CS562 Final Project Report

Real Time Volumetric Shadows

Alex Demirci Nijmeijer

DigiPen Institute of Technology Europe-Bilbao

Abstract

Volumetric lighting is an effect that allows to perceive the beam of a light source. It can help with adding depth and dramatism to a scene, overhauling the overall image.

This report is about the implementation of a paper called Real Time Volumetric Shadows using Polygonal Light Volumes by Markus Billeter, Erik Sintorn and Ulf Assarsson.

This report will discuss what is the problem to solve, how the paper solves it, my solution, how other papers approach this effect and any improvements that can be made.

1. Introduction

Volumetric lighting is mainly caused by light scattering, according to B. Kruppa & G. Strube (1994: 157), a term referring to physical processes involving the interaction of light and matter. It is because of such interactions that the beam of a light can be physically seen. This beam can help both artistically and narratively in both movies and videogames. Volumetric lighting can add depth to a scene, and it can also help to direct the eye towards a specific position on the screen.

This type of effect has existed for a long time in computer graphics and has been used an infinite number of times in animated movies, however, these accurate simulations are usually computed using path tracing or similar techniques which are, as for now, not suitable for real time applications.

It is possible to solve such problem using a simpler model that only considers single scattering in a homogeneous medium, which is accurate enough to be able to run at real time framerates.

2. Solution Proposed

The solution's objective is to compute the amount of light from a point towards an observer in a homogeneous medium, that is, from a light towards the camera in a medium that stays the same.

This amount of light, called airlight, is given by the following integral [MBSA10].

$$La(d0, d1) = \int_{d0}^{d1} \beta k(\alpha) \frac{I_0 e^{\beta d(x)}}{d(x)^2} e^{-\beta x} dx \quad (1)$$

This equation is solved in [SRNN05]. β symbolizes the scattering coefficient of the medium and I_0 the radiant intensity of the light source. What equation 1 determines is the airlight along a segment between the points d0 and d1.

The model used does not consider airlight in non-lit areas (areas occluded by an object) and hence only lit areas contribute to the total airlight. It is then possible to express the total airlight as a sum.

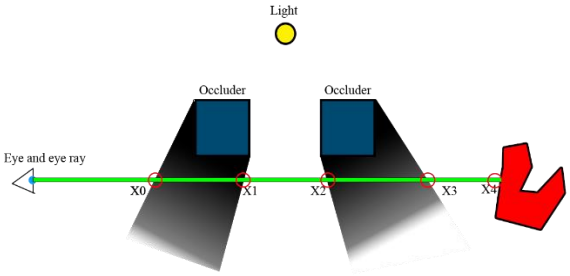


Figure 1: *Example of a scene where two occluders generate shadows that are intersected by the view ray. The view ray starts at x = 0 and enters an unlit region at x = x0, and x = x2, enters a lit region at x = x1 and x = x3 and intersects an object at x = x4.*

As mentioned earlier, only lit regions contribute to the total airlight. Hence, taking the example in Figure 1:

$$LaTotal = La(0, x0) + La(x1, x2) + La(x3, x4).$$

It quickly becomes apparent that detecting all these regions is not trivial and if done can reduce performance [MBSA10].

Using the Additivity of intervals in integrals property, it is possible to split the integral between two points A and B into two different ones at a new point C. It is then possible to rewrite the equation above as:

$$LaTotal = La(0, x0) - La(0, x1) + La(0, x2) - La(0, x3) + La(0, x4).$$

At this stage, all computations use the eye's position and the intersection with a boundary along the view ray. It is important to note how every transition from an unlit to a lit region is subtracted and every transition from a lit to an unlit region is added. This will be represented by the parameter Sn.

Combining everything together the total airlight along a single view ray is [MBSA10]:

$$LaTotal = \sum_0^n Sn\ La(0, x_n) \qquad (2)$$

As mentioned before, Sn will be 1 if going from a lit to an unlit region and -1 if going from an unlit to a lit region.

As the paper suggests, creating an enclosed volume will make the computation of the total airlight extremely simple. The idea is to use shadow maps to generate a mesh grid that matches the resolution of the shadow map. Then the shadow map depths are read, and the mesh vertices are displaced by such depth. Finally, the border edges of the mesh are connected to the light source, creating a directly illuminated volume for it. All of this is done in the GPU following McCool's implementation [McC00].

Once the directly illuminated volume is generated, it is possible to render it and compute the total airlight. Each fragment shader call would represent an eye-ray intersection with the boundaries. Each fragment will compute a term in equation 2 and Sn will be determined by front facing and back facing triangles. Front facing triangles will represent that the ray is entering a lit region and Sn will be -1, and vice versa.

The total sum of the equation is computed using additive blending which allows to sum all the results along the view ray. As the paper mentions, it is very desirable to render the mesh with depth testing disabled, but because of inaccuracies of the shadow map, artifacts will appear [MBSA10]. This happens because some fragments will be inside the scene geometry and some other not.

