CS562 Project 3

Synopsis

Enhance the previous projects with Physically Based Lighting (PBS), and Image Based Lighting (IBS), and a bit of global tone-mapping.

- PBS: a micro-facet BRDF to provide realistic surface lighting calculations.
- IBL: an HDR environment map to provide surround lighting.
- Tone Mapping and Linear Color space: The mapping of a High-Dynamic-Range image onto a Low-Dynamic-Range device.

Instructions

PBS: Implement a physically-based micro-facet BRDF lighting calculation. This requires the same information as Phong lighting, but interprets the quantities differently, and calculates the results very differently and far more accurately. The details are below.

IBL: Supply lighting for your scene using an HDR environment image. This replaces the constant ambient light term (from previous lighting models) with a detailed calculation of the environment's lighting effect at each point in the scene. See:

http://http.developer.nvidia.com/GPUGems3/gpugems3_ch20.html

for details.

Tone Mapping and Linear Color space: Use a simple *gamma compression* tone mapping to display your HDR calculations on an ordinary LDR screen. Do all your lighting calculations in linear color space by mapping inputs from sRGB to linear color spaces, and output colors in the reverse direction. The details are below.

See the supplied sample-filter.zip for a bit of code demonstrating how to read and write images of format *.hdr (a.k.a. RGBE, a.k.a. RADIANCE) (in file sample-filter.cpp). This makes use of a widely and freely available library for reading and writing such files (in files rgbe.cpp and rgbe.h). This sample also demonstrated the absolutely simple magic of OpenMP for making your loops multi-threaded. (See the single line pragma.)

What to hand in

Please submit a zip file containing the relevant code and a project report (DOC, DOCX, ODT, PDF, ... format). Your report should contain sufficient screen captures (and accompanying text) to demonstrate the correctness of your project, including:

- Images demonstrating the accurate lighting/interaction of object(s) with an environment.

I require the program code only for completeness sake; I do not intend to compile the code to check for correctness of the project. This is to especially allow you to implement your project in your game without my having to learn to build and run the game. This is not an issue if you use my default framework.

Lighting

The full lighting calculation involving a BRDF $f(...)$, is

$$\int_{\Omega} f(\omega_i, \omega_o)\, L_i(\omega_i) \cos\theta_i \, d\,\omega_i \qquad (1)$$

with L_i representing a light's brightness, and the cosine term being the usual $(N \cdot L)$. This is modified an various way for real-time lighting calculations:

- The integral is approximated by a sum over individual lights (or pixels)
- Various parts of the integral can be factored out and pre-calculated.

Physically Based Lighting

The full BRDF with Lambertian diffuse term and micro-facet specular term is:

$$f(L, V, N) = \frac{K_d}{\pi} + \frac{D(H)\ G(L, V, H)\ F(L, H)}{4\,(L \cdot N)(V \cdot N)} \qquad (2)$$

where L, V are the unit length vectors toward the light and eye, N is the surface normal, and H is the half-way vector L+V scaled to unit length.

A nice starter set of D, G, and F functions is:

- Phong with roughness parameter α : 0(rough) … ∞ (mirror)

  $$D(H) = \frac{\alpha+2}{2\pi}(N \cdot H)^{\alpha}$$

- A well known approximation to more carefully derived shadow terms:

  $$\frac{G(L, V, H)}{(L \cdot N)(V \cdot N)} = \frac{1}{(L \cdot H)^2} \text{ or even } ... = 1$$

- Schlick's approximation of the Fresnel equation:

  $$F(L, H) = K_s + (1 - K_s)(1 - L \cdot H)^5$$

Tone Mapping and Linear Color space

Tone mapping: Since HDR images can contain colors in the range $[0 \ldots \infty]$ some attempt must be made to convert an infinite range into the displayable range $[0 \ldots 1]$. An easy global tone-mapping calculation is

$$C_{out} = \left(\frac{e\ C}{(e\ C + (1,1,1))} \right)^{1/2.2}$$

where e is a total exposure control (in the range 0.1 to 10000 for various of my test IBL images), and the exponent does the conversion from linear to gamma color space for final display. (See next paragraph.) For a little extra control, you may want to include an extra adjustable contrast parameter in the exponent $c/2.2$.

Color spaces: All lighting/color calculations should take place in **linear** color space while most input images and all output devices expect values in a non-linear color space (e.g., **gamma** color space or **sRGB** color space). Color values should be converted from sRGB to linear on input and back to sRGB just before output to the display device. The simplest such conversion calculations are:

sRGB to linear: $C_{\text{linear}} = C_{\text{sRGB}}^{2.2}$

linear to sRGB: $C_{\text{sRGB}} = C_{\text{linear}}^{1/2.2}$

or look up (on Wikipedia) the more accurate/complex conversion.

