8.5 Designing a Renderer for Multiple Lights: The Light Pre-Pass Renderer

Wolfgang Engel, Rockstar Games

Renderer design is like building a foundation for a house. The house might end up bigger or smaller than the foundation or too heavy. Knowing what will stand on the foundation up-front and doing some educated guessing about the soil and future extensions is a requirement to building a stable house.

In software design terms the foundation is the renderer design, and the house, size, and weight are the graphics requirement that is hopefully documented before the project starts. These requirement have a tendency to change during the project.

This article will focus on a renderer design that supports a huge number of lights. This design was implemented in games such as *GTA IV* and *Midnight Club Los Angeles*.

It will cover three different renderer design patterns that were used in the game development process in the last eight years to solve this problem: Z pre-pass renderer, deferred renderer, and light pre-pass renderer.

Z Pre-Pass Renderer

The design pattern in this article that is labeled "Z pre-pass renderer" was used by John Carmack in *DOOM III*.

The idea is to construct a depth-only pass (Z pre-pass) first and therefore fill the Z buffer with depth data, and at the same time fill the Z culling. Then render the scene using this occlusion data to prevent pixel overdraw. This approach is used by hardware vendors in the design of their hardware and is very common today.

Rendering depth only can be done at two to eight times the speed compared to a combined color and depth write. Figure 8.5.1 shows the render passes.

FIGURE 8.5.1 Render passes in a Z pre-pass renderer.

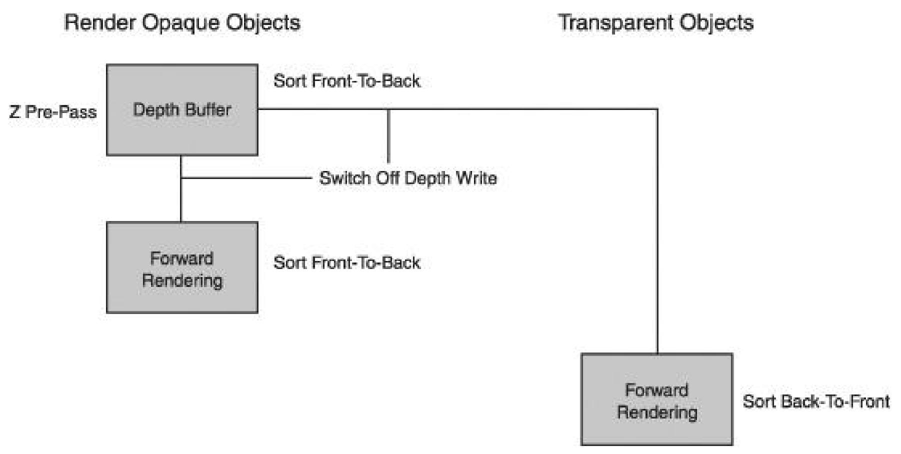


Figure 8.5.1 shows the opaque and transparent rendering passes in a Z pre-pass renderer.

After the opaque objects are drawn into the depth buffer, the depth writes to the depth buffer are switched off, and transparent objects are rendered front-to-back. This makes sure that the transparent objects do not write into the depth buffer at any time and are also not considered in the Z pre-pass.

A naïve multi-light solution that accompanies a Z pre-pass renderer design pattern would just render a limited number of lights in the pixel shader. Assuming that a pixel shader is written for up to eight point lights, this would mean that we can draw eight point lights per draw call, independent of whether the objects are opaque or transparent.

Although the shaders for opaque and transparent objects would remain similar, it would be necessary to split up geometry following the number of lights for certain game objects. This might be expensive because current graphics hardware has long pipelines, and it is very sensitive to the number of draw calls.

A more advanced approach stores light source properties such as position, light color, and other light properties in textures following a 2D grid that is laid out in the game world. An index texture would get the ID of the visible lights from the grid, and then the light properties of all those lights are fetched from the light property textures.

Because the texture fetch from the light property textures would depend on the result of a index texture fetch, current hardware will be slower with many texture fetches like this.

The advantage of this approach is that the rendering path regarding opaque and transparent objects would be quite similar.

The cost of using many lights with this approach led to the adoption of a render design pattern that is now called the deferred renderer.