This can easily be fixed by disabling depth testing and passing the depth buffer to the fragment shader and basically clamping the values to the scene's depth and then processing the fragment as usual. This means that if the current fragment is beyond the maximum stored depth in the depth buffer, it will be set to have that depth.

In addition to that, the paper suggests to use Adaptive Tessellation. This is because the algorithm's performance depends only on the resolution of the shadow map. The higher the resolution, the more vertices are generated for the mesh. Flat surfaces do not require of a

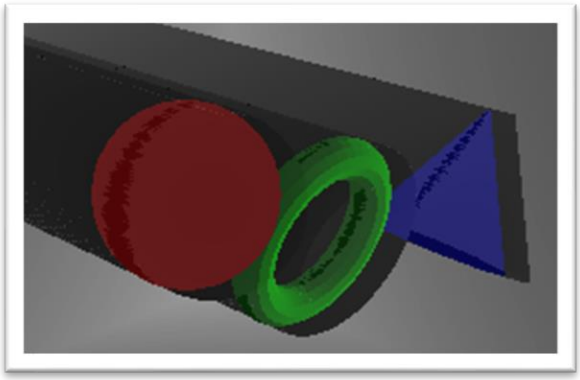


Figure 2: *Surface acne produced by the innacuracies of a shadow map with a resolution of 512x512. It is possible to see how some of the fragments are inside the primitives producing this so called "surface acne". Increasing the shadow map resolution mitigates the artifact but does not make it disappear.*

large amount of vertices whereas places in the mesh where there is a lot of variance in depth and height do. Tessellating the mesh will reduce the number of vertices significantly, improving the overall performance.

3. My Implementation

In this case, the volumetric shadows were implemented exclusively for directional lights. A shadow map needs to be generated with a square resolution and an orthogonal projection. The orthographic projection takes 4 parameters: left, right, top and bottom.

These four values need to be the same in order to form a completely square projection. This value will be referred as orthogonal scale.

Generating the shadow map is extremely straightforward, in the deferred shading pipeline, the scene needs to be drawn from the lights point of view and stored as a texture with the depth values.

3.1 Generating the geometry on the CPU

In order to generate the geometry, what was done was to set an initial position at the point (-1, 1, 0) and compute the distance each points need to have with each other to be evenly spaced in the grid. As the mesh, in model coordinates, goes from -1 to 1 in both x and y axis, the total distance is 2. Dividing 2 by the resolution on either axis results in the mentioned distance.

$$Distance = \frac{2}{resolution}$$

Once the distance between the points is obtained, it is possible to loop through rows and columns and offset the initial point to fill the grid.

```
for (int row = 0; row < resolution.y; ++row)
{
	for (int col = 0; col < resolution.x; ++col)
	{

		glm::vec3 point = initPos;
		point.x += (col * distanceBetweenPoints);
		point.y -= (row * distanceBetweenPoints);
		mPoints[(size_t)columns * (size_t)row + (size_t)col] = point;
	}
}
```

As it is possible to see, the push_back() function of vectors is not used, this is because this function allocates memory every time its capacity is reached, hence, all of this memory is allocated before hand and then the array is accessed via the subscript operator for efficiency. The resizing is the multiplication of the width and height of the resolution + 1. This extra space in the array is where the lights position is stored in order to connect the border edges of the mesh later.

Generating the grid in a brute force manner will have a huge impact on performance and how memory is used because of the duplication of triangles. That is why element buffer objects were used in order to reduce the duplication of points and give a boost to performance.

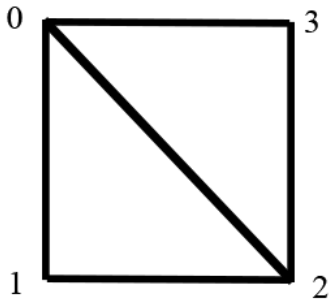


Figure 3: *Illustration on how vertex indices are stored. Not using element buffer objects would increase the number of vertices the model uses. Both triangles use vertex 0 and 2 and it is then tempting to generate the left triangle with 3 vertices and right triangle with 3 more vertices, with a total of 6. It is possible to reduce this to only 4. In the case of a 4x4 grid not using EBOs would yield 54 vertices and using them only 16. It is possible to see how quickly this can scale with larger resolution grids.*

Once the vertices and indices are computed and stored, they are sent to the GPU. It is important to note that only the position attribute is used as no other type of attributes such as UVs or normals are required.

```
// upload the vertices
glBindBuffer(GL_ARRAY_BUFFER, mVbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3) * mPoints.size(),
reinterpret_cast<void*>(mPoints.data()), GL_STATIC_DRAW);

// upload the indices
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mEbo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned) * mElements.size(),
mElements.data() , GL_STATIC_DRAW);

// set the attribute for positions only
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(glm::vec3), (void*)0);
```

In terms of matrix transformations, the model to world matrix is computed in a very simple way. For translation, the plane is placed at the light's position, for rotation, it is rotated to face the same direction as the light and for scale, it is scaled by the orthogonal scale in both x and y and by Far – Near for the z.