***When to use these conversions*:**

- **Input conversion to linear space:** Most image/texture are in sRGB space because they are meant to display nicely on sRGB devices. Convert these. Input RGB colors should also be converted (or you may just get use to input them in linear space).
- **Calculation**: Nothing changes in your calculation except you can be assured that adding and scaling colors actually makes physical sense now.
- **Output**: Convert final linear space color to sRGB space and output. If your colors range is not ranged $[0 \ldots 1]$ then tone-map into that range before outputting. It is convenient to allow interactive control over the tone-mapping range and gamma parameters.

Image Based Lighting

Image based lighting uses a single HDR image, wrapped around a sky-dome (or equivalent), for lighting a scene. The purpose of this project is to replace the hefty amount of precomputation needed by most IBL algorithms with something that can be calculated in real-time. All calculations are driven by the lighting equation along with a micro-facet BRDF (Equations (1) and (2) above.)

A sample full resolution HDR image

Low resolution irradiance HDR image

Diffuse calculation equations:

The diffuse portion of the BRDF placed in the lighting equation

$$\int_{\Omega} \frac{K_d}{\pi} L_i(\omega_i) \cos\theta_i \, d\omega_i = \frac{K_d}{\pi} \int_{\Omega} L_i(\omega_i) \cos\theta_i \, d\omega_i = \frac{K_d}{\pi} irradiance(N)$$

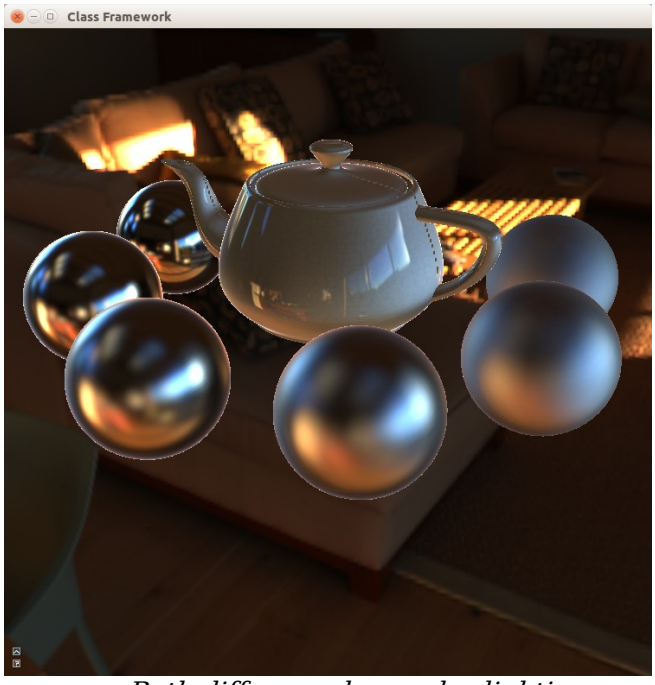
shows that the integral is a function of only N, and so can be precalculated and accessed with a single lookup in the direction N. Such a map is called an irradiance map., and for this project we will simply use a pre-calculated irradiance map. Later we will consider slow-and-easy methods to calculated such. Later still we will consider fast-and-complex ways to generate them at application startup time.

Diffuse calculation usage:

The usage of an irradiance map accessed via normal direction

$$\text{diffuse color} = \frac{K_d}{\pi} irradiance(N)$$

Diffuse lighting only,
from the irradiance map.



Both diffuse and specular lighting

Specular calculation equations: ***(See http.developer.nvidia.com/GPUGems3/gpugems3_ch20.html for details.)***

The specular portion of the lighting requires an approximation of the lighting integral

$$\int_{\Omega} f(\omega_i, \omega_o)\, L_i(\omega_i) \cos\theta_i \, d\,\omega_i$$

where the function $f(...)$ is now the specular portion of the BRDF

$$f(L,V) = \frac{D(H)\;\; G(L,V,H)\;\; F(L,H)}{4\,(L\cdot N)\,(V\cdot N)}.$$

The Monte-Carlo approximation to this integral is

$$\frac{1}{n}\sum_{k=1}^{n} \frac{f(\omega_k, V)\, L_i(\omega_k) \cos\theta_k}{p(\omega_k)}$$

where the n directions ω_k are chosen with probability $p(\omega_k)$. With some cleverness, we choose $p(\omega_k) = D(H)$ to cancel each other out giving a Monte-Carlo approximation of

$$\frac{1}{n}\sum_{k=1}^{n} \frac{G(\omega_k, V, H)\;\; F(\omega_k, H)}{4\,(\omega_k\cdot N)\,(V\cdot N)}\, L_i(\omega_k) \cos\theta_k \qquad (3)$$

Specular calculation usage:

The usage of the this in the lighting shader becomes:

- Choose N (20 to 40) random directions ω_k according to probability $p(\omega_k) = D(H)$. See next section. (This is "importance sampling.)
- For each ω_k, evaluate the incoming light $L_i(\omega_k)$ by accessing the HDR image for each direction vector ω_k. For the "filter" step of the paper, read this texel from the MIPMAP at a carefully calculated level (using GLSL's **textureLod** function):

  $$level = \frac{1}{2}\log_2\left(\frac{WIDTH * HEIGHT}{N}\right) - \frac{1}{2}\log_2(D(H))$$

  Hints: GLSL provides log2() for logarithms in base 2.
  Note: Experience seems to suggest that this equation from the paper produces values that are to high and spread out. Try $(1/2)\log_2(D(H)/4)$ for the second term.
- Evaluate the Monte-Carlo estimator (eq (3) above) for each pair ω_k and $L_i(\omega_k)$.
- Sum them up, compute average, and add the results to the previous diffuse calculation for the final lighting calculation.

