SPHERICAL HARMONICS ALGORITHM FOR DYNAMIC LIGHT IN REAL TIME

Lien Muguercia Torres

Facultad 5 Entornos Virtuales
Universidad de las Ciencias Informáticas
Ciudad Habana, Cuba
lmuguercia@uci.cu

ABSTRACT

Traditionally, efficient shading and light mapping has been an obstacle in computer graphics. Reaching desired realism levels requires a high source consume and sometimes it doesn't accomplish existing expectative.

The intention of this work is adding more realism to virtual scenes through dynamic light using GPU.

Starting off of the study of many algorithms, it proposes many algorithms to work whit light through shader (program used to determine the final surface properties of an object or image. This can include arbitrarily complex descriptions of light absorption and diffusion, texture mapping, reflection and refraction, shadowing, surface displacement and post-processing effects)but we'll propound the Spherical Harmonics algorithms and implemented in OpenGL Shading Language (GLSL).

Keywords: GPU, shader, post-processing effects.

INTRODUCTION

The recent trend in graphics hardware has been to replace fixed functionality with programmability in areas that have grown exceedingly complex.

Two such areas are vertex processing and fragment processing. Vertex processing involves the operations that occur at each vertex, most notably transformation and lighting. Fragments are per-pixel data structures that are created by the rasterization of graphics primitives.

A fragment contains all the data necessary to update a single location in the frame buffer. Fragment processing consists of the operations that occur on a per-fragment basis, most notably reading from texture memory and applying the texture value(s) at each fragment.

With the OpenGL Shading Language, the fixed functionality stages for vertex processing and fragment processing have been augmented with programmable stages that can do everything the fixed functionality stages can doand a whole lot more. The OpenGL Shading Language allows application programmers to express the processing that occurs at those programmable points of the OpenGL pipeline.

The OpenGL Shading Language code that is intended for execution on one of the OpenGL programmable processors is called a shader. The term OpenGl shader is sometimes used to differentiate a shader written in the OpenGL Shading Language from a shader written in another shading language such as RenderMan.

In this article I explore how the OpenGL Shading Language can help us implement such models so that they can execute at interactive rates on programmable graphics hardware. We look at some lighting models that provide more flexibility and give more realistic results than those built into OpenGL's fixed functionality rendering pipeline. Much has been written on the topic of lighting in computer graphics. We only examine a few methods an propound the implementation of one . Hopefully, you'll be inspired to try implementing some others on your own.

BASIC

Why shader?

By exposing support for traditional rendering mechanisms, OpenGL has evolved to serve the needs of a fairly broad set of applications. If your particular application was well served by the traditional rendering model presented by OpenGL, you may never need to write shaders. But if you have ever been frustrated because OpenGL did not allow you to define area lights, or because lighting calculations are performed per-vertex rather than per-fragment or, if you have run into any of the many limitations of the traditional OpenGL rendering model, you may need to write your own OpenGL shader.

With each new generation of graphics hardware, more complex rendering techniques can be implemented as

OpenGL shaders and can be used in real-time rendering applications. Here's a brief list of what's possible with OpenGL shaders:

- Increasingly realistic lighting effects area lights, soft shadows
- Advanced rendering effects global illumination, ray-tracing,
- Animation effects key frame interpolation, particle systems, procedurally defined motion
- User programmable anti-aliasing methods
- General computation sorting, mathematical modelling, fluid dynamics, and so on

Many of these techniques have been available before now only through software implementations. If they were at all possible through OpenGL, they were possible only in a limited way. The fact that these techniques can now be implemented with hardware acceleration provided by dedicated graphics hardware means that rendering performance can be increased dramatically and at the same time the CPU can be off-loaded so that it can perform other tasks.

LIGTHING

In the real world, we see things because they reflect light from a light source or because they are light sources themselves. In computer graphics, just as in real life, we won't be able to see an object unless it is illuminated or emits light. To generate more realistic images, we need to have more realistic models for illumination, shadows, and reflection than those we've discussed so far.