3.2 Geometry Displacement

Displacing the mesh is done every frame, because otherwise, the mesh would not adapt to dynamic environments (which could be desirable in some cases). The OpenGL function glReadPixels is used to read what is stored in the light's shadow map. These values are stored in a vector and then fed to the model to update its vertex z position.

The depth values read range from [0, 1] where 0 is at the light's near plane and 1 at the light's far plane. This is the reason why the z component of the scale matrix is the subtraction of the Far and Near distances. A depth of 1 in the shadow map indicates that the point corresponding to that pixel on the grid is at z = Far and the same applies for the Near plane.

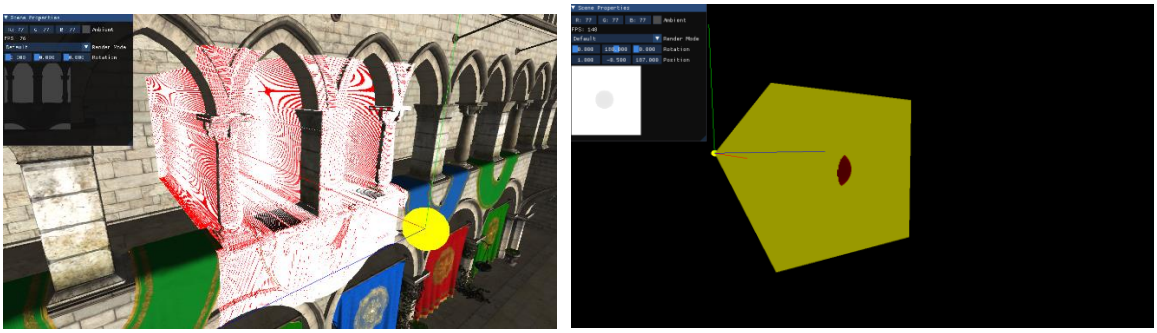
Reading the pixels from the shadow map is something to consider in terms of performance, as this function will make the pipeline stall until the frame is done and the amount of time reading increases the higher the resolution of the texture is.

```
std::vector<float> imageData((size_t)mShadowRes.x * (size_t)mShadowRes.y);

glPixelStorei(GL_PACK_ALIGNMENT, 1);

glReadPixels(0, 0, mShadowRes.x, mShadowRes.y, GL_DEPTH_COMPONENT
            , GL_FLOAT, imageData.data());
```

Updating the vertices depths component consists of iterating in a loop and assigning the newly obtained values. Once the new depths are set, it is the time to connect the border edges to the light's position. In order to do that it is necessary to go through the top, bottom, left and right of the plane and add the new indices to the array. For connecting the top plane, it is necessary to go through the first row and all columns of the grid and



(a) (b)

Figure 4: *In (a) a visualization of a 512x512 mesh grid facing in the direction of the light and displaced by the z values from the shadow map in Sponza and in (b) the final directly illuminated volume with a yellow shading and enclosing a sphere.*

connect the vertices in a counterclockwise order. The same applies for the left, right and bottom planes but of course connecting vertices from all rows of the first column, all rows of the last column and all columns of the last row respectively.

3.3 Shading

At this point all is left to do is to render the volume and apply some shade to it. Just applying some translucency generates somewhat interesting results.

In order to render the mesh correctly, it is done after all standard passes, that is, the geometry pass, lighting pass, decal pass, etc. Depth testing and face culling are disabled, and additive blending is used in order to accumulate the results.

The vertex shader is straightforward, in this case it transforms the vertices to clip space, view space and light view space. These las two are then sent to the fragment shader to be interpolated.

```
gl_Position = (uP * uV * uM) * vec4(aPos.x, aPos.y, aPos.z, 1.0);

// vertex position in view space
boundPos_cam = vec3(uV  * uM * vec4(aPos.x, aPos.y, aPos.z, 1.0));

// vertex position in light view space
boundPos_light = vec3(uV_light  * uM * vec4(aPos.x, aPos.y, aPos.z, 1.0));
```

The solution of equation 1 solved in [SRNN05] is not used, as convincing effects can be achieved in other ways without making so many computations.

The fragment shader receives the interpolated vertex positions in both view space and light view space and a texture containing the depth information of the scene (other parameters are also passed such as color, matrices, etc.).

