Light Indexed Deferred Lighting

By Damian Trebilco
dtrebilco@gmail.com

December 2007
(Revised January 2008)

Table of Contents

Abstract....................3
Introduction....................3
Rendering Concept....................3
Light Indexed Deferred Rendering....................4
    Light Index Packing – CPU Sorting....................4
    Light Index Packing – Multi-pass Max Blend Equation....................5
    Light Index Packing – Bit Shifting....................6
    Light Index Geometry Lighting....................8
Combining with other Rendering Techniques....................9
    Multi-sample Anti Aliasing....................9
        MSAA Technique 1 - MSAA Texture Surface....................9
        MSAA Technique 2 - Light Volume Front Faces....................10
        MSAA Technique 3 - Nearest Fragment Edge....................10
    Transparency....................10
    Shadows....................11
        No Combined Shadows....................11
        Shadow volumes....................11
        Shadow Maps....................11
    Constraining Lights to Surfaces....................11
    Multi-Light Type Support....................12
Lighting Technique Comparison....................12
    Light Indexed Deferred Rendering vs. Standard Deferred Rendering....................12
    Light Indexed Deferred Rendering vs. Multi-pass Forward Rendering....................12
    Light Indexed Deferred Rendering vs. Multi-light Forward Rendering....................13
Future Work....................13
    Fake Radiosity....................13
    Buffer Sharing....................14
Conclusion....................14
References....................15

Abstract

Current rasterization based renderers utilize one of two main techniques for lighting, forward rendering and deferred rendering. However, both of these techniques have disadvantages. Forward rendering does not scale well with complex lighting scenes and standard deferred rendering has high memory usage and trouble with transparency and MSAA.
This paper aims to explore a middle ground between these two lighting techniques with the aim of keeping the key advantages of both. This is achieved with deferring lighting by storing a light index value where light volumes intersect the scene.

Introduction

Current rasterization based renderers utilize one of two main techniques for lighting, forward rendering and deferred rendering[1] [2] [3].

The most basic lighting technique is forward rendering which calculates and renders the final lit surface color during rasterization of vertex data. Forward rendering is simple, memory efficient and fast on simple scenes. However, when multiple lights are involved, a renderer must either render X number of lights per pass (and code the shader for each light combination) or X passes per object (with lighting for each type) which has a high vertex processing cost. A renderer using this technique also has to perform calculations to determine object and light intersections – which may be a significant amount of time for complex scenes.

A more complex lighting technique is deferred rendering in which attributes for each fragment are saved out during rasterization of vertex data. These attributes typically include fragment position, normal vector and material properties. Light volume geometry is then rendered into the scene which access these surface attributes and light the scene. This technique handles many lights with linear scaling and lights all surfaces equally with only one pass of the scene vertex data. However this technique suffers from high memory usage as it must store surface attributes and is difficult to combine with techniques like MSAA and transparency. Another problem with this technique is that all surfaces are lit equally which makes a custom lighting scheme on a per-surface basis difficult.

This paper aims to explore a middle ground between these two lighting techniques with the aim of keeping the key advantages of both.

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 initial implementation stored the lighting contribution at each fragment by calculating an attenuated light vector, for each light, at the fragment position. These vectors were then summed together by adding them to six axis vectors in view space (+x, -x, +y, -y, +z, -z). The theory being:

$$(\vec{L1}\cdot\vec{N})+(\vec{L2}\cdot\vec{N})+(\vec{L3}\cdot\vec{N})=(\vec{L1}+\vec{L2}+\vec{L3})\cdot\vec{N}$$

Where L1-L3 are lighting vectors and N is the normal of the surface.

However, it soon became apparent that while the above vector math is correct, negative lighting values need to be clamped to zero which the above equation does not do. Adding in this clamping results in an equation that does not equal:

$$max(\vec{L1} \cdot \vec{N}, 0) + max(\vec{L2} \cdot \vec{N}, 0) + max(\vec{L3} \cdot \vec{N}, 0) \neq max((\vec{L1} + \vec{L2} + \vec{L3}) \cdot \vec{N}, 0)$$

In order to properly clamp the light vectors, access to the fragment's normal is needed. This is essentially standard deferred rendering. This approach also did not account for specular and other lighting techniques that require a light direction. (Only good for isotropic materials)

The next approach involved storing light direction, color and attenuation at each fragment. However this approach had a very limited number of lights that could influence each fragment. This limitation was due to render target limits and buffer storage space.

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 indexes can then be used in a fragment shader to lookup 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 (depth testing only) and render light volumes into a light index texture. Standard deferred lighting / 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 step two and three occurs when lights overlap. If no light volumes overlapped, step two could simply write the light index to the texture and be directly accessed in step three.

