IV 2

Physically Based Deferred Shading on Mobile

Ashley Vaughan Smith and Mathieu Einig

2.1 Introduction

In order for graphical applications to achieve maximum performance and therefore maximum graphical quality, they need to utilize memory bandwidth as best as possible. This is especially true on mobile devices without large, fast DDR RAM, like discrete GPUs have, and where power is limited through battery life.

This bandwidth bottleneck is even more tangible in the context of deferred shading renderers, where large G-buffers need to be stored and retrieved multiple times during the rendering process. It is possible to take advantage of the fast on-chip memory that exists on tile-based GPUs to prevent unnecessary data transfers, which improves power consumption and increases performance.

This chapter discusses how to achieve minimum main memory bandwidth utilization along with the tradeoffs and benefits to doing so, including power usage. Also discussed is how to take advantage of the savings in time spent reading and writing to main memory by implementing a physically based deferred rendering pipeline. (See the example in Figure 2.1.)

2.2 Physically Based Shading

One key aspect of rendering is trying to reproduce real-world materials in a convincing way. This is a problem that has to be solved both on the engineering side (how do I render shiny metals?) but also on the art side (what colors do I need for a gold material in this specific renderer?), usually leading to either the creation of an impractical number of material shaders that have their own sets of constraints or to simpler shaders that cannot approximate most common materials.

Figure 2.1. An efficient deferred shading renderer.

Physically based shading (PBS) is an attempt at solving the rendering equation [Kajiya 86], but with a more unified shading model than its ad-hoc predecessors [Pharr and Humphreys 04]. While there has been an increasing amount of attention on PBS in the past few years, it should be noted that it does not involve any fundamentally new concept. It should instead be seen as a set of criteria and constraints (for both the developers and artists) that, if respected, should produce an image with plausible materials in most scenarios. Some of the key concepts are as follows:

- Energy conservation: you cannot reflect more light than you receive. This means that the specular intensity is inversely proportional to its size.
- Everything has Fresnel reflections.
- Lighting calculations need to be done in linear space to achieve correct output.

It also formalizes the material parameters:

- Albedo: Formerly known as the diffuse map, with a few notable differences: lighting information should not be present, and this texture is mostly uniform. Metals do not have an albedo color.
- Reflectance: Formerly known as the specular map, expect that in PBS, this is mostly a material constant. Only metals have color information in their reflectance.

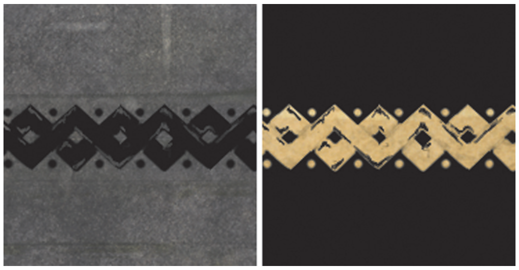

Figure 2.2. Specular workflow: albedo (left) and reflectance (right).

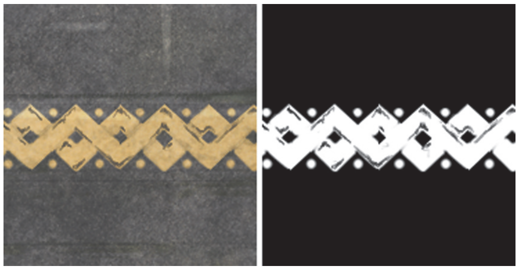

Figure 2.3. Metallicness workflow: albedo/reflectance texture (left) and metallicness texture (right).

- Roughness: This is the micro surface data (i.e., the bumps that are too high frequency to be stored in a normal map). It defines the “blurriness” of the specular highlights. This is where artists should focus most of the details. It should also be noted that the roughness textures’ mipmap levels should be generated from the original texture and the normal map.

Figure 2.2 shows an example of albedo and reflectance textures for a material with stone and gold.

The albedo and reflectance textures are nearly mutually exclusive: Metals do not have a diffuse component, and insulators can be assumed to have the same constant specular color. This means that they can both be merged into a single texture. An extra mask, representing which areas are to be treated as specular, has to be created. But this saves memory and could be stored as the alpha channel for convenience. See Figure 2.3 for the same material represented using the metallicness workflow.