The final goal of the fragment shader is to compute the airlight at the current intersection, but first, the scene's depth buffer must be sampled in order to clamp the intersection z value in case it is beyond of what the buffer has stored. To do that, gl_FragCoord is used to determine which UVs the mesh occupies in screen space. Once that is known, the depth buffer is sampled. In order to do comparisons both depths need to be in the same space. As the fragment depth is interpolated in view space, the read depth must also be converted into view space. For that, this depth that ranges from [0, 1] is converted to NDC space where it ranges [-1, 1] and then multiplied by the inverse of the projection matrix and applying perspective division in order to bring it to the correct space.

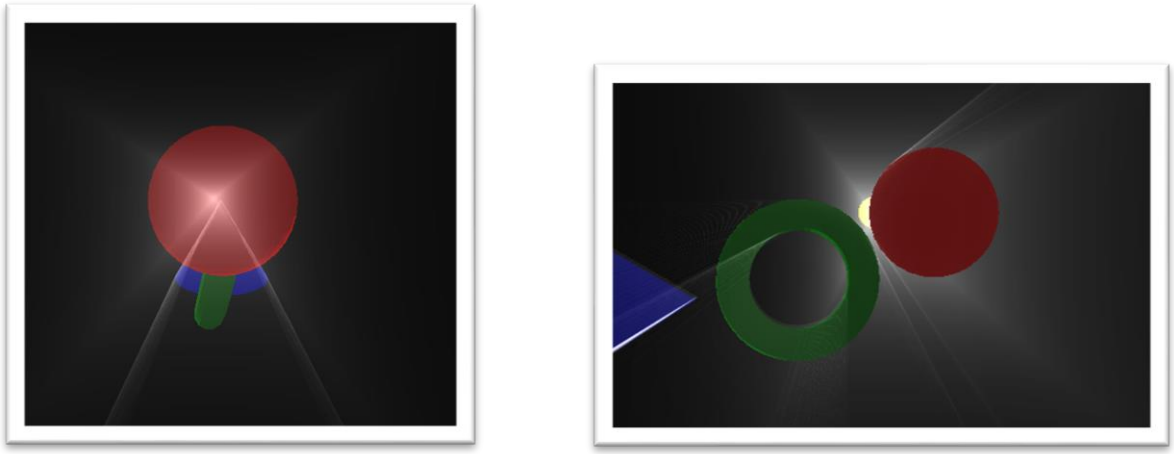
```
// read the depth buffer
vec2 screenSize = textureSize(uDepthMap, 0);
vec2 newUvs = vec2(gl_FragCoord.x / screenSize.x, gl_FragCoord.y / screenSize.y);

float depth = texture(uDepthMap, newUvs).r;

// convert depth to view space
vec4 ndcPos = vec4(newUvs.x, newUvs.y, depth, 1.0f) * 2.0f - 1.0f;
vec4 viewPos = inverse(uPersp) * ndcPos;
viewPos /= viewPos.w;

vec3 boundPosition_cam = boundPos_cam;
float myDepth = boundPos_cam.z;
```

When both depths are in view space, only a clamp is necessary in order to avoid seeing the light volume when an object is in front of it.



(a) (b)

Figure 5: *(a) shows a scene with three objects with depth testing disabled and no z clamping in the fragment shader. The light volume behind the objects can be seen. (b) shows the same scene but passing the depth buffer texture into the shader and clamping the z. In this way, the light volume behind is not seen through the objects.*

As mentioned before, each fragment will compute a term of equation 2. The built-in variable, gl_FrontFacing is used to determine if the current fragment is front or back facing. If it is front facing, Sn will be -1 (entering a lit region) and 1 otherwise.

```
// entering a lit region
if(gl_FrontFacing)
	Sn = -1.0f;
// entering an unlit region
if(!gl_FrontFacing)
	Sn = 1.0f;
```

Once again, a different solution to equation 1 was used to obtain convincing results. The distance from the viewer to the fragment is computed and then divided by the total distance the viewer can see. This yields a distance between [0, 1], where 1 means that the fragment is at the viewer's far plane and 0 at the viewer's near plane. Multiplying this by a parameter will make the effect more or less intense, giving a sense of depth or density. The greater this value, the denser the light will be and less will be seen through it. It is also important to simulate a loss of strength the further away the fragment is from the light source. Using the same idea as before, it is possible to divide the distance of the fragment to the light by the total distance the light can see. Reversing this result (1 - x) will change the range and thus values far away will be closer to 0 and values nearby the light will be closer to 1. The basic idea is to multiply the airlight by this value, by doing this, far away fragments will be more attenuated and nearby fragments will be brighter.

