1

III

Real-Time Lighting via Light Linked List

Abdul Bezrati

1.1 Introduction

Deferred lighting has been a popular technique to handle dynamic lighting in video games, but due to the fact that it relies on the depth buffer, it doesn't work well with translucent geometry and particle effects, which typically don't write depth values. This can be seen in Figure 1.1, where the center smoke effect and the translucent water bottles are not affected by the colorful lights in the scene.

Common approaches in deferred engines have been to either leave translucent objects unlit or apply a forward lighting pass specifically for those elements. The forward lighting pass adds complexity and an extra maintenance burden to the engine.

At Insomniac Games, we devised a unified solution that makes it possible to light both opaque and translucent scene elements (Figure 1.2) using a single path. We have named our solution Light Linked List (LLL), and it requires unordered access views and atomic shader functions introduced with DirectX 10.1–level hardware.

The Light Linked List algorithm shares the performance benefits of deferred engines in that lighting is calculated only on the pixels affected by each light source. Furthermore, any object not encoded in the depth buffer has full access to the lights that will affect it. The Light Linked List generation and access is fully GPU accelerated and requires minimal CPU handholding.

1.2 Algorithm

The Light Linked List algorithm relies on a GPU-accessible list of light affecting each pixel on screen. A GPU Linked List has been used in the past to implement

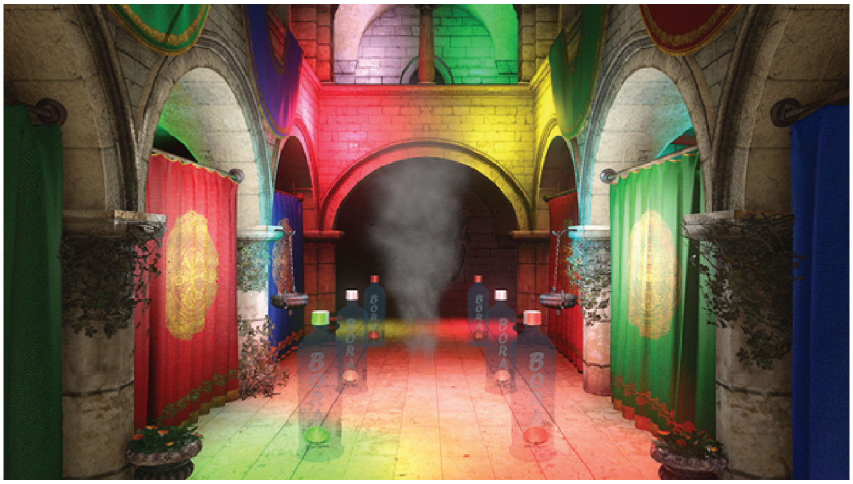

Figure 1.1. Smoke effect and translucent water bottles don't receive any scene lighting in a traditional deferred lighting engine.

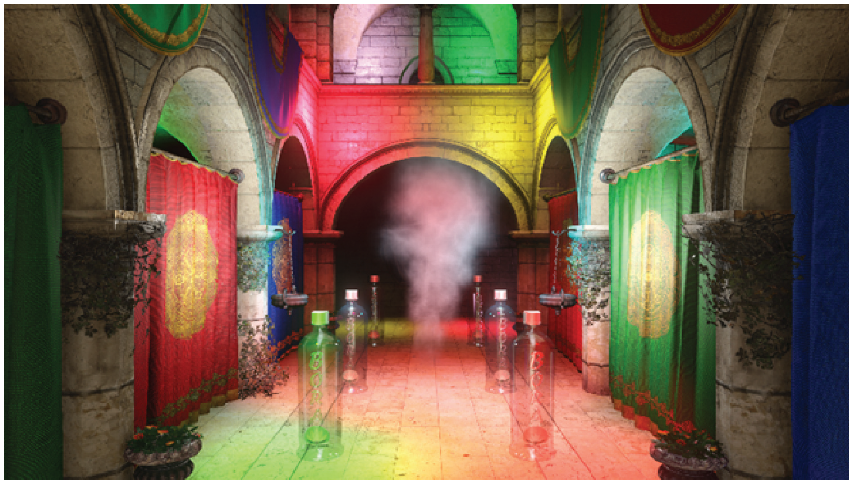

Figure 1.2. Smoke effects and translucent water bottles receive full-scene lighting via the LLL.