Using a metallicness workflow requires less memory; however, it also introduces visual artifacts due to texture compression and filtering. The metallicness map should normally be a nearly binary texture, but because of bilinear filtering or mipmapping, sharp transitions between metal and dielectric will become

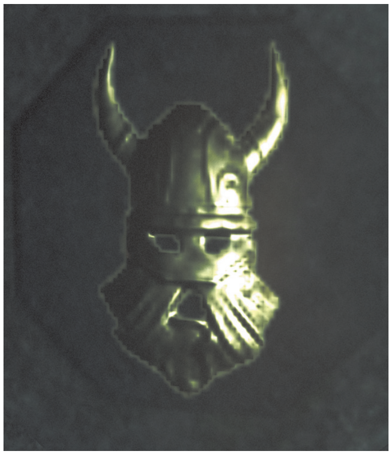

Figure 2.4. Bright halos caused by the transition between the gold and stone materials.

smooth gradients. This causes the renderer to interpret the color texture as both an albedo and specular map, which may then lead to unnaturally shiny or matte halos, as shown in Figure 2.4.

These issues can generally be fixed by eroding or dilating the metallicness mask around the problematic areas.

2.3 An Efficient Physically Based Deferred Renderer

With the ever-increasing computing power available on mobile chips, state-of-the-art rendering techniques and paradigms such as physically based shading are becoming increasingly feasible on mobile. A deferred renderer is a common way to achieve detailed dynamic lighting [Smith 14]. In the next section we detail how to create an efficient deferred physically based renderer.

2.3.1 A Bandwidth Friendly G-Buffer Setup

Standard deferred shading renderers use multiple passes, each rendering to their own framebuffer, as shown in Figure 2.5. At the end of each render, the content of the on-chip memory is transferred to the main memory, which consumes a prohibitive amount of bandwidth for a mobile GPU.

However, given that each pass relies solely on the previous output and only requires data from the same pixel, it is possible to merge them all into a single large pass and to use the Framebuffer Fetch mechanism [Khronos 13] to bypass the intermediary data transfer. The OpenGL ES Framebuffer Fetch extension

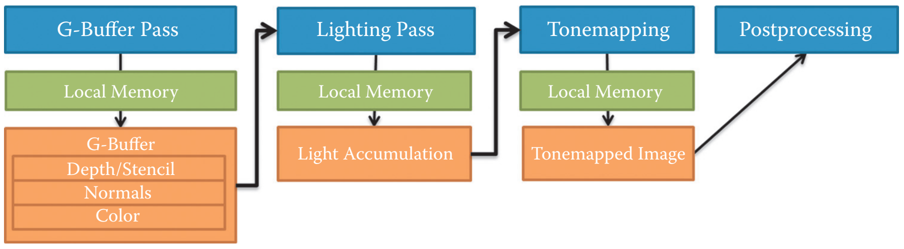

Figure 2.5. Standard deferred shading pipeline.

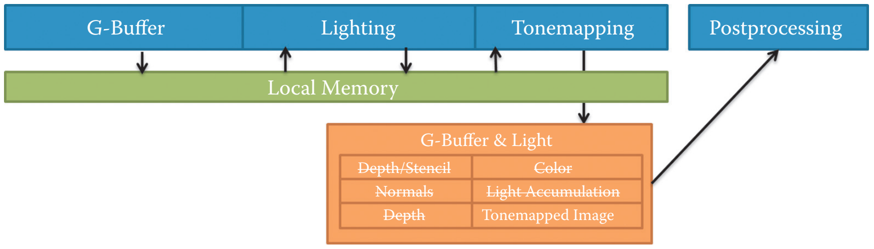

Figure 2.6. Bandwidth-friendly pipeline using Framebuffer Fetch.

allows the shader to treat what is traditionally the output as an input/output register.

Figure 2.6 shows how the pipeline can be modified to limit the amount of data transfer for the G-buffer, lighting, and tonemapping passes. These three passes are merged into one, which has several implications on the G-buffer layout: The HDR light accumulation buffer and tonemapped output are both added to the G-buffer. Furthermore, in order to access the geometry depth in the lighting stage, we pass the current depth through in local memory. Because only the tonemapped output is needed in the subsequent passes, all other attachments should be explicitly discarded to avoid unnecessary data transfer using `glInvalidateFramebuffer()`. It is also possible to reduce the G-buffer size by recycling the normal or color attachments into the tonemapped output.