Missing features

Two features of the original paper have been left out. An initial implementation without them seems to produced reasonable results. This opinion may change with more experience.

1. The mapping distortion factor d(u) which should be in the $\log_2(D)$ part of the above level calculation accounts for the vast distortion near the poles of the spherical map. This may become important if one uses a environment map with important details around the poles.
2. The transformation to a dual paraboloid map is used to avoid trouble around the the edges/poles/seam of the HDR environment map. I suggest waiting until you are actually bothered by such problems before implementing this solution to them.

Odds and ends

Longitude-latitude Sphere map:

The sphere map for an environment image is slightly different from the usual longitude-latitude map because we are viewing the image from the inside. This explains the $1/2-\ldots$ portions in both the following equations.

The map from a direction vector N to a texture coordinate (u,v) is:

$$N \;\Rightarrow\; (u,v) = \left(\frac{1}{2} - \frac{atan(N.y, N.x)}{2\pi},\; \frac{acos(N.z)}{\pi}\right)$$

The inverse map from a texture coordinate to a direction vector is:

$$(u,v) \;\Rightarrow\; \left(\cos(2\pi(1/2-u))\sin(\pi v),\; \sin(2\pi(1/2-u))\sin(\pi v),\; \cos(\pi v)\right) \qquad (4)$$

How to choose N random directions ω_k according to probability $p(\omega_k) = D(H)$

These steps start by building the proper distribution around the Z-axis, and finish by rotating it to the reflection direction.

1. Choose pairs of uniformly distributed random numbers (ξ_1, ξ_2) **very** carefully. (See below.)
2. Skew the points to match the distribution $D(H)$: $(u,v) = \left(\xi_1,\; \cos^{-1}\left(\xi_2^{\frac{1}{\alpha+1}}\right)/\pi\right)$,
3. Form the direction vector L for that pair using eq (4).
4. Form a rotation that takes the Z-axis to the reflection direction $R = 2(N \cdot V)N - V$.
   $A = normalize(Z_{axis} \times R)$
   $B = normalize(R \times A)$
   $\omega_k = normalize(L.x\ A + L.y\ B + L.z\ R)$
   Precalculate R, A, and B before entering the loop for the many ω_k. In the above equation, ω_k is unit-length mathematically, but because of occasional roundoff errors, I found I needed to normalize it anyway. (This does not make **any** sense to me, because I can't see how the remaining calculations with ω_k in the BRDF/Lighting equations are sensitive to such small roundoff errors. Nevertheless, they seem to be. **This warning may save you hours of debugging to rid the final image of 10-20 black pixels. Yuck.**)

Building a low-discrepency list or pairs of pseudo-random points

(See the Wikipedia page http://en.wikipedia.org/wiki/Low-discrepancy_sequence)

Choose N (about 20-40 for this project) and build N pairs of points with this algorithm. Placing both N and the array of floats in a C-style struct makes it easy to send this data to the GPU. See the next section for that.

```
struct {
   float N;
   float hammersley[2*N]; }  block;
block.N = N; // N=20 ... 40 or whatever ...

int kk;
int pos = 0;

for (int k=0;  k<N;  k++) {
   for (p=0.5f, kk=k, u=0.0f;  kk;  p*=0.5f, kk >>= 1)
      if (kk & 1)
         u += p;
   float v = (k + 0.5) / N;
   block.hammersley[pos++] = u;
   block.hammersley[pos++] = v; }
```

Sending a block of data to shader as a GLSL uniform block

After creating the data block as above, send it to the GPU as a GL_UNIFORM_BUFFER:

```
unsigned int id, bindpoint;
```

```
glGenBuffers(1, &id);
bindpoint = 1;  // Increment this for other blocks.
glBindBufferBase(GL_UNIFORM_BUFFER, bindpoint, id);
glBufferData(GL_UNIFORM_BUFFER, sizeof(block), &block, GL_STATIC_DRAW);
```

After creating the lighting shader, attach the buffer to a uniform variable:

```
int loc = glGetUniformBlockIndex(shaderprogram, "HammersleyBlock");
glUniformBlockBinding(program, loc, bindpoint);
```

In the shader program declare the block, but access **N** and **hammersley** directly:

```
uniform HammersleyBlock {
  float N;
  float hammersley[2*100]; };
```