Order Independent Transparency as well as Indirect Illumination [Gruen and Thibieroz 10].

For each new frame in the game, we populate the linked list of lights and we later access it to evaluate lighting at each pixel.

1.2.1 GPU List Structure

For efficient lighting using the Light Linked List, we need to store each light's minimum and maximum depths so we can quickly reject any pixel that is outside of the light's boundaries. We also store the index of the light into a global array, where we keep each light's attributes such as colors, radii, intensities, etc.

Finally, we store a link to the next light at the current screen pixel: The Light Linked List algorithm follows a LIFO convention, where the last linked element stored is evaluated first.

```
struct LightFragmentLink
{
  float m_MinDepth;  // Light minimum depth at the current pixel
  float m_MaxDepth;  // Light maximum depth at the current pixel

  uint m_LightIndex; // Light index into the full information array
  uint m_Next;       // Next LightFragmentLink index
};
```

1.2.2 GPU Structure Compressed

Because memory can be scarce on some systems and in an effort to reduce bandwidth usage, we chose to compress the `LightFragmentLink` structure and shave off half of the original memory requirements.

Both minimum and maximum light depths were converted to half precision and packed into a single unsigned integer `uint`. HLSL provides the useful intrinsic `f32tof16` to convert from full precision to half precision float. The light index was compressed from 32 to 8 bits, which puts an upper limit of 256 maximum visible lights at any frame. In practice, we found out that our scenes rarely ever exceed 75 lights per shot, but if the need for more than 256 lights ever comes up, we can either allocate more bits for the index or place it back in its own unsigned integer.

The link to the next fragment bits were reduced from 32 down to 24 bits in order to fit with the 8-bit light index. A 24-bit unsigned integer allows for more than 16 million actively linked fragments at once. The compressed `LightFragmentLink` structure stands at 8 bytes, whereas previously it required 16 bytes of memory.

```
struct LightFragmentLink
{
   uint m_DepthInfo; // High bits min depth, low bits max depth
   uint m_IndexNext; // Light index and link to the next fragment
};
```

1.2.3 Required Resources

To generate the Light Linked List, we use a total of four buffers, though the algorithm can easily be modified to require only three.

The first buffer is a pool of all the `LightFragmentLinks` that can be allocated and linked during a single frame. This resource is a read and write structured buffer:

```
RWStructuredBuffer < LightFragmentLink > g_LightFragmentLinkedBuffer
```

The `LightFragmentLink` minimum and maximum depth values will be generated in separate draw calls, and thus we need a buffer to temporarily store one value while waiting for the matching depth to render. The second required buffer is a read and write byte address buffer:

```
RWByteAddressBuffer                   g_LightBoundsBuffer
```

The third buffer is also a read and write byte address buffer that will be used to track the index of the last `LightFragmentLink` placed at any given pixel on screen:

```
RWByteAddressBuffer                   g_LightStartOffsetBuffer
```

The final buffer is an optional depth buffer that will be used to perform software depth testing within a pixel shader. We chose to store the depth as linear in a FP32 format instead of the typical hyper values.

1.2.4 Light Shells

To render the dynamic lights into the LLL buffers, we represent the lights as geometry: Point lights are represented by spheres (Figure 1.3), spotlights are represented by cones, and area lights are represented by boxes.

To perfectly represent a sphere or a cone in 3D space with polygons, we need an extremely well-tessellated mesh, which places a heavy burden on both memory resources and GPU rendering time. To work around the tessellation problem, we resort to creating coarsely tessellated geometry that is oversized enough to fully contain the original high-resolution mesh.

1.3 Populating the Light Linked List

The pixel shader that generates the light linked list can be described in three steps. The first step is to perform a software depth test to reduce the number of `LightFragmentLinks` allocated in a single frame. The depth test is followed by collecting the light's minimum and maximum depth, before moving forward with the allocation of a `LightFragmentLink` element.