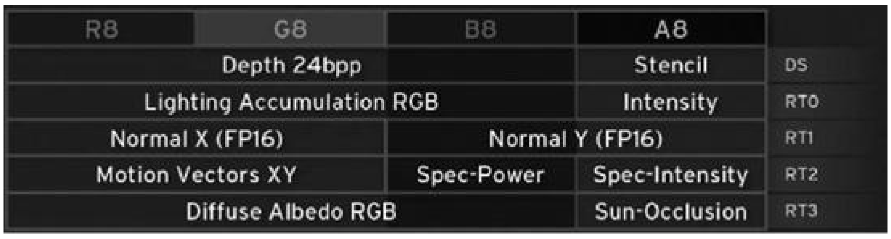
Deferred Renderer

The underlying idea of a deferred renderer is based on a paper from SIGGRAPH 1988 [Deering].

Similar to the Z pre-pass renderer, the deferred renderer pattern splits up rendering into two passes. While the Z pre-pass renderer only fills depth values to utilize the hardware depth and occlusion culling in the second pass, the deferred renderer renders all data necessary to render lights and shadows into render targets. This first pass is usually called the G-Buffer write or geometry pass. In the following second pass, called the lighting pass, each light is additively blended into the light buffer while solving the whole lighting equation each time. During each of these passes shadows are calculated.

A typical G-Buffer stores the data shown in Figure 8.5.2 [Valient].

FIGURE 8.5.2 G-buffer layout (courtesy of Michal Valient).



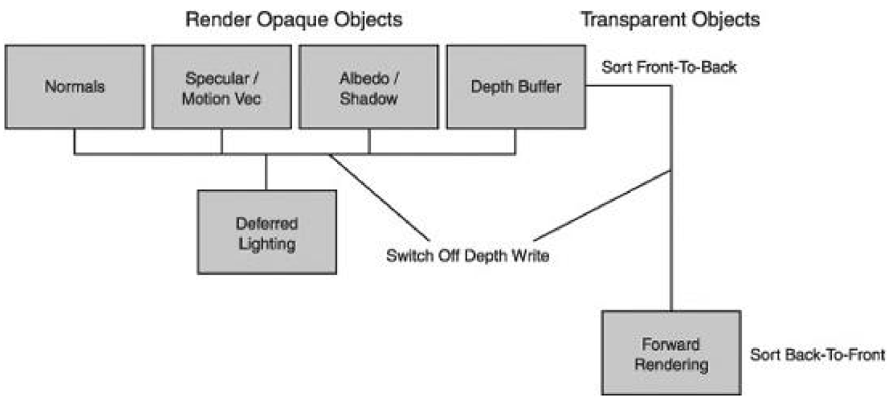
R8	G8	B8	A8	
Depth 24bpp			Stencil	DS
Lighting Accumulation RGB			Intensity	RT0
Normal X (FP16)		Normal Y (FP16)		RT1
Motion Vectors XY		Spec-Power	Spec-Intensity	RT2
Diffuse Albedo RGB			Sun-Occlusion	RT3

Together with the transparent rendering path, a simple overview of the render passes is shown in Figure 8.5.3.

Using a G-Buffer is possible because data can be written at the same time into several render targets in a what are called multiple render targets (MRTs).

Killzone 2 uses all four render targets that belong to an MRT and one depth buffer in the first pass, the geometry pass. The render targets RT1, RT2, and RT3 and the depth buffer are filled in during this pass. Based on this data, the light buffer in RT0 is filled in during the lighting pass.

FIGURE 8.5.3 Render passes in a deferred renderer.



In general, the G-Buffer holds material data of all objects that are visible in the scene, motion blur vectors, depth data to reconstruct position, and stencil data. For all the objects, it holds the different specular properties, normals, and the different color values, so objects can differ in the way their specular reflection and albedo are calculated.

The main advantage of using a deferred renderer is the huge number of lights that can be additively blended into the light buffer independent of geometry restrictions or geometry draw calls. Additionally, it only requires rendering all opaque objects once for the main scene (apart from the scene rendering, geometry will need to be rendered several times for reflections, shadows, etc.).

Because transparent objects can't be rendered into the depth buffer, they need to be handled in a dedicated pass. In current games this pass is similar to what is described above for a Z pre-pass renderer. With the latest DirectX 10 hardware, newer techniques such as reverse depth peeling [Thibieroz07] or a stencil-routed K-buffer [Bavoil] can be used to catch several layers of depth, making it possible to render transparent objects like opaque objects.

