.beyond3d.com/content/articles/19/1, March 11, 2007.

[Shish05] Shishkovtsov, Oles, “Deferred Shading in STALKER,” *GPU Gems 2*, Addison-Wesley Professional, 2005, pg. 143.

[Koonce07] Koonce, Rusty, “Deferred Shading in Tabula Rasa,” *GPU Gems 3*, Addison-Wesley Professional, 2007, pg. 429.

[Engel08] Engel, Wolfgang, “Designing a Renderer for Multiple Lights - The Light Pre-Pass Renderer” in this volume.

[Farrar07] Farrar, Timothy, “Output data packing,” OpenGL forum post available online at http://www.opengl.org/discussion_boards/ubbthreads.php?ubb=showflat&Number=230242#Post230242, October 31, 2007.

[Dach06] Dachsbacher, Carsten and Stamminger, Marc, “Splatting indirect illumination,” *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, March 14–17, 2006, Redwood City, California.