The last step is to multiply the airlight value by the color of the light.

```
float La = (Sn * normalizedDistance * distFromLight * uDensity);

FragColor = vec4(La, La, La, La) * uColor;
```

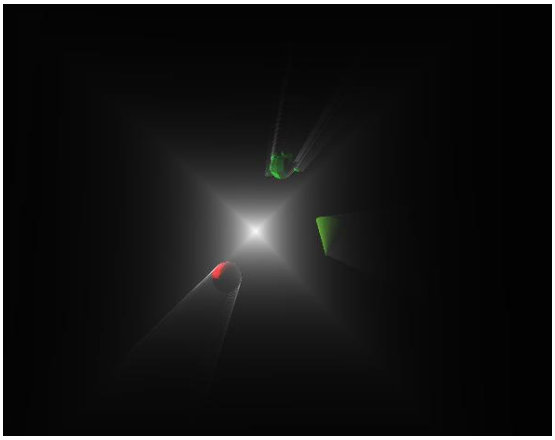


Figure 6: *Scene with three objects lit by a white light using the described formula.*

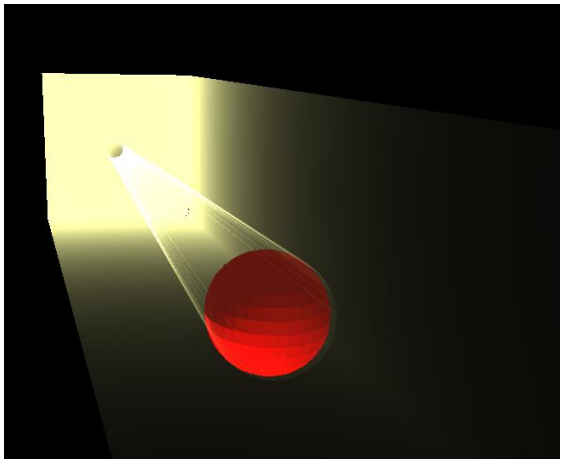


Figure 7: *A yellowish light illuminating a sphere. In this case the airlight does not change the further away it is from the light source and hence the displaced grid can be seen.*

4. Demo

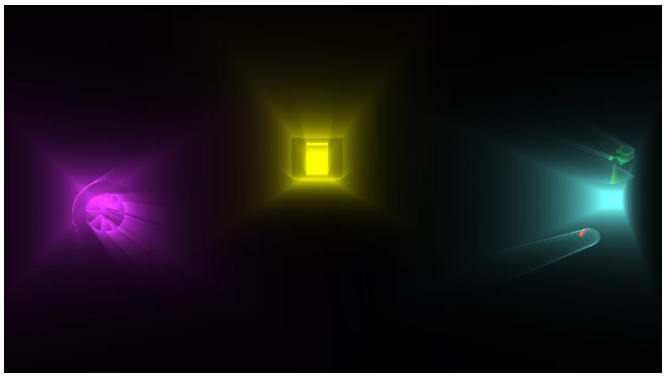
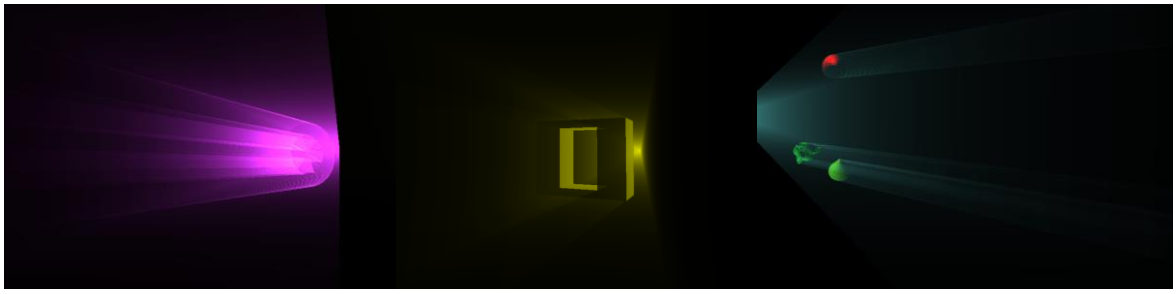


Figure 8: *Main screenshot of the demo. The demo shows a scene divided into three parts: left, center and right, each part with its own light. Each light has a different intensity and color, no falloff and a 512x512 resolution shadow map.*



(a) (b) (c)

Figure 9: (a) *Shows the left section of the demo. A pink light is illuminating a spinning vent and hence light rays do not get through the vent's blades. (b) Shows the center section, a yellow light pointing at a window, simulating the effect light would have when entering a closed building through doors, windows or any other open space. (c) Shows the right most section, where a blue light points towards three objects (Suzanne, a sphere and a cone) spinning in circles. Suzanne and the cone also rotate respect to themselves, again, generating new volumetric shadows.*