Hemisphere Lighting

We know at the fixed functionality lighting model built into OpenGL and developed shader code to mimic the fixed functionality behavior. However, this model has a number of flaws, and these flaws become more apparent as we strive for more realistic rendering effects. One problem is that objects in a scene do not typically receive all their illumination from a small number of specific light sources. Interreflections between objects often have noticeable and important contributions to objects in the scene. The traditional computer graphics illumination model attempts to account for this phenomena through an ambient light term. However, this ambient light term is usually applied equally across an object or an entire scene. The result is a flat and unrealistic look for areas of the scene that are not affected by direct illumination.

Another problem with the traditional illumination model is that light sources in real scenes are not point lights or even spotlightsthey are area lights. Consider the indirect light coming in from the window and illuminating the floor and the long fluorescent light bulbs behind a rectangular translucent panel. For an even more common case, consider the illumination outdoors on a cloudy day. In this case, the entire visible hemisphere is acting like an area light source. In several presentations and tutorials, Chas Boyd, Dan Baker, and Philip Taylor of Microsoft described this situation as Hemisphere Lighting and discussed how to implement it in DirectX. Let's look at how we might create an OpenGL shader to simulate this type of lighting environment.

The idea behind hemisphere lighting is that we model the illumination as two hemispheres. The upper hemisphere represents the sky, and the lower hemisphere represents the ground. A location on an object with a surface normal that points straight up gets all of its illumination from the upper hemisphere, and a location with a surface normal pointing straight down gets all of its illumination from the lower hemisphere (see Figure 1). By picking appropriate colors for the two hemispheres, we can make the sphere look as though locations with normals pointing up are illuminated and those with surface normals pointing down are in shadow.

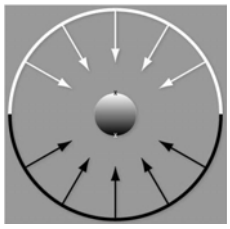

Figure 1: A sphere illuminated using the hemisphere lighting model

To compute the illumination at any point on the surface, we must compute the integral of the illumination received at that point:

$Color = a \cdot SkyColor + (1 - a) \cdot GroundColor$
where
$a = 1.0 - (0.5 \cdot \sin(q))$ for $q < 90°$
$a = 0.5 \cdot \sin(q)$ for $q > 90°$
q = angle between surface normal and north pole direction

But we can actually calculate a in another way that is simpler but roughly equivalent:

$a = 0.5 + (0.5 \cdot \cos(q))$

This approach eliminates the need for a conditional. Furthermore, we can easily compute the cosine of the angle between two unit vectors by taking the dot product of the two vectors. This is an example of what Jim Blinn likes to call "the ancient Chinese art of chi ting." In computer graphics, if it looks good enough, it is good enough. It doesn't really matter whether your calculations are physically correct or a colossal cheat.

One of the issues with this model is that it doesn't account for self-occlusion. Regions that should really be in shadow because of the geometry of the model appear too bright.

Image-Based Lighting

If we're trying to achieve realistic lighting in a computer graphics scene, why not just use an environment map for the lighting? This approach to illumination is called Image-Based Lighting; it has been popularized in recent years by researcher Paul Debevec at the University of Southern California. Churches and auditoriums may have dozens of light sources on the ceiling. Rooms with many windows also have complex lighting environments. It is often easier and much more efficient to sample the lighting in such environments and store the results in one or more environment maps than it is to simulate numerous individual light sources.

The steps involved in image-based lighting are:

- Use a Light Probe (e.g., a reflective sphere) to capture (e.g., photograph) the illumination that occurs in a real-world scene. The captured omni-directional, high-dynamic range image is called a Light Probe Image.
- Use the light probe image to create a representation of the environment (e.g., an environment map).
- Place the synthetic objects to be rendered inside the environment.
- Render the synthetic objects by using the representation of the environment created in step 2.

On his Web site (http://www.debevec.org/), Debevec offers a number of useful things to developers. For one, he has made available a number of images that can be used as high-quality environment maps to provide realistic lighting in a scene. These images are high dynamic range (HDR) images that represent each color component with a 32-bit floating-point value. Such images can represent a much greater range of intensity values than can 8-bit-per-component images. For another, he makes available a tool called HDRShop that manipulates and transforms these environment maps. Through links to his various publications and tutorials, he also provides step-by-step instructions on creating your own environment maps and using them to add realistic lighting effects to computer graphics scenes.

