Real-Time Relighting of Compressed Panoramas

Tien-Tsin Wong and Siu-Hang Or, The Chinese University of Hong Kong, and Chi-Wing Fu, Indiana University, Bloomington

ttwong@acm.org, shor@cse.cuhk.edu.hk, cwfu@acm.org

Panorama is a simple representation for modeling large-scale complex backgrounds without paying the expensive rendering cost. It can be applied in computer games to increase visual richness. However, the static panoramic images do not allow the dynamic changing of lighting (known as "dynamic lighting" in the game community). The ability to change lighting allows the game developer to create a dramatic atmosphere. The simplest solution is to render the complete geometric models during the game execution. Unfortunately, unless specially tuned, real-time rendering of complex scenes is usually not possible due to the necessary computations. It would be ideal if we can "relight" (modify the lighting) the panoramic image without referring to the complex geometry. Wong et al. [Wong01] proposed a "relightable" panorama representation that is in between the static panoramic image and geometric model (see the spectrum in Figure 1). It allows real-time dynamic lighting without rendering the complex geometry.

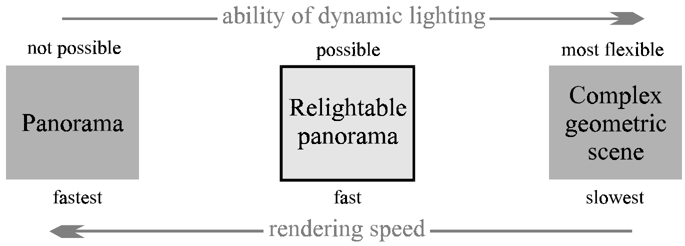


Figure 1. A spectrum of representations for modeling complex scenes.

The relightable panorama is actually a set of hundreds of reference panoramic images, each with a different lighting condition. By looking up the pixel values in this huge data set, we can realistically simulate the effect under new lighting conditions [Wong97]. The trade-off of this representation is its enormous storage requirement. Thus, compression is a must. The challenge is how to achieve real-time relighting from this compressed panorama. In this article, we describe a hardware-assisted relighting method using programmable graphics hardware. It can be executed on consumer graphics hardware, more specifically, the nVidia GeForce3 or compatible graphics boards.

Compression of Relightable Panoramas

The reference panoramic images are created with a directional light source as the sole illuminant. Each of them corresponds to a distinct light vector (**L**) sampled on the spherical grid, shown on the left hand side of Figure 2. The massive color values are then rebinned in such a way that all values related to the same pixel window are grouped together. The right hand side of Figure 2 shows the result of rebinning. Each tile corresponds to one such group of color values. It tells us the color of that pixel when the scene is illuminated by a varying directional light. Note that the smoothness of color in a tile facilitates the compression.

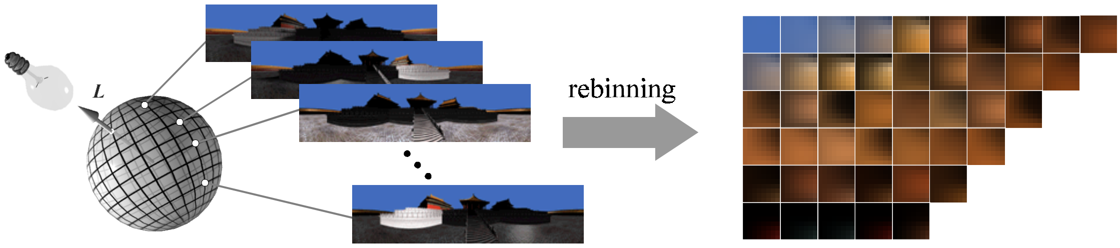


Figure 2. Reference panoramas are rebinned to maximize data correlation for compression.

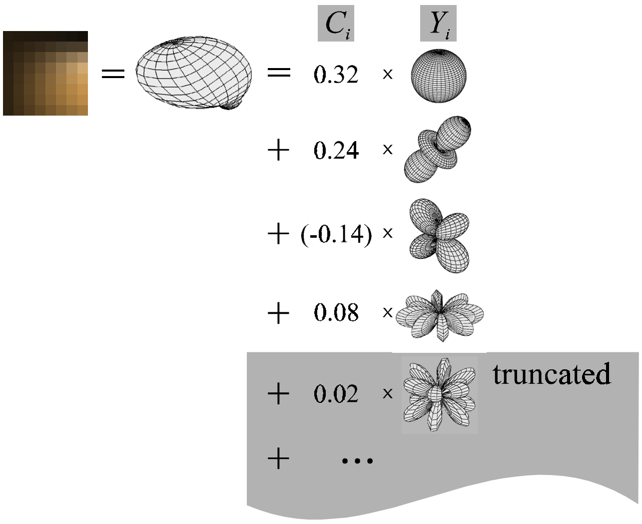


Figure 3. Color values associated with the same pixel window can be decomposed into spherical harmonic domain where high-frequency components are dropped to reduce storage.

