WebGL Insights

Edited by **Patrick Cozzi**

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN A K PETERS BOOK

Section V
Rendering

Rendering is the generation of an image, given a scene description of geometry, materials, lights, and a camera. It is made up of two areas: finding visible surfaces and shading. Shading is the simulation of material and light to produce a color and is the primary topic of chapters in this section.

A common theme throughout this section is WebGL implementations of algorithms traditionally implemented in more feature-rich desktop graphics APIs—and how to get the best possible performance out of them. This can be something as simple as how to implement packing in GLSL, despite its lack of bitwise operators, to how to do volumetric rendering without 3D textures to simulating `EXT_shader_texture_lod` with octahedral environment mapping.

Deferred shading is a technique that decouples lighting from geometric complexity by performing lighting in a postprocessing step. This allows a large number of dynamic lights and simplifies engine design by only requiring one shader per material and per light type. Deferred shading became popular using desktop graphics APIs in 2008. We are now starting to see deferred shading with WebGL. In Chapter 15, "Deferred Shading in Luma," Nicholas Brancaccio discusses Luma, a physically based renderer for interior architectural spaces. He explores how to perform deferred rendering without multiple render targets through chroma subsampled lighting and creative packing of g-buffer parameters in a conventional floating-point render target.

Image-based lighting (IBL) uses processed image data, stored with high dynamic range (HDR), to properly represent a wide range of intensities to compute lighting. In Chapter 16, "HDR Image-Based Lighting on the Web," Jeff Russell explains how to implement IBL within the limitations of WebGL. This includes memory, performance, and visual-quality trade-offs of HDR decoding in GLSL and HDR transmission from the server to the client. For environment maps, octahedral environment mapping is used to store a cube map in a 2D texture to allow mipmap level of detail (LOD) selection without `EXT_shader_texture_lod`.

Many data sets, especially those in biomedical imaging, are naturally represented as 3D volumes. In Chapter 17, "Real-Time Volumetric Lighting for WebGL," Muhammad Mobeen Movania and Feng Lin discuss how volume rendering with half-angle slicing can be implemented within the constraints of WebGL using `OES_texture_float`, `OES_texture_float_linear`, and `WEBGL_draw_buffers`. This chapter includes an overview of the theory, a walk-through of the JavaScript and GLSL code, a detailed performance analysis of direct volume rendering and half-angle slicing, and CPU versus GPU approaches.

Terrain rendering is a popular area since it has so many real-world and game-uses cases. Terrain presents geometric LOD challenges and opportunities for clever shading. In Chapter 18, "Terrain Geometry—LOD Adapting Concentric Rings," Florian Bösch presents a terrain rendering approach that is well suited to WebGL since it is very light on the CPU, pushing most of the work to the GPU. Topics include LOD and geomorphing with nested grids, and shading by combining a derivative map and detail mapping. Check out the online demo as you read.

16

HDR Image-Based Lighting on the Web

Jeff Russell

16.1 Introduction

Image-based lighting (IBL) is a family of techniques for illuminating surfaces using processed image data. Such images can be prerendered, captured on the fly, or obtained through photography. Regardless of their source, the use of these images for lighting has several advantages for render quality, not least of which is the inclusion of both direct and indirect illumination from a surrounding scene.

Image-based lighting is not new and has in fact become a prevalent rendering technique in games and visual simulations today. As we turn our attention to graphics on the web, and WebGL in particular, it becomes clear that most if not all IBL techniques should be feasible with some modification on this new platform.

This chapter will focus on the aspects of a WebGL implementation that may differ from those of other platforms. Concerns specific to compatibility and performance will be addressed in order to better reflect the broad hardware demographic of the web today, with a special emphasis on mobile devices. This assumes that the reader has a basic familiarity with image-based lighting; for a broader introduction to the topic, see Debevec [Debevec 02].

16.2 High Dynamic Range Encoding

Image-based lighting requires use of high dynamic range (HDR) texture data to properly represent the full range of luminosities present in a given image. Such values often span ranges far outside those of typical display technology, requiring both increased range and precision. Several encodings exist to address these needs; however, at the time of this writing few are well suited for web deployment.

Many WebGL implementations expose extensions such as `OES_texture_float` and `OES_texture_half_float`, providing support for 32-bit and 16-bit floating point values, respectively. At first glance, these formats would seem to be an ideal means for storing HDR data, and indeed they are in large part made available for exactly that purpose. There are, however, several drawbacks to their use. Because they are optional extensions, many implementations do not provide them, and when others do, they often do not supply functionality for linear filtering. On top of this, these formats double or quadruple memory and bandwidth requirements compared to a typical 8-bit format, which can create performance problems on mobile devices and increase page load times.