Following Debevec's guidance, I purchased a 2-inch chrome steel ball from McMaster-Carr Supply Company (http://www.mcmaster.com). We used this ball to capture a light probe image from the center of the square outside our office building in downtown Fort Collins, Colorado. We then used HDRShop to create a lat-long environment map and a cube map of the same scene. The cube map and lat-long map can be used to perform environment mapping. That shader simulated a surface with an underlying base color and diffuse reflection characteristics that was covered by a transparent mirror-like layer that reflected the environment flawlessly.

We can simulate other types of objects if we modify the environment maps before they are used. A point on the surface that reflects light in a diffuse fashion reflects light from all the light sources that are in the hemisphere in the direction of the surface normal at that point. We can't really afford to access the environment map a large number of times in our shader. What we can do instead is similar to what we discussed for hemisphere lighting. Starting from our light probe image, we can construct an environment map for diffuse lighting. Each texel in this environment map will contain the weighted average (i.e., the convolution) of other texels in the visible hemisphere as defined by the surface normal that would be used to access that texel in the environment.

Again, HDRShop has exactly what we need. We can use HDRShop to create a lat-long image from our original light probe image. We can then use a command built into HDRShop that performs the necessary convolution. This operation can be time consuming, because at each texel in the image, the contributions from half of the other texels in the image must be considered. Luckily, we don't need a very large image for this purpose. The effect is essentially the same as creating a very blurry image of the original light probe image. Since there is no high frequency content in the computed image, a cube map with faces that are 64 x 64 or 128 x 128 works just fine.

A single texture access into this diffuse environment map provides us with the value needed for our diffuse reflection calculation. What about the specular contribution? A surface that is very shiny will reflect the illumination from a light source just like a mirror. A single point on the surface reflects a single point in the environment. For surfaces that are rougher, the highlight defocuses and spreads out. In this case, a single point on the surface reflects several points in the environment, though not the whole visible hemisphere like a diffuse surface. HDRShop lets us blur an environment map by providing a Phong exponenta degree of shininess. A value of 1.0 convolves the environment map to simulate diffuse reflection, and a value of 50 or more convolves the environment map to simulate a somewhat shiny surface.

The shaders that implement these concepts end up being quite simple and quite fast. In the vertex shader, all that is needed is to compute the reflection direction at each vertex. This value and the surface normal are sent to the fragment shader as varying variables. They are interpolated

across each polygon, and the interpolated values are used in the fragment shader to access the two environment maps in order to obtain the diffuse and the specular components. The values obtained from the environment maps are combined with the object's base color to arrive at the final color for the fragment.

The ÜberLight Shader

We've discussed lighting algorithms that simulate the effect of global illumination for more realistic lighting effects. Traditional point, directional, and spotlights can be used in conjunction with these global illumination effects. However, the traditional light sources leave a lot to be desired in terms of their flexibility and ease of use.

Ronen Barzel of Pixar Animation Studios wrote a paper in 1997 that described a much more versatile lighting model specifically tailored for the creation of computer-generated films. This lighting model has so many features and controls compared to the traditional graphics hardware light source types that its RenderMan implementation became known as the "überlight" shader (i.e., the lighting shader that has everything in it except the proverbial kitchen sink). Larry Gritz wrote a public domain version of this shader that was published in Advanced RenderMan: Creating CGI for Motion Pictures, which he coauthored with Tony Apodaca. A Cg version of this shader was published by Fabio Pellacini and Kiril Vidimice of Pixar in the book GPU Gems, edited by Randima Fernando. The full-blown überlight shader has been used successfully in a variety of computer-generated films, including Toy Story, Monsters, Inc., and Finding Nemo. Because of the proven usefulness of the überlight shader, this section looks at how to implement its essential features in the OpenGL Shading Language.

In movies, lighting helps to tell the story in several different ways. Sharon Calahan gives a good overview of this process in the book Advanced RenderMan: Creating CGI for Motion Pictures. This description includes five important fundamentals of good lighting design that were derived from the book Matters of Light & Depth by Ross Lowell:

- Directing the viewer's eye
- Creating depth
- Conveying time of day and season
- Enhancing mood, atmosphere, and drama
- Revealing character personality and situation

Because of the importance of lighting to the final product, movies have dedicated lighting designers. To light computer graphics scenes, lighting designers must have an intuitive and versatile lighting model to use.

