Souhrnná zpráva za rok 2011 z projektu "Vývoj softwaru v oblasti CAD systémů, 3D grafiky a vizualizace grafických scén"

Jan Pečiva

Faculty of Information Technology, Brno University of Technology

Abstract

Traditional ray tracing algorithms tend to provide photorealistic results but at high computing costs. Rendering times of minutes or days are not exceptional. On the other side, hardware accelerated OpenGL rendering can provide real-time interaction with virtual environment with unnoticeable rendering times. This paper attempts to bring these two together and attempts to give an answer on the difficulty of implementing real-time photorealistic rendering. The paper presents case study on mimicking of POV-Ray photorealistic rendering with accelerated OpenGL pipeline. The study shows the opportunity to accelerate some photorealistic algorithms by real-time approaches while, on the other side, it locates the parts that are difficult to replace by traditional real-time rendering paradigms. Particularly, it is shown how to implement primary and shadow rays and POV-Ray-like material model using accelerated OpenGL pipeline using modern shader technology. On the other side, the difficulties of reflected and refracted rays implementation using real-time rendering approaches is discussed.

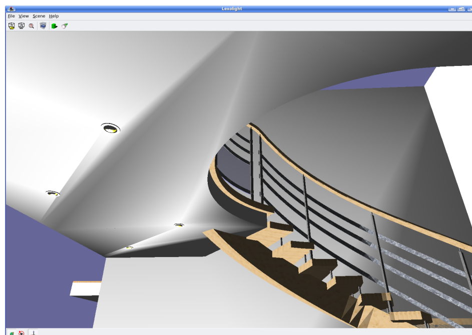

Figure 1: Lighting artifacts on a scene using old-fashioned OpenGL Gouraud shading with advanced light setups

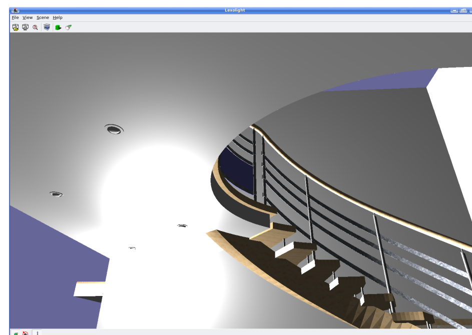

Figure 2: Per-pixel lighting in a scene rendered using shader technology

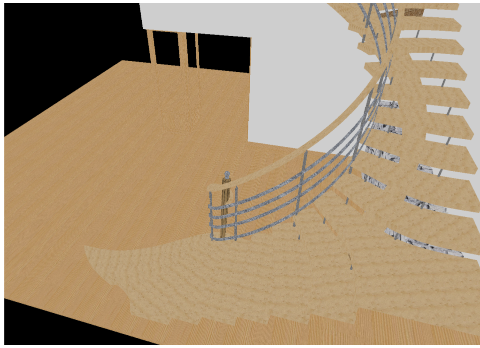

Figure 3: POV-Ray rendering using only level 0 rays

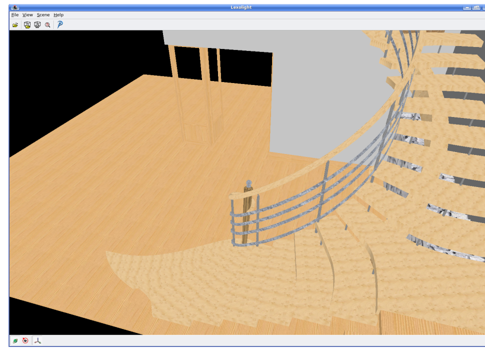

Figure 4: OpenGL rendering using ambient light

1 Introduction

Photorealistic rendering is a very important domain in computer graphic because of the realism that is often expected from graphics applications. Beginnings of photorealistic rendering can be traced back to the '60s to the invention of the ray casting algorithm [Appel 1968], followed later by ray tracing algorithm [Whitted 1980]. Ray tracing and other methods of photorealistic rendering, however, tend to be computationally very expensive even for today's hardware, often forcing users to wait minutes, or even days for high quality results. On the other side, many applications of real-time computer graphics desiring for higher realism exist. They can not accept computationally expensive algorithms of photorealistic computer graphics. Their users require high productivity and interaction with the graphics scenes. Such requirements are difficult to achieve in non-real time applications.

