Real-time Global Illumination Using Voxel Cone Tracing

Fredrik Präntare (prantare@gmail.com)

December 2016

Abstract

This report presents a real-time global illumination algorithm that was implemented during the course "Advanced Game Programming" (alias TSBK03) at Linköping University. The presented algorithm supports indirect diffuse lighting, glossy reflections, refractive lighting (transparency), soft shadows, and emissive surfaces at interactive frame rates. The algorithm is a 2 pass real-time global illumination technique based on lighting voxelization and cone tracing. The direct lighting of a scene is voxelized in real-time using the GPU hardware rasterizer. The voxelized lighting is represented using a 3D texture, and filtered linearly using a 3D mipmapping scheme. Voxel cones are then traced throughout the different mipmap levels in order to approximate indirect illumination. 3 different cones are introduced: shadow cones, diffuse cones, and specular cones. The specular cones are used in order to create glossy reflections and refractive lighting, the diffuse cones are used in order to emulate indirect diffuse reflections, and the shadow cones are used in order to approximate soft shadows and rough ambient occlusion. The technique is view independent, almost scene independent, and supports fully dynamic environments.

1 Introduction

Algorithms for global illumination improve the quality of rendering by adding more diverse and realistic lighting, such as indirect lighting, glossy reflections and caustics; effects depicted in figure 1.

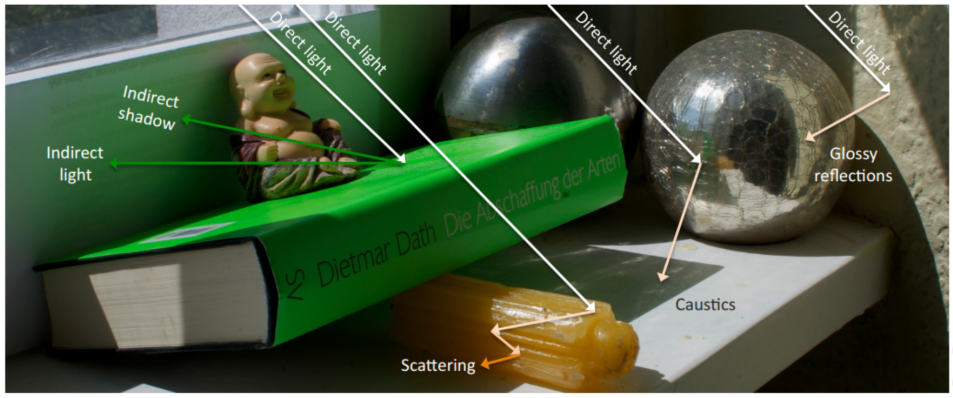

Figure 1: Photography of different lighting effects.
Image on courtesy of Tobias Ritschel. [RDGK12]

A generalization of many such rendering algorithms was formulated by J. T. Kajiya in 1986 [Kaj86]. The formulation, which is now known as the rendering equation, is an integral part of many well-known realistic global illumination techniques, such as ray tracing [Whi80] and photon mapping [Jen96]. Rendering algorithms attempt to solve the rendering equation approximately, which today's personal computers have a hard time to accomplish at high accuracies while maintaining interactive frame rates. Most global illumination techniques are either static solutions, such as radiosity [GTGB84] and ray tracing [Whi80, Jen96], or lack in features or quality, such as ambient illumination.

This report presents a real-time global illumination algorithm that is able to render many high quality indirect lighting phenomena at interactive frame rates. The algorithm is based on a three-step rendering algorithm that was introduced by Crassin et al. in 2011 [CNS+11]. The presented algorithm takes advantage of a voxelization pass, in which the direct lighting of the scene is voxelized in real-time using the GPU hardware rasterizer. The voxelized lighting is represented using a cubic 3D texture, and is linearly filtered using a 3D mipmapping scheme. Voxel cones are then traced throughout the filtered mipmap in order to approximate indirect lighting for Lambertian, transparent and glossy surfaces.

2 Previous work

Rendering of dynamic indirect lighting in 3D scenes is usually based on rendering meshes that consist of triangles, where lighting for each pixel is partially

re-computed every frame. Many such algorithms work inherently in triangle-space, since ray-triangle intersection tests make it possible to decide whether a pixel should be illuminated or not. As the number of triangles and vertices increases, algorithms that work in triangle-space become less appealing due to their computational complexity. Other adaptive and efficient solutions are thus needed. Some of the most efficient real-time global illumination algorithms for rendering triangle-based scenes work in image-space and ignore off-screen information [RGS09, NSW09], and there are many promising real-time algorithms for voxel-based global illumination that work in voxel-space and consider off-screen information [THGM11, CNS+11, LWDB10].