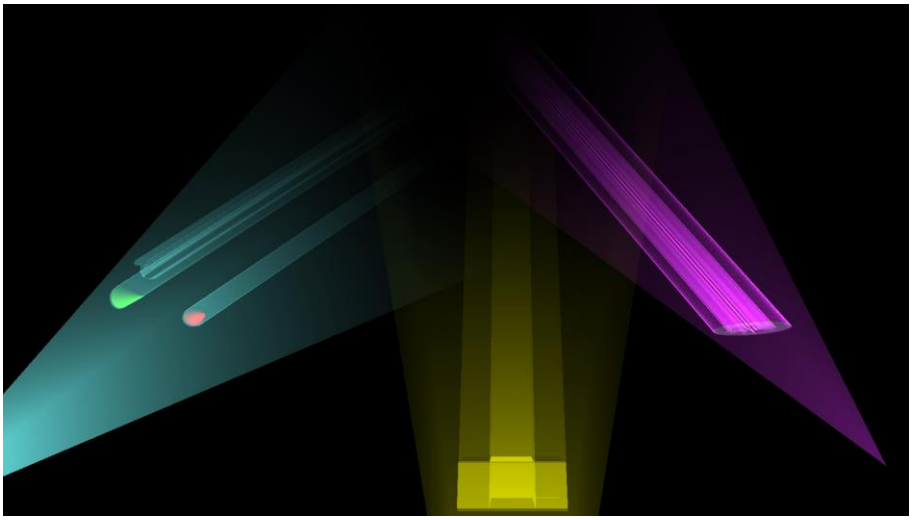


Figure 10: *Top view of the demo scene where all three sections are shown from an aerial view. It is possible to see how light goes through the window in the center section and how all three lights lose intensity the further away the fragment is from the source.*

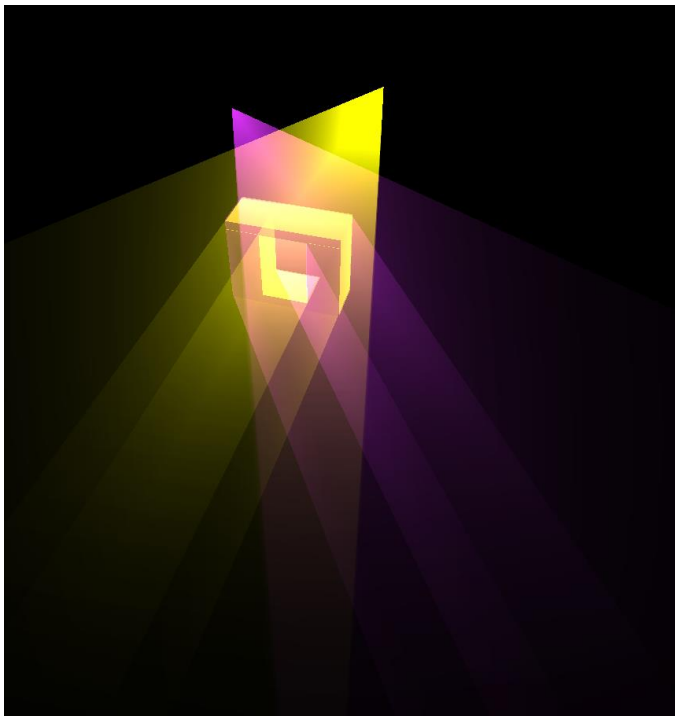


Figure 11: *Center section being illuminated by two lights from different points and angles. As it is possible to see, light volumes do not interact with each other and hence two shadows are generated for the window.*

4.1 How to use it

The demo is equipped with several sliders and buttons to change the look of it. It is also possible to move the camera around freely using the W, A, S, D keys to move forward, left, back and right respectively. Holding the right click and moving the mouse makes it possible to look around.

There is a GUI window showing the frames per second of the scene and three selectables under it referring to each one of the three lights. Each light has the name of the section it corresponds to. The sections are named from the starting position. If a selectable is clicked, the corresponding light will be selected and underneath some parameters will be able to be tweaked.

The first one is the light's rotation, which as its name suggests, will rotate the light and make it face a different direction. The second one is the light's position. The third slider is the Density slider, increasing the slider will make the light more intense and decreasing it will make the light fade out. The fourth parameter is the falloff. The falloff describes how much the light decays the further away it is from the light. Having a falloff of 0 means no decay. The greater this value, the less far away the light will reach.

The next parameter is the light's color, which will change the color of the light and the light volume generated. Underneath, it is possible to find a resolution combo, which allows to select the resolution of the shadow map for the currently selected light. Low resolutions will generate a low number of vertices and inaccurate meshes, and high resolutions will generate accurate meshes but at the cost of performance.

The last thing shown is an image of the shadow map the currently selected light is rendering. It can be used to visualize how each light sees and how resolution changes affect the final effect.