There are several challenges with a deferred renderer. Reading and writing a G-Buffer consisting of four or five render targets substantially increases the hardware requirements. The hardware needs to support multiple render targets (MRTs) and needs to have support for a high amount of memory read bandwidth.

Because of the memory bandwidth requirements, several techniques were developed that are used to optimize memory bandwidth usage.

One simple way to optimize the bandwidth is by scissoring out the 3D bounding box volume of the light projected into a 2D rectangle [Placeres]. On recent NVIDIA hardware depth bounds can act as a 3D scissor that would scissor out in all three dimensions.

Another way to reduce the bandwidth usage is to render convex geometry and then use the depth buffer to reject rendering of non-lit pixels. For a point light this would be a sphere; for a spotlight it would be a spherical cone. When the camera is inside this volume, only the back faces of the volumes are rendered. Additionally, the back-facing pixels of this volume are only rendered when the depth buffer visibility test fails. This can be achieved by inverting the depth test when rendering the volumes (`D3DCMP_GREATER` instead of `D3DCMP_LESSEQUAL`) [Thibieroz04] (depending on the hardware platform, Z culling can rely on the direction of the depth test, and therefore changing this direction can decrease performance).

A more sophisticated way is to use the stencil test similar to the depth-fail stencil shadow volume technique. This requires rendering two passes. First, the bounding volumes are rendered into the stencil buffer, and then the light is blit into the light buffer.

When drawing the back-facing light volume geometry, the stencil test increments when the depth buffer test fails with `D3DCMP_GREATER` instead of `D3DCMP_LESSEQUAL`. When drawing front-facing light volumes, the depth test is set to D3DCMP_LESSEQUAL, and the stencil test decrements when the depth test fails. In the second pass, only lit pixels are rendered where the stencil value is greater than or equal to 1 [Hargreaves][Valient]. The stencil buffer is cleared to its default value during the blit. Hierarchical stencil support or stencil culling can additionally speed up the blit.

Hardware MSAA support is usually more complicated with a deferred renderer than with a Z pre-pass renderer because MSAA is encapsulated on the PC with DirectX in the runtime. On this platform programmers do not have much control over where and how the MSAA will happen.

On the Xbox 360 and PS3 platforms, hardware MSAA is possible but can be quite expensive because the whole G-Buffer needs to run in MSAA'ed resolution.

A different challenge with a deferred renderer is material variety. Because space in the G-Buffer is very limited, the variety of materials that can be used compared to a Z pre-pass renderer is quite small. Additionally, for all materials a very similar lighting equation needs to be used. Otherwise, pixel shader switching in the lighting stage would be too expensive.

The restricted material support and the huge amount of memory bandwidth required led to the idea of a light pre-pass renderer.

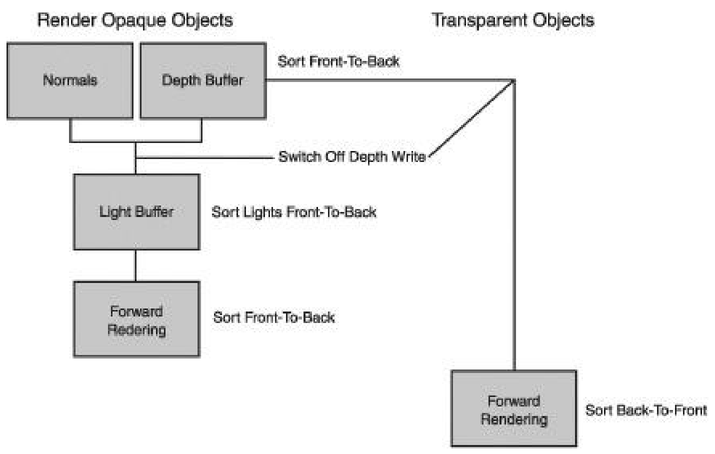
Light Pre-Pass Renderer

While a deferred renderer stores material properties in a G-Buffer, a light pre-pass renderer stores depth and normals in one or two render targets. In a second rendering pass, the light pre-pass renderer stores the light properties of all lights in a light buffer.

Because the light buffer only stores light properties and does not require rendering the whole lighting equation with shadows, reflections, and other effects, the cost per light can be lower compared to a deferred renderer.

