Supplement for
Real-Time Soft Shadows in Dynamic Scenes using Spherical Harmonic Exponentiation

Zhong Ren[1*] Rui Wang[1*] John Snyder[2] Kun Zhou[3] Xinguo Liu[3]
Bo Sun[4†] Peter-Pike Sloan[5] Hujun Bao[1] Qunsheng Peng[1] Baining Guo[3]

[1] Zhejiang Univ. [2] Microsoft Research [3] Microsoft Research Asia [4] Columbia Univ. [5] Microsoft Corporation

This document contains supplemental material for our Siggraph submission, including more details about the run-time shading implementation and more extensive results.

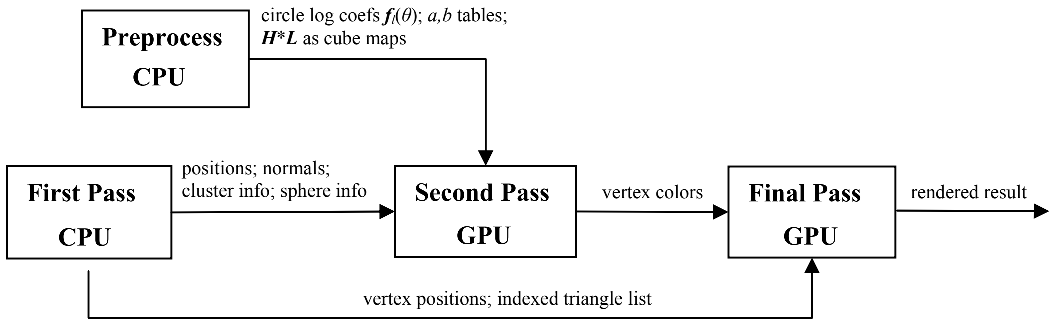
1 Run-Time Shading Implementation on the GPU

Given the blocker spheres at a vertex, we accumulate the blocker visibility log vectors $\boldsymbol{f}[i]$ to yield the total log visibility vector $\boldsymbol{f}$, and then perform SH exponentiation to transform the log vector back to a product space visibility vector, $\boldsymbol{g} = \exp_*(\boldsymbol{f})$. The result is then used in the $(\boldsymbol{H}(N)*\boldsymbol{L})\cdot\boldsymbol{g}$ shading computation. To save an SH product, we precompute $\boldsymbol{L}_H(N)=\boldsymbol{H}(N)*\boldsymbol{L}$, stored as a cube map, and do a cube map lookup using the normal vector N as the index.

To map this computation to the GPU, we have two basic choices:

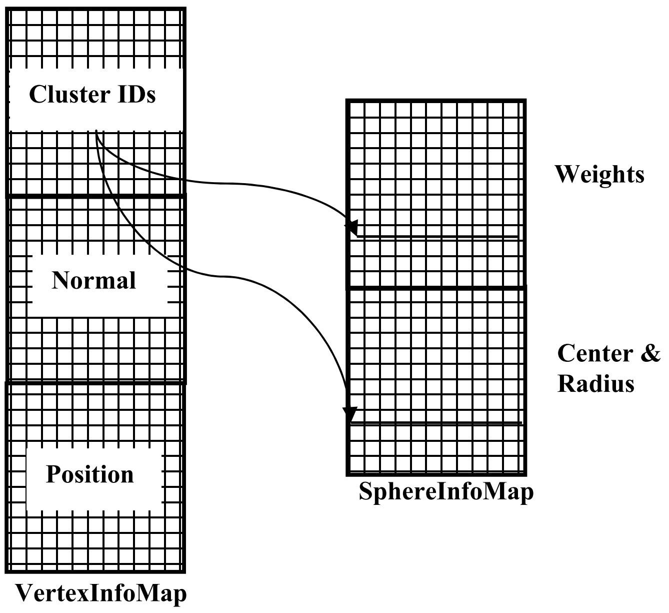
1. *Per-pixel shading*. A list of spherical blockers, parameterized by their center and radius, are packed into textures to be passed into a pixel shader. Per-pixel shading provides a high-quality result, but is too expensive for real-time shading. In one experiment using a 3.2GHz PC with 1GB memory and an Nvidia 6800GT graphics card, only 7~8 Hz was achieved for a small scene consisting of 32 blockers, rendered in a moderately-sized window of 600×600.
2. *Per-vertex shading*. Many tables are involved in the shading computation, including lists of spherical blockers, circle blocker log coefficients, $\boldsymbol{f}_l(\theta)$, and a and b coefficients used in the optimal linear (OL and HYB) exponentiation method. The NVidia GeForce6 series graphics cards support texture fetching in the vertex shader, so this is feasible. However, the operation is much slower than in the pixel shader, and provides an unsatisfactory performance gain over the CPU implementation.