For the best results in lighting a scene, it is crucial to make proper decisions about the shape and placement of the lights. For the überlight lighting model, lights are assigned a position in world coordinates. The überlight shader uses a pair of superellipses to determine the shape of the light. A superellipse is a function that varies its shape from an ellipse to a rectangle, based on the value of a roundness parameter. The superellipse function is defined as

$$\left(\frac{x}{a}\right)^{\frac{2}{d}} + \left(\frac{y}{b}\right)^{\frac{2}{d}} = 1$$

As the value for d nears 0, this function becomes the equation for a rectangle, and when d is equal to 1, the function becomes the equation for an ellipse. Values in between create shapes in between a rectangle and an ellipse, and these shapes are also useful for lighting. This is referred to in the shader as barn shaping since devices used in the theater for shaping light beams are referred to as barn doors.

It is also desirable to have a soft edge to the light, in other words, a gradual drop-off from full intensity to zero intensity. We accomplish this by defining a pair of nested superellipses. Inside the innermost superellipse, the light has full intensity. Outside the outermost superellipse, the light has zero intensity. In between, we can apply a gradual transition by using the smoothstep function.

Two more controls that add to the versatility of this lighting model are the near and far distance parameters, also known as the cuton and cutoff values. These define the region of the beam that actually provides illumination (see Figure 2). Again, smooth transition zones are desired so that the lighting designer can control the transition. Of course, this particular control has no real-world analogy, but it has proved to be useful for softening the lighting in a scene and preventing the light from reaching areas where no light is desired.

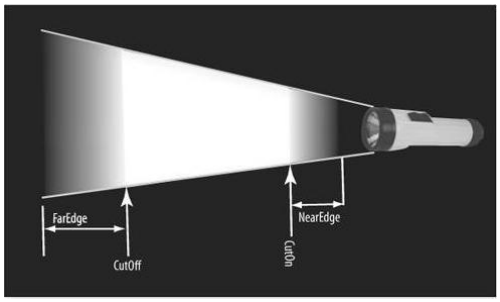

Figure 2: Effects of the near and far distance parameters for the überlight shader

The überlight shader as described by Barzel and Gritz actually has several additional features. It can support multiple lights, but our example shader showed just one for simplicity. The key parameters can be defined as arrays,

and a loop can be executed to perform the necessary computations for each light source. In the following chapter, we show how to add shadows to this shader.

Lighting with Spherical Harmonics

In 2001, Ravi Ramamoorthi and Pat Hanrahan presented a method that uses spherical harmonics for computing the diffuse lighting term. This method reproduces accurate diffuse reflection, based on the content of a light probe image, without accessing the light probe image at runtime.

The light probe image is pre-processed to produce coefficients that are used in a mathematical representation of the image at runtime. The mathematics behind this approach is beyond the scope of this book (see the references at the end of this chapter if you want all the details). Instead, we lay the necessary groundwork for this shader by describing the underlying mathematics in an intuitive fashion. The result is remarkably simple, accurate, and realistic, and it can easily be codified in an OpenGL shader. This technique has already been used successfully to provide real-time illumination for games and has applications in computer vision and other areas as well.

Spherical harmonics provides a frequency space representation of an image over a sphere. It is analogous to the Fourier transform on the line or circle. This representation of the image is continuous and rotationally invariant. Using this representation for a light probe image, Ramamoorthi and Hanrahan showed that you could accurately reproduce the diffuse reflection from a surface with just nine spherical harmonic basis functions. These nine spherical harmonics are obtained with constant, linear, and quadratic polynomials of the normalized surface normal.

Intuitively, we can see that it is plausible to accurately simulate the diffuse reflection with a small number of basis functions in frequency space since diffuse reflection varies slowly across a surface. With just nine terms used, the average error over all surface orientations is less than 3% for any physical input lighting distribution. With Debevec's light probe images, the average error was shown to be less than 1% and the maximum error for any pixel was less than 5%.