Many solutions exist for packing HDR data into smaller memory footprints. The Red/Green/Blue/Exponent (RGBE) encoding stores a shared exponent byte in the fourth color channel of a 32-bit image [Ward 97]. This allows for representation of a wide range of luminosities in a compact form, though it does require additional instructions in a shader to unpack and apply the exponent. A similar encoding that uses the CIE LUV color space is "LogLUV" [Ward 98].

A simpler option available to us involves storing a separate scale value in the alpha channel, and decoding these Red/Green/Blue/Multiplier (RGBM) values through simple multiplication in the shader [Karis 09]. This has several advantages. First, like RGBE and LogLUV, it does not require use of any extensions, which makes it workable on all platforms that support the base WebGL specification. Second, it uses much less memory and hence bandwidth during render time than most floating point formats. Finally, it is very simple to encode and decode, requiring fairly little ALU workload.

Listing 16.1 Basic RGBM decoding.

```
mediump vec3 decodeRGBM(mediump vec4 rgbm) {
  return rgbm.rgb * rgbm.a * maxRange;
}
```

Linear interpolation of such RGBM values during texture sampling is technically incorrect, and a naive decoding, as shown in Listing 16.1, will often produce banding artifacts. Filtering precision can be a significant factor in this; some devices perform texture filtering at 8-bit precision, which can worsen banding significantly. The performance advantage of using built-in texture filtering is significant enough that we need to explore ways of minimizing its side effects on RGBM textures.

One method of correcting banding is reducing the range of our "M" multiplier. An 8-bit value could theoretically be interpreted as a linear multiplier on a range as wide as [0, 255]. While this is feasible on some devices, it tends to stretch the limits of texture filtering

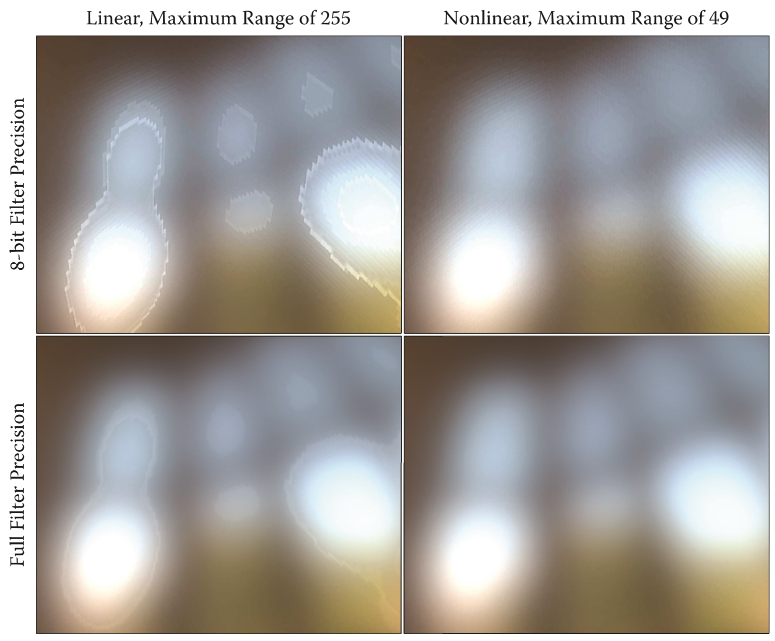

Figure 16.1

Comparison of texture filtering results with RGBM encoding. Linear RGBM with a maximum range of 255 (left column) compared with nonlinear RGBM with a maximum range of 49 (right column). The top row has been rendered on a mobile GPU with 8-bit filter precision.

precision on others. We find that reducing the mapped range of M to [0,7] greatly reduces the appearance of filtering artifacts.

We additionally adopt a nonlinear transformation akin to a gamma curve in order to reduce color banding in the color components. The rationale is analogous to that behind gamma compression, though in our case it will have the additional benefit of using nonlinearity to greatly expand dynamic range. sRGB color space conversion suits this purpose well and is available on many GPUs; however, WebGL does not expose this, so we instead adopt a simpler exponential curve. By applying an exponent of two, we both increase the upper limit on our range from 7 to 49 and provide better precision for values closer to zero, where banding is most apparent (Figure 16.1).

