A Programmable System for Artistic Volumetric Lighting: Supplemental Document

Derek Nowrouzezahrai[1] Jared Johnson[2] Andrew Selle[2] Dylan Lacewell[2] Michael Kaschalk[2] Wojciech Jarosz[1]

[1]Disney Research Zürich [2]Walt Disney Animation Studios

Abstract

This supplemental document includes RenderMan™ shader code for our shading model, illustrations corresponding to the *Target Matching* experiment discussed in Section 8.2 of the paper, as well as a schematic comparing the geometric setups of the old (without beams) and final (with beams) keyhole animation sequences.

1 RenderMan™ Shader Code

The shader code listed below can be used to generate physically-based beam shading. Coupled with the beam data provided in the supplemental file, this shader can be used to generate the scattering-only results from the first image of Figure 4 in the paper. In order to match our results, a glass shader must be applied to the bumpy sphere geometry, in conjunction with our shader and the provided beam data.

2 Torus Target Matching

Figure 2 illustrates three animation sequences of a torus, immersed in a volumetric medium, rotating over a blue area light source. The top animation sequence was created by an artist familiar with our system. The bottom two rows of the figure are animation sequences generated by a different artist, unfamiliar with our system, first using traditional animation and rendering techniques and then using our system.

The artist was asked to comment on the experience of replicating this simple animation with and without our system. Section 8.2 in our paper describes the artist's primarily positive feedback regarding the contrast between using our system and using the alternative, more complex and indirect methods used to generate the middle row results.

3 Keyhole Schematics

Figure 1 visualizes the emissive surface geometry used in the original keyhole sequence, and beams generated using a physical simulation in the final keyhole sequence. The former required a tedious, manual placement and animation of emissive surfaces, whereas artists simply "sprinkled" beam seed points along the door cracks and used standard particle simulation tools to evolve the beams over time.

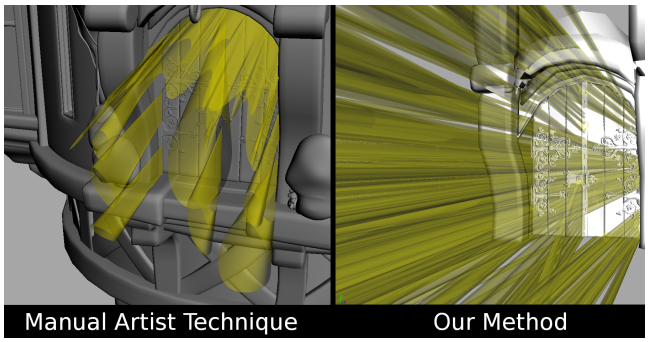


Figure 1: *Manual placement and animation of emissive surface geometry for the original keyhole sequence (left), compared to automatically simulated beam geometry used in the final keyhole sequence (right).*

