Volumetric Shadows using Polygonal Light Volumes

Supplemental Material

Markus Billeter, Erik Sintorn, and Ulf Assarson

February 22, 2012

Abstract

This document provides supplemental material to simplify implementing volumetric shadows using our technique described in the paper *Volumetric Shadows using Polygonal Light Volumes [BSA10]*.

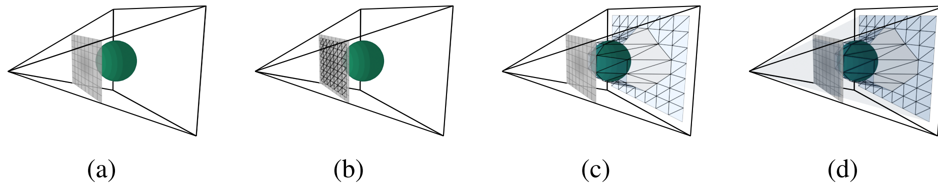
1 Implementation Details

To summarize, our shafts of light algorithm consists of the following steps:

1. Create the shadow map by rendering the scene into a depth buffer (see Figure 1).
2. Render scene from the camera with diffuse lighting, attenuating incoming and outgoing light due to absorption and scattering in media (see code listing below). Hard shadows on surfaces can be added in this step using standard shadow mapping.
3. Construct the mesh from the shadow map, with or without adaptive tesselation. The simplest is to skip adaptive tessellation, which is fast as long as the shadow map resolution is not very large (less than $4k \times 4k$).
4. Render the mesh with depth testing disabled and additive blending enabled. The fragment shader evaluates the in-scattering, including attenuation, for a non-shadowed ray from the eye to the fragment's position. If fragment belongs to a back-facing mesh polygon, the contribution is negated (see code listing below).

Since there are several options for the implementation of the in-scattering and attenuation computations, we here include the fragment shaders associated with step 2 and 4.

```
// Step 2:
// In function main() of the fragment shader, when rendering the
// scene from the camera:
```



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

Figure 1: *Creation of the shadow volume mesh from the shadow map. (a) First, the shadow map is rendered. (b) Then, the mesh is constructed with one vertex per shadow map pixel. The mesh triangles should face the light source. In (c), the mesh vertices are displaced by the depth values in the shadow map so that each vertex is the corresponding shadow map pixel's world space coordinate. Finally, in (d) the mesh is closed by the four light frustum planes.*

```
for(int i=0; i<NUM_LIGHTS; i++) {
    // Compute fragment color from surface shading, e.g.  using
    // the standard ambient, diffuse, specular shading model
    vec3f color_from_shading = ...
    // Compute light attenuation factor
    vec3 lightToObj = lightposition[i] - objPos;
    float beta = 0.04; // optical thickness
    float attenuation = 0.1 *
        exp(-beta * length(lightToObj))/ length(lightToObj)^2 *
        exp(-beta * length(objPos));
    // lightintensity = 1000.0f is a reasonable value
    gl_FragData[0].xyz += attenuation * lightintensity[i] *
        lightcolor[i] * color_from_shading;
```

While the light is attenuated on its way to the surface point, the surface point also receives in-scattering enhancing the light intensity. This latter term is, however, as computationally expensive to compute per pixel as the full in-scattering towards the eye for the whole image. Therefore, a plausible approximation is to assume that the attenuation from the light to the surface point is low per unit step compared to the attenuation from the point to the eye.

```
// Step 4:  Computing airlight with shadows.
// Render the shadow volume mesh with depth testing disabled
// and additive blending enabled.
// Fragment shader:
uniform sampler2D camera_z;
in vec3 objPos, lightPos, viewPos;
void main()
{
    float facing = gl_FrontFacing ?  -1.0 :  1.0;
```