One of the first successful attempts at interactive voxel-based indirect illumination was accomplished by Thiedemann et al. in 2011. Their solution was based on a combination of path tracing and a scene voxelization technique called atlas-based boundary voxelization. However, their algorithm only supports a few indirect lighting phenomena for near-field single bounce illumination, but still proved to be a glance into the future of voxel-based global illumination. [THGM11]

A new voxel-based indirect illumination algorithm was presented in "Interactive Indirect Illumination Using Voxel Cone Tracing" by Crassin et al. in 2011. They used the GPU in order to generate filtered mipmaps for voxel-based lighting representations in real-time. A sparse voxel octree was introduced in order to increase rendering performance by using less samples and smaller step distances during cone tracing, and a new splatting scheme for light injections. Their algorithm supports many lighting effects at interactive frame rates, such as glossy reflections, ambient occlusion, and indirect diffuse lighting, and exhibits almost scene-independent performance. Due to the octree-voxelization scheme their algorithm also manages to render complex scenes in real-time. Their three-step voxel cone tracing algorithm is depicted in figure 2. [CNS+11]

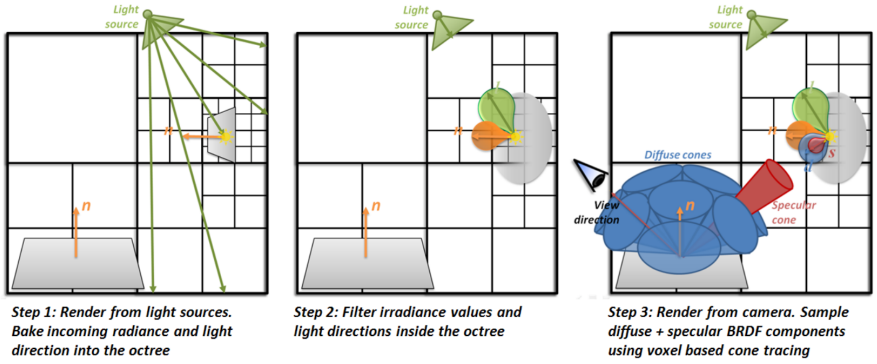

Figure 2: The three major steps of the voxel-based global illumination algorithm described by Crassin et al.
Image on courtesy of Cyril Crassin.

3 Algorithmic overview

The proposed voxel-space global illumination algorithm for triangle-based scene rendering is based on 2 passes: lighting voxelization and cone tracing. The first pass, i.e. lighting voxelization, uses the geometry shader to voxelize direct lighting in real-time using the GPU hardware rasterizer and a simplified version of a voxelization scheme described by Crassin et al. in "Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer" [CG12]. The voxelized lighting is represented using a 3D texture, which is used to take advantage of the filtering capabilities of modern GPUs by generating 3D mipmaps using linear filtering. Paths are then traced throughout the filtered direct lighting in order to approximate indirect lighting. Each path starts at a pixel fragment in world-space, and uses the filtered voxel-based representation in order to accumulate indirect lighting. The approach is obviously similar to ray marching, but much more effective due to using larger (and varying) sample steps.

The choice of bidirectional reflectance distribution function (BRDF) and bidirectional transmittance distribution function (BTDF) can alter the behaviour of the path traces, and how the sampling occurs along the path, thus making it possible to emulate different types of materials and indirect lighting effects. The choice of BRDF and BTDF also makes it possible to, in some cases, use under-sampling in order to increase performance, since larger sample steps are more efficient (but yields blurrier results). Larger sample steps are not necessarily bad in terms of quality, since they can be used to efficiently approximate high quality diffuse lighting, soft shadows, and rough glossy surfaces. The traced paths are called cones, and 3 different cones (shadow, diffuse and specular) are used in order to approximate indirect lighting. All tracing can be done using the fragment shader in a single OpenGL program, making an implementation relatively easy to manage. The voxelization and 3D mipmapping schemes are run every frame in order to support fully dynamic environments without lighting artifacts.

3.1 Pass 1: Lighting Voxelization and Filtering

A 3D texture that represents the direct lighting of the scene needs to be generated before any cone tracing can take place. This can be done using the voxelization scheme described by Crassin et al. in "Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer" [CG12]. The main parts of their voxelization scheme are roughly described in figure 3. In our case we use an almost identical approach to Crassin et al., but exchange the conservative rasterization with a non-conservative technique.

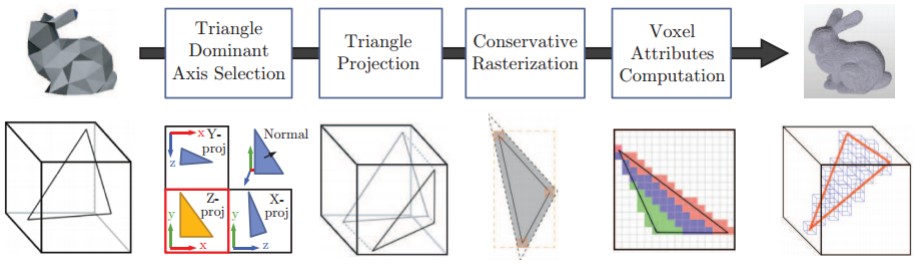

Figure 3: Illustration of the voxelization scheme described by Crassin et al. *Image on courtesy of Cyril Crassin. [CG12]*

A non-conservative voxelization scheme without memory barriers will, in practice, introduce artifacts due to either missing small triangles, or due to overlapping triangles and their corresponding voxels being written to the same texel during a single rendering pass. However, the voxelization approach using non-conservative rasterization can be implemented using only a few lines of geometry shader code in GLSL. The image store functionality (made available in OpenGL version 4.2) of modern GPUs can store fragments directly to a 3D texture, making it possible to voxelize without writing to an intermediary frame buffer.