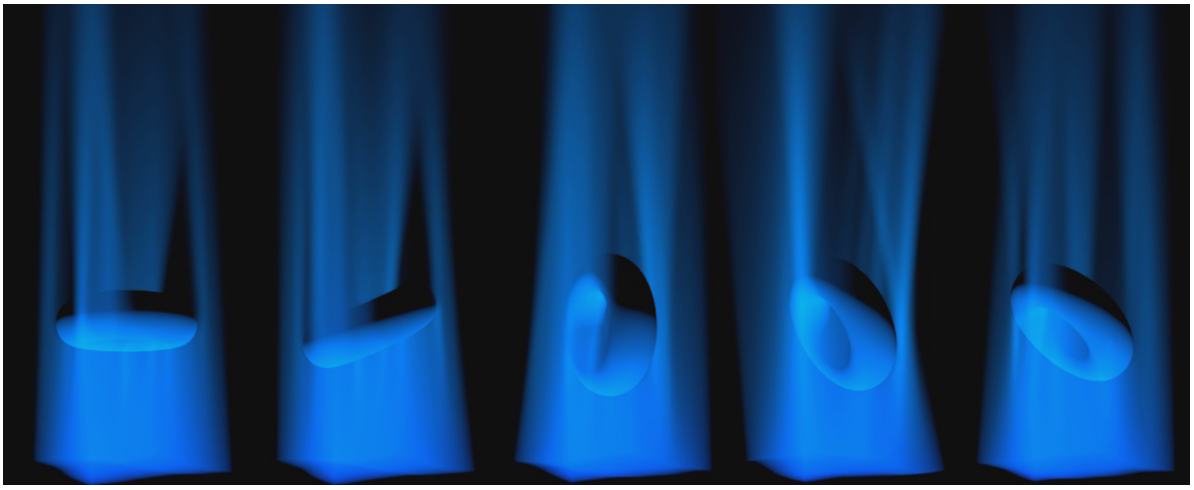
4 Heterogeneous Media

For physically-based lighting, photon beams can easily handle heterogeneous media as well. To accomplish this, the attenuation along each beam f_b could be computed using a 1D texture (see Figure 4 of the paper) that stores the accumulated optical depth. The attenuation towards the eye f_e would also need to compute the transmittance, which could be accomplished using a deep shadow map. Automatic setting of physical scattering parameters in heterogeneous media is an interesting area of future work.

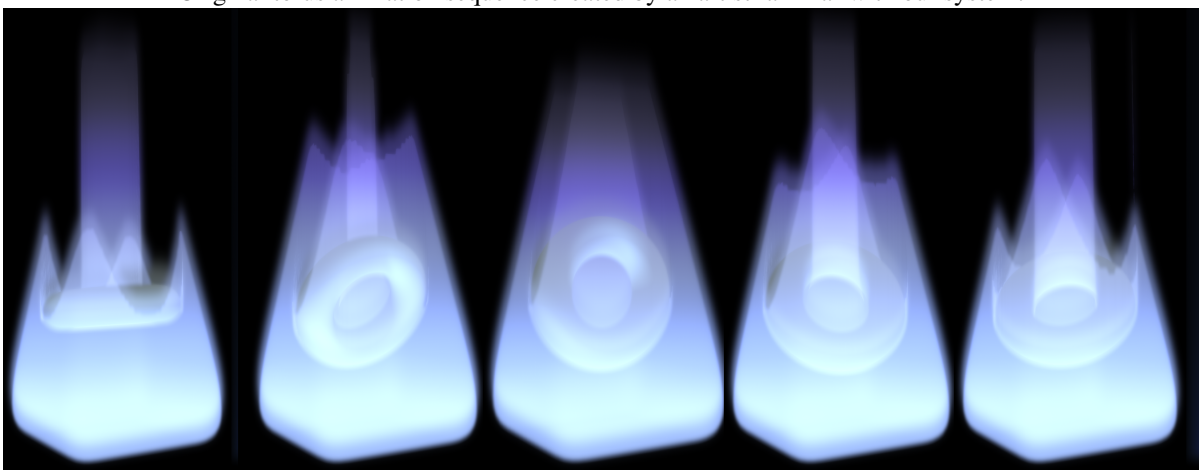
As seen in Figure 4 in the paper, our artistic abstraction provides even more flexibility to incorporate variation along the length of the beams and towards the camera. Such spatial variations can be injected using procedural functions such as Perlin noise or values stored in voxel data. To obtain physically-plausible results, however, it may be desirable to consider two aspects of physical lighting:

- evaluating the attenuation functions based on world-space locations and distances (and not just parametric distances) gives the sense that these details are due to heterogeneous density in the scene and not attached to the beams,
- the attenuation functions should be monotonically decreasing along the beams and monotonically increasing towards the camera.