We instead use a more complicated approach based on the OGL pixel-buffer-object (PBO) and vertex buffer object (VBO) extensions. The idea is to perform per-vertex shading but computed using the pixel engine. Because the NVidia 6800/7800 has 16 fragment processors, compared with 6 vertex processors, and also because texture fetching in the pixel shader is much more efficient than in the vertex shader, this approach gives us more than a 10× performance gain over the CPU implementation.


Preprocess CPU
circle log coefs $\boldsymbol{f}_l(\theta)$; a,b tables; $\boldsymbol{H}*\boldsymbol{L}$ as cube maps
First Pass CPU
positions; normals; cluster info; sphere info
Second Pass GPU
vertex colors
Final Pass GPU
rendered result
vertex positions; indexed triangle list

In a preprocessor, the log visibility coefficients $\boldsymbol{f}_l(\theta)$ and a,b tables of the OL method are prepared and passed into the shader. To handle local light sources, another 2D table of zonal harmonic (ZH) coefficients, $\boldsymbol{L}_H(\theta,\varphi)$, is also

[*] This work was done while Zhong Ren and Rui Wang were interns at Microsoft Research Asia
[†] This work was done while Bo Sun was an intern at Microsoft Research



prepared. At runtime, the first pass prepares vertex info (position, normal, and receiver cluster id) and sphere info (center and radius of the blocker spheres from the "cut" of the sphere tree hierarchy based on the bounding sphere of the receiver cluster, as well as the weight vector $\boldsymbol{w}[i]$ for each of these blocker spheres) on the CPU and passes them into the shader. In the second pass, a quad is rendered with the pixel shader on. Each pixel in the frame buffer corresponds to a vertex to be shaded. The result is then dumped from the frame buffer to the vertex buffer and used as a color array. The final pass simply invokes glDrawElements to draw all the triangles in the scene to be shaded. In summary, the first pass generates vertex and blocker information on the CPU, the second pass does the shading on the GPU, and the final pass renders the resulting shaded vertices to the screen, also on the GPU.

The second pass reads from a texture called the *vertex info map* which stores the position, normal and receiver cluster id for an array of pixels, each representing a vertex. Since each vertex in the same cluster shares the same sphere tree cut, the shader uses the cluster id to locate the corresponding row in the *sphere info map*, where the center, radius and weights of each sphere in the cluster's blocker cut are stored. In a "while" loop, each sphere i in the row, representing the cut list, is processed via equation (38) in the paper and the log visibility vector is accumulated. Dynamic branching is used here to avoid self shadow problems, following the rules in Section 6.2 of the paper. After accumulating all the blockers in the cut list, the shader evaluates the hybrid (HYB) algorithm, which combines the optimal linear evaluation with DC isolation and scaling/squaring to perform the SH exponential, yielding the visibility vector. A dot product with the pre-computed $\boldsymbol{L}_H(N)$ then yields the color for the pixel (really, the vertex).

To indicate when the while loop above should terminate, we store a "terminator sphere" at the end of sphere list having a radius of 0. The size of the sphere info map texture specifies the maximum number of blocker spheres we can store in any cut list. We use space for 128 or 256 blocker spheres in our examples.

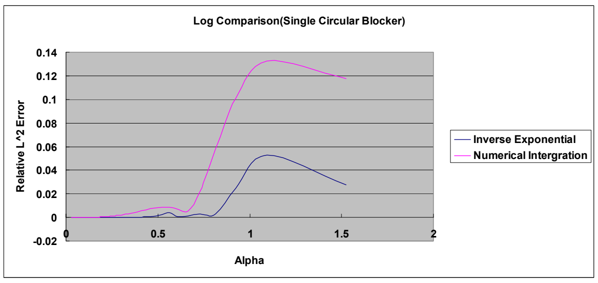
The PBO/VBO extensions of OGL are nothing more than a video memory management API, and are used in combination to eliminate a read-back from video memory to host memory between the second and final pass. The result of the second pass is transferred from the frame buffer to a PBO via a glReadPixels call. This performs a video-memory to video-memory copy which can be done very fast, over 300Hz for a 512×512 RGBA frame buffer. The PBO then becomes a VBO to make the shaded colors available as vertex information.