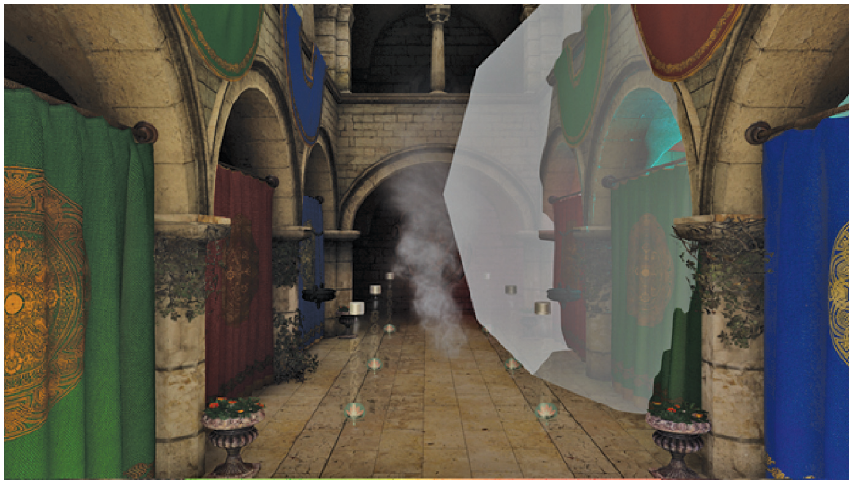

Figure 1.3. The light shell displayed in gray is used to describe a point light in the scene.

1.3.1 Software Depth Test

The Light Linked List algorithm allocates and links `LightFragmentLink` elements when the back faces of the light geometry get rasterized and sent to the pixel shader. In the common scenario where the scene geometry intersects the light sources, the hardware depth test can let the front faces pass through but occlude the back faces and thus interfere with the allocation of a `LightFragmentLink` (Figure 1.4).

To guarantee that back faces get processed by the pixel shader, we disable the hardware depth test and only perform the software test against the front faces; this will be explained in detail in the next section.

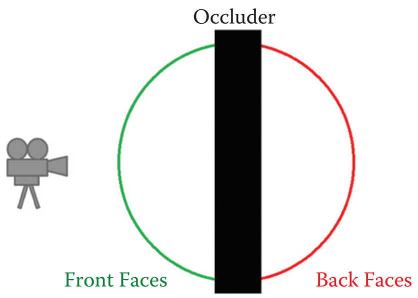

Figure 1.4. Front faces in green pass the hardware depth test, whereas back faces fail.

1.3.2 Depth Bounds

The `LightFragmentLink` structure stores both minimum and maximum light depth; however, those two values are rasterized by the hardware and sent to the pixel shaders at different times: The minimum depth will be carried through by the light geometry's front faces, whereas the maximum depth will be provided by the geometry's back faces.

We first draw the light geometry with back-ace culling turned on to allow rasterization of only the front faces. A pixel is determined to belong to a front- or back-facing polygon by the use of the HLSL semantic `SV_IsFrontFace`.

We perform a software depth test by comparing the light depth against the scene's depth. If the test fails, we turn the light depth into a negative value. If the test passes, we leave the target value unaltered.

The light's incoming depth is stored in an unsigned integer's lower 16 bits, the global light index in the upper 16 bits, and this value is then written to the `g_LightBoundsBuffer` resource.

```
// Detect front faces
if(front_face == true)
{
   // Sign will be negative if the light shell is occluded
   float depth_test = sign( g_txDepth[vpos_i].x - light_depth);

   // Encode the light index in the upper 16 bits and the linear
   // depth in the lower 16
   uint bounds_info = (light_index << 16) | f32tof16( light_depth*
       depth_test );

   // Store the front face info
   g_LightBoundsBuffer.Store( dst_offset, bounds_info );

   // Only allocate a LightFragmentLink on back faces
   return;
}
```

Once we have processed the front faces, we immediately rerender the light geometry but with front-face culling enabled.

We fetch the information previously stored into `g_LightBoundsBuffer`, and we decode both the light ID and the linear depth. At this point, we face two scenarios.

In the first scenario, the ID decoded from the `g_LightBoundsBuffer` sample and the incoming light information match. In this case, we know the front faces were properly processed and we proceed to check the sign of the stored depth: if it's negative we early out of the shader since both faces are occluded by the regular scene geometry.