Figure 8.5.4 shows the render passes of a light pre-pass renderer:

FIGURE 8.5.4 Render passes in a light pre-pass renderer.



For opaque objects there are three distinct rendering passes. The first pass fills up the depth buffer and the normal buffer.

Similar to all the other renderer design patterns, the light pre-pass renderer renders into the depth buffer first. Similar to a deferred renderer, it can also fill in a normal buffer in this render pass.

Normals in the normal buffer can be stored in view space [Placeres] or world space [Thibieroz04]. With an 8:8:8:8 render target, world space normals will be of better quality when they are stored in spherical coordinates.

Without support for MRT, the depth buffer and normal buffer would need to be filled in two render passes. Compared to a deferred renderer this allows a decrease of the minimum hardware requirements.

The light buffer stores light properties. These are all the properties that are used to differ light sources from each other.

In case of a Blinn-Phong lighting model, a simplified lighting equation for several point lights can look like this:

$$I = \text{Ambient} + \sum_i \text{Attenuation}_i(N.L_i * \text{Diffuse}_{\text{Color}} * \text{Diffuse}_{\text{Intensity}} + (N.H_i)^n * \text{Specular}_{\text{Intensity}})$$

EQUATION 8.5.1 All terms that hold the letter i as subscript need to be stored in the light buffer.

To save space in the light buffer, these light-dependent terms need to be stored by considering their locality in space by applying the diffuse term N.L and the attenuation factor to all of them. A typical light buffer might therefore hold the following terms:

$$\text{Channel 1:} \sum_i N.L_i * \text{Diffuse}_{\text{Red}} * \text{Attenuation}_i$$

$$\text{Channel 2:} \sum_i N.L_i * \text{Diffuse}_{\text{Green}} * \text{Attenuation}_i$$

$$\text{Channel 3:} \sum_i N.L_i * \text{Diffuse}_{\text{Blue}} * \text{Attenuation}_i$$

$$\text{Channel 4:} \sum_i N.L_i * (N.H_i)^n * \text{Attenuation}_i$$

EQUATION 8.5.2

Using the diffuse term N.L to restrict the specular term in channel 4 is done to restrict the specular reflection to the areas where diffuse lighting is visible. This is also necessary in case the normal and the light vector point in opposite directions; for example, in case the light vector is pointing toward the viewer and is occluded behind an object [Engel].

Having a light buffer set up like this packs the data tightly. Depending on the light overdraw and with the help of a scale value, an 8:8:8:8 render target should be sufficient to store all the lights.

The number of light properties that are stored in a light buffer like this should be enough to reconstruct the Blinn-Phong lighting model in a later rendering pass while using the light buffer as one light source as shown in Equation 8.5.3.

$$I = \text{Ambient} + (\text{LightBuffer}_{123} * \text{Diffuse}_{\text{Intensity}} + \text{LightBuffer}_4 * \text{Specular}_{\text{MaterialColor}} * \text{Specular}_{\text{Intensity}})$$

EQUATION 8.5.3

The approach covered so far limits the specular reflections to the shininess value of the light source but does not consider the material shininess.

If the specular term can be separated in a later rendering pass, a dedicated material shininess value can be applied as shown in Equation 8.5.4.

$$\left(\sum_i (N.H_i)^n\right)^{mn}$$ EQUATION 8.5.4

A specular term consisting of a light shininess and a material shininess property behaves differently than the specular term that is traditionally used in renderers. Adding up lights with a light shininess value and then applying the result to a material shininess value will show different results than having only one material shininess value that is added up in the light buffer. Because lights in the modern world can have very different specular characteristics, introducing a light shininess value should mimic reality better.

Applying a material shininess value to the specular reflection can be done in several ways.

1. Similar to a deferred renderer, a material specular shininess value can be stored in the normal or depth buffer (stencil area).
2. Moving into a different color space that reuses some of the ideas here to achieve a tighter packed render target.
3. A separate term can be stored to reconstruct the specular term in a later rendering pass.
4. The diffuse term stored in the first three channels of the light buffer can be converted to luminance and then used to reconstruct the specular reflection term.
5. The common rules for constructing a specular term can be bended by creating a new term that fits better into this renderer design.