To be more precise, we use the following code sequence:

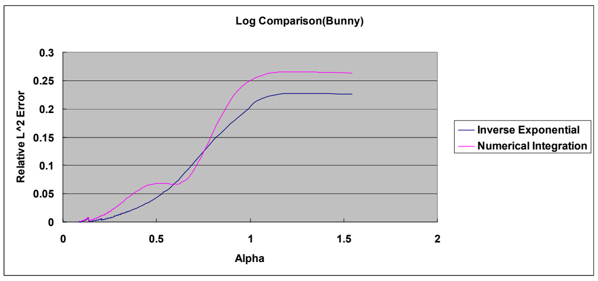
```
#define BUFFER_OFFSET(i) ((char*)NULL + (i))

glGenBuffersARB(1, &H);
// perform second pass
...
glBindBufferARB(GL_PIXEL_PACK_BUFFER_ARB, H); // bind H as target of glReadPixels
glReadPixels(0, 0, nRes, nRes, GL_RGB, GL_FLOAT, BUFFER_OFFSET(0));

// perform final pass
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glVertexPointer(…);
glBindBufferARB(GL_ARRAY_BUFFER, H);   // bind H as vertex source
```

```
glColorPointer(…, BUFFER_OFFSET(0));   // use H as vertex color source

glDrawElements(…);

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);

...
```

2 More Results



(a) single spherical blocker



(b) bunny blocker

Figure 1: SH Log method quantitative comparison. We plot relative RMS error for two examples: (a) a single spherical blocker, and (b) the bunny blocker with n_S=63 overlapping spheres, as a function of the angle subtended at the receiver point in radians. The numerical integration method uses equation (25) in the paper, with a threshold ε=0.02. The inverse exponential method uses equation (30), with the same threshold of ε=0.02. Both methods use PS*-9 to evaluate SH exponential and only differ in their method of computing log when constructing the table of circular blockers. Error is defined using an L^2 norm based on difference with product space rendering. The plots show the inverse exponential technique reduces error significantly.

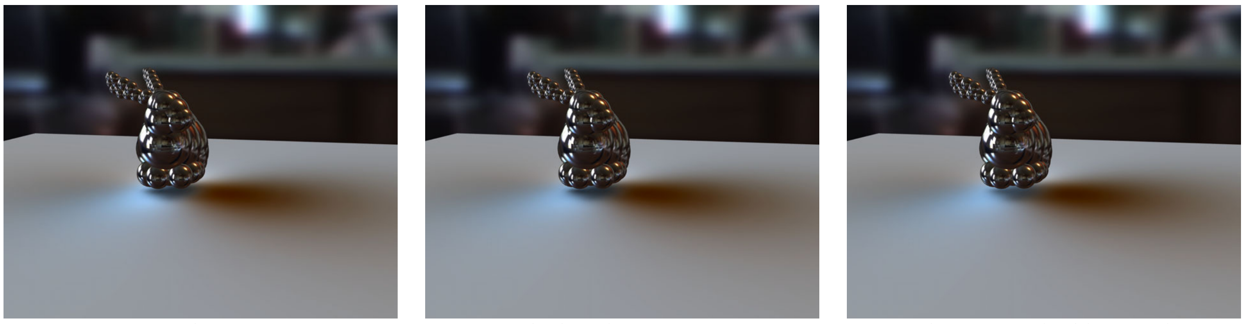

(a) product space (b) log=inverse exp (c) log=numerical integration

Figure 2: SH Log method visual comparison. (a) is accumulated in product space and forms the "ground truth" rendering. (b) and (c) both use log space accumulation, but with different methods for computing SH log (see previous figure). Computing log using numerical integration with clipping yields a more error, in the form of blur, compared to the inverse exponential method. Note the darkest part of the shadow directly underneath the bunny.