This pipeline can be improved further: A whole G-buffer is allocated, even though it is never actually written to since its attachments are discarded. Its main use is only to define the data layout to be used for the render pass. The Pixel Local Storage (PLS) extension [Khronos 14] fixes this issue by letting the developer define the data layout in the shaders and write arbitrary data to the on-chip pixel memory. Figure 2.7 shows the same pipeline, improved with PLS: The G-buffer storage is no longer needed, and only one simple RGB color attachment is created for the whole deferred shading renderer.

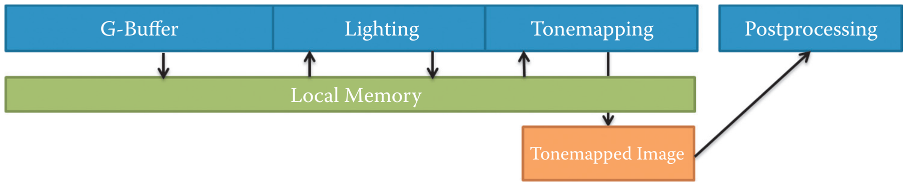

Figure 2.7. Optimal pipeline with Pixel Local Storage.

```
layout(rgb10a2) highp vec4 lightAccumulation_padding;
layout(r32f) highp float depth;
layout(rgba8) highp vec4 normals_roughness;
layout(rgba8) highp vec4 baseColour_padding;
layout(rgba8) highp vec4 specularColour_padding;
```

Listing 2.1. Specular G-buffer setup (160 bits, 142 bits actually used).

PowerVR Rogue series 6 GPUs have 128 bits of per-pixel on-chip memory. It is possible to access up to 256 bits per pixel, at the cost of performance: The spilled data will be transferred back to the main memory, unless it is small enough to fit in a cache. This means that ideally the whole G-buffer and light accumulation buffer would need to fit in 128 bits for a bandwidth-efficient renderer.

Using the specular workflow leads to the data layout in Listing 2.1. Although it is the most flexible and straightforward in terms of assets, and best in terms of quality, it requires 160 bits of storage per pixel, which makes it suboptimal for the hardware because 32 bits may be spilled to main memory per pixel.

Using the metallicness workflow allows us to pack the whole pixel data into 128 bits (see Listing 2.2), which fits nicely in the per-pixel on-chip memory without spilling.

2.3.2 Fast Gamma Approximation

The albedo and reflectance channels of the G-buffer should be stored in gamma space to prevent banding issues. This conversion to and from gamma space is

```
layout(rgb10a2) highp vec4 lightAccumulation_padding;
layout(r32f) highp float depth;
layout(rgba8) highp vec4 normals_roughness;
layout(rgba8) highp vec4 albedoOrReflectance_metallicness;
```

Listing 2.2. Metallicness G-buffer setup (128 bits, 126 bits used).

```
vec3 toGamma(vec3 linearValue) {
  return pow(linearValue, vec3(2.2));
}
vec3 toLinear(vec3 gammaValue) {
  return pow(gammaValue, vec3(1.0/2.2));
}
```

Listing 2.3. Manual gamma correction.

```
vec3 toGamma(vec3 linearValue) {
  return linearValue*linearValue;
}
vec3 toLinear(vec3 gammaValue) {
  return sqrt(gammaValue);
}
```

Listing 2.4. Pseudo-gamma correction.

usually done for free by the hardware as long as the G-buffer relevant attachments are set up as sRGB. However, because the optimized pipeline does not write to a physical G-buffer, this has to be done manually by simply raising to the power of 2.2 (see Listing 2.3).

This can be optimized by assuming a gamma of 2.0, which simplifies the gamma conversion and can be a good compromise between visual quality and speed (see Listing 2.4).

2.3.3 Lighting

We have defined how to input the material parameters to the rendering system. We now define how to input the lighting information. We use and extend physically based rendering techniques from different sources [Lagarde 12]. The types of information we need to include are static diffuse lighting, static specular lighting, and dynamic specular lighting.