While this article focuses on the third, fourth, and fifth approach, Pat Wilson's article [Wilson] shows how to use a LUV color space to store more light properties in one 8:8:8:8 render target and therefore keep the specular term separable as shown in Equation 8.5.4.

Storing an Additional Diffuse Term

To reconstruct the specular term in the later rendering pass, an additional diffuse term can be stored. All the light properties that would need to be stored with the new term at the end of the list would look as shown in Equation 8.5.5.

$$\text{Channel 1:} \sum_i N.L_i * \text{Diffuse}_{Red} * \text{Attenuation}_i$$

$$\text{Channel 2:} \sum_i N.L_i * \text{Diffuse}_{Green} * \text{Attenuation}_i$$

$$\text{Channel 3:} \sum_i N.L_i * \text{Diffuse}_{Blue} * \text{Attenuation}_i$$

$$\text{Channel 4:} \sum_i N.L_i * (N.H_i)^n * \text{Attenuation}_i$$

$$\text{Channel 5:} \sum_i N.L_i * \text{Attenuation}_i$$

EQUATION 8.5.5

With the additional term we can reconstruct the specular term as shown in Equation 8.5.6.

$$\left(\sum_i N.L_i * (N.H_i)^n * \text{Attenuation}_i\right) / \left(\sum_i N.L_i * \text{Attenuation}_i\right)$$ EQUATION 8.5.6

The result of this equation can be used to apply a material shininess value as shown in Equation 8.5.4.

Because the diffuse and the specular terms are now separable in a later rendering pass, a wide range of different materials can be supported.

The disadvantage of this approach is that it requires an additional channel to store the fifth term, exceeding the number of channels of a four-channel render target and therefore requiring an additional render target for the light buffer.

Converting the Diffuse Term to Luminance

Instead of storing a diffuse term in a separate render target as shown above to reconstruct the specular term in a later rendering pass, the diffuse term stored in channels 1–3 can be converted to luminance. This should be equal to an approximation of the value stored in channel 5 of Equation 8.5.5. It can be used like the diffuse term in Equation 8.5.6. Converting the diffuse term to luminance can be done by taking a dot product between the value triple [0.2126, 0.7152, 0.0722] [EngelPOSTFX] and the term stored in the fourth channel.

Bending the Specular Reflection Rules

Another approach bends the rules about how to apply shininess. This can be done by applying the material shininess value to the term that is stored in the light buffer in the fourth channel as shown in Equation 8.5.7.

$$\left(\sum_i N \cdot L_i * (N \cdot H_i)^n * \text{Attenuation}_i\right)^{mn}$$

EQUATION 8.5.7

The main advantage of this solution is that all the light properties fit into four channels of a render target, and at the same time different material properties can be applied to surfaces. Equation 8.5.8 lists a few ideas on what can be done in the forward rendering path.

$$I = \text{Ambient} + \left\{\begin{matrix}\text{Minnaert}+\\ \text{Subsurface}+\\ \text{Reflections}+\end{matrix}\right\}(\text{LightBuffer}_{123} * \text{Diffuse}_{\text{Intensity}} + (\text{LightBuffer}_4)^{mn}\{\text{Fresnel}\} * \text{Specular}_{\text{MaterialColor}})$$

EQUATION 8.5.8

The terms in the curly brackets represent some of the material variety that is possible within the light pre-pass renderer (see the appendix at the end of this article for more information).

Comparison and Conclusion

Compared to a deferred renderer, the light pre-pass renderer offers more flexibility regarding material implementations. Compared to a Z pre-pass renderer, it offers less flexibility but a flexible and fast multi-light solution.

The main disadvantage compared to a deferred renderer is the requirement to render all geometry for the main view twice, but this is what a Z pre-pass renderer requires as well, and compared to the amount of geometry that needs to be rendered in case there are, for example, four shadow maps and reflections, the proportional increase of vertex throughput for the additional geometry pass for opaque objects should be moderate.

Because the light pre-pass renderer only fetches two textures for each light, the read memory bandwidth is lower than for a deferred renderer.

Using MSAA in the light pre-pass renderer can be more efficient than with the deferred renderer. The depth buffer and the back buffer need to be MSAA'ed. Additionally, the normal buffer and the light buffer can be MSAA'ed too.