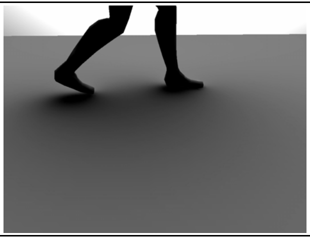
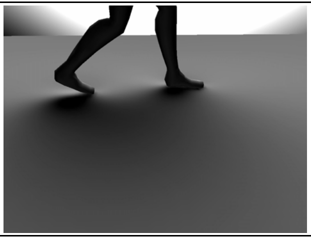
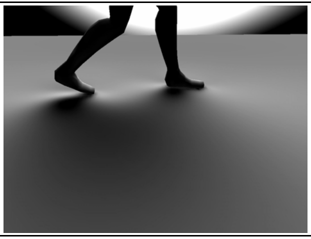
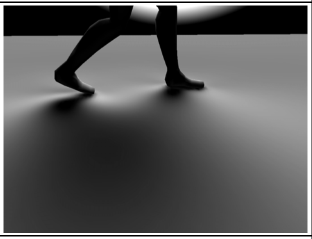
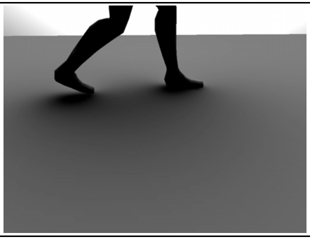
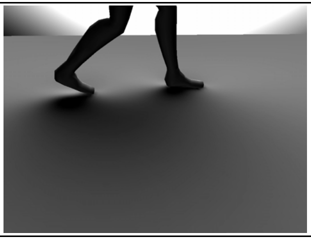
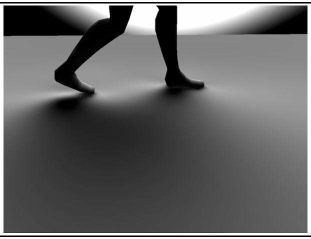
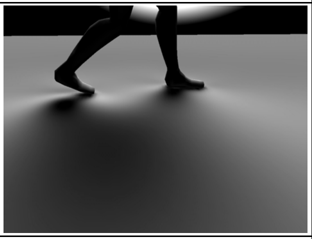
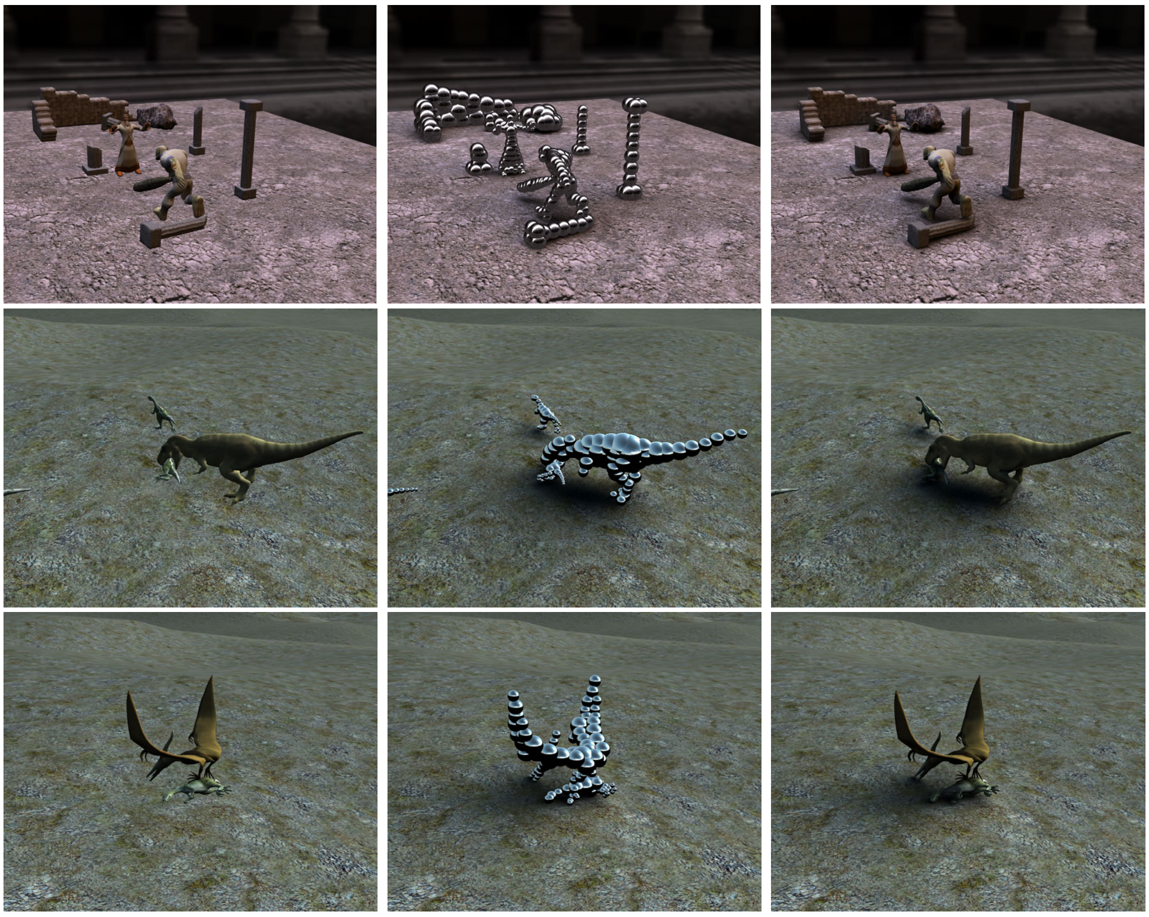
n=3	*n*=4	*n*=5	*n*=6
			