An offline renderer is used to generate HDR lightmaps that are used to represent static diffuse lighting information such as shadows and radiosity. As these textures can take up a large memory footprint, they should be compressed to reduce memory usage and improve texture throughput. One such compression format is ASTC, which supports HDR data; however, not all devices currently support HDR ASTC. Non-HDR compression formats such as PVRTC can be used along with an encoding method, RGBM [Karis 09], in which a second texture is used as a scale factor to enable HDR output. The RGB channels should be compressed, but the scale factor channel should be left uncompressed to prevent serious block artifacts. See Listing 2.5 on how to decode this value. Diffuse

```
const float RGBM_SCALE = 6.0;
vec3 RGBMSqrtDecode(vec4 rgbm) {
  vec3 c = (rgbm.a * RGBM_SCALE) * rgbm.rgb;
  return c*c;
}

dualRGBM.rgb = texture(LightmapTexture, uv).rgb;
dualRGBM.a = texture(LightmapTextureScale, uv).r;
vec3 diffuseLight = RGBMDecode(dualRGBM);
diffuseLight *= surfaceColour * (1.0 - surfaceMetallicness);
```

Listing 2.5. Code for calculating the static diffuse lighting information.

lighting can be output into the light accumulation buffer in the geometry pass of the technique. Note that metallic materials do not reflect diffuse lighting.

We use image-based lighting as the input to the static specular lighting. An offline renderer is used to produce a cubemap that represents the lighting from a certain point in space. These cubemaps are converted to Prefiltered mipmaped radiance environment maps (PMREM) using the modified AMD cubemap gen tool [Lagarde 12]. A PMREM is a cubemap filtered by integrating the radiance over a range of solid angles of the hemisphere depending on the mipmap level. During the lighting stage, the surface roughness is used to select which mipmap levels of the cubemap should be sampled (see Listing 2.6). As with the lightmaps, the environment maps are stored in RGBM HDR. Listing 2.6 shows how the static specular lighting is computed, including the implementation of Shlick's approximation of Fresnel [Lagarde 12].

These static specular lights can be rendered in the same way as other dynamic lights in a deferred renderer, with a transparent mesh encompassing the light bounds.

```
// From seblagarde.wordpress.com/2011/08/17/hello-world
vec3 fresRough(vec3 specCol, vec3 E, vec3 N, float smoothness) {
  float factor = 1.0 - clamp(dot(E, N), 0.0, 1.0);
  return specCol + (max(vec3(smoothness), specCol) - specCol) *
    pow(factor, 5.0);
}
vec3 iblSpec(float roughness, vec3 specCol, vec3 N, vec3 V) {
  vec3 iblFresnel = fresRough(specCol, V, N, 1.0 - roughness);
  vec3 refDir = reflect(-V, N);
  float mipLevel = 8.0 * roughness; // 8 mips total
  vec3 envMapResult = RGBMDecode(
    textureLod(SpecularProbe, refDir, mipLevel));
  return envMapResult * iblFresnel;
}
```

Listing 2.6. Code for computing the image-based specular contribution.

For dynamic specular lighting we use the GGX equation [Walter et al. 07]. This takes into account roughness and is designed for a physically based pipeline.

2.3.4 Decals Made Easy

Decals are a popular method for adding extra details to the scene by adding layers of textured transparent geometry. However, this is often limited by the use of fixed function blending, meaning that the decal will be composited to the scene with either an addition or an interpolation.

The Framebuffer Fetch and Pixel local Storage mechanisms allow developers to do programmable blending, meaning complete freedom in how the decals will affect the G-buffer. More importantly, some G-buffer attachments are packed in ways that would prevent naive blending to work properly [Pranckevičius 10], but would be trivial with programmable blending.

It also makes environmental awareness (i.e., knowing what is behind the decal) completely free in terms of bandwidth because the G-buffer no longer needs to be flushed to VRAM before being sampled as a texture.

Finally, programmable blending makes it easy for developers to write selectively to specific outputs (e.g., a decal that only modifies the normals). Writing selectively to framebuffer attachments has always been possible, but with programmable blending, it is no longer defined in the renderer itself but in the shader. This makes it completely transparent to the application, which is convenient and slightly more efficient. With the Pixel Local Storage extension, this goes even further as there is no concept of framebuffer attachment, giving a very fine-grained control over what gets written to what.