Because the light buffer only has to hold light properties, the cost of rendering one light source is lower than for a similar setup in a deferred renderer. For example, a directional light rendered with shadows, reflections, and all the other scene properties would take about 3.3 ms in a deferred renderer. With a light pre-pass the same light on the same hardware platform would take 0.7 ms to render into the light buffer, and probably the rest of the time would be spent in the later rendering pass. By increasing the number of directional lights, the cost per light will go down in the case of the light pre-pass renderer. Similar characteristics apply for point and spot lights.

The light pre-pass renderer is scalable on less powerful platforms. It does not require MRT support, and it consumes less read memory bandwidth compared to a deferred renderer. Therefore, it is suitable for platforms like the Wii or in general platforms that feature a DX8.1-capable graphics card.

Acknowledgments

I would like to thank, in no particular, order Kenneth Hurley, Steve Reed, Thomas Johnstone, Raymond Kerr, Ray Tran, Christina Coffin, Simon Brown, Pat Wilson, Matthias Wloka, Michal Valient, Martin Mittring, Marco Salvi, and Michael Krehan for discussing various techniques described in this article with me. Many of those discussions have spiked new ideas or new thoughts that found their way into this article.

Appendix: Applying Different Materials with a Light Pre-Pass Renderer

Here is how the approximated skin model in the NVIDIA SDK 9.5 named lambSkinDusk [NVIDIA] can be used in a light pre-pass renderer. The original code is below.

```
float ldn = dot(L,N);


float diffComp = max(0,ldn);


Diffuse = float4((diffComp * DiffColor).xyz,1);


float subLamb = smoothstep(-RollOff,1.0,ldn) - smoothstep(0.0,1.0,ldn);


subLamb = max(0.0,subLamb);
```

```
Subsurface = subLamb * SubColor;
```

The code for the light pre-pass looks like this:

```
// convert the diffuse term in the first three channels

// of the Light Buffer to luminance

float Lum = dot(LightBuffer.rgb,float3(0.2126, 0.7152, 0.0722));


// the content of LightBuffer.rgb contains the same content

// as the Diffuse variable above

float subLamb = smoothstep(-RollOff, 1.0, Lum) - smoothstep(0.0, 1.0, Lum);
subLamb = max(0.0,subLamb);

Subsurface = subLamb * SubColor;
```

Using a Minnaert lighting model can be done like this [Hurley]:

```
// convert the diffuse term in the first three channels

// of the Light Buffer to luminance

float Lum = dot(LightBuffer.rgb,float3(0.2126, 0.7152, 0.0722));



// N.L^k * V.N^1-k

float Minnaert = pow(Lum, k) * pow(VN, 1-k);

Minnaert *= MaterialColor;
```

References

[Bavoil] Louis Bavoil, Kevin Myers, “Deferred Rendering using a Stencil Routed K-Buffer,” *ShaderX*^{6}

[Deering] Michael Deering “The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics,” SIGGRAPH 1988

[Engel] Wolfgang Engel, *Programming Vertex and Pixel Shaders*, pp. 123–127, Charles River Media, 2004, ISBN 1-58450-349-1

[EngelPOSTFX] Wolfgang Engel, “Post-Processing Pipeline,” www.coretechniques.info/index_2007.html

[Hargreaves] Shawn Hargreaves, “Deferred Shading,” www.talula.demon.co.uk/DeferredShading.pdf

[Hurley] Kenneth Hurley, “Minnaert Shading,” www.realistic3d.com/minnaert.htm

[NVIDIA] NVIDIA SDK 9.5, lambSkinDusk example on the following page: http://developer.download.nvidia.com/SDK/9.5/Samples/effects.html

[Placeres] Frank Puig Placeres, “Overcoming Deferred Shading Drawbacks,” pp. 115–130, *ShaderX*^{5}

[Thibieroz04] Nick Thibieroz, “Deferred Shading with Multiple-Render Targets,” pp. 251–269, *ShaderX*^{2} *– Shader Programming Tips & Tricks with DirectX*^{9}

[Thibieroz07] Nick Thibieroz, “Robust Order-Independent Transparency via Reverse Depth Peeling in DirectX® 10,” *ShaderX*^{6}

[Valient] Michal Valient, “Deferred Rendering in Killzone 2,” www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf

[Wilson] Pat Wilson, “Light Pre-Pass Renderer: Using a LUV Color Model,” *ShaderX*^{7}