SH product space			
			
1.9x	3.4x	6.5x	11.0x
SH log space (using PS*-2 for exponentiation)			

Figure 3: SH order comparison, product space vs. log space. The lighting applied is as close to a delta function as allowed by the SH projection at each order *n*. No windowing is applied. The factors below the log space images are the total rendering speedup observed in a CPU implementation when accumulating blockers in log space rather than product space and using PS*-2 for exponentiation. Approximation error from doing the computation in log space is very difficult to see at any of these SH orders. Higher-order SH vectors allow "peakier" lights and sharper shadows, and also increase the acceleration obtained using log space instead of product space.



(a) no shadows (b) shadows, showing blocker approx (c) shadows, drawing meshes

Figure 4: GPU-rendered examples. SH exponential is evaluated using HYB. SH order is *n*=4.

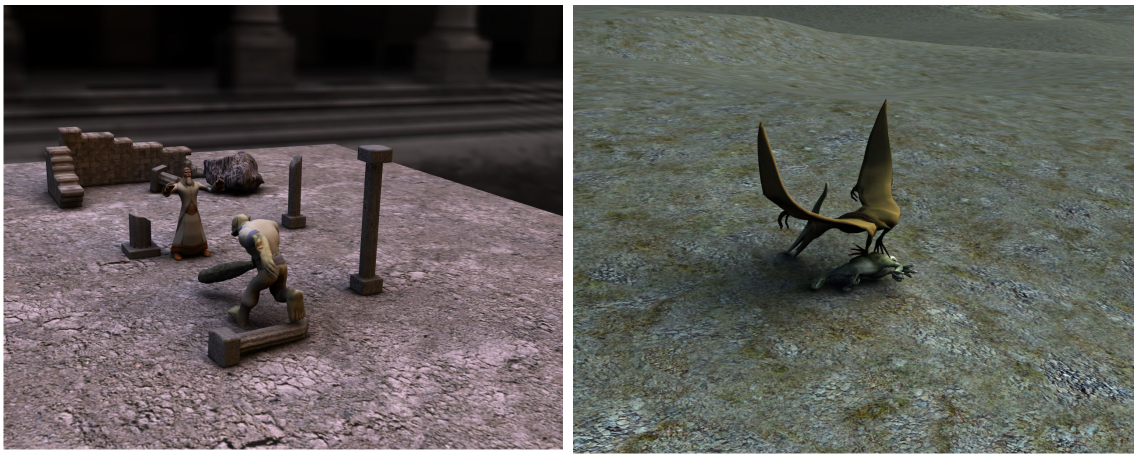


Figure 5: GPU-rendered examples zoomed up.