Areas desiring real-time photorealistic rendering include, for instance, CAD and architecture applications for interior design. Architects often want to immediately see the esthetic value of the designed model. Presently, they are forced either to compromise productivity by waiting for the results of the photorealistic rendering, or to compromise visual quality by throwing away photorealism and by using standard approaches of real-time computer graphic instead. However, visual quality is often essential while standard approaches of real-time computer graphics usually tend to provide poor results on advanced scene lighting setups. If real-time photorealistic visualizations would be possible, they would provide high quality visualizations, making architects and designers much more time effective while increasing the final quality of the designed models. Other applications include visualizations for civil engineering, scientific research, computer games, and visualizations in general.

Recent enhancements in programability of the graphics processor, particularly shaders and per-pixel lighting (see figures 1 and 2), gives a question whether it would be possible to implement and accelerate some photorealistic algorithms using standard approaches of real-time computer graphics. This paper does so by a case study of mimicking POV-Ray rendering with accelerated OpenGL pipeline [Wright 2004] while evaluating speed up factors of accelerated parts and describing difficulties with algorithms that are difficult to match with current OpenGL rendering paradigm.

2 POV-Ray Architecture

POV-Ray will be divided into three main parts for the purpose of this report:

- ray tracing rendering
- scene description and modelling library
- material modelling algorithms

As POV-Ray functionality set is quite large, only core functionality will be investigated and accelerated.

POV-Ray ray tracing rendering is delivering photorealistic results on one side while requiring much computing resources on the other side. Rendering times of hours or days are not exceptional. Mimicking ray tracing in the OpenGL rendering pipeline is a challenging task as they are using an opposite approach: POV-Ray is tracing rays from the

camera while OpenGL is projecting the scene geometry to the camera. This small difference brings various challenges to various light effects that will be discussed through the paper.

The scene description language of POV-Ray is very robust. It starts from triangle meshes and atmospheric effects and finishes with splines, blobs, run-time evaluated functions, and animations. In this paper, we will focus just on triangle meshes because that is the natural representation to OpenGL while all other representations can be tessellated – producing a triangle representation.

Modelling of materials is more advanced in POV-Ray than in standard OpenGL. It includes even multiple layers of material and procedurally generated material surfaces. Many of these advanced material approaches should be implementable in OpenGL shaders. This paper will not go to the excessive material functionalities of POV-Ray, rather mimicking of core material functionality will be discussed in section 3.2.

3.1 Ray Tracing

POV-Ray is a ray tracer. It casts a ray or a number of rays through each pixel of the image, rendered by camera. These rays are called primary rays. The ray travels through the scene until it hits a scene object, a light, or escapes the scene. If some object is hit, the object's material is processed and computed color is assigned to the ray. If it is primary ray, the ray's color is immediately assigned to the rays image pixel. However, depending on the material properties and the light setup, material processing may cause additional rays to be cast from the point of the object hit to various directions. Such rays are called secondary rays. When these rays hit other surfaces in the scene, these secondary rays may spawn recursively additional secondary rays. If the light is hit, it assigns the ray the light's color. If the ray leaves the scene, it is usually assigned the background's or sky's color.

POV-Ray assigns the level's number to the rays cast. The primary rays cast from the camera are level 0 rays. When a surface is hit by a ray of level n and it emits a new ray or several rays, they are said to be of level n+1. Tracking of the ray level avoids infinite recursions in some scenes and limits the rendering time by giving a maximum ray level limit. POV-Ray sets the limit to 5 by default while the user may increase it as needed, for example, on scenes with many mirrors.