Each tile is in fact a spherical function (table), as every color value inside corresponds to a sample on sphere. To compress the tile, spherical harmonic transform [Courant53] (analogous to a Fourier transform in spherical domain) is applied. The spherical harmonic transform decomposes a spherical function into a series of low- and high-frequency components. As illustrated in Figure 3, each component is the product of a coefficient C_i (the numeric value) and a basis function Y_i (the bumpy complex object). Since the spherical function is usually smooth, high-frequency components can be dropped to reduce storage, while preserving the accuracy. The idea is similar to sacrificing high-frequency DCT (Discrete Cosine Transform) components in JPEG coding. By keeping the first k coefficients, we can reconstruct a color value within the tile by linearly combining C_i and Y_i,

$$\sum_{i=0}^{k-1} C_i Y_i(\mathbf{L}). \qquad (1)$$

A useful value range of k is 16 to 25. Coefficients C_i are the constants to store. The basis functions Y_i return scalar values. They are mathematically defined and do not need to be stored. The input to these functions is the light vector $\mathbf{L}$ that actually looks up the sample on sphere.

In other words, after the spherical harmonic transform, the array of tiles in Figure 2 is converted to an array of k-dimensional coefficient vectors, as shown in Figure 4. Let's call these vectors the SH vectors from now on, where SH stands for spherical harmonic. The total size of the SH vectors is much smaller than that of the original color values. Hence, compression is achieved. The detail formulas of the spherical harmonic transform and its inverse (reconstruction) are listed in the Appendix.

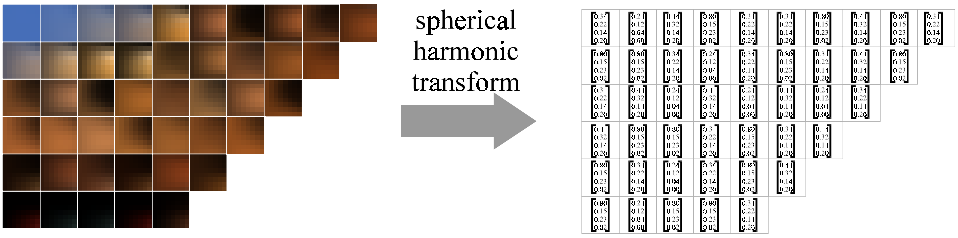


Figure 4. The spherical harmonic transform converts the color tiles (1 tile for each pixel window) to SH vectors for storage reduction.

Equation 1 is the key to relighting. Basis function Y_i is a function of light vector $\mathbf{L}$. Given a specific $\mathbf{L}$, Equation 1 allows us to quickly look up (reconstruct) a pixel value due to that lighting direction without reconstructing the whole color tile. To relight the whole panorama, Equation 1 is performed for all pixels in the panorama. Thus, this computation is fully *parallelizable*. This is why we can use SIMD-based programmable graphics hardware [Lindholm01] to achieve real-time relighting.

In order to parallelize the computation, we need to rebin the array of k-dimensional SH vectors to form k SH maps, as in Figure 5. They are rebinned in the following manner. The first coefficients of all SH vectors are grouped to form the first SH map. This process is repeatedly applied to form other SH maps. Interestingly, each SH map is an image of real values (can be positive or negative). These SH maps are the data to be stored on disk and they will be loaded into memory to relight the panorama when the program starts up.