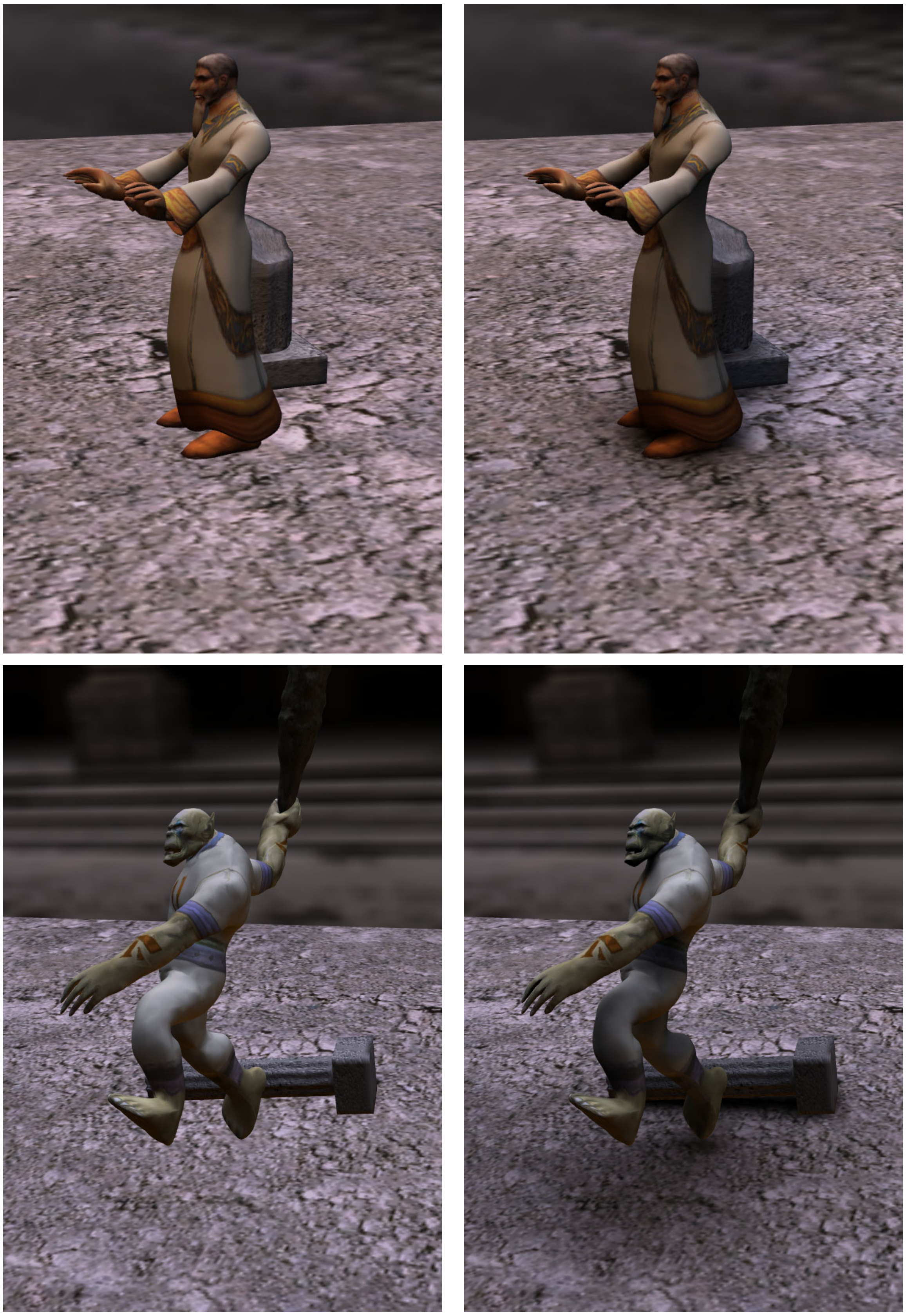
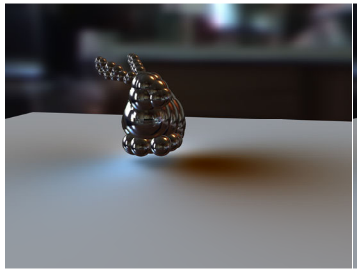
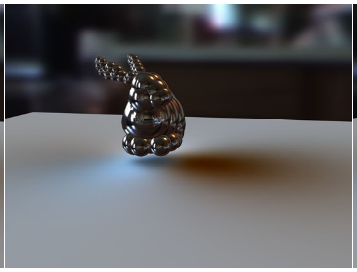
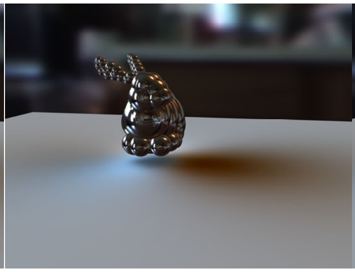
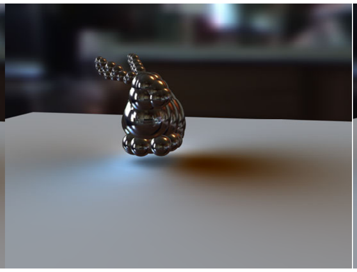


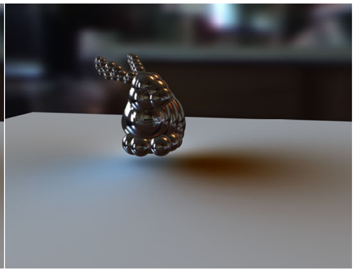
Figure 6: Self-shadowing examples, rendered on the GPU. Note the shadow under the wizard's beard and arms, and from his gown onto his feet. The troll's arm casts a shadow onto his body and his head onto his neck.