3.2 Ambient Scene: Level 0 Rays

The idea of mimicking POV-Ray with the standard OpenGL pipeline leads immediately to the crucial question whether POV-Ray's ray casting can be replaced by the rendering pipeline. The approach of OpenGL is to project the scene's geometry to the camera's rendering plane and to change the pixel's colors as the geometry is rasterized and fragments are processed. POV-Ray's approach is the opposite. The rays are cast through the rendering plane and the color of the pixels are determined by the scene geometries hit by the rays. If only direct rays (e.g. level 0) are considered, both approaches should be interchangeable. The experiment of figures 3 and 4 proves the idea. Both images are of the same quality while POV-Ray's image took 4 seconds to render and the OpenGL's one 2.2 milliseconds (i7-920@2.66GHz, GeForce GTX 260). An acceleration factor of 1800 is the first achievement of this paper.

However, limiting rays to level 0 only prevents the scene geometry to be illuminated by light sources and only the ambient component of the light model affects visual appearance. To include light sources, rays of level 1 need to be introduced, providing the secondary rays cast from the intersection point to the light sources.

3.2 POV-Ray's Surface Model

To accelerate POV-Ray's level 1 rays in OpenGL, it is necessary to properly model POV-Ray's interaction of a ray with a surface. If the ray hits the object's surface any combination of four things might happen:

- absorption – the light ray or part of its intensity is absorbed, lowering its intensity and possibly altering its hue by absorption of some wavelengths only
- reflection – the light ray is reflected in one or more directions, secondary rays are cast

<table>
<tr><th>POV-Ray's Finish component</th><th>Required rays to be cast</th><th>Finish component description</th><th>Implementability in OpenGL</th></tr>
<tr><td>Ambient</td><td>no ray casting required</td><td>pigment multiplied by ambient component and global ambient intensity</td><td>yes, using ambient color and global ambient light</td></tr>
<tr><td>Diffuse</td><td rowspan="3">secondary ray is cast to each light source to test its visibility</td><td>for each visible light source compute sum of diffuse equations (Lambertian reflectance [Phong 1973]) of the pigment multiplied by diffuse component, and color of the ray cast to the light source. Brilliance parameter may modify distribution of the reflection.</td><td rowspan="3">yes, but light source visibility requires accelerated shadow algorithms to be used. For example, shadow maps or shadow volumes techniques can be used.

Per-pixel lighting requires color components to be computed using shaders because OpenGL's per vertex lighting provides often unacceptable results (see figure 1).</td></tr>
<tr><td>Phong</td><td>for each visible light source compute phong highlight [Phong 1973] of pigment, phong intensity, light source ray color. Pigment color affects the ray if metallic is specified.</td></tr>
<tr><td>Specular</td><td>for each visible light source compute specular highlight [Blinn 1977][Phong 1973] of pigment, specular intensity and light source ray color. Pigment color affects the ray if metallic is specified.</td></tr>
<tr><td>Reflection</td><td>one secondary ray cast in direction of reflection vector</td><td>color of reflection ray is multiplied with reflection intensity</td><td rowspan="2">OpenGL provides direct support for transparency, but robust implementation of reflection and refraction is not trivial. Possible solutions are discussed in the paper.</td></tr>
<tr><td>Transparency</td><td>one secondary ray cast into the object</td><td>the color of the ray is multiplied by transparent intensity. The direction of the ray may be affected by refraction</td></tr>
</table>

Table 1: Relation between POV-Ray materials and OpenGL materials

- refraction – transparent and translucent materials may cause a portion of the light ray to refract into the object
- fluorescence and emission – the surfaces may emit the light of different wavelengths, possibly altering the spectrum of the ray

POV-Ray models the surface absorption like OpenGL. It uses the RGB light color model. If the surface if purely blue, it absorbs all red and green component of incoming light rays. If it is 50% grey, half of the light ray's intensity is absorbed. Various reflection types are modelled by ambient, diffuse, specular, phong, and reflection material components. Refraction is modelled by the component of the same name, by the densities of refraction environments, and the amount of transparency of a given object. Fluorescence and emission effects are simulated by the ambient component or material components set in a way that the amount of outgoing light is bigger than the amount of incoming light. This may happen, for example, when one of the material components is bigger than 1.0.