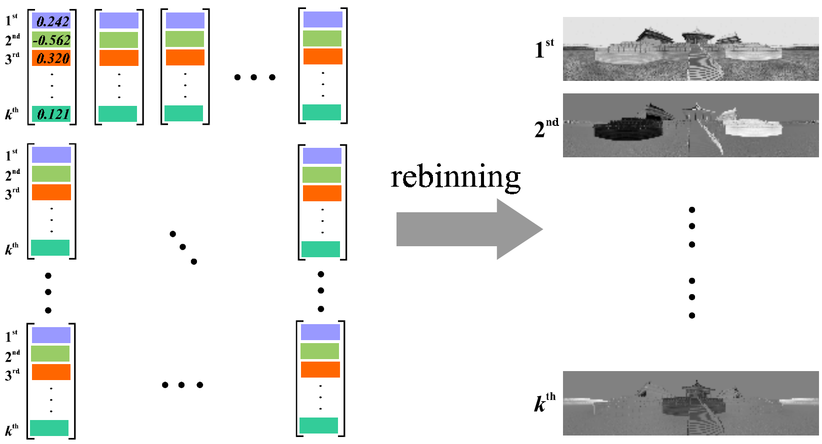


Figure 5. The array of k-dimensional SH vectors can be rebinned to form k SH maps on the right.

Relighting by a Directional Source

Linear Combination

It is interesting that relighting the panorama is actually a reconstruction process that computes Equation 1 for every pixel in the panorama. This process requires a light vector $\mathbf{L}$ to lookup a color value in the encoded data. Let's first consider the simplest case, relighting by a directional light source. In the case of a directional source, every pixel gets the same light vector $\mathbf{L_0}$ and hence the same $Y_i(\mathbf{L_0})$. Therefore, the relighting can be formulated as a linear combination of SH maps and *scalars* $Y_i(\mathbf{L_0})$, in Figure 6.

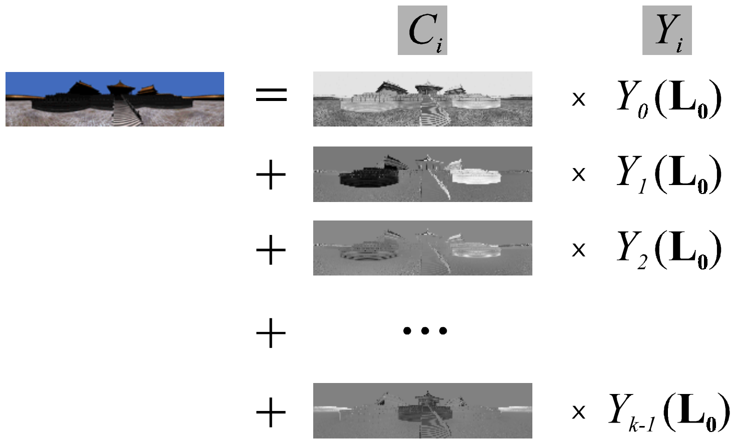


Figure 6. Relighting by a directional light is a simple linear combination of SH maps and scalars $Y_i(\mathbf{L_0})$.

To relight in real-time, we put the SH maps into the texture buffers. They are then blended together with $Y_i(\mathbf{L_0})$ as the scaling factors. It seems that this kind of texture blending can be easily achieved using OpenGL. Unfortunately, both the SH map and $Y_i(\mathbf{L_0})$ may contain negative values and signed textures are not supported in standard OpenGL. Our solution is to write shaders on programmable graphics hardware.

Shader Implementation

There are two main types of shaders, vertex shaders and fragment shaders [Lindholm01]. For a directional light source, we mainly use the fragment shader that takes care of per-pixel operations. However, the latest development of programmable graphics hardware only supports precision-limited operations (much less than the 32-bit floating point) in the fragment shader and more seriously, there is no standard shading language supported by all manufacturers. Cg (C for graphics) programming language [Cg02] inherits many high-level features of the RenderMan shading language [Upstill90]. It is a promising standard, but supporting hardware is not available at the time of preparation of this article. Therefore, we have to choose one non-standard shader extension of OpenGL for our development.

In particular, we use the nVidia GeForce3 graphics board and its OpenGL shader extension. The fragment shader of the nVidia OpenGL extension is divided into a texture shader and register combiner. The texture shader handles how a texture is fetched from memory. The register combiner handles the per-pixel operations between fetched textures. Therefore, our relighting is mainly executed in the register combiner.

First of all, the SH maps are loaded into the texture units of the graphics board. Since the number of texture units in one graphics board is usually limited, the relighting process has to be divided into multiple passes. Figure 7 shows an illustration of such multi-pass summation. The input of the fragment shaders is textures, while the output is the framebuffer or pbuffer. In particular, our graphics board supports 4 texture units, named as `tex0`, `tex1`, `tex2`, and `tex3`.