Figure 12: *Top part of the GUI, showing the FPS, and the three selectables representing the three lights on the scene.*

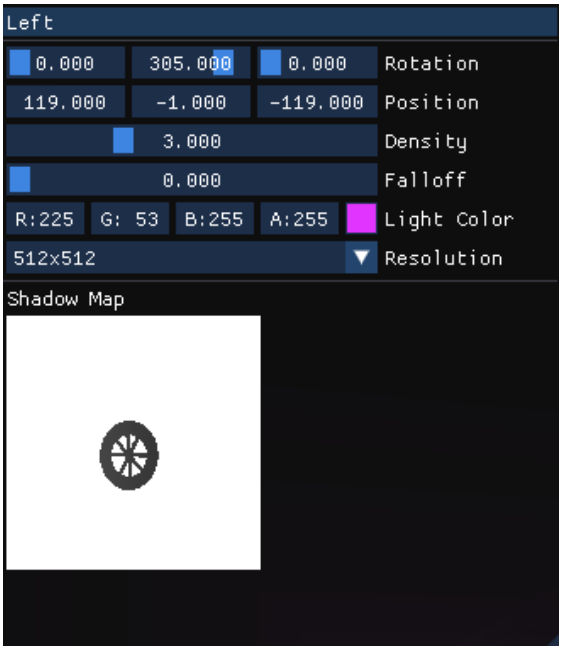


Figure 13: *Once a light is selected, it is possible to tweak some of its values, such as rotation, position, density, falloff, its color and resolution, and to visualize its shadow map.*

5. Possible Improvements

As mentioned before, this implementation does not use the GPU to nor generate the mesh nor displace the vertices once the light's shadow map is read. An obvious improvement is to use geometry shaders to generate and displace the vertices in order to improve performance.

The original paper [MBSA10] implements adaptive tessellation in order to reduce polygon count and improve framerates whereas this implementation does not.

Another improvement to mention is to make this effect general for all kind of lights: point lights, spotlights and directional lights and not only the last one listed. In order to implement point lights, it would be necessary to take six pictures of the scene each frame to generate

the shadow maps and the proceed in the same way as with directional lights. Spotlights would be a bit more complex, as these have a rounded nature the fragment shader would need to discard and attenuate fragments at the edges to make the volume have this circular shape.

As of for now, some artifacts appear around the extruded shadows of objects. This is most certainly because of some shadow map inaccuracies. As it is possible to see in the demo, these artifacts do not appear in square objects but do appear in objects that have rounder edges and shapes. This happens because the shadow map cannot store all the smooth information of these objects and hence stair like patterns appear. In the fragment shader, parts of this stairs are seen as front faces instead of back faces and hence airlight is accumulated wrongly in these parts. Increasing the shadow map resolution does help but does not completely solve the problem.

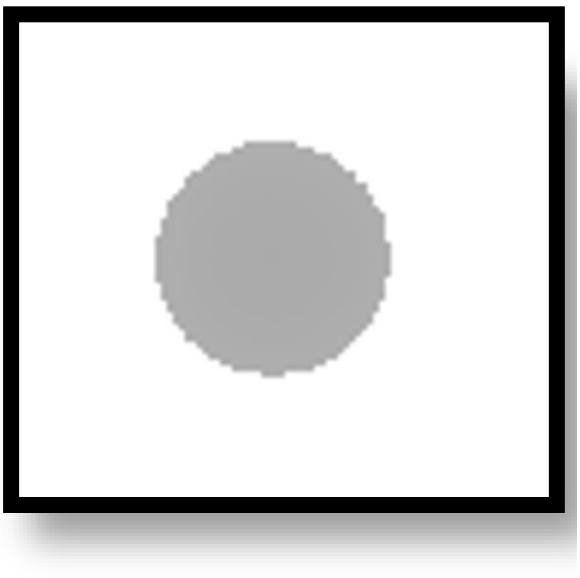


Figure 14: *256x256 shadow map of a light rendering a sphere. As it is possible to see, the round shape is not captured correctly and the mentioned staircase like pattern appears. Depending on the point of view of the viewer, the horizontal or vertical fragments around the edge are seen as back or front faces wrongly.*

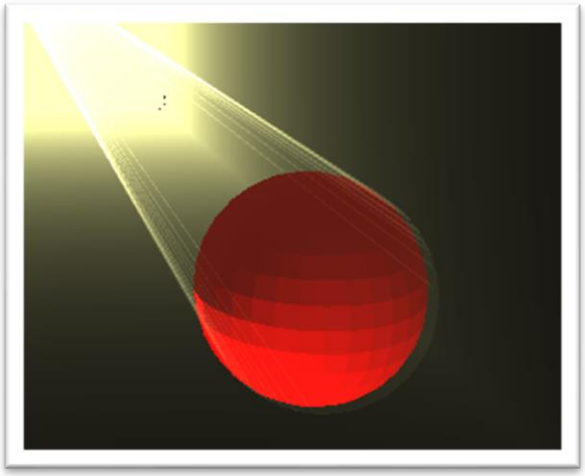


Figure 15: *Resulting image from figure 14. White stripes appear along the extruded shadow due to these fragments being faced incorrectly.*

Another improvement that can be made is the interaction between lights. As seen in figure 11, a light A illuminating another light B should make this last one be affected by A. The amount depends a little bit on how intense both lights are. It would also be an improvement to blend the colors of both lights together and most certainly not generate shadows for both.