The second scenario occurs when the decoded ID doesn't match the light information provided by the back faces. This scenario can happen when the frustum near clip intersects the light geometry. In this case, the minimum depth to be stored in the `LightFragmentLink` is set to zero.

```
// Load the content that was written by the front faces
uint bounds_info = g_LightBoundsBuffer.Load( dst_offset );

// Decode the stored light index
uint stored_index = (bounds_info >> 16);

// Decode the stored light depth
float front_depth = f16tof32( bounds_info >> 0 );

// Check if both front and back faces were processed
if(stored_index == light_index)
{
  // Check the case where front faces rendered but were occluded
  // by the scene geometry
  if(front_depth < 0)
  {
      return;
  }
}
// Mismatch, the front face was culled by the near clip
else
{
front_depth = 0;
}
```

1.3.3 Allocation of LightFragmentLink

Now that we know both minimum and maximum light depths are available to us, we can move forward with the allocation of a `LightFragmentLink`. To allocate a `LightFragmentLink`, we simply increment the internal counter of our `StructuredBuffer` containing all the fragments. To make the algorithm more robust and to avoid driver-related bugs, we must validate our allocation and make sure that we don't overflow:

```
// Allocate
uint  new_lll_idx = g_LightFragmentLinkedBuffer.IncrementCounter();

// Don't overflow
if(new_lll_idx >= g_VP_LLLMaxCount)
{
    return;
}
```

Once we have allocated a `LightFragmentLink`, we need to update our second `RWByteAddressBuffer` to keep track of the last inserted LLL element. Again, we make use of the HLSL atomic function `InterlockedExchange`:

```
uint     prev_lll_idx;

// Get the index of the last linked element stored and replace
// it in the process
g_LightStartOffsetBuffer.InterlockedExchange( dst_offset, new_
   lll_idx, prev_lll_idx );
```

At this point, we have all four of the required values to populate and store a valid `LightFragmentLink`:

```
// Encode the light depth values
uint light_depth_max = f32tof16( light_depth );// Back face depth
uint light_depth_min = f32tof16( front_depth );// Front face depth

// Final output
LightFragmentLink element;

// Pack the light depth
element.m_DepthInfo = (light_depth_min << 16) | light_depth_max;

// Index/Link
element.m_IndexNext = (light_index << 24) | (prev_lll_idx &
    0xFFFFFF);

// Store the element
g_LightFragmentLinkedBuffer[ new_lll_idx ] = element;
```

1.4 Accessing the Light Linked List

Accessing the Light Linked List is the same whether your engine uses a deferred or a forward renderer.

The first step is to convert the incoming pixel position from viewport space to an LLL index, and we do so by first converting the vPos to the LLL resolution, as shown below.

```
uint lll_x     = uint( (vpos_f.x / g_VP_Width ) * g_VP_LLLWidth );
uint lll_y     = uint( (vpos_f.y / g_VP_Height) * g_VP_LLLHeight );
uint src_index = lll_y * g_VP_LLLWidth + lll_x;
```

With the LLL index calculated, we fetch our first link from the unordered access view resource `g_LightStartOffsetView` and we start our lighting loop; the loop stops whenever we find an invalid value.

```
uint src_index = ScreenUVsToLLLIndex( screen_uvs );
uint first_offset = g_LightStartOffsetView[ src_index ];

 // Decode the first element index
 uint element_index = (first_offset & 0xFFFFFF);
```

```
   // Iterate over the Light Linked List
   while( element_index != 0xFFFFFF )
   {
      // Fetch
       LightFragmentLink element   = g_LightFragmentLinkedView[↩
           element
       _index];

       // Update the next element index
       element_index         = (element.m_IndexNext & 0xFFFFFF);
        ...
   }
```

Once we have acquired a valid `LightFragmentLink`, we decode the stored light depths and we perform a simple bounds test against the incoming pixel: if the pixel lies outside the light's bounds, we skip the rest of the lighting loop.

```
// Decode the light bounds
float light_depth_max     = f16tof32( element.m_DepthInfo >> 0 );
float light_depth_min     = f16tof32( element.m_DepthInfo >> 16 );

// Do depth bounds check
if( (l_depth > light_depth_max) || (l_depth < light_depth_min) )
{
   continue;
}
```