The first part of the voxelization scheme, i.e. selection of dominant axis for projection, is accomplished by computing the areas of all possible triangle projections onto the three axial planes (the xy-projection, yz-projection and xz-projection). The dominant axis is then selected as the axis that corresponds to the largest projected area (i.e. the normal of the projection). Crassin et al. describes this process as selecting the axis that maximizes $\boldsymbol{l_{x,y,z}} = |\boldsymbol{n} \cdot \boldsymbol{v_{x,y,z}}|$, where $\boldsymbol{n}$ is the triangle normal and $\boldsymbol{v_{x,y,z}}$ the three base axes of the scene. When the dominant axis has been selected, the geometry shader generates triangles for rasterization according to the chosen axis. For more specific details, please see the paper by Crassin et al. [CG12]

The rendering viewport must be set to the same dimensions as the cubic 3D texture in order to generate voxel-fragments that correspond to positions in the 3D texture. This is done using the OpenGL call `glViewport`. The 3D texture must be cubic since the hardware-based mipmapping scheme only supports high performance filtering for cubic textures.

Since we are not writing to a frame buffer by using the image store functionality, it's safe to disable color writing, depth testing and depth writing by using the calls `glDisable(GL_DEPTH_TEST)`, `glDisable(GL_CULL_FACE)` and `glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE)`.

The only voxel-fragment attributes that we are interested in are the interpolated normals and their voxel-fragment world positions, since those are what we need to compute direct lighting using the Blinn-Phong lighting model. The world positions are passed using `gl_Position`, while the normals must be treated as a separate attribute. The computed direct lighting information is

stored in the voxel texture as a single `RGBA8` value, where the alpha channel is used for transparency. Uniforms can be treated and used as usual. A set of 2D triangles are thus passed through the shader pipeline into the fragment shader. Each such triangle is rasterized and turned into a set of voxels with direct lighting information which we store in our target 3D texture of `RGBA8`s. The direct lighting of a triangle-based scene is in this way stored and represented as a three-dimensional `RGBA8` voxel texture. A directly lit scene (using the Blinn-Phong lighting model) and its voxelized representation is visualized in figure 4.

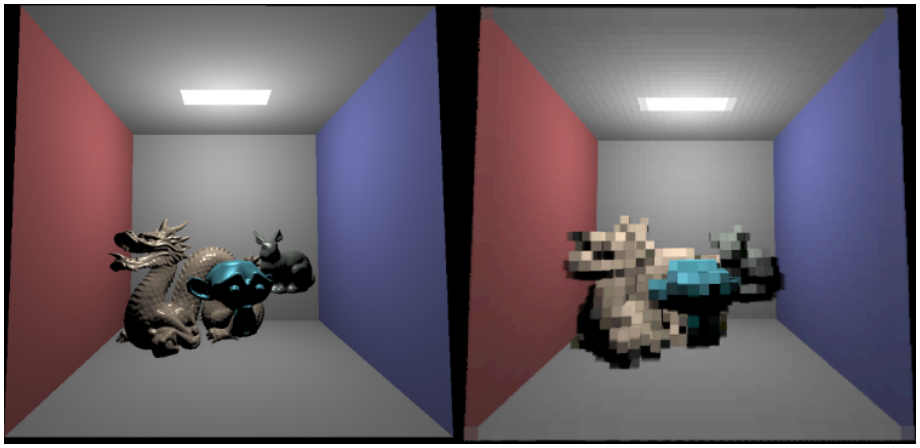

Figure 4: A visualization of the direct lighting (left) of a scene stored as a `RGBA8` voxel texture (right).

By having the direct lighting stored and represented using a 3D texture, we can now filter the 3D texture using a OpenGL-based 3D mipmapping scheme. This is done using the OpenGL call `glGenerateMipmap(GL_TEXTURE_3D)`, and having the filtering settings of the 3D texture set to `GL_LINEAR_MIPMAP_LINEAR` and `GL_NEAREST` for the `GL_TEXTURE_MIN_FILTER` and `GL_TEXTURE_MAG_FILTER` settings respectively. A few filtered mipmap levels are visualized in figure 5.