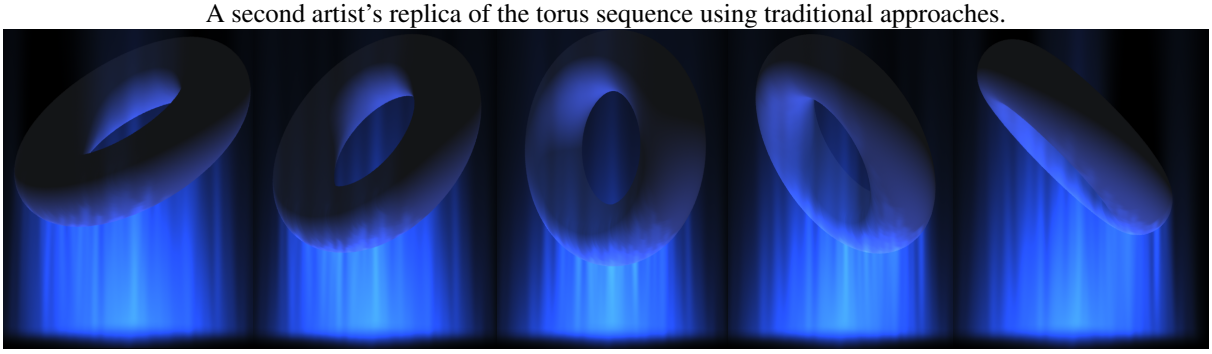
```
/*
 * BEAM SHADER SOFTWARE
 * Copyright 2011 Disney Enterprises, Inc. All rights reserved
 *
 * Redistribution and use in source and binary forms, with or without modification, are permitted
 * provided that the following conditions are met:
 *
 * * Redistributions of source code must retain the above copyright notice, this list of conditions
 *   and the following disclaimer.
 *
 * * Redistributions in binary form must reproduce the above copyright notice, this list of
 *   conditions and the following disclaimer in the documentation and/or other materials
 *   provided with the distribution.
 *
 * * The names "Disney", "Walt Disney Pictures", "Walt Disney Animation Studios" or the names
 *   of its contributors may NOT be used to endorse or promote products derived from this
 *   software without specific prior written permission from Walt Disney Pictures.
 *
 * Disclaimer: THIS SOFTWARE IS PROVIDED BY WALT DISNEY PICTURES AND CONTRIBUTORS "AS IS" AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT AND TITLE ARE DISCLAIMED.
 *
 * IN NO EVENT SHALL WALT DISNEY PICTURES, THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR
 * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
 * BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND BASED ON ANY THEORY OF LIABILITY, WHETHER
 * IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN  IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
 */
class beamshader
(
  color sigma_s = 0.1;  // scattering coefficient
  color sigma_a = 0.1;  // absorption coefficient
  varying color power=0; // beam power
  varying float width=-1; // beam width (redundant for RICurves)
  uniform float beamLength=0; // beam length
  uniform float beamDir[]=; // beam direction
  float boost = 1; // constant linear scaling of intensity
)
{
  constant float phase = 1.0/(4*PI);
  constant float EPSILON = 1e-6;

  float kernelBiweight(float y) {
    float x = clamp(y,0.0,1.0);
    return (15.0/16.0)*(1-x*x)*(1-x*x);
  }

  // Pre-integrated biweight kernel function
  float kernelBiweight(float u; float du) {
    float s = clamp(2*u-1-du,-1.0,1.0);
    float t = clamp(2*u-1+du,-1.0,1.0);
    float s3 = s*s*s, t3 = t*t*t, s5 = s3*s*s, t5 = t3*t*t;
    return (15.0/16.0)*(t5/5.0 - 2.0*t3/3.0 + t - s5/5.0 + 2.0*s3/3.0 - s)/(t-s);
  }

  color transmittance(float distance) {
    color sigma_t = sigma_s + sigma_a;
    return color(exp(-comp(sigma_t,0)*distance), exp(-comp(sigma_t,1)*distance), exp(-comp(sigma_t,2)*distance));
  }
  public void surface(output color Ci,Oi) {
    float eyeDist = length(I);
    float sampleDist = beamLength * v;
    vector Inorm = I / eyeDist;
    vector beamDirnorm = normalize(vector "world" (beamDir[0], beamDir[1], beamDir[2]));

    // Beam blur across width and beam ends
    float blur = kernelBiweight(u, du);
    float distancefromend = beamLength - sampleDist;
    if( distancefromend < width ) blur *= kernelBiweight(1.0 - distancefromend / width);
    if( sampleDist < width ) blur *= kernelBiweight(1.0 - sampleDist / width);

    // Inverse sine computation
    float cos_e_b = beamDirnorm . Inorm;
    float invSin = sqrt(1.0 - cos_e_b * cos_e_b);
    invSin = 1.0 / max(EPSILON, invSin);
    Ci = transmittance(eyeDist) * transmittance(sampleDist) * invSin;
    Ci *= boost * power * phase * sigma_s * 2.0 * blur / width;
    Ci = clamp( Ci, color(0,0,0), color(1.0,1.0,1.0) );
  }
}
```



Original torus animation sequence created by an artist familiar with our system.



A second artist's replica of the torus sequence using traditional approaches.



The second artist's replica of the torus sequence using our system.

Figure 2: *An artist unfamiliar with the beam analogy or our system was asked to replicate the animation sequence in the top row using traditional animation techniques (middle row) and then using our system (bottom row).*