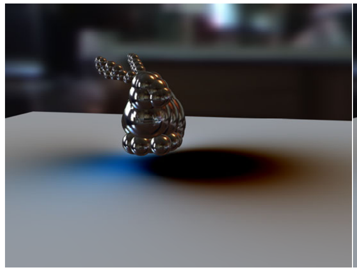

TRIP
RMS (L^2 Norm) = 0.00000
RMS (Shaded) = 0.000000
#SH squares: 0
#SH products: 494,790
@ 2.47 Hz

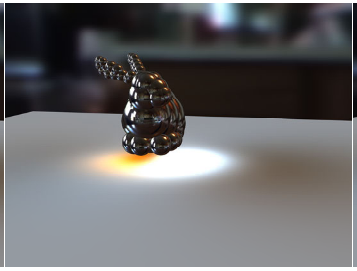

PS*-9
RMS (L^2 Norm) = 0.009122
RMS (Shaded) = 0.002740
#SH squares: 54,022
#SH products: 32,768
@ 3.98 Hz

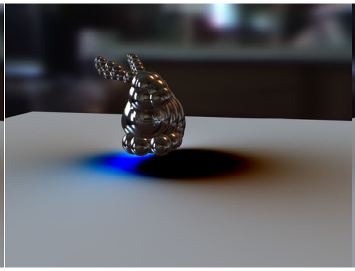

PS*-2
RMS (L^2 Norm) = 0.01009
RMS (Shaded) = 0.002875
#SH squares: 45,518
#SH products: 0
@ 7.53 Hz

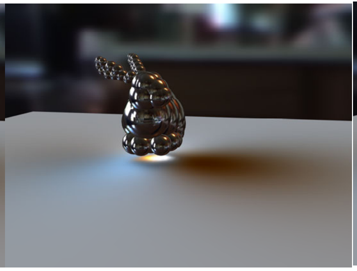

HYB (GPU)
RMS (L^2 Norm) = 0.009133
RMS (Shaded) = 0.002795
#SH squares: 30,784
#SH products: 0
@ 82.60 Hz

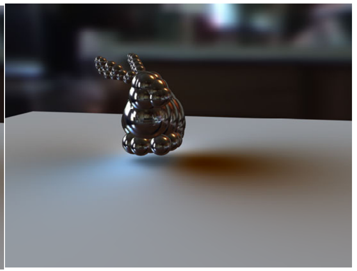

OL (GPU)
RMS (L^2 Norm) = 0.017135
RMS (Shaded) = 0.0241328
#SH squares: 0
#SH products: 0
@ 90.35 Hz


PS-2
RMS (L^2 Norm) = 0.085408
RMS (Shaded) = 0.222010
#SH squares: 0
#SH products: 0
@ 8.11 Hz

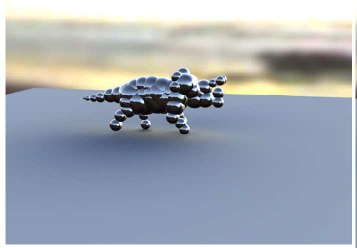

PS-3
RMS (L^2 Norm) = 0.095092
RMS (Shaded) = 0.371026
#SH squares: 0
#SH products: 16,384
@ 7.64 Hz

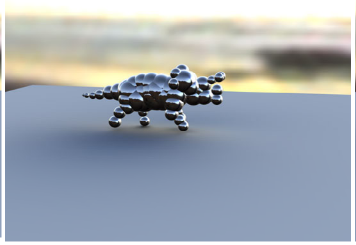

PS-4
RMS (L^2 Norm) = 0.144249
RMS (Shaded) = 0.750868
#SH squares: 0
#SH products: 49,152
@5.70 Hz

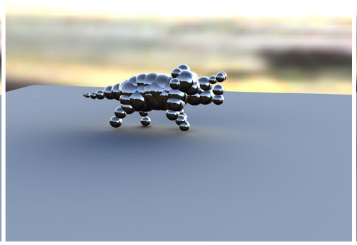

PS-21
RMS (L^2 Norm) = 0.0055160
RMS (Shaded) = 0.0046910
#SH squares: 0
#SH products: 327,680
@ 3.27 Hz