Listing 16.2 RGBM encoding and decoding, with nonlinearity and range reduction to reduce filtering artifacts.

```
highp vec4 encodeRGBM(highp vec3 rgb) {
  highp vec4 r;
  r.xyz = (1.0/7.0) * sqrt(rgb);
  r.a = max(max(r.x, r.y), r.z);
  r.a = clamp(r.a, 1.0/255.0, 1.0);
  r.a = ceil(r.a * 255.0)/255.0;
  r.xyz/= r.a;
  return r;
}

mediump vec3 decodeRGBM(mediump vec4 rgbm) {
  mediump vec3 r = rgbm.rgb * (7.0 * rgbm.a);
  return r * r;
}
```

As is evident in Listing 16.2, encoding RGBM values require several more instructions than decoding, although both operations are fairly brief. In many cases this is not a problem, as IBL image data are prepared infrequently during runtime, if at all. However, in situations where RGBM encoding is to be used for render targets, the use of floating-point texture formats may be preferable when they are available, particularly if blending is required.

The relatively limited range of RGBM values is another possible drawback. For most scenarios we have found that preconvolved, pre-exposed light data fit well within this modest span. Preconvolved images, such as those used for diffuse and specular lighting, tend to fit well, as any strong highlights in the original image tend to "spread out" their energy as a result of convolution. The use of tone mapping filters may in some cases also hide a lack of range. However, applications wishing to faithfully represent very high luminance values may need to switch to a higher dynamic range format such as RGBE or floating point textures.

RGBM texture data are well suited for compact transmission over the web. In theory, the 32-bit color values map well into the portable network graphics (PNG) image format for which all browsers provide support. PNG images use lossless compression, which is a necessity for RGBM data; use of JPEG or other lossy formats results in severe reconstruction errors.

In practice, several popular browsers have been found to premultiply alpha values after PNG decoding, which introduces pronounced color banding in RGBM color data (this occurs regardless of the "`UNPACK_PREMULTIPLY_ALPHA_WEBGL`" pixel storage setting). Our solution is to compress the RGBM data ourselves. If we store each color plane separately, readily available content encodings such as gzip can provide compression ratios comparable to PNG. This requires client-side decompression and re-interleaving of color channel data, but is well worth the savings in transmission times. Any convenient image container format may be used for transmission, including DirectDraw Surface (DDS) or custom layouts.

16.3 Environment Mapping

Key to any image-based lighting system is the layout of the environment images themselves. For any image to be usable for lighting, it must cover the entire sphere of possible directions. There are, of course, many layouts with this property, but far and away the most popular is cube mapping. WebGL has good support for cube maps in the base specification, supporting all texture formats, filtering, and more.

It is also important for an IBL system to provide the ability for texture artists to specify, usually through a grayscale mask, differing roughness values. This gives control over the apparent "shininess" of a surface, affecting the size of specular highlights and reflections. This is often known as "gloss mapping" or "roughness mapping" and is a nearly indispensable tool for creating believable surfaces.

Roughness mapping typically interacts with cube map reflections through the manual selection by the shader of different mipmap levels according to roughness values. Environment cube maps are specially prepared such that they contain different convolutions of the base image in each mipmap level—typically becoming "blurrier" as the mip dimensions reduce. In this way the shader can easily control the appearance of environment reflections in much the way it would for analytical light sources (Figure 16.2). Third-party tools for preparing convolutions of this sort are readily available; see www.hdrshop.com, www.knaldtech.com/lys/ or code.google.com/p/cubemapgen/.

Herein lies a difficulty for implementation in WebGL, as mipmap level of detail (LOD) selection is not supported in the core specification. The `EXT_shader_texture_lod` extension seeks to address this shortcoming; however, at the time of this writing it is not widely supported. In order to make universal use of roughness mapping possible in WebGL today, an alternative means of storage is needed for preconvolved environment maps.

By packing multiple maps into a two-dimensional texture atlas, we can access them with simple texture coordinate transformations, requiring no mipmap selection. However, since cube maps are not easily packed in this way, this again poses our earlier problem of mapping the environment sphere onto a flat surface. The puzzle is as old as map making, and we come to it with the added restriction of requiring a fast, simple formulation with minimal distortion.