However, modelling POV-Ray's material properties with OpenGL is not an easy task because the surface materials of POV-Ray are very robust and include material libraries and procedural functions. Because covering the material libraries and procedural functions would be time expensive work and would provide enough material for a paper dedicated just to this topic, this paper will be limited just to the core material modelling capabilities of POV-Ray.

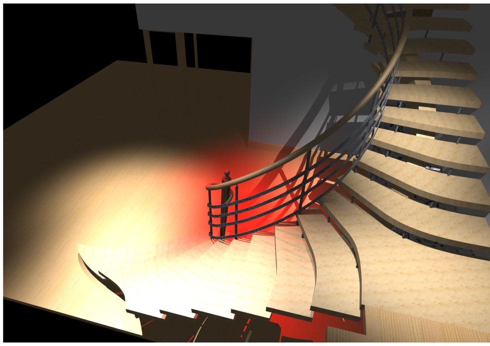

Figure 5: POV-Ray rendered image using rays level 0 and 1

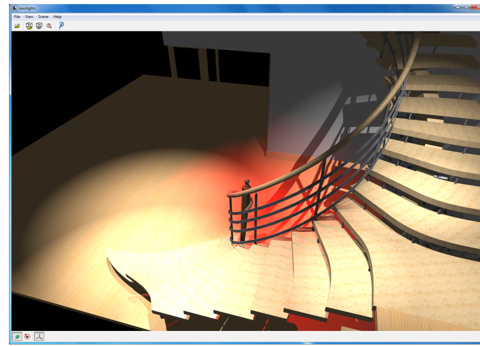

Figure 6: OpenGL rendered image using per-pixel lighting

POV-Ray calls the set of material surface properties "texture" (note the name collision with OpenGL's "texture"). The texture is composed of pigment, finish, and normal. The pigment can be a color, image map (e.g. 2D texture), or color map (procedural texture). Colors and image maps are the functionalities that have equivalents in OpenGL. Color maps are more complicated. They map floating point values in a range from 0.0 to 1.0 to color values specified in the map. The floating point values can be generated by various mapping functions and be modified by functions like turbulence and frequency. These procedural functions can be probably implemented using shaders, however they are out of scope of this paper.

Finish is the next item of POV-Ray's surface texture. Knowing the pigment, e.g. surface color at a given point – this may include texture mapping and filtering – finish gives the amount of ambient, diffuse, specular, phong, reflection, and refraction component. All these values are floats, usually between 0.0 and 1.0 to specify range from absence (value 0.0) to full intensity (1.0) of a given material component. Negative values and values over 1.0 can be used as well to create special or unrealistic effects. These intensities just multiply their value with the pigment color, resulting in the color of a particular component. Mimicking of all POV-Ray's finish material functionality in accelerated OpenGL is described summarized in the table 1.

The last POV-Ray's material surface property is normal. It is designed for surface normal manipulation. It can be specified by a bump map or a procedural approach. The procedural generators are out of the scope of this paper, so only bump maps are considered. The bump maps have mandatory support in OpenGL since version 1.3 [Segal 2001], thus they can be implemented.

3.3 Secondary Rays

After the investigation of primary rays (level 0) and color processing on POV-Ray's surface textures and seeing that it is possible to implement them in OpenGL, it is possible to come to the next step: POV-Ray's level 1 rays. When a level 0 ray cast from the camera hits a surface of an object, several secondary rays of level 1 may be cast. Secondary rays called shadow rays are sent to each light source to see whether anything is in the way and obscures the light source. If there is nothing in the way, the intersection point is illuminated by the particular light source and the color of the light source is assigned to the ray. If the light is obscured by an object, the black color is assigned to the ray instead.