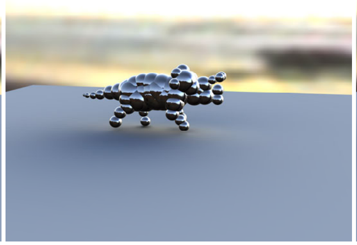

PS-30
RMS (L^2 Norm) = 0.002121
RMS (Shaded) = 0.009653
#SH squares: 0
#SH products: 458,752
@ 2.61 Hz

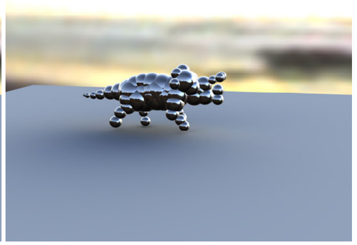
$Norm_{max}$	*16.50*
#Spheres	*63*
Receiver Plane	*128X128*
Env	*Kitchen*

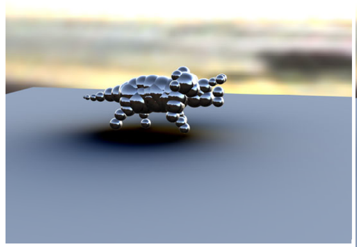

TRIP
RMS (L^2 Norm) = 0.00000
RMS (Shaded) = 0.000000
#SH squares: 0
#SH products: 570,962
@ 2.14 Hz

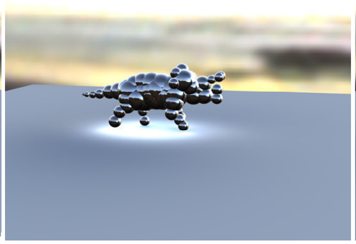

PS*-9
RMS (L^2 Norm) = 0.003261
RMS (Shaded) = 0.004354
#SH squares: 51,940
#SH products: 32,768
@ 3.78 Hz

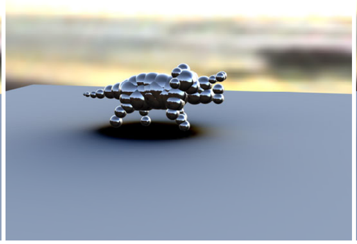

PS*-2
RMS (L^2 Norm) = 0.004136
RMS (Shaded) = 0.004436
#SH squares: 28,257
#SH products: 0
@ 6.85 Hz

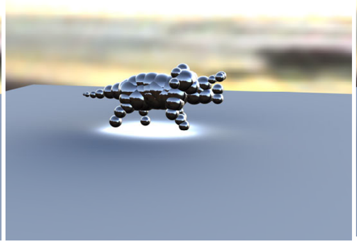

HYB (GPU)
RMS (L^2 Norm) = 0.004035
RMS (Shaded) = 0.004446
#SH squares: 18,849
#SH products: 0
@ 85.01 Hz

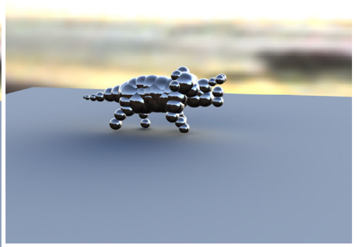

OL (GPU)
RMS (L^2 Norm) = 0.011595
RMS (Shaded) = 0.012560
#SH squares: 0
#SH products: 0
@ 92.30 Hz


PS-2
RMS (L^2 Norm) = 0.049708
RMS (Shaded) = 0.147138
#SH squares: 0
#SH products: 0
@ 7.88 Hz


PS-3
RMS (L^2 Norm) = 0.040657
RMS (Shaded) = 0.162911
#SH squares: 0
#SH products: 16,354
@ 6.57 Hz


PS-4
RMS (L^2 Norm) = 0.038439
RMS (Shaded) = 0.228313
#SH squares: 0
#SH products: 32,768
@ 5.88 Hz


PS-5
RMS (L^2 Norm) = 0.034559
RMS (Shaded) = 0.241002
#SH squares: 0
#SH products: 49,152
@ 5.01 Hz


PS-17
RMS (L^2 Norm) = 0.0029233
RMS (Shaded) = 0.00583762
#SH squares: 0
#SH products: 245,760
@ 3.05 Hz

$Norm_{max}$	*12.76*
#Spheres	*63*
Receiver Plane	*128X128*
Env	*Beach*