If our pixel lies within the light's bounds, we decode the global light index stored in the `LightFragmentLink` and we use it to read the full light information from a separate global resource.

```
// Decode the light index
uint light_idx = (element.m_IndexNext >> 24);

// Access the light environment
GPULightEnv light_env = g_LinkedLightsEnvs[ light_idx ];
```

1.5 Reduced Resolution

One way to reduce the memory footprint of the algorithm is to shrink the resolution at which the Light Linked List is stored. Running at a full resolution of 1080p and assuming an even light distribution of 32 lights per pixel, the total memory required for the linked list would be

$$1920 \times 1080 \times 32 \times \texttt{LightFragmentLink} = 506.25 \text{ MB}.$$

In practice, generating the Light Linked List at one quarter of the native game resolution, or even one eighth, is largely sufficient and reduces the required memory footprint by a significant amount:

$$(1920 \div 8) \times (1080 \div 8) \times 32 \times \texttt{LightFragmentLink} = 7.91 \text{ MB}.$$

1.5.1 Depth Down-Sampling

For engines that perform either a depth prepass or have a G-buffer, layer we need to down-sample the depth buffer to match the resolution of the LLL.

Scaling down the depth buffer must be done via point sampling and the use of the function `max` to avoid missing light information due to aggressive Z-culling. To speed up the down-sampling of the depth buffer, we make extensive use of the `GatherRed` function, which allows us to read four depth samples at once. Below is an example of how to down-sample a full-resolution depth buffer down to one eighth across the width and height:

```
float4 d4_max;

{
 float4 d4_00 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2↩
     (-3, -3));
 float4 d4_01 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2↩
     (-1, -3));
 float4 d4_10 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2↩
     (-3, -1));
 float4 d4_11 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2↩
     (-1, -1));
          d4_max = max(d4_00, max( d4_01, max( d4_10, d4_11)));
}

{
  float4 d4_00 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2↩
      (-3, 3));
  float4 d4_01 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2↩
      (-1, 3));
  float4 d4_10 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2↩
      (-3, 1));
  float4 d4_11 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2↩
      (-1, 1));
        d4_max = max(d4_max, max(d4_00, max( d4_01, max( d4_10, ↩
            d4_11))));
}

{
  float4 d4_00 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2↩
      (3, -3));
  float4 d4_01 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2↩
      (1, -3));
  float4 d4_10 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2↩
      (3, -1));
  float4 d4_11 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2↩
      (1, -1));
        d4_max = max(d4_max, max(d4_00, max( d4_01, max( d4_10, ↩
            d4_11))));
}

{
    float4 d4_00 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2(↩
         3, 3));
    float4 d4_01 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2(↩
         1, 3));
    float4 d4_10 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2(↩
         3, 1));
    float4 d4_11 = g_txDepth.GatherRed(g_samPoint, screen_uvs, int2(↩
         1, 1));
          d4_max = max(d4_max, max(d4_00, max( d4_01, max( d4_10, ↩
              d4_11))));
  }

  // Calculate the final max depth
  float depth_max = max(d4_max.x, max( d4_max.y, max(d4_max.z, ↩
      d4_max.w)));
```

1.6 Conclusion

The Light Linked List algorithm helped us to drastically simplify our lighting pipeline while allowing us to light translucent geometry and particle effects, which were highly desirable. With Light Linked List, we were able to match or improve the performance of our deferred renderer, while reducing memory use. Additionally, the flexibility of Light Linked List allowed us to easily apply custom lighting for materials like skin, hair, cloth, and car paint.

In the future, we intend to further experiment with a more cache-coherent layout for the `LightFragmentLink` buffer, as this seems likely to yield further performance improvements.

Bibliography

[Gruen and Thibieroz 10] Holger Gruen and Nicolas Thibieroz. "Order Independent Transparency and Indirect Illumination Using Dx11 Linked Lists." Presentation at the Advanced D3D Day Tutorial, Game Developers Conference, San Francisco, CA, March 9–13, 2010.