Each spherical harmonic basis function has a coefficient that depends on the light probe image being used. The coefficients are different for each colour channel, so you can think of each coefficient as an RGB value. A preprocessing step is required to compute the nine RGB coefficients for the light probe image to be used. Ramamoorthi makes the code for this pre-processing step available for free on his Web site. I used this program to compute the coefficients for all the light probe images in Debevec's light probe gallery as well as the Old Town Square light probe image and summarized the results in Figure 3.

Coefficient	Old Town Square	Grace Cathedral	Eucalyptus Grove	St. Peter's Basilica	Uffizi Gallery
L00	.87 .88 .86	.79 .44 .54	.38 .43 .45	.36 .26 .23	.32 .31 .3
L1m1	.18 .25 .31	.39 .35 .60	.29 .36 .41	.18 .14 .13	.37 .37 .4
L10	.03 .04 .04	-.34 -.18 -.27	.04 .03 .01	-.02 -.01 .00	.00 .00 .0
L11	-.00 -.03 -.05	-.29 -.06 .01	-.10 -.10 -.09	.03 .02 .00	-.01 -.01 -.0
L2m2	-.12 -.12 -.12	-.11 -.05 -.12	-.06 -.06 -.04	.02 .01 .00	-.02 -.02 -.0
L2m1	.00. .00 .01	-.26 -.22 -.47	.01 -.01 -.05	-.05 -.03 -.01	-.01 -.01 -.0
L20	-.03 -.02 -.02	-.16 -.09 -.15	-.09 -.13 -.15	-.09 -.08 -.07	-.28 -.28 -.3
L21	-.08 -.09 -.09	.56 .21 .14	-.06 -.05 -.04	.01 .00 .00	.00 .00 .0
L22	-.16 -.19 -.22	.21 -.05 -.30	.02 .00 -.05	-.08 -.03 .00	-.24 -.24 -.2

Coefficient	Galileo's Tomb	Vine Street Kitchen	Breezeway	Campus Sunset	Funston Beach Sunset
L00	1.04 .76 .71	.64 .67 .73	.32 .36 .38	.79 .94 .98	.68 .69 .7
L1m1	.44 .34 .34	.28 .32 .33	.37 .41 .45	.44 .56 .70	.32 .37 .4
L10	-.22 -.18 -.17	.42 .60 .77	-.01 -.01 -.01	-.10 -.18 -.27	-.17 -.17 -.1
L11	.71 .54 .56	-.05 -.04 -.02	-.10 -.12 -.12	.45 .38 .20	-.45 -.42 -.3
L2m2	.64 .50 .52	-.10 -.08 -.05	-.13 -.15 -.17	.18 .14 .05	-.17 -.17 -.1
L2m1	-.12 -.09 -.08	.25 .39 .53	-.01 -.02 .02	-.14 -.22 -.31	-.08 -.09 -.1
L20	-.37 -.28 -.29	.38 .54 .71	-.07 -.08 -.09	-.39 -.40 -.36	-.03 -.02 -.0
L21	-.17 -.13 -.13	.06 .01 -.02	.02 .03 0.03	.09 .07 .04	.16 .14 .1
L22	.55 .42 .42	-.03 -.02 -.03	-.29 -.32 -.36	.67 .67 .52	.37 .31 .2

Figure 3: Spherical harmonic coefficients for light probe images

The equation for diffuse reflection using spherical harmonics is

$$\text{Diffuse} = c_1 L_{22} (x^2 - y^2) + c_3 L_{20} z^2 + c_4 L_{20} - c_5 L_{20} + 2c_1 (L_{2-2} xy + L_{21} xz + L_{2-1} yz) + 2c_2 (L_{11} x + L_{1-1} y + L_{10} z$$

(1)

The constants c1c5 result from the derivation of this formula and are shown in the vertex shader code in Listing 1. The L coefficients are the nine basis function coefficients computed for a specific light probe image in the preprocessing phase. The x, y, and z values are the coordinates of the normalized surface normal at the point that is to be shaded. Unlike low dynamic range images (e.g., 8 bits per color component) that have an implicit minimum value of 0 and an implicit maximum value of 255, high dynamic range images represented with a floating-point value for each color component don't contain well-defined minimum and maximum values.

The minimum and maximum values for two HDR images may be quite different from each other, unless the same calibration or creation process was used to create both images. It is even possible to have an HDR image that contains negative values. For this reason, the vertex shader contains an overall scaling factor to make the final effect look right.