Spherical projections making direct use of latitude and longitude provide conceptually straightforward mappings; however, they come with the significant drawback of having polar discontinuities. This results in a highly uneven distribution of texels at poles,

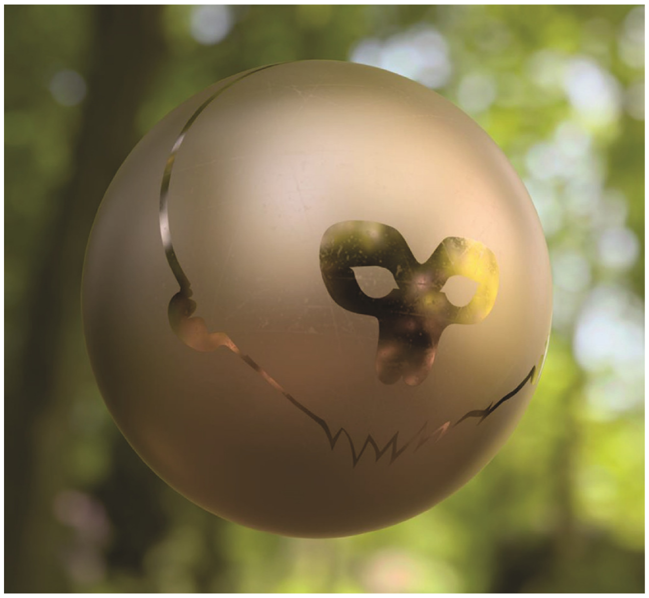

Figure 16.2

Roughness mapping used in conjunction with cube-mapped environment reflections.

as well as filtering artifacts, which can be difficult to fully mask. Dual paraboloid mapping [Heidrich 98] avoids this polar distortion, having only modest discontinuities. It does, however, provoke somewhat uneven texel distribution and does not make efficient use of texture space, as much goes unused around its circular borders.

Octahedral environment mapping [Engelhardt 08] provides a good compromise between simplicity, distortion, and speed, with performance and appearance very similar to cube mapping. Additionally, its two-dimensional surface lies in a perfect square, which means no space goes unused in an atlas layout. The technique maps three-dimensional vectors onto the surface of an octahedron, or “double pyramid” (Figure 16.3).

The three-dimensional surface of the unit octahedron is defined by $|x| + |y| + |z| = 1$. From this relation, unnormalized vectors can be quickly projected into the octahedron’s two-dimensional surface. Points with positive Y values are simply “flattened” onto the XZ-plane, and those in the opposite hemisphere are unfolded to fill the corners (see [Engelhardt 08] for a more detailed description). Listing 16.3 displays code to perform this mapping.

Listing 16.3 Octahedral projection from unnormalized 3D vector to octahedral UV coordinates.

```
mediump vec2 octahedralProjection(mediump vec3 dir) {
  dir/= dot(vec3(1.0), abs(dir));
  mediump vec2 rev = abs(dir.zx) - vec2(1.0,1.0);
  mediump vec2 neg = vec2(dir.x < 0.0 ? rev.x : -rev.x,
                          dir.z < 0.0 ? rev.y : -rev.y);
  mediump vec2 uv = dir.y < 0.0 ? neg : dir.xz;
  return 0.5*uv + vec2(0.5,0.5);
}
```

With this environment projection in hand, we are now able to build a full atlas of as many convolutions as we wish. An atlas comprising a vertical column of a few images makes for easy construction through simple memory concatenation and allows simple shader logic for selecting convolutions. A shader making use of such an atlas is also free to

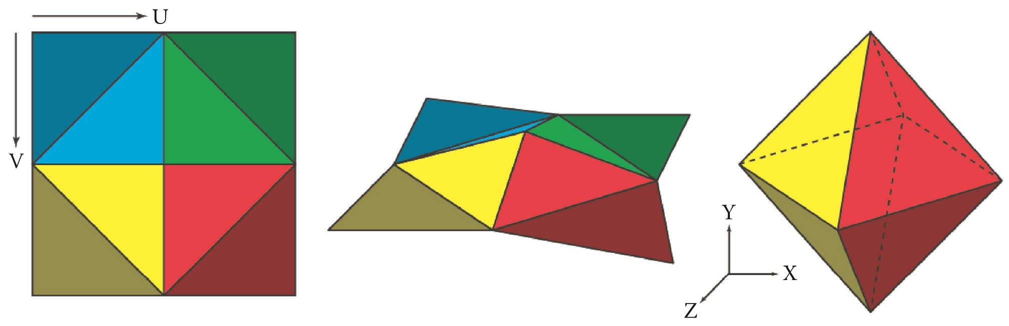

Figure 16.3

Octahedral environment map layout in 2D coordinates (left), and folded into three dimensions (middle and right).