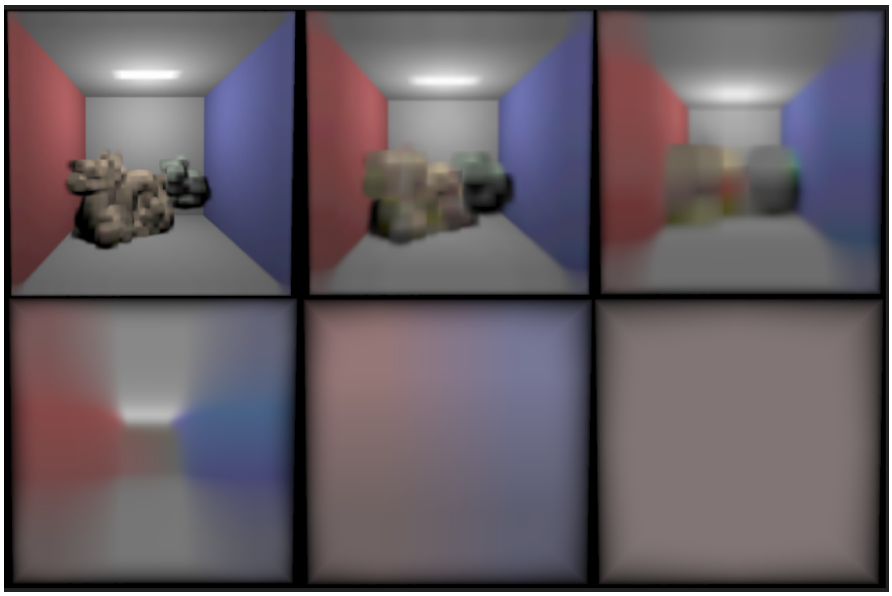

Figure 5: Visualization of increasing mipmap levels of a linearly filtered 3D texture.

Finally, the voxel texture must be cleared before every voxelization pass, otherwise there will be leftover voxels from previous voxelization passes. This is done using the OpenGL call `glClearTexImage`, which was introduced in OpenGL 4.4. It's also possible to clear the voxel texture by using a clearing program, or by simply using the much slower process of reallocating an empty 3D texture every frame. All of these different approaches were tested, and the new `glClearTexImage` call was clearly advantageous in terms of both performance and simplicity.

The voxelization and 3D mipmapping schemes must be run every frame in order to support fully dynamic environments, otherwise indirect lighting wont be synchronized with movements in the scene. This is not necessarily a bad thing, since it could potentially increase performance by only updating indirect lighting sporadically.

The voxelization scheme can also be implemented using CUDA or OpenCL programs, or more interestingly using the new `GL_NV_conservative_raster` OpenGL voxelization extension by NVIDIA.

3.2 Pass 2: Voxel cone tracing

Voxel cone tracing is used in a second pass to approximate indirect lighting and shadows. Every object is rendered using an OpenGL program that takes advantage of tracing cones throughout a linearly filtered voxel texture that represents the direct lighting of the scene. The implementation described in this

paper uses a Blinn-Phong lighting model for direct lighting, but could easily be exchanged for any other lighting model. All cone tracing takes place in the fragment shader, and the vertex shader is a simple and typical vertex shader that consist of the whole matrix transformation pipeline (i.e. model, view and projection matrices).

Three different specialized cones are introduced in order to approximate different lighting phenomena: diffuse cones, specular cones, and shadow cones. One cone type could theoretically be used for all phenomena, but specialized cones turned out to be more flexible for development and experimentation of new tracing techniques. For example, the indirect diffuse cones take advantage of undersampling, while the shadow cones use multisampling to increase shadow quality.

3.2.1 The diffuse cone

The diffuse cones are traced in order to approximate indirect diffuse light using the Lambertian lighting model. 9 lighting accumulation cones are traced from every fragment: 1 forward cone in the direction of the normal, and 8 side cones each rotated 45° from the normal toward the surface, and with 45° increments around the normal, as depicted by figure 6.

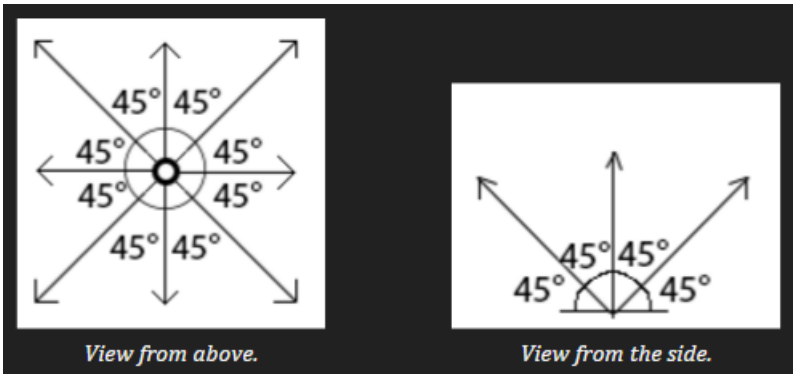

Figure 6: Illustration of the angles of which the indirect diffuse cones are emitted.