If the surface has a non-zero reflection component, another secondary ray is cast in the direction of the reflection vector. If the surface is not completely opague, e.g. it is transparent or semi-transparent, another ray is cast into the object while a refraction effect may apply. The secondary rays are processed recursively in the same way as the primary rays, making secondary reflections and refractions, tertiary reflections, etc. At the end of each ray level n evaluation, the color of all the rays of the level n+1 are taken and included in the color computation of the ray of the level n. Finally, when level 0 ray evaluation is completed, its color is assigned to the pixel of the image.

Visual results are shown in the figure 5 for POV-Ray while using level 0 and 1 rays only, and figure 6 for the OpenGL accelerated implementation.

POV-Ray's light type	Description	Implementability in OpenGL
Point light	The light placed in the scene shining equally in all directions	Similar to OpenGL's point light except that OpenGL's light does not cast shadows by default. One of shadow techniques need to be utilized. Lighting should be implemented per-pixel, for example, in shaders.
Parallel light	The light whose rays come in parallel in certain direction	Similar to OpenGL's directional light with the exception of shadows. One of shadow techniques need to be used.
Spot light	Like the point light but the light is restricted to a cone in some direction. The light intensity in the cone can be modulated. POV-Ray uses radius, falloff, and tightness parameters.	Similar to OpenGL's spot light except shadows and light intensity modulation. Shadows need to be implemented. Intensity modulation inside the cone is different from the OpenGL's built-in spotlight. However, shaders are usually used to implement per-pixel lighting and POV-Ray's light modulation can be computed there as well.
Cylindrical light	Like spot light but it is constrained by a cylinder. It is useful for effects light laser beams. Technically, it is not based on parallel light as could be expected, but on point light whose rays are constrained by the cylinder.	Can be implemented in OpenGL shaders. Shadow technique needs to be used.
Area light	Finite 1D or 2D rectangular area providing flat panel light. Technically, it is implemented using array of point lights. As a side effect, the light provides soft shadows.	Can be implemented as an array of point lights. Alternatively, some area light model [Au 2007] implemented using shaders can be used for instance.

Table 2: POV-Ray's lights and their implementability by OpenGL

3.4 POV-Ray Lights

Casting of secondary rays, particularly shadow rays cast towards light sources, is influenced by the light type. As can be expected, POV-Ray uses a more advanced lighting model than built-in OpenGL lights. Anyway, the limited built-in light model was made obsolete in OpenGL 3.0 and programmers are encouraged to create their own lighting effects using shaders.

POV-Ray uses several types of lights: point light, parallel light (like OpenGL's directional light), spot light, cylindrical light, and area light. The first three have equivalents in OpenGL, the fourth should be possible to implement in shaders and the last one – area light – can be implemented as an array of point lights. That is actually the way that area light is implemented in POV-Ray. All the details of POV-Ray's lights and their implementability by OpenGL are described in the table 2. Implementability was verified for point light, directional light and spot light. Cylindrical light is rarely used, and area light is just the array of point lights in POV-Ray. To prove implementability of the lights, point light, directional light and spot light were implemented in OpenGL's shaders. The results are shown in the figures. Because of the space limitations, we show just the spot light scene as the spot light is the most complex type when considering mentioned light

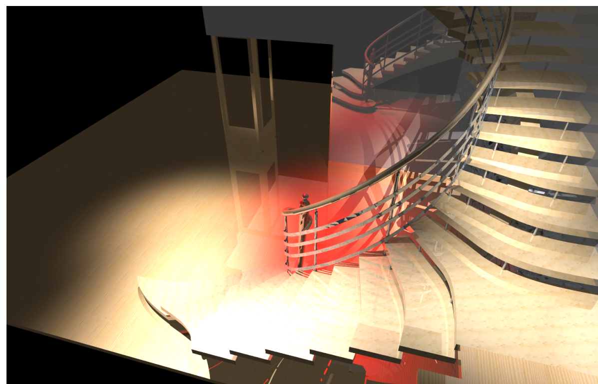

Figure 7: POV-Ray rendered image using rays level 0,1, and 2

types and their POV-Ray implementation. A summary of lights implementability is shown in the table 2.