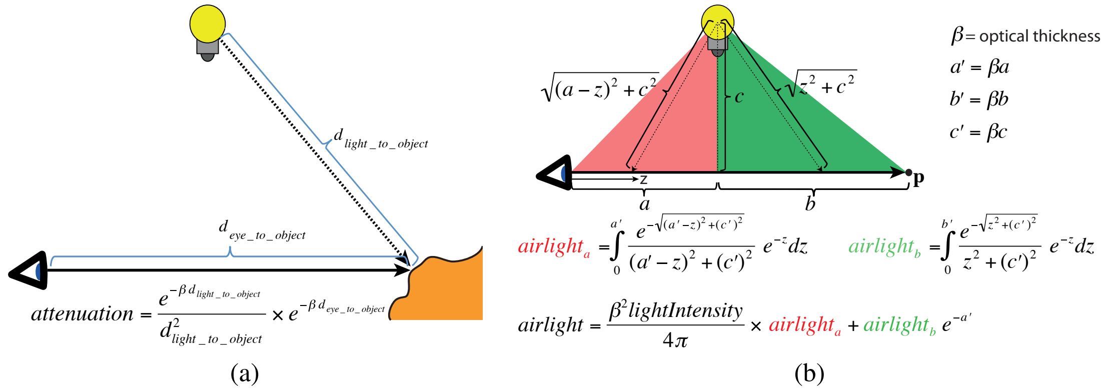


Figure 2: *(a) Illustration of the light attenuation factor used after the lighting computations for standard surface shading (see fragment shader of step 2). (b) Computation of airlight contribution (in-scattering and attenuation) along an unoccluded ray, from the view point to a certain position* **p** *(see the airlight()-function of the fragment shader for step 4). Since* airlight_a *and* airlight_b *only depend on* a', b' *and* c', *they are precomputed into a 2D-texture and looked up in run time by the fragment shader of step 4. The contribution is split into two parts to simplify storage in the lookup table. The first part (red) is between the eye up to the point of projection of the light onto the view ray, and the second part is the remaining distance up to the point* **p**. *The airlight for each part is the integration for each z-value of the attenuated light reaching* z, *scaled by the square of the distance to the light source and then attenuated by the distance* z *to the eye (see formulas for* airlight_a *and* airlight_b*). Since* a *and* b *could be negative – if the projection of the light source is behind the eye or beyond* **p** *– the fragment shader deals with these two special cases as well. The total airlight for the ray, is the sum of the airlight contribution for* a *and* b, *where the latter is also scaled by the attenuation for distance* a' *to the eye. The total is also multiplied by the phase function* $1/4\pi$ *and the optical thickness* β *and finally multiplied by* $\beta\times$ lightIntensity. *The lookup table is downloadable at: http://www.cse.chalmers.se/~billeter/pub/volumetric/.*

```
    vec3 op = objPos; // mesh's fragment position
    vec3 myz; // z-value in depth buffer (scene-fragment)
    myz = texelFetch2D(camera_z, ivec2(gl_FragCoord.xy),0).rgb;
    op = (myz.z >= op.z) ?  myz :  op; // keep z closest to eye
    float ai = facing * airlight(viewPos-lightPos, op-lightPos);
    gl_FragData[0].xyz = ai * light_color;
}
```

The function `airlight()` returns the amount of airlight (inscattering including attenuation) for an unshadowed ray between the eye and the mesh fragment, where the latter is clamped to the scene fragment if that is closer to the eye than the mesh fragment. Figure 2 illustrates the computation of the airlight. Here follows the fragment shader code:

```
// Step 4:  fragment shader continued...
uniform sampler2D LUT; // airlight lookup table
float map_x(float t)
{
    return sign(t)*((log(abs(t))+16.1181)/17.9099)/2+0.5;
}
float map_y(float t)
{
    return (log(abs(t))+16.1181)/17.9099;
}
vec2 airlight_components(float tao, float tbo, float tlo)
{
    // The lookup table (LUT) is downloadable from here:
    // http://www.cse.chalmers.se/ billeter/pub/volumetric/airlight-
    // lookup-loglog-premult-1024x512.gz
    float at = texture2D(LUT, vec2(map_x(tao), map_y(tlo))).r;
    float bt = texture2D(LUT, vec2(map_x(-tbo),map_y(tlo))).r;
    return vec2(at,bt)
}
float airlight(vec3 viewPos, vec3 objPos)
{
    vec3 v=-viewPos;
    vec3 d = -viewPos + objPos;

    float dao = dot(v, normalize(d));
    float dbo = length(d) - dao;
    float dlo = sqrt( dot(v,v) - dao*dao );

    float beta = 0.04; // optical thickness
    float tao = beta * dao;
    float tbo = beta * dbo;
    float tlo = beta * dlo;

    vec2 ab = airlight_components( tao, tbo, tlo );

    float ae = 1, be = 1;
    if( dao > 0 && dbo < 0 )
        be = exp( -beta * length(d) );
    else if( dao > 0 && dbo > 0 )
        be = exp( -tao );
    else if( dao < 0 && dbo > 0 )
        ae = be = exp( -tao );
    else
        ae = be = -1000000;

    float abc = sign(tao) * ab.x * ae + sign(tbo) * ab.y * be;
```

```
    // lightintensity = 1000.0f is a reasonable value.  The
    // division by tlo is because we premultiply the lookup
    // table by tlo to get a better range of precission.
    float ret = beta*beta*lightIntensity / (4*PI) * abc / tlo;
    return clamp( ret, 0, 1e7 );
}
```

Figure 3: *Here is the result, showing the Sibenik-scene with real-time volumetric shadows in homogeneous participating media. This scene uses two light sources (~90 fps, resolution: 1280x1024, Geforce GTX480).*

References

[BSA10] Markus Billeter, Erik Sintorn, and Ulf Assarson. Volumetric shadows using polygonal light volumes. In *Proceedings of High Performance Graphics 2010*, pages 39–45, June 2010.