A cone is terminated whenever its path accumulator reaches an alpha value higher than 1.0, or when a sample volume is outside of the voxel texture. The resulting color uses a simple polynomial correction curve to alter the intensity of the accumulated color.

3.2.2 The specular cone

The specular cones are traced in order to approximate indirect specular and refractive light. The indirect specular light is accomplished using a perfectly reflected cone, while the refractive light is computed using a refracted cone

(using Snell's law). It would be possible to use other reflection and refraction models, but it turns out that a perfectly reflective cone is enough to emulate glossy reflections, mirrors and rough specular surfaces, and a single refracted cone is enough to emulate many different transparent materials.

3.2.3 The shadow cone

The shadow cones are traced in order to create soft shadows. The shadow cone is in concept a very cheap shadow ray. However, we are not trying to find out whether a surface is occluded; we are only interested in how dense the traced volumetric path is in terms of non-transparent voxels. Denser paths thus yields more occluded fragments.

4 Results and discussion

The results[1] shown in this section all ran at frame rates above 100 frames per second using a NVIDIA GeForce GTX 970 GPU and an Intel 2500K CPU using a 64^3 voxel texture at a resolution of 1280x720 pixels.

The presented voxel cone tracing algorithm improved performance of a factor of 800.000 over a naïve CPU-based ray tracing implementation[2]. The performance test scene contained a Cornell box with a white emissive lamp and a glossy dragon. The visual differences are obvious, and can be seen in figure 7.

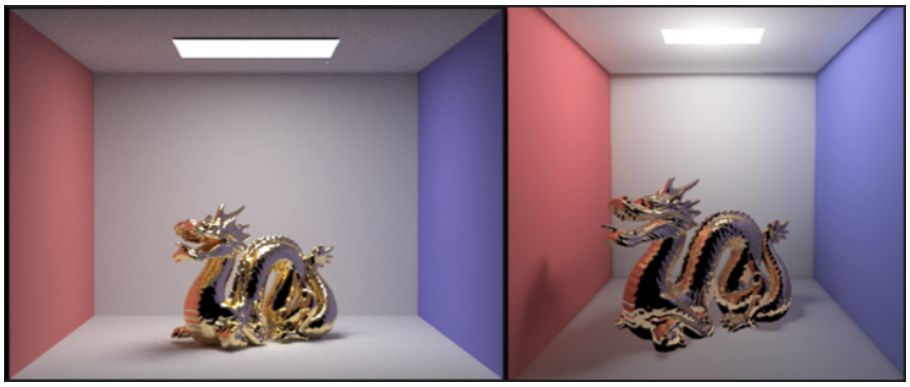

Figure 7:
(L) Monte Carlo ray tracer: $>$ 7200000 ms (more than 2 hours).
(R) Voxel cone tracing: $\sim$ 9 ms.

Multiple different materials can be modelled using the presented algorithm, such as fuzzy glass, sharp gold, and clay-like materials, as seen in figure 8.

[1] The source code, and a video of the implementation running in real-time, can be found at the author's Github repository: https://github.com/Friduric

[2] The CPU-based ray tracer was implemented by Kadie Jaffe.