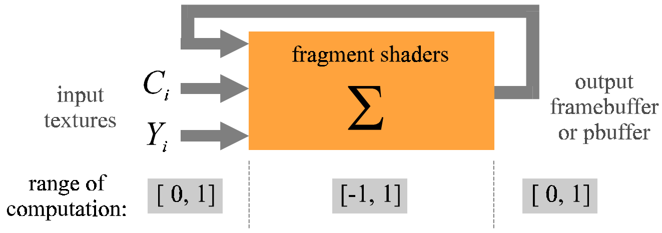


Figure 7. Fragment shader is used to sum C_iY_i iteratively. One major concern is that the numeric range of computation must be carefully handled.

Note that both the input textures and the output buffer cannot represent negative values. The computation of negative values is only allowed inside the shader. Therefore, the values in SH maps are mapped from the domain of [-1,1] to the range of [0,1] before loading into the texture units. These values are then "`expand()`" to [-1,1] in the register combiner and further mapped back to [0,1] right before outputting to the framebuffer or pbuffer. The following OpenGL code fragment loads a SH map into the texture unit, `tex0`.

```
...
glEnable(GL_TEXTURE_SHADER_NV);
glActiveTextureARB(GL_TEXTURE0_ARB);   // activate tex0
(SHmap[0]).bind();                     // load the buffer to texture unit
(SHmap[0]).enable();
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
          GL_TEXTURE_RECTANGLE_NV);
...
```

In Listing 1, we list the code of the register combiner used for linearly combining 4 SH maps. The language syntax of register combiner is a little bit hard to understand. There can be several combiners in one shader, each enclosed by the construct `rgb{}`. All operations in `rgb{}` are executed for RGBA channels. In the following shader code, we use 4 combiners to scale and sum 4 SH maps (loaded into `tex0` to `tex3`). The scalars Y_i are input through the color constants `const0` and `const1`. The string "`%f`" allows us to "`sprintf`" the scalars Y_i into the shader code. The values of Y_i for R, G & B channels are the same and must be mapped to [0,1]. The first combiner scales C_0 by Y_0 and C_1 by Y_1. They are then summed and stored into the intermediate buffer `spare1`. The function `expand()` maps values from [0,1] to [-1,1]. The next two combiners accumulate C_2Y_2 and C_3Y_3. The last combiner maps the total sum (stored in `spare1`) from the range [-1,1] to [0,1] right before output. Due to the non-negative restriction of both the input textures and the output buffer, we have to do this complex mapping within the shader.

Listing 1. Register Combiner in OpenGL.

```
!!RC1.0
{ # the first combiner computes C0Y0 + C1Y1
    const0 = (%f, %f, %f, 0.0);                    # Y0
    const1 = (%f, %f, %f, 0.0);                    # Y1
    rgb {
        discard = expand(tex0) * expand(const0);   # a = C0 * Y0
        discard = expand(tex1) * expand(const1);   # b = C1 * Y1
        spare1  = sum();                           # a + b
    }
}
```

```
{ #  the second combiner computes C0Y0 + C1Y1 + C2Y2
    const0 = (%f, %f, %f, 0.0);                     #  Y2
    rgb {
        discard = expand(tex2) * expand(const0);    #  c = C2 * Y2
        discard = spare1;                           #  a+b
        spare0  = sum();                            #  a+b+c
    }
}
{ #  the third combiner computes (C0Y0 + C1Y1 + C2Y2 + C3Y3) / 2
    const0 = (%f, %f, %f, 0.0);                     #  Y3
    rgb {
        discard = expand(tex3) * expand(const0);    #  d = C3 * Y3
        discard = spare0;                           #  a+b+c
        spare1  = sum();                            #  a+b+c+d
        scale_by_one_half();                        #  (a+b+c+d)/2
    }
}
{ #  the last combiner maps the total sum from [-1,1] to [0,1]
    const0 = (0.5, 0.5, 0.5, 0.0);
    rgb {
        discard = spare1;                           #  (a+b+c+d)/2
        discard = const0;                           #  0.5
        spare0  = sum();                            #  (a+b+c+d)/2 + 0.5
    }
}
out.rgb = spare0;                                   # output RGB value
out.a   = unsigned_invert(zero);                    # alpha = 1.0
```

Since the linear combination is divided into multiple passes, the intermediate result (output of each pass) is stored in a `pbuffer`, instead of outputting to the framebuffer. The pbuffer is actually a framebuffer, but it can also be fed to the register combiner as input texture. Therefore, the output of one pass can be imported to another pass for further accumulation.