3.5 Higher Level Rays

When we attempted to analyze implementability of level 2 rays in OpenGL, we found it very difficult, particularly when considering the general case. Following the goals set in the beginning of this paper, this section successfully identifies one particular difficulty: to mimics POV-Ray rays starting from level 2. Additionally, the section is going to discuss available options that should be addressed in the future.

Rays of higher level starting from level 2 bring very cool effects, like mirroring, reflections and refractions. One POV-Ray example is shown in the figure Error: Reference source not found. However, these rays are not straightforward to implement for the general case. Some GPU approaches follow that can be used to reach similar effects as those created by level 2 rays:

- scene mirroring – there are various approaches to implement mirrors in OpenGL. Usually, they just duplicate the scene behind the mirroring surface while using stencil test to limit rendering just to mirroring surface.
- render to texture – the mirrored or reflected geometry can be pre-rendered to the texture that is applied to the surface afterwards.
- environment mapping – similar to render to texture approach, but it usually renders all surrounding environment (360 degrees) to, for example, cube map. The pre-rendered environment is then mapped to the geometry using, for instance, texgen OpenGL functionality or shaders. Effects like mirrors or surface reflections are easily implementable using environement mapping.
- GPU raytracing – there were number of attempts to perform ray tracing on GPU, such as [Garanzha 2010][Shih 2009][Horn 2007][Gunther 2007].

The first three OpenGL-based options seem appropriate to create nice reflections and refraction effects on planar surfaces. However, they are breaking projection coherency that was present with primary rays and shadow level 1 rays. Breaking of this coherency may result in a huge number of projections and pre-rendering to texture. Theoretically, each surface that is not coplanar with any other surfaces may require its own projection. Such solution may turn to be very performance expensive. Moreover, curved surfaces, such as NURBS, exhibit even more problems. They can not be processed directly as they are not planar surfaces. They can be tesselated, but to reach high quality visual results, too many not coplanar surfaces may be produced, possibly harming performance too much.

Another approach can be to start GPU ray tracing at level 2 rays. Although GPU ray tracing still does not reach the performance of the standard OpenGL rendering paradigm, it may outperform OpenGL approaches for level 2 rays. Further investigation would be necessary to clarify the best approach that may even lead to a hybrid solutions – rendering big planar surfaces in OpenGL and ray tracing the small or curved surfaces.

4 Experiments

All the algorithms developed in this paper were tested as a part of our Lexolights open source project available at: http://lexolight.sourceforge.net. The website includes win32 binaries covering the functionalities mentioned in this paper. Namely, all POV-Ray core material functionality is implemented, e.g. all major elements of POV-Ray's texture finish, with exception of reflection and refraction elements that would require support of level 2 rays. Next covered area is lights that include point, directional, and spot light while we used shadow maps [Williams 1978] and LiSPSM [Wimmer 2004] for shadow rays visibility tests. Lexolight is implemented using OpenSceneGraph (http://www.openscenegraph.org) – high level rendering library built on the top of OpenGL. As some of our algorithms were better suited to be included directly in OpenSceneGraph, such as POV-Ray scene converter and exporter, we submitted them to be included in the upcoming release for the profit of the open source community.

4.1 Performance

To evaluate the performance gains, several measurements were made for POV-Ray rendered scenes. Then, the same scene was rendered by hardware accelerated OpenGL. The results are summarized in the table 3. We used the scene composed of approximately 83000 triangles visualized on high and low performance CPUs and a variety of graphics cards ranging from hi-end, through mobility versions, to old low-end GPUs. POV-Ray rendering took from few seconds to about a minute. Rendering of the same scene for level 0 and level 1 rays using POV-Ray's accelerated OpenGL approach took less than 150ms even on very old mobile fill-rate limited graphics card. For nowadays hi-end graphics cards, the speed up factor stayed far above 1000. We consider such speed up an interesting result. It shows a potential to accelerate level 0 and level 1 rays of ray tracers. Another option can be to further investigate acceleration of level 2 rays or to consider whether a hybrid solution would be the best option for close-to-photorealistic real-time rendering.