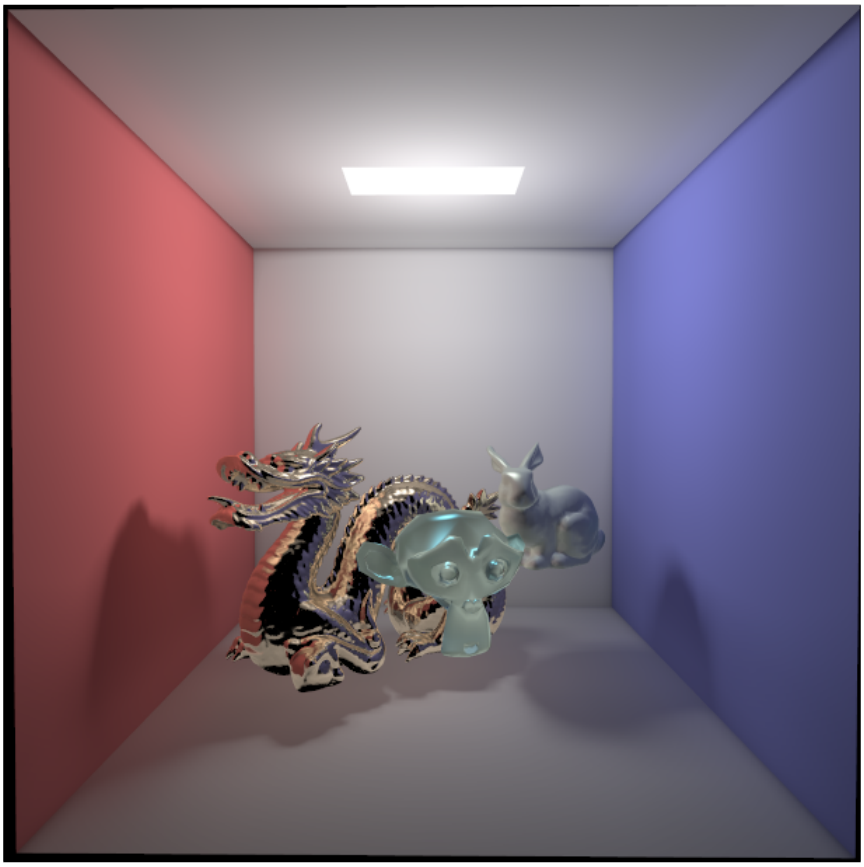

Figure 8: The presented algorithm manages to render many different materials. In this image there is a golden dragon (perfectly specular surface), a fuzzy glass monkey (a semi-transparent Susanne), and a clay-like bunny (semi-specular surface).

The diffuse cones make it possible to emulate high quality color bleeding, as seen in figure 9. There's no noise due to sampling from filtered data. There are also no obvious artifacts, which shows promise in terms of using undersampling for indirect diffuse lighting. Using undersampling for specular reflections and refraction might be harder, since those generally need higher precision in order to yield realistic and satisfying results.

The specular cones make it possible to emulate specular reflections, as seen in figure 10. It is also possible to achieve different types of specular reflections, such as rough and glossy surfaces, by adjusting the specular diffusion factor, as visualized in figure 11.

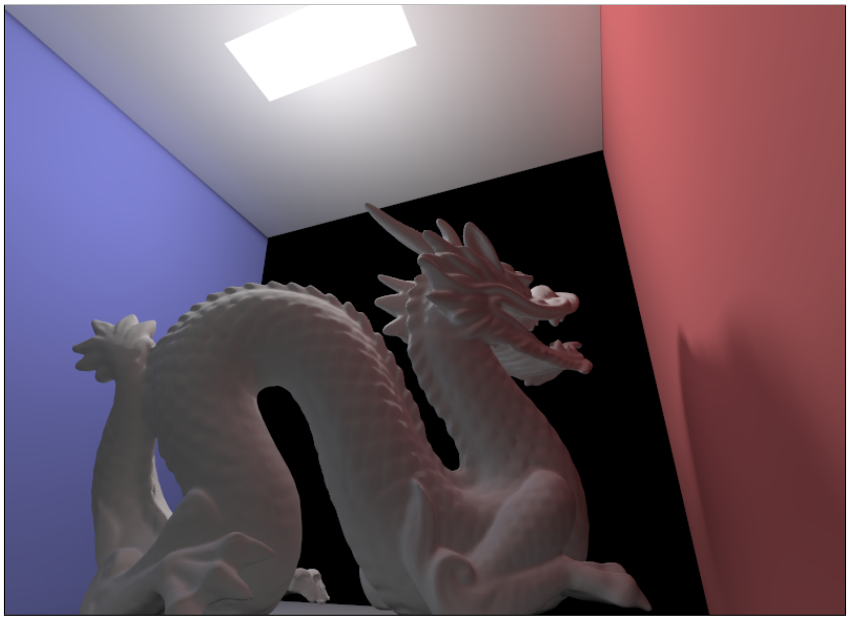

Figure 9: The diffuse cones manages to emulate color bleeding and indirect diffuse light.

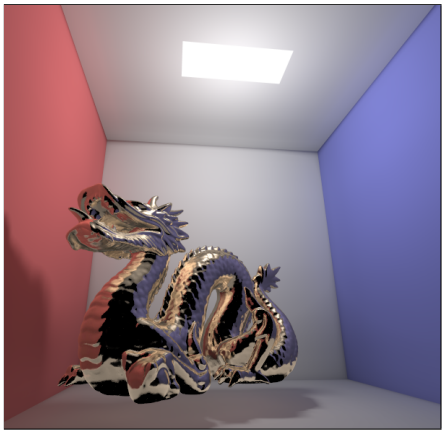

Figure 10: The specular cones manages to emulate specular reflections.

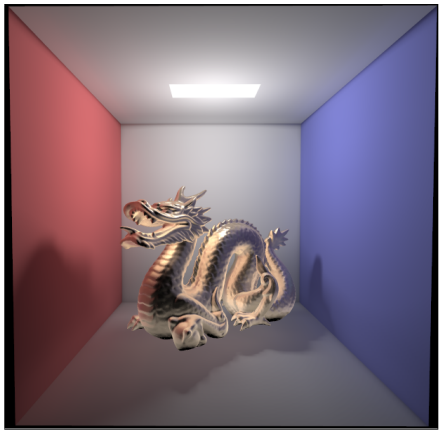

Figure 11: Using a specular diffusion factor makes it possible to emulate rougher glossy materials.

Emissive materials can be emulated using both specular and diffuse cones, as seen in figure 12. Emissive materials are emulated by simply adding an emissive color to each voxel-fragment, and cost practically nothing in terms of computational effort.