Cg Implementation

The register combiner is relatively difficult to use and its operations are low in precision. In contrast, the high-level Cg shader language is easier to implement and it supports high-precision operations. In Listing 2, we have also implemented the relighting using Cg. The following Cg shader scales and sums 4 SH maps.

Listing 2. Register combiner in Cg.

```
// input data type
struct InputData{
   float3 TexelPos0:TEX0;
   float3 TexelPos1:TEX1;
   float3 TexelPos2:TEX2;
   float3 TexelPos3:TEX3;
};

// output data type
struct OutputData{
   float4 Color:COL;
};
```

```
// Main shader function
OutputData main2(
   InputData in,
   uniform sampler2D tex0 : texunit0,   // C0
   uniform sampler2D tex1 : texunit1,   // C1
   uniform sampler2D tex2 : texunit2,   // C2
   uniform sampler2D tex3 : texunit3,   // C3
   uniform float4 Y0,                   // Y0
   uniform float4 Y1,                   // Y1
   uniform float4 Y2,                   // Y2
   uniform float4 Y3,                   // Y3
)
{
   OutputData out;
   float4     accum;

   // Ci  in textures are mapped to [-1,1] before multiplying with Yi
   accum =         (f4tex2D(tex0,in.TexelPos0.xy)-0.5)*2 * Y0;
   accum = accum + (f4tex2D(tex1,in.TexelPos1.xy)-0.5)*2 * Y1;
   accum = accum + (f4tex2D(tex2,in.TexelPos2.xy)-0.5)*2 * Y2;
   accum = accum + (f4tex2D(tex3,in.TexelPos3.xy)-0.5)*2 * Y3;
   accum = (accum/2.0) + 0.5;      // map from [-1,1] to [0,1]

   out.Color = accum;
   return out;
}
```

In fact, Cg supports more textures than our register combiner implementation. However, for the purpose of comparison, both implementations are developed to perform equally well. In Cg, both the input textures and the output buffer still do not support negative values. Therefore it is still necessary to map the values of input textures from [0,1] to [-1,1] and the output values from [-1,1] to [0,1]. Function `f4tex2D()` retrieves the RGBA value of a texel as a floating point from the texture (e.g. `tex0`), given a 2D coordinate (e.g. `in.TexelPos0.xy`).

Relighting by a Point Source

Per-pixel Linear Combination

Relighting panoramas by a point light source is basically the same as that of a directional light source. The major difference is that the light vector **L** is *different* for each pixel. Hence, $Y_i(\mathbf{L})$ are different for each pixel. Figure 8 illustrates such difference when compared to Figure 6. $Y_i(\mathbf{L})$ are maps of scalars instead of a single scalar. Instead of a linear combination of images as in Figure 6, we now have a per-pixel linear combination of color values. The major difficulty of point-source relighting is the computation of Y_i. To do so, we first need to compute the light vector for each pixel.