2.4 Experiments

Tests were performed on a consumer-available device with a PowerVR series 6 GPU. An analysis of the application using the PowerVR SDK shows that the optimized renderers (Framebuffer Fetch and PLS) execute fewer rendering tasks—meaning that the G-buffer generation, lighting, and tonemapping stages are properly merged into one task. It also shows a clear reduction in memory bandwidth usage between the on-chip and the main memory: a 53% decrease in reads and a 54% decrease in writes (see Figure 2.8). All these optimizations result in a slightly lower frame time but in much lower power consumption, as shown in Figure 2.9. This means longer battery life on mobile devices.

2.5 Conclusion and Future Work

In this chapter, we presented an efficient physically based deferred shading renderer targeted at mobile devices. We have shown how the traditional deferred

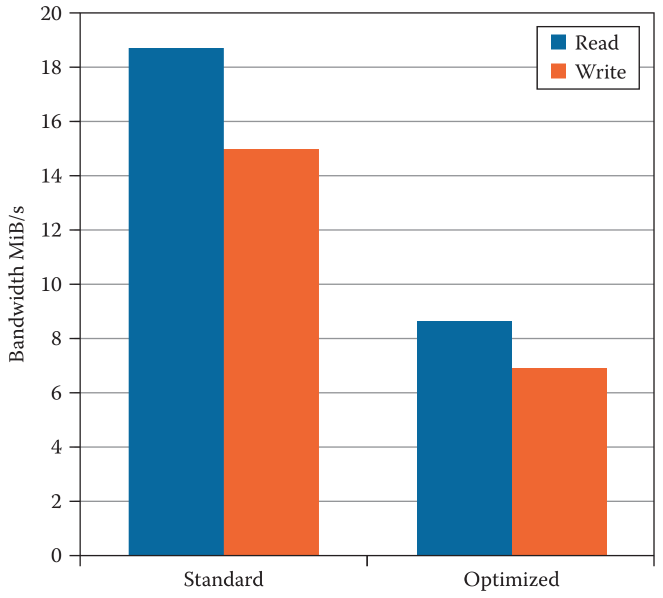

Figure 2.8. Bandwidth comparison.

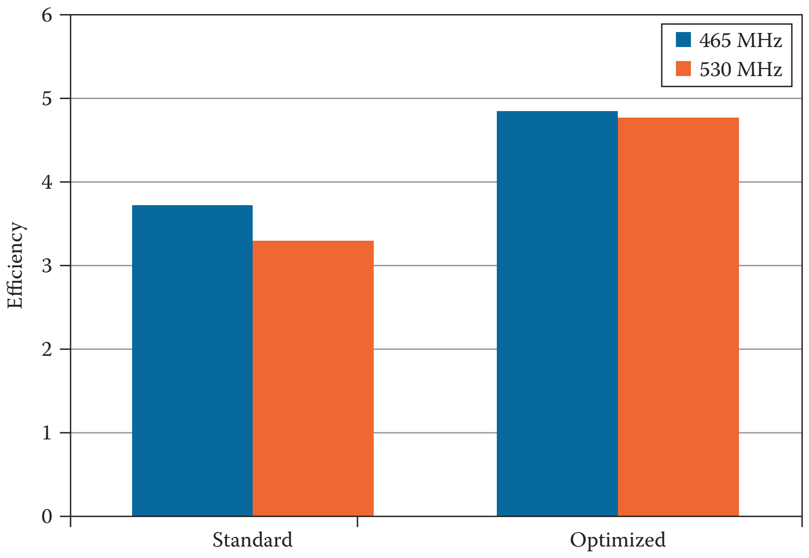

Figure 2.9. Efficiency in FPS (frames per second) per Watt of total system power.

```
layout(r32f) highp float posx;
layout(r32f) highp float posy;
layout(r32f) highp float posz;
layout(rg16f) highp vec2 lightAccumRG;
layout(rg16f) highp vec2 lightAccumB_normalsX;
layout(rg16f) highp vec2 normalsYZ;
layout(rg16f) highp vec2 roughness_metallicness;
layout(r11f_g11f_b10f) highp vec3 albedoOrReflectance;
```