Figure 12: It's a simple matter to emulate high performance emissive materials using voxel cone tracing.

Transparent materials with varying refractive indices and diffusivity is created using refractive cones. The results show promise in terms of using voxel cone tracing for transparency, but multiple bounces are needed in order to create more realistic transparent materials. The current implementation only uses a single refraction, which gives a rough approximation of refractive materials. A glass-like bunny can be seen in figure 13.

Approximations to soft shadows and ambient occlusion can be created using shadow cone tracing. The shadows become blurrier further away from the light sources due to using a inverse square fall-off curve for shadow diffusion. Emissive surfaces do not cast shadows, and need to be complemented with a point light in order to do so. The soft shadows look fairly good in real-time scenarios, but both flat and imprecise in still images. The soft shadows can be seen in most example images, such as in figure 14. Ambient occlusion could be improved by using a separate method, such as the one desribed by Crassin et al. in their paper from 2011 [CNS+11].

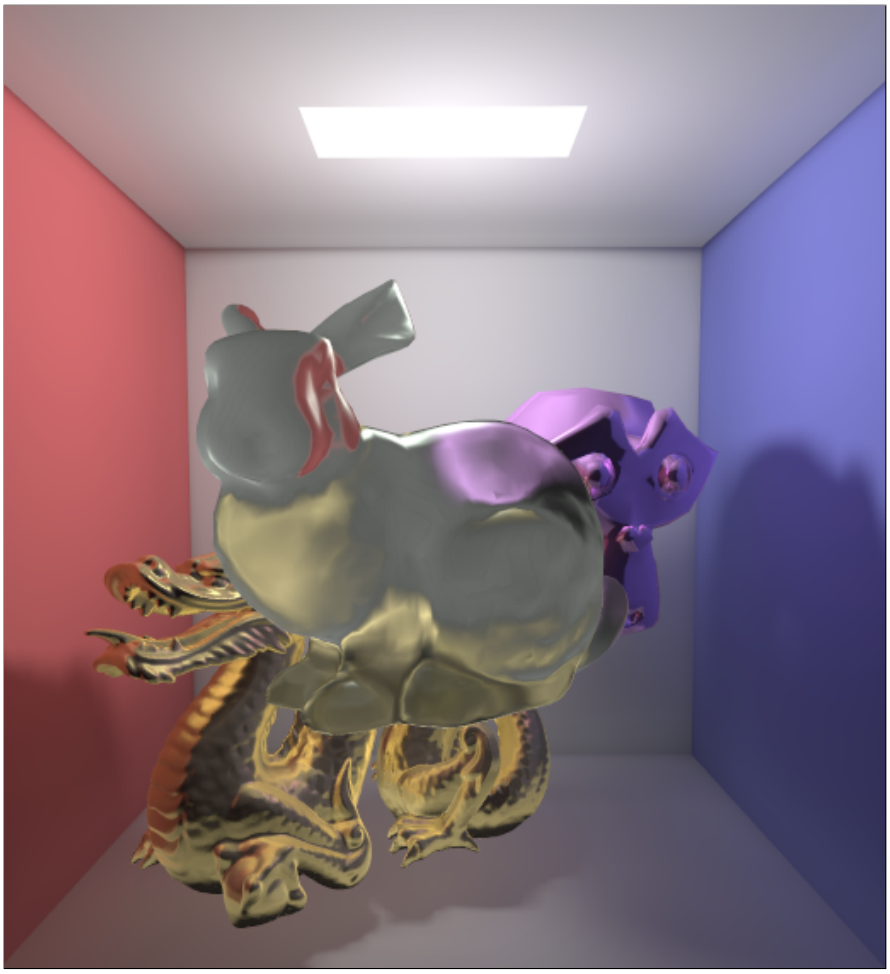

Figure 13: A transparent bunny that refracts light bounces from the dragon, walls, and monkey head (Susanne).