The vertex shader that encodes the formula for the

nine spherical harmonic basis functions is actually quite simple. When the compiler gets hold of it, it becomes simpler still. An optimizing compiler typically reduces all the operations involving constants. The resulting code is quite efficient because it contains a relatively small number of addition and multiplication operations that involve the components of the surface normal.

```
varying vec3 DiffuseColor;
uniform float ScaleFactor;

const float C1 = 0.429043;
const float C2 = 0.511664;
const float C3 = 0.743125;
const float C4 = 0.886227;
const float C5 = 0.247708;

// Constants for Old Town Square
lighting
const vec3 L00  = vec3( 0.871297,
0.875222,  0.864470);
const vec3 L1m1 = vec3( 0.175058,
0.245335,  0.312891);
const vec3 L10  = vec3( 0.034675,
0.036107,  0.037362);
const vec3 L11  = vec3(-0.004629, -
0.029448, -0.048028);
const vec3 L2m2 = vec3(-0.120535, -
0.121160, -0.117507);
const vec3 L2m1 = vec3( 0.003242,
0.003624,  0.007511);
const vec3 L20  = vec3(-0.028667, -
0.024926, -0.020998);
const vec3 L21  = vec3(-0.077539, -
0.086325, -0.091591);
const vec3 L22  = vec3(-0.161784, -
0.191783, -0.219152);
```

```
void main()
{
    vec3 tnorm    = normal-
ize(gl_NormalMatrix * gl_Normal);

    DiffuseColor =  C1 * L22 *
(tnorm.x * tnorm.x - tnorm.y *
tnorm.y) +
                    C3 * L20 *
tnorm.z * tnorm.z +
                    C4 * L00 -
                    C5 * L20 +
                    2.0 * C1 * L2m2 *
tnorm.x * tnorm.y +
                    2.0 * C1 * L21  *
tnorm.x * tnorm.z +
                    2.0 * C1 * L2m1 *
tnorm.y * tnorm.z +
                    2.0 * C2 * L11  *
tnorm.x +
                    2.0 * C2 * L1m1 *
tnorm.y +
                    2.0 * C2 * L10  *
tnorm.z;

    DiffuseColor *= ScaleFactor;

    gl_Position = ftransform();
}
```

Listing 1 Vertex shader for spherical harmonics lighting

```
varying vec3 DiffuseColor;

void main()
{
    gl_FragColor = vec4(DiffuseColor,
1.0);
}
```

Listing 2 Fragment shader for spherical harmonics lighting

Once again, our fragment shader has very little work to do. Because the diffuse reflection typically changes slowly, for scenes without large polygons we can reasonably compute it in the vertex shader and interpolate it during rasterization. As with hemispherical lighting, we can add procedurally defined point, directional, or spotlights on top of the spherical harmonics lighting to provide more illumination to important parts of the scene.

Coefficients for some of Paul Debevec's light probe images provide even greater color variations. We could make the diffuse lighting from the spherical harmonics computation more subtle by blending it with the object's base color.

CONCLUSION

Now the programmable graphics hardware has freed us from the shackles of the traditional hardware lighting equations, we are free to implement and experiment with a variety of new techniques. Some of the techniques we explored are both faster and more realistic than the traditional methods.

Such light probe images can either be preprocessed and used to compute spherical harmonic basis function coefficients that can be used for simple and high-performance lighting.

REFERENCES

Ramamoorthi, Ravi, and P. Hanrahan, 2001, *An Efficient Representation for Irradiance Environment Maps*, Computer Graphics (SIGGRAPH 2001 Proceedings), pp. 497500.

Baldwin, Dave, October 2001, *OpenGL 2.0 Shading Language White Paper*, Version 1.0, 3Dlabs.

ATI developer Web site. http://www.ati.com/developer

Akenine-Möller, Tomas, and E. Haines, 2002, *Real-Time Rendering, Second Edition*, AK Peters, Ltd., Natick, Massachusetts. http://www.realtimerendering.com

BIOGRAPHIES

Wesley, A., 2006, *OpenGL Shading Language 2nd Edition*. Orange Book: Addison Wesley Professional