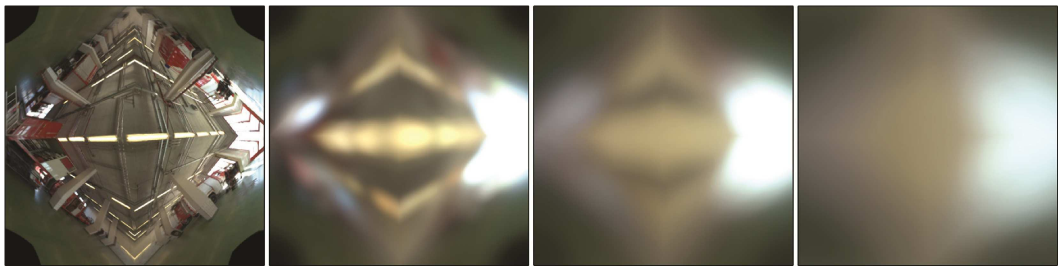

Figure 16.4

Example of octahedral environment maps, convolved for the Phong BRDF with varying levels of surface roughness.

take samples from different convolutions and blend between them, achieving a smoother transition between roughnesses.

Octahedral maps do have some difficulties with texture filtering. Specifically, they do not tile: Their edges will filter improperly if the texture wrap parameter is set to "`REPEAT`." Use of octahedral maps in texture atlases poses similar problems, causing neighboring maps to filter with one another at map boundaries. This is best resolved by adding a single pixel of padding around each map in the atlas, and altering the shader code to adjust the sample coordinates accordingly.

Octahedral environment maps (Figure 16.4) incur a small performance penalty in exchange for the added compatibility they provide. We have measured a difference of roughly six additional instructions as compared to traditional cube mapping, though this varies with hardware characteristics and compilers. This cost is incurred for each unique direction sampled, but can be amortized for multiple samples with the same direction vector (as in the case of gloss mapping).

Use of an octahedral layout directly for render targets can prove challenging. Engelhardt [Engelhardt 08] provides a brief discussion of a method based on splitting rendered triangles across the eight faces to ensure proper perspective. In practice we have found it preferable to simply remap a cube render target into the octahedral layout as a postprocess. This can be quickly performed by rendering flat the octahedral geometry itself, with cube map texture coordinates assigned to each vertex.

Diffuse IBL data sets generally do not have need of multiple convolutions, freeing them of the restrictions that motivate the use of octahedral maps for reflections. Diffuse convolutions can therefore still make use of cube maps or any other convenient mapping. Such convolutions are also amenable to representation with the spherical harmonic basis functions [Ramamoorthi 01], which provide an extremely compact reproduction of low-frequency image data. For diffuse lighting we have found the spherical harmonic representation preferable to images, due to its flexibility, simplicity, and small memory footprint.

16.4 Conclusion

With the modifications discussed here, WebGL is well suited today for the use of image-based lighting across the panoply of devices and platforms that is the web. Through careful selection of HDR texture encodings, many of the drawbacks of floating point formats can be eliminated while retaining a useful dynamic range; by adopting octahedral environment mapping, we can create a fully featured IBL renderer without relying on device-specific extensions.

It is our hope that as the WebGL standard progresses, some of the limitations outlined in this chapter will be overcome in the same ways they have been in other environments, eliding the need for much special treatment. Image-based lighting is likely to remain a relevant rendering technique for years to come, and the web, being no exception, will see the benefit of solid implementations.

Bibliography

[Debevec 02] Paul Debevec. “Image-Based Lighting.” USC Institute for Creative Technologies, http://ict.usc.edu/pubs/Image-Based%20Lighting.pdf, 2002.

[Engelhardt 08] Thomas Engelhardt and Carsten Dachsbacher. “Octahedron Environment Maps.” Vision, Modeling and Visualization, http://www.vis.uni-stuttgart.de/~dachsbcn/download/vmvOctaMaps.pdf, 2008.

[Heidrich 98] Wolfgang Heidrich and Hans-Peter Seidel. “View-Independent Environment Maps.” Eurographics Workshop on Graphics Hardware. http://www.cs.ubc.ca/~heidrich/Papers/GH.98.pdf, 1998.

[Karis 09] Brian Karis. “RGBM Color Encoding.” http://graphicrants.blogspot.com/2009/04/rgbm-color-encoding.html, 2009.

[Ramamoorthi 01] Ravi Ramamoorthi and Pat Hanrahan. “An Efficient Representation for Irradiance Environment Maps.” http://graphics.stanford.edu/papers/envmap/envmap.pdf, 2001.

[Ward 97] Gregory Ward Larson. “Radiance File Formats.” http://radsite.lbl.gov/radiance/refer/filefmts.pdf, 1997.

[Ward 98] Gregory Ward Larson. “The LogLuv Encoding for Full Gamut, High Dynamic Range Images.” *Journal of Graphics Tools*, 3(1):15–31, 1998.