	Level 0 Rays Acceleration			Level 1 Rays Acceleration			Level 2 Rays
	POV-Ray rendering (level 0 rays)	OpenGL ambient rendering	Speed-up factor	POV-Ray rendering (level 1 rays)	OpenGL per-pixel ligting	Speed-up factor	POV-Ray rendering (level 2 rays)
i7-920 @2.66GHz, GeForce GTX 260	4s	2.2ms	1800	14s	4.4ms	3200	21s
Core 2 Duo @ 2.00GHz, Radeon HD 3670 Mobility	8s	7.4ms	1100	21s	18ms	1200	33s
Athlon XP 2000+, Radeon HD 2600 XT	12s	11.4ms	1100	37s	20.5ms	1800	58s
Core 1 Duo @ 1.83GHz, Radeon X1300M	13s	33ms	400	36s	133ms	270	55s

Table 3: Performance comparison
(screen size: 1440x1050, one omnidirectional light, 83000 triangles)

5 Conclusions

This paper investigated the realization of photorealistic rendering in real-time, accelerated by the graphics hardware. The investigation was made on a case study by mimicking of POV-Ray with OpenGL's rendering paradigm accompanied with the latest programmable shader technologies.

The presented study proved that the core POV-Ray material and lighting functiontionality can be implemented in accelerated OpenGL. It turned out that it is very easy to accelerate primary rays (e.g. level 0 rays) cast by POV-Ray when rendering the scene and a speed up factor of 400 was measured even on very old hardware. Secondary rays of level 1 were accelerated as well while together with level 0 rays, the speed up factor was over 3000 for modern hardware. Rays of level 2 and higher turned out to be difficult to be implemented in accelerated OpenGL. They may require additional research efforts and using of advanced rendering techniques as was discussed in section 3.5.

Future research should extend acceleration of rays of level 0 and 1 to higher level rays. Although it may be difficult to do so, it would enable additional visual effects and may even lead to ideas of accelerating global illuminations methods, such as radiosity.

References

AU, A. 2007. A simple area light model for GPUs. In *Shader X5*, W. Engel, Ed. Charles River Media, Chapter 2.1, 63—67.

APPEL, A. 1968. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30--May 2, 1968, Spring Joint Computer Conference* (Atlantic City, New Jersey, April 30 - May 02, 1968). AFIPS '68 (Spring). ACM, New York, NY, 37-45. DOI= http://doi.acm.org/10.1145/1468075.1468082

BLINN, J. F. 1977. Models of light reflection for computer synthesized pictures. SIGGRAPH Comput. Graph. 11, 2 (Aug. 1977), 192-198. DOI= http://doi.acm.org/10.1145/965141.563893

GARANZHA, K., LOOP, C., 2010. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing, Computer Graphics Forum 29, 2. Proceedings of Eurographics 2010, Norrköping, Sweden.

GUNTHER, J., POPOV, S., SEIDEL, H., AND SLUSALLEK, P. 2007. Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In Proceedings of the 2007 IEEE Symposium on interactive Ray Tracing (September 10 - 12, 2007). IEEE/Eurographics Symposium on Interactive Ray Tracing. IEEE Computer Society, Washington, DC, 113-118. DOI= http://dx.doi.org/10.1109/RT.2007.4342598

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree GPU raytracing. In *Proceedings of the 2007 Symposium on interactive 3D Graphics and Games* (Seattle, Washington, April 30 - May 02, 2007). I3D '07. ACM, New York, NY, 167-174. DOI= http://doi.acm.org/10.1145/1230100.1230129

PHONG, B. T. 1973 *Illumination for Computer-Generated Images.* Ph.D. Thesis. UMI Order Number: AAI7402100., The University of Utah.