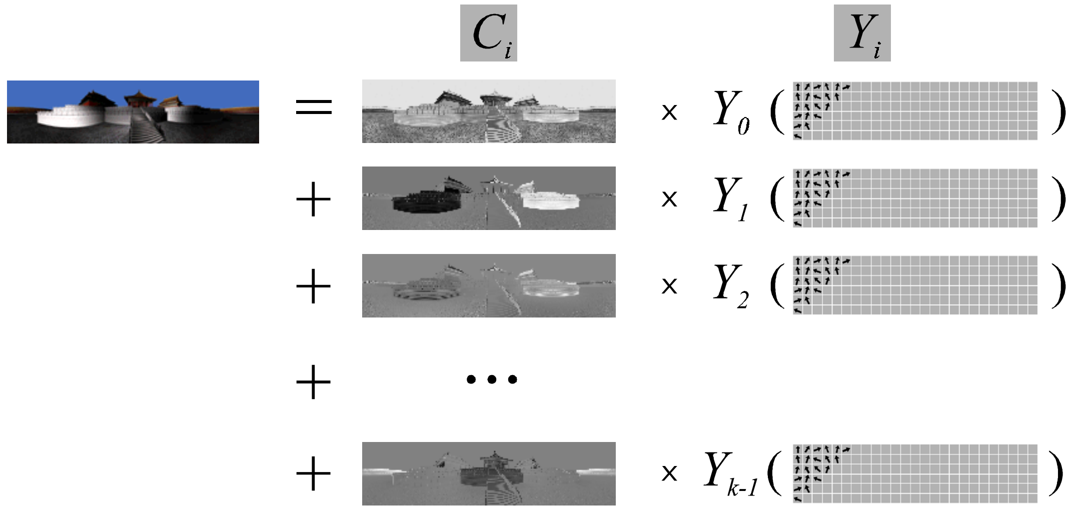


Figure 8. Relighting by a point light source requires a per-pixel linear combination of color values.

Computing the Light Vector

For relighting of a non-directional light source, depth map of the panorama is required due to the heterogeneous nature of light vector. The light vector is computed by the following equation given the depth value.

$$\mathbf{L} = \mathbf{S} - \left(\mathbf{E} - \frac{\mathbf{V}}{|\mathbf{V}|} d \right), \qquad (2)$$

where **S** is the position of the point light source; **E** is the viewpoint associated with the panorama; **V** is the viewing direction associated with the interested pixel; and d is the depth value of that pixel. Since **E**, **V** and d are all known and only **S** varies during run-time, we can pre-compute $\mathbf{P} = \mathbf{E} - d\,\mathbf{V}/|\mathbf{V}|$ (the intersection points between the viewing rays **V** and the scene) and store them as a vector map. Figure 9 shows the steps to convert the depth map to a map of intersection point that is further converted to a map of light vectors.

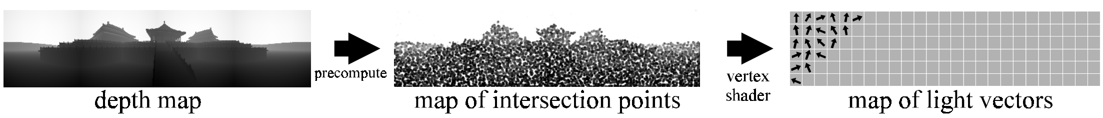


Figure 9. Steps to prepare the map of light vector.

To obtain the light vector of each pixel in real-time, we first model every pixel as a vertex. At each vertex, we execute the following vertex shader code fragment to compute **L**.

```
!!VP1.0
...
# v[1]  : intersection point, P        o[TEX1]: computed light vector, L
# c[10]: light source position, S      c[11]:   unity vector [1,1,1,1]
ADD o[TEX1],    c[10],    -v[1];      # L = S - P
MOV o[TEX1].w, c[11].x;               # L.w = 1.0
...
```

Intersection point **P** is input to the vertex shader as a vertex attribute `v[1]`. The light source position is input through register `c[10]`. Register `c[11]` is simply a unity vector [1,1,1,1] because we cannot specify constant 1 in the vertex shader. The computed light vector **L** is output to texture unit `tex1` (`o[TEX1]`) for computing Y_i.

Computing the Basis Functions

Once the light vector map is cooked, it can be used to compute Y_i. However, Y_i is rather complicated (see Appendix) and impractical to compute in real-time. Since each Y_i function is a spherical function and **L** specifies a direction, our solution is to model each Y_i function as a cube-map and use **L** as a look-up vector, as shown in Figure 10. The left hand side of Figure 10 shows one such cube-map, Y_{18}. All these cube-maps can be pre-computed. Note that Y_i may contain negative values, therefore care must be taken to handle the range problem as was done with the register combiner.

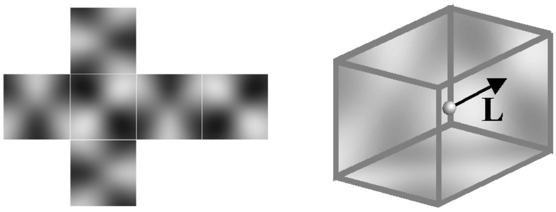


Figure 10. The function Y_i is modeled as a cube-map texture and the light vector **L** is used to look up the corresponding value of $Y_i(\mathbf{L})$.