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

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 indexes and a RGBA8 light index texture, four overlapping light indexes 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. (eg. 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 light index color buffer to zero.
- Enable writing to the Red channel only and render light volumes from light data array one.
- Enable writing to the Green channel only and render light volumes from light data array two.
- Enable writing to the Blue channel only and render light volumes from light data array three.
- Enable writing to the Alpha channel only and render light volumes from light data array four.

The advantages of this method are:

- No unpacking in the fragment shader is needed.

The disadvantages to packing light indexes this way are:

- 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 pre-sorted 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 indexes based on the fragment pass was suggested by Timothy Farrar[4], and has been modified to better support light indexed rendering.

Assuming 8 bit light indexes and a RGBA8 light index texture, four overlapping light indexes 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 stencil to increment on stencil pass and set the stencil compare value to only pass on values < 2. (only allow a max 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 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 advantages of this method are:

- Simple unpack

The disadvantages to packing light indexes this way are:

- Only supports a max of 4 overlapping lights.
- Requires two passes of the light volumes for 4 light indexes.
- Uses the stencil buffer – may be needed by shadow volumes or the light volume passes. If only two light indexes are needed, the stencil does not need to be used.
- If only 1 or 3 lights hit a fragment, light index 1, 2 or 3, 4 will be the same index.

Using this packing method, lights can be prioritized by using the high and low indexes for primary important lights and the mid range indexes 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 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 indexes and a RGBA8 light index texture, four overlapping light indexes 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 8bit index value into four 2bit values and output each 2 bit value into RGBA channels as high bits. eg. 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 will use 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 below:

Lights per-fragment	***Scene Light count***	***Details***
2	15	1 x 8bit channel Light index written directly out (4 bits)
4	15	2 x 8bit channels Light index split into 2x2bit values
8	15	1 x RGBA8 surface Light index split into 4x1bit values
16	15	2 x RGBA8 surfaces (2 render targets) Light index split into 4x1bit values Output to one render target at a time and use the stencil buffer like in Multi-pass max blend equation when 8 overlaps has been reached. (requires two passes of light volume geometry)
1	255	1 x 8bit channel Light index written directly out.(8 bits)
2	255	2 x 8bit channels Light index split into 2x4bit values
4	255	1 x RGBA8 surface Light index split into 4x2bit values
8	255	2 x RGBA8 surfaces (2 render targets) Light index split into 8x1bit values
16	255	4 x RGBA8 surfaces (4 render targets) Light index split into 8x1bit values Output to two render targets at a time and use the stencil buffer like in Multi-pass max blend equation when 8 overlaps has been reached. (requires two passes of light volume geometry)
1	65535*	2 x 8bit channels Light index split into 2 x 8bit values
2	65535	1 x RGBA8 surface Light index split into 4 x 4bit values
4	65535	2 x RGBA8 surfaces (2 render targets) Light index split into 8x2bit values
8	65535	4 x RGBA8 surfaces (4 render targets) Light index split into 16x1bit values

The advantages of this bit packing method are:

- Scales based on how many scene lights and the overlap requirements.
- Typically renders in a single pass without using additional buffers (eg stencil).

The disadvantages to packing light indexes this way are:

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

* Note that when 65535 lights are used, it is advised to split the light index into two indexes and lookup into a 256x256 light data table.

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 indexes.

If using multiple light indexes, 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.

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 paper 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 light index and the y axis as 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 it was 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 32bit 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 8bit light indexes (255 lights) the total size of these light data textures is a tiny 5KB.

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. (eg 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.