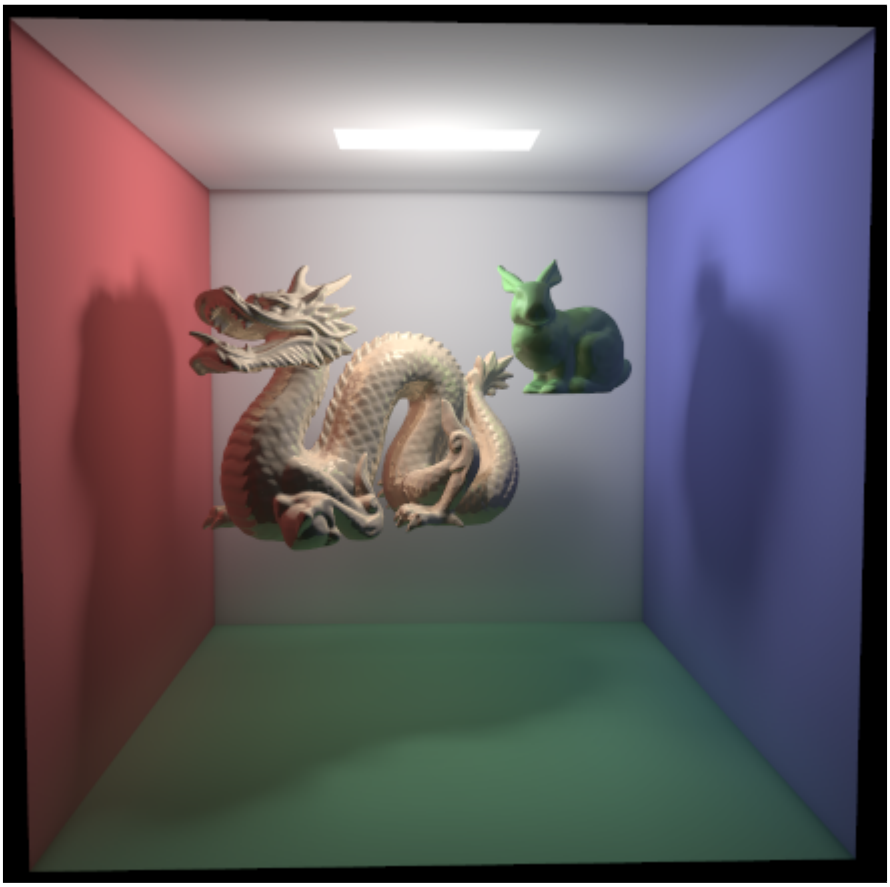

Figure 14: The presented algorithm can create rough approximations of soft shadows and ambient occlusion.

Even though voxel cone tracing can produce impressive real-time global illumination, it's still not a method without problems and artifacts. One such problem is flickering, or shimmering, which cannot be seen in still images. The flickering effect is produced when two triangles intersect, and their voxel-fragments are rasterized at different orders during two subsequent rendering passes. This cannot be solved by using the same rendering order for every rendering pass, since the GPU may still decide to rasterize at a different ordering. Crassin et al. suggests calculating a moving average for each voxel using atomic counters and memory barriers in order to remove shimmering artifacts. Another problem with voxel cone tracing is light leakage through thin meshes, which was expected due to Crassin et al. having the same problem with their implementation. [CNS+11]

5 Conclusion

Voxel cone tracing is obviously a promising technique for indirect lighting emulation in interactive media, and the presented real-time global illumination algorithm manages to render many indirect lighting phenomena at interactive frame rates, such as glossy reflections, transparency and diffuse reflections.

The algorithm generally yields high quality results in simple scenes at high frame rates. The results of the presented algorithm are interesting, but there are still many questions that need to be answered, such as how to accomplish caustics. There are many improvements that could improve performance, such as implementing a sparse voxel octree, making the presented algorithm a viable alternative to other real-time global illumination algorithms, such as techniques working in image-space or triangle-space.

In the future it would be interesting to study if it is possible to accomplish high performance approximative caustics using voxel cone tracing, and how well the presented algorithm works with other materials, such as liquid and gas. It would also be interesting to improve the shadow and refractive cones, and study new undersampling schemes.

References

[CG12] Cyril Crassin and Simon Green. Octree-based sparse voxelization using the gpu hardware rasterizer. *OpenGL Insights*, 2012.

[CNS+11] Cyril Crassin, Fabrice Neyret, Miguel Saintz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. *Pacific Graphics 2011*, 30(7), 2011.

[GTGB84] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. *ACM SIGGRAPH Computer Graphics*, 18(3), 1984.

[Jen96] Henrik Wann Jensen. *Global Illumination using Photon Maps*. Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering), 1996.

[Kaj86] James T. Kajiya. The rendering equation. *ACM Siggraph Computer Graphics*, 20(4):143–150, 1986.

[LWDB10] J. Laruijssen, R. Wang, Ph. Dutre, and B.J. Brown. Fast estimation and rendering of indirect highlights. *Computer Graphics Forum*, 29(4), 2010.

[NSW09] Greg Nichols, Jeremy Shopf, and Chris Wyman. Hierarchical image-space radiosity for interactive global illumination. *Computer Graphics Forum*, 28(4):1141–1149, 2009.

[RDGK12] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz. The state of the art in interactive global illumination. *Computer Graphics Forum*, 31(1):0–1, 2012.

[RGS09] Tobias Ristchel, Thorsten Grosch, and Hans-Peter Seidel. Approximating dynamic global illumination in image space. *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, 2009.

[THGM11] Sinje Thiedemann, Niklas Henrich, Thorsten Grosch, and Stefan Müller. Voxel-based global illumination. *Symposium on Interactive 3D Graphics and Games. ACM*, pages 103–110, 2011.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *ACM Siggraph 2005 Courses*, 23(6):343–349, 1980.