To setup the cube-map texture, the following extended OpenGL code is used.

```
...
glActiveTextureARB(GL_TEXTURE1_ARB);
cubemap[0].bind();
cubemap[0].enable();
glTexEnvi(GL_TEXTURE_SHADER_NV, GL_SHADER_OPERATION_NV,
          GL_TEXTURE_CUBE_MAP_ARB);
...
```

Attenuation

To model the distance fall-off effect of a point light source, we can simply multiply the linearly combined result by an attenuation map, as demonstrated in Figure 11. This multiplication is implemented in the last pass of the fragment shader. The attenuation map is obtained from the map of light vector (un-normalized) in Figure 9. The attenuation formula we use is $C_0 / |\mathbf{L}|$, where C_0 is a user-defined constant; and $|\mathbf{L}|$ is the magnitude of **L** or the distance from the light source to the intersection point.

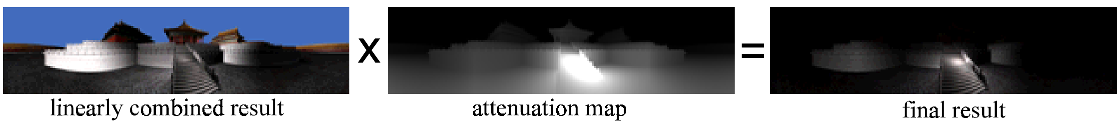
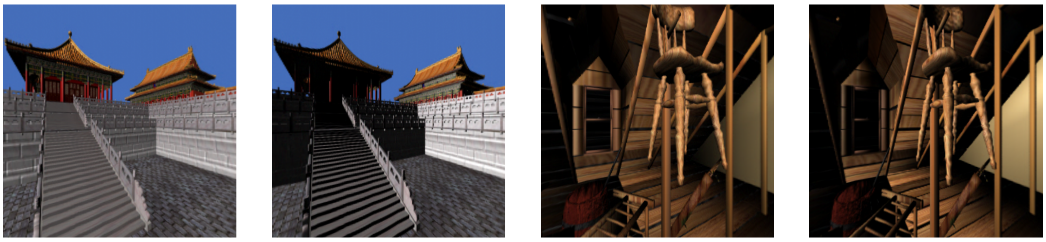


Figure 11. The distance fall-off effect can be simulated by multiplying the linearly combined result with an attenuation map.

Results

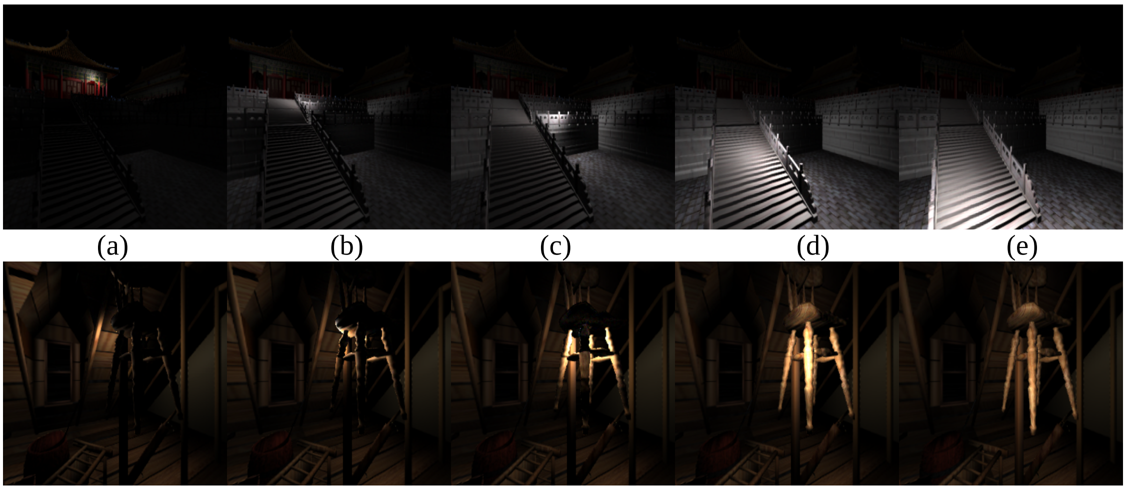
Figure 12 shows the perspective snapshots of relighting two panoramas, ‘forbid’ and ‘attic,’ by a moving directional light source. The geometric version of ‘forbid’ and ‘attic’ contain 500k and 1M triangles respectively. Thus dynamic lighting is not possible for these geometric counterparts. The image resolution of both scenes is 1024×256. The number of spherical harmonic components for both scenes is 16. The sizes of ‘forbid’ and ‘attic’ on disk are 2.8MB and 3.6MB, respectively. Since the computation of directional-source relighting is less intensive, the frame rate achieved is around 114 fps.