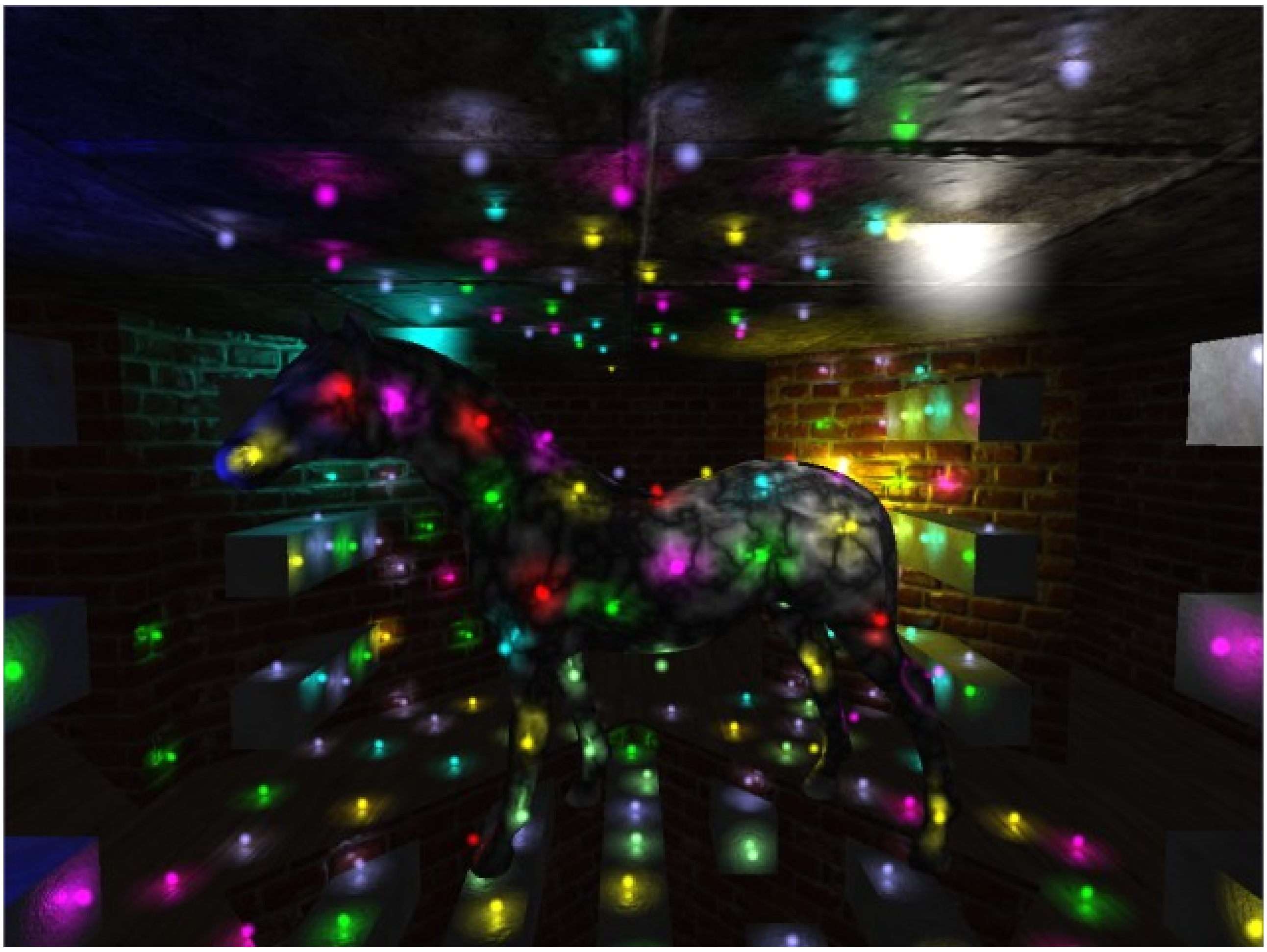

Illustration 1: Screen shot from test application. Scene consists of ~40 000 triangles with 255 lights. Running on a Nvidia Geforce 6800 GT at 1024x768 resolution the application compares Light Indexed Deferred Rendering(LIDR) with multi-pass forward rendering. Frame rates measured: 82 FPS with 1 overlap LIDR, 62 FPS with 2 overlap LIDR, 37FPS with 4 overlap LIDR and 12 FPS with multi-pass forward rendering.

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 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 Light Indexed Deferred rendering.

MSAA Technique 1 - MSAA Texture Surface

Render all targets (depth/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 a 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.

1) Already large "fat buffers" are made 2/4/8x the size.
2) Lighting calculations have to be done for all samples (2/4/8x the fragment work) or samples have to be interpolated which only give approximate lighting. Light Indexed Deferred rendering 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 – then rendering front faces and only accepting when depth is less than 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 bit packing of light indexes. By only rendering front faces all light volumes that intersect the ray from the eye to the surface will be counted in the bit 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.

MSAA Technique 3 - Nearest Fragment Edge

When a fragment looks up into a non-MSAA full-screen texture, it typically uses texture coordinates that sample the center of the texel/pixel it is in. If this center location is not covered by the rendered polygon – the light indexes retrieved may be incorrect.

If you can adjust the texture lookup coordinates to sample the nearest texel/pixel on the polygon when this occurs – the light indexes retrieved should be within one pixel of accuracy. This accuracy may be good enough for foreground objects, but may break down for distant objects.

This adjustment could be accomplished by searching surrounding fragments when this case occurs. This searching could possibly be done by calculating the surface gradient of the polygon. However, if a polygon is made up of mostly sub-pixel fragments, (long thin polygon case) searching for a nearest fragment edge may fail. Due to the complexity and possible errors with this technique, this technique may not be practical[5].