Listing 2.7. Future metallicness workflow PLS G-buffer setup (256 bits).

```
layout(r32f) highp float posx;
layout(r32f) highp float posy;
layout(r32f) highp float posz;
layout(rg16f) highp vec2 lightAccumRG;
layout(rg16f) highp vec2 lightAccumB_roughness;
layout(r11f_g11f_b10f) highp vec3 normals;
layout(r11f_g11f_b10f) highp vec3 albedo;
layout(r11f_g11f_b10f) highp vec3 reflectance;
```

Listing 2.8. Future specular workflow PLS G-buffer setup (256 bits).

shading pipeline could be improved using OpenGL ES extensions such as Framebuffer Fetch and Pixel Local Storage to significantly reduce the amount of necessary bandwidth and therefore power and battery usage. We have proposed an efficient 128-bit G-buffer for PLS that allows state-of-the-art physically based shading, shown how to integrate high-quality static and dynamic lighting, and explained how decals could be made more complex at a lower performance cost.

In the next series of PowerVR GPUs, the available fast storage will increase from 128 to 256 bits per pixel. This gives application developers a wider variety of techniques and optimizations they can take advantage of.

Two examples of possible future G-buffer layouts using 256 bits per pixel are shown in Listings 2.7 and 2.8. Instead of storing depth, the world position is stored, which saves an expensive operation in the lighting stage. All the surface parameters are stored with higher precision, meaning they can all be stored in linear space. This saves an expensive conversion during both the geometry and lighting passes. Normals are also stored in higher precision for better quality. Finally, these G-buffer layouts are also very efficient in terms of ALU usage due to the fact that they are composed mostly of FP16 and FP32 registers, which do not require any packing and unpacking on PowerVR GPUs.

Such a large local storage could also be used for advanced effects such as Order Independent Transparency [Bjørge et al. 14], which could easily be integrated into our proposed deferred pipeline. Transparent objects would be rendered between the lighting and tonemapping stages, reusing the same PLS storage and overwriting it, keeping only the light accumulation data from the previous stage.

Bibliography

[Bjørge et al. 14] Marius Bjørge, Sam Martin, Sandeep Kakarlapudi, and Jan-Harald Fredriksen. "Efficient Rendering with Tile Local Storage." In *ACM SIGGRAPH 2014 Talks*, *SIGGRAPH '14*, pp. 51:1–51:1. New York: ACM, 2014.

[Kajiya 86] James T. Kajiya. "The Rendering Equation." *SIGGRAPH Comput. Graph.* 20:4 (1986), 143–150.

[Karis 09] Brian Karis. "RGBM Color Encoding." http://graphicrants.blogspot.co.uk/2009/04/rgbm-color-encoding.html, 2009.

[Khronos 13] Khronos. "Framebfufer Fetch." https://www.khronos.org/registry/gles/extensions/EXT/EXT_shader_framebuffer_fetch.txt, 2013.

[Khronos 14] Khronos. "Pixel Local Storage." https://www.khronos.org/registry/gles/extensions/EXT/EXT_shader_pixel_local_storage.txt, 2014.

[Lagarde 12] Sébastien Lagarde. "AMD Cubemapgen for Physically Based Rendering." https://seblagarde.wordpress.com/2012/06/10/amd-cubemapgen-for-physically-based-rendering, 2012.

[Pharr and Humphreys 04] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation.* San Francisco: Morgan Kaufmann Publishers, Inc., 2004.

[Pranckevičius 10] Aras Pranckevičius. "Compact Normal Storage for Small G-Buffers." http://aras-p.info/texts/CompactNormalStorage.html, 2010.

[Smith 14] Ashley Vaughan Smith. "Deferred Rendering Techniques on Mobile Devices." In *GPU Pro 5*, edited by Wolfgang Engel, pp. 263–272. Boca Raton, FL: A K Peters/CRC Press, 2014.

[Walter et al. 07] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. "Microfacet Models for Refraction Through Rough Surfaces." In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, *EGSR'07*, pp. 195–206. Aire-la-Ville, Switzerland: Eurographics Association, 2007.