One more improvement that can be made is to make the airlight formula view independent and more precise. The one used generates interesting results but is not near perfect and as mentioned, the effect changes depending on how close the viewer is from the directly illuminated volume.

5.1 Other approaches

One of other possible implementations is to trace the view volume and accumulate lit samples in a texture using shadow map comparisons. The final intensity depends on the number of lit samples. The texture is then combined with the scene and edge blurred due to aliasing issues in the border edges [VLJ08]

Other possible approach is to implement the effect as a post process. Using the stencil buffer, emissive parts of the image are rendered normally while setting a stencil bit. Afterwards, the rest of the objects are renderer

without the stencil bit. In the post processing part, only fragments with the stencil bit set will contribute to the additive blended effect. [KMEA07].

One more approach suggests dividing the frustum in different planes and project the shadow generated by an object into the sampling planes [JM04].

6. Problems during development

The first problem encountered during development was to face the mesh grid in the same direction as the light. After some work, I used the inverse of the glm lookAt function to make the plane be oriented correctly. Generating the mesh on the CPU was not much of an issue and neither was it reading the shadow map texture and updating the vertices' depth.

I struggled a little bit with making the mesh dynamic while keeping framerates high. As updating the mesh on the CPU is slow, I had to find a way to make it as fast as possible which was done by pre allocating all necessary memory and reusing it.

Reading through the paper, I found it difficult to understand how the airlight integral was implemented in the code. The paper where this integral is solved does so in a very clean way, but it got quite complex and out of scope [SRNN05]. Hence, finding a proper equation to simulate the desired effect has been difficult, especially making it view independent, which it still is not.

I also struggled with depth comparisons. Passing the depth of the scene as a texture to the fragment shader was no challenge, but the result was not what I expected. If an object outside the illuminated volume is in front of it from the point of view of the eye, the fragments of the illuminated volume get clamped to the stored depth and giving the look as if some light rays are going through the object, making it look as if it was translucent in some way.

Other than that, comping up with an interesting demo was difficult. The effect somewhat worked in Sponza, which was the initial objective, but the last-mentioned issue made it look very unnatural and strange, as the light could be seen through the walls and hence had to create a demo where this did not happen but still was interesting enough.

7. Conclusions

This solution is quite easy to implement and if done correctly can give a fresh and unique look to any scene. Despite all troubles, the effect is still somewhat believable in isolated scenarios. It would have been great to implement the mesh generation on the GPU and experiment and learn about other types of shaders.

As the paper suggests, the borders of the mesh should be connected to the light's positions, forming a cone-like shape. In my opinion this does not look as good or coherent with directional lights, as light rays are parallel, and it would be impossible for them to follow this type of trajectory.

As the performance of the algorithm just depends on the resolution of the shadow map it is great for any kind of scene. Besides that, in most games shadow mapping is already a must in order to generate shadows, so this approach integrates quite well in the rendering pipeline.

Overall, it is an easy and stunning effect to implement that brings visual quality to another level.

8. Bibliography

Mayinger, F. (Ed.). (2013). *Optical Measurements: Techniques and Applications.* Springer Science & Business Media.

[MBSA10] Billeter, M., Sintorn, E., & Assarsson, U. (2010). *Real time volumetric shadows using polygonal light volumes*. https://www.cse.chalmers.se/~uffe/volumetricshadows.pdf

[SRNN05] Sun, B., Ramamoorthi, R., Narasimhan, S. G., & Nayar, S. K. (2005). *A practical analytic single scattering model for Real time rendering*. https://cseweb.ucsd.edu/~ravir/papers/singlescat/scattering.pdf

[McC00] McCool, M. D. (2000, January 1). *Shadow Volume Reconstruction from depth maps. ACM Transactions on Graphics.* https://dl.acm.org/doi/pdf/10.1145/343002.343006

[VLJ08] *Volume Light*. (2008, January 15). *NVIDIA Developer*. https://developer.download.nvidia.com/SDK/10.5/direct3d/Source/VolumeLight/doc/VolumeLight.pdf

[KMEA07] Mitchell, K. (2007). Chapter 13. *Volumetric light scattering as a post-process. NVIDIA Developer*. https://developer.nvidia.com/gpugems/gpugems3/part-ii-light-and-shadows/chapter-13-volumetric-light-scattering-post-process

[JM04] Mitchell, J. (2004). Light shafts - AMD. *Rendering Shadows in Participating Media*. https://developer.amd.com/wordpress/media/2012/10/Mitchell_LightShafts.pdf