(a) (b) (c) (d)

Figure 12. Relighting by a moving directional light source. (a) & (b): forbidden city (forbid). (c) & (d): attic.

Figure 13 shows the sequences of images relit by a moving point light source. The 'forbid' panorama is relit by a moving point light source in Figures 13(a)-(e). In Figures 13(f)-(j), the point light source passes under the hanging chair in the scenes. Note how the illumination is accounted for even though the geometric model is not present. The relighting of point light source is computationally intensive. Hence the frame rate achieved is 20 fps. All timing statistics are recorded on a Pentium IV 1.5GHz equipped with the nVidia GeForce3 graphics accelerator.



(a) (b) (c) (d) (e)

(f) (g) (h) (i) (j)

Figure 13. Relighting by a moving point light source. (a)-(e): A point source moves in the forbidden city. (f)-(j): A point source passes under the hanging chair.

Conclusion

In this article, we have illustrated how to relight compressed panoramas in real-time. Hardware-assisted shader programming is used to achieve the goal. The relighting process is basically a linear combination of SH maps either image-wise (directional light source) or pixel-wise (point light source). Due to the limitation of current programmable graphics hardware, special care is needed. We expect the programmability and rendering performance will be further improved as the functionality of next-generation hardware increases. Interested readers are referred to the companion CD-ROM as well as the following web site for updated demo, tools and source code:

http://www.cse.cuhk.edu.hk/~ttwong/demo/panoshader/panoshader.html

Appendix

The SH vector can be obtained through the following integration,

$$C_{l,m} = \int_0^{2\pi}\int_0^{\pi} P(\theta,\phi) Y_{l,m}(\theta,\phi)\sin\theta d\theta d\phi \ .$$

Parameter (θ,ϕ) specifies a direction **L** in spherical coordinate system. Parameters l and m index the order of C and Y, where l = 0, 1, 2, … and m = -l, …, l. They are more convenient in a mathematical sense. The reason we use the notation i in previous sections is for simplicity in discussion. The spherical harmonic basis functions $Y_{l,m}(\theta,\phi)$ are recursively defined as,

$$Y_{l,m}(\theta,\phi) = \begin{cases} N_{l,m} Q_{l,m}(\cos\theta)\cos(m\phi) & \text{if } m > 0 \\ N_{l,0} Q_{l,0}(\cos\theta)/\sqrt{2} & \text{if } m = 0 \\ N_{l,m} Q_{l,m}(\cos\theta)\sin(|m|\phi) & \text{if } m < 0 \end{cases},$$

where

$$N_{l,m} = \sqrt{\frac{2l+1}{2\pi}\frac{(l-|m|)!}{(l+|m|)!}} \ ,$$

and

$$Q_{l,m}(x) = \begin{cases} 1 & \text{if } l = m = 0 \\ (1-2m)\sqrt{1-x^2}\,Q_{m-1,m-1}(x) & \text{if } l = m \neq 0 \\ (2m+1)xQ_{m,m}(x) & \text{if } l = m+1 \\ \frac{2l\text{-}1}{l\text{-}m} xQ_{l-1,m}(x) - \frac{l+m-1}{l-m} Q_{l-2,m}(x) & \text{otherwise.} \end{cases}$$

References

[Cg02] http://www.cgshaders.org/.

[Courant53] Courant, Richard and Hilbert, David, *Methods of Mathematical Physics*, Interscience Publisher, Inc., 1953.

[Lindholm01] Lindholm, Erik, et al, "A User-Programmable Vertex Engine," Proceedings of SIGGRAPH 2001, August 2001, pp. 149-158.

[Upstill90] Upstill, Steve, *The RenderMan Companion*, Addison Wesley, 1990.

[Wong01] Wong, Tien-Tsin, et al, "Interactive Relighting of Panoramas," IEEE Computer Graphics & Applications, Vol. 21, No. 2, March-April 2001, pp 32-41.

[Wong97] Wong, Tien-Tsin, et al, "Image-based Rendering with Controllable Illumination," Proceedings of the 8-th Eurographics Workshop on Rendering, St. Etienne, France, June 1997, pp 13-22.