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 will be 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 Light Indexed Deferred Rendering with shadows depending on what shadowing technique is used.

No Combined Shadows

The easiest way to integrate Light Indexed Deferred Rendering (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 using the Light Volume Front Faces technique for MSAA / Transparency support.

Shadow Maps

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

1. 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 will need to be bit-packed/unpacked in a way similar to the light index values, and accessed in the forward pass to attenuate the lighting.
2. 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 6 consecutive shadow map indexes) If using this technique, it is recommended that only a limited light index range has shadows. eg. Light indexes 0...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

Light Indexed Deferred Rendering vs. Standard Deferred Rendering

Advantages of LIDR:

- Uses forward rendering so no need for "fat buffers" to store normal/position type data.
- Can layer on existing light schemes.
- Small buffers size (varies depending on how many lights per fragment are supported)
- Light calculations like the reflection vector only needs to be calculated once.
- MSAA can be supported with fewer resources.
- Transparency can be supported.

Disadvantages of LIDR:

- Exotic lighting types are harder to support (eg. projected texture light)
- Need to set a limit on how many lights can hit each fragment. (current implementation has a max of 16)
- Need to pass the vertex geometry twice – once fore depth pre-pass and once for the forward pass. Note that the depth pre-pass is not vital for LIDR but it allows a lot of optimization.
- Shadows are harder to support.

Light Indexed Deferred Rendering vs. Multi-pass Forward Rendering

Advantages of LIDR:

- Can render lots of lights with only a fragment size cost per light.
- Only two passes of the scene geometry – depth only pass then a forward render color pass.

- Do not have to break geometry up into pieces for individual lighting – can render huge vertex buffers.
- No Object->Light interactions need to be calculated on the CPU (for non-shadowing lights)
- Light calculations like reflection vectors only needs to be calculated once and texture lookups and filtering only need to be done once.
- Can render "mesh-shaped" lights - not limited to sphere/cone shaped lights.

Disadvantages of LIDR:

- Exotic lighting types are harder to support (eg. projected texture light)
- Need to set a limit on how many lights can hit each fragment. (current implementation has a max of 16)
- Requires a full screen buffer to store light index data
- All scene shaders need to be updated to support LIDR
- Slower on scenes that have few objects and lights.
- Shadows are harder to support.

Light Indexed Deferred Rendering vs. Multi-light Forward Rendering

Advantages of LIDR:

- Can render lots of lights with only a fragment size cost per light.
- Do not have to break geometry up into pieces for individual lighting – can render huge vertex buffers.
- No Object->Light interactions need to be calculated on the CPU (for non-shadowing lights)
- Can render "mesh-shaped" lights - not limited to sphere/cone shaped lights.

Disadvantages of LIDR:

- Exotic lighting types are harder to support (eg. projected texture light)
- Need to set a limit on how many lights can hit each fragment. (current implementation has a max of 16)
- Requires a full screen buffer to store light index data
- Can require two passes of scene geometry – depth only pass then a forward render color pass.
- Slower on scenes that have few objects and lights.
- Shadows are harder to support.

Future Work

Fake Radiosity

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

- Render the scene from light projection and store depth and color values.
- Render hundreds of small point light spheres into the scene using LIDR. Inside the vertex

shader, lookup the depth texture from the previous light projection pass to position each light in the scene. Lookup 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[6].

Buffer Sharing

Most implementations of LIDR will need a separate buffer to store the light indexes. However, on some hardware it may be possible to re-use the final color buffer as the light index texture. This is due to the fact that accesses to the index texture are only ever for the current fragment position.

Conclusion

The technique presented in this paper "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 deferred rendering difficult 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.

References

[1] **Photo-realistic Deferred Lighting**
http://www.beyond3d.com/content/articles/19/1

[2] **Deferred Shading in STALKER**
Oles Shishkovtsov (GSC Game World) GPU Gems 2 pg 143

[3] **Deferred Shading in Tabula Rasa**
Rusty Koonce (NCsoft Corporation) GPU Gems 3 pg 429

[4] **Output data packing** – OpenGL forum post
http://www.opengl.org/discussion_boards/ubbthreads.php?ubb=showflat&Number=230242#Post230242

[5] **MSAA - Detecting edge fragments** – OpenGL forum post
http://www.opengl.org/discussion_boards/ubbthreads.php?ubb=showflat&Number=230471#Post230471

[6] **Splatting indirect illumination**
Carsten Dachsbacher , Marc Stamminger, Splatting indirect illumination, Proceedings of the 2006 symposium on Interactive 3D graphics and games, March 14-17, 2006, Redwood City, California