SEGAL, M., AKELEY, K. 2001. *The OpenGL Graphics System: A Specification (Version 1.3)*. Silicon Graphics, Inc. Available at: http://www.opengl.org/documentation/specs/

SHIH, M., CHIU, Y., CHEN, Y., AND CHANG, C. 2009. Real-Time Ray Tracing with CUDA. In Proceedings of the 9th international Conference on Algorithms and Architectures For Parallel Processing (Taipei, Taiwan, June 08 - 11, 2009). A. Hua and S. Chang, Eds. Lecture Notes In Computer Science, vol. 5574. Springer-Verlag, Berlin, Heidelberg, 327-337. DOI= http://dx.doi.org/10.1007/978-3-642-03095-6_32

WHITTED, T. 1980. An improved illumination model for shaded display. *Commun. ACM* 23, 6 (Jun. 1980), 343-349. DOI= http://doi.acm.org/10.1145/358876.358882

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. SIGGRAPH Comput. Graph. 12, 3 (Aug. 1978), 270-274. DOI= http://doi.acm.org/10.1145/965139.807402

WIMMER, M., SCHERZER, D., PURGATHOFER, W. 2004. Light Space Perspective Shadow Maps, In Rendering Techniques 2004 (Proceedings Eurographics Symposium on Rendering), p. 143-151. June 2004.

WRIGHT, R. S. AND LIPCHAK, B. 2004 *OpenGL Superbible (3rd Edition)*. Sams.

Additional Images

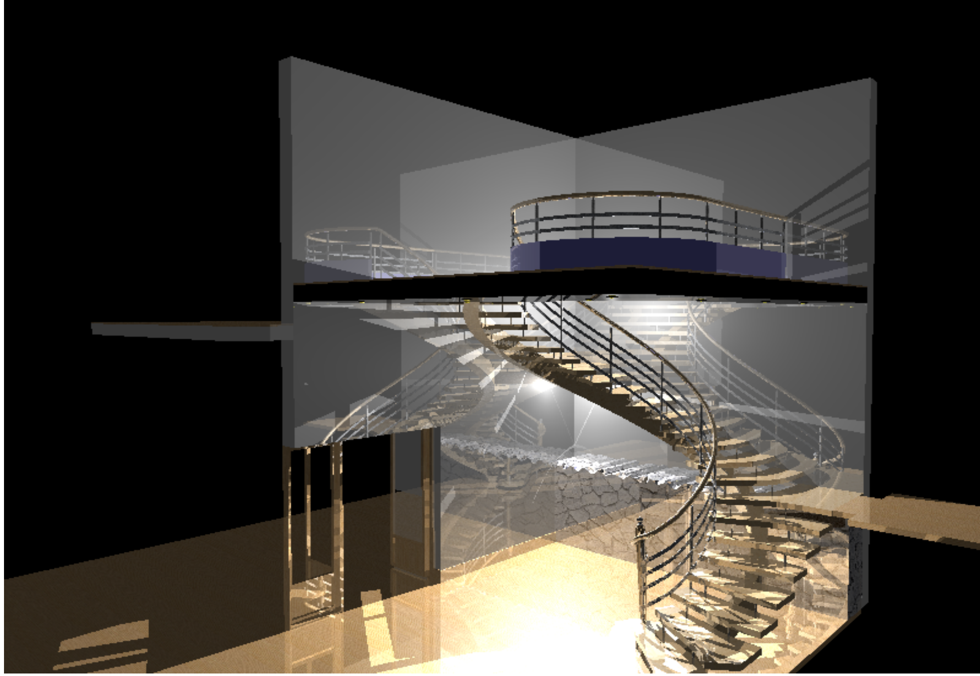

Figure 8: Visualization of stairs with reflective floor and walls made of glass