Screen Space Cone Tracing for Glossy Reflections

Lukas Hermanns[1], Tobias Franke[2], and Arjan Kuijper[1,2(✉)]

[1] Technische Universität Darmstadt, Darmstadt, Germany
[2] Fraunhofer IGD, Darmstadt, Germany
arjan.kuijper@igd.fraunhofer.de

Abstract. Indirect lighting (also Global Illumination (GI)) is an important part of photo-realistic imagery and has become a widely used method in real-time graphics applications, such as Computer Aided Design (CAD), Augmented Reality (AR) and video games. Path tracing can already achieve photo-realism by shooting thousands or millions of rays into a 3D scene for every pixel, which results in computational overhead exceeding real-time budgets. However, with modern programmable shader pipelines, a fusion of ray-casting algorithms and rasterization is possible, i.e. methods, which are similar to testing rays against geometry, can be performed on the GPU within a fragment (or rather pixel-) shader. Nevertheless, many implementations for real-time GI still trace perfect specular reflections only. In this work the advantages and disadvantages of different reflection methods are exposed and a combination of some of these is presented, which circumvents artifacts in the rendering and provides a stable, temporally coherent image enhancement. The benefits and failings of this new method are clearly separated as well. Moreover the developed algorithm can be implemented as pure post-process, which can easily be integrated into an existing rendering pipeline.

1 Introduction and Motivation

In this work a novel method for real-time glossy reflections is presented. This method can be implemented as a pure post-process, which simplifies the effort, of integrating it into an existing graphics application, considerably. Similar algorithms are already implemented, not only in video games, but also in CAD and AR applications [2,4,16] and reflections for indirect lighting are essential in GI simulations [14]. These reflections increase the photo-realism of Computer Generated Imagery (CGI) drastically.

Indirect lighting is implemented in a broad range of GI applications since many years, particularly in film productions such as Terminator 2 (1991), Avatar (2009), and the first feature-length computer animated motion picture Toy Story (1995) [5]. From the very beginning, developers are striving to reduce the calculation overhead or rather to accelerate the rendering process. This is because accurate rendering can take several hours or days even on high performance computer systems, depending on the scene and shading complexity. Therefore the

S. Lackey and R. Shumaker (Eds.): VAMR 2016, LNCS 9740, pp. 308–318, 2016.
DOI: 10.1007/978-3-319-39907-2_29

costs and duration to produce photo-realistic imagery can be very high. To solve this, there are many algorithms to achieve photo-realism by approximating the influence of indirect light, for both real-time and non real-time rendering [12]. Some of these simulate the indirect light by distributing many small direct light emitters throughout the scene, called Virtual Point Light (VPL), but which then requires new optimizations or rather hierarchies of spatial data structures [10,14]. Still others store and propagate the indirect light through the scene within a volume, called Light Propagation Volume (LPV), which then causes a large memory footprint for large scenes and high resolution volumes. However, methods like VPL and LPV lack an important part of indirect light, which will be discussed in this work and is shown in the above image. Our novel approach for real-time reflections is a rough approximation but still produces plausible effects of indirect lighting. It is fast, fully dynamic, and can easily be integrated into an existing rendering pipeline, because it is a pure post-process. Hence, the result of an intermediate render pass is the only input for this effect. No information about the scene is required.

2 Related Work

In this section we will analyze several approaches for real-time reflections and then make a comparison to summarize their benefits and failings.

2.1 Analytical Area Lights

We start out with a simple method for glossy reflections which deals with direct lighting only: Analytical Area Light (AAL). The three basic models of light shapes are: Directional-, Point-, and Spot light. In classical fixed function pipelines these are the usual lighting primitives to shade geometry. All of them only have a position and optionally a direction, but neither an area nor volume. This makes it very easy and fast to determine the light intensity for each pixel, because only a scalar product and the distance between that pixel and the respective light source must be computed. However, every light source in reality has a volume. This can be simulated more plausible with AALs, which cover a specific area on the screen. The formulas for these light models resemble the basics of ray tracing. Ray intersection tests are performed against all AALs for every pixel, otherwise the nearest distance between the current pixel and the primitive light geometry is computed. Since these light models require more complex and more flexible calculations, they became popular with the advances of programmable shader units in modern hardware, but the research around them has begun long before [3]. In addition, glossy reflections can be simulated easily by adding a further light attenuation depending on the surface roughness. Contrariwise, they only allow direct lighting, since with indirect lighting everything is treated as a light source, but AALs only provide limited shapes, such as cuboids, spheres, cylinders, and capsules. Although there are more complex AALs, such as fractals, these shapes are not enough to represent an entire reflectable scene. Therefore, one can use them only as direct light emitters.

Advantages: (1) accurate calculation for incident radiance and (2) easy to implement. Disadvantage: direct lighting only.

2.2 Planar Reflections

We continue with one of the oldest methods for reflections of indirect lighting, which is still used in modern 3D engines: Planar Reflections. For planar reflections there are two major methods: the first one is to use a stencil buffer and the second one is to use a render target (also framebuffer). With the first method, the stencil buffer operates as the name implies like a stencil. It allows to reject pixels outside the stencil pattern. The reflected scene should only be visible inside this pattern, and to generate it, the reflective geometry is rendered with certain render states enabled. We can divided this algorithm into the following steps: 1. Clear frame- and stencil buffers. 2. Render scene with default settings but without reflective geometry. 3. Render reflective geometry into stencil buffer. 4. Render scene inside the stencil pattern with mirrored view transformation. In the second method, the steps of rendering the actual and the mirrored scene are in the reverse order. We first have to render the mirrored scene into the render target. Then the render target serves as the source of the texture (and also of the reflective light color), which is mapped onto the reflective geometry, when the actual scene is rendered. In some cases this method may be used but it has a larger memory footprint. This is a very simple method which allows perfect specular reflections with correct geometry reconstruction. The correct reconstruction originates from the fact that the scene is rendered a second time, which can be very time consuming. This also means that the performance depends on the scene complexity. Another disadvantage is that this only works for planar reflectors such as mirrors or flat water puddles, because the mirroring is due to a mirrored matrix transformation, which does not allow any distortions. Note that this method has nothing to do with ray tracing or something similar. It is just a secondary rasterization pass. However, there are also hybrid approaches which combine geometry- and image-based rendering, such as Forward Mapped Planar Mirror Reflections [1]. Such a technique is used in the idTech 3 game engine (1999) and even older ones.

Advantages: (1) perfect reconstruction of reflected geometry and (2) Easy to implement. Disadvantages: (1) Depends on scene complexity and (2) Planar reflectors only.

2.3 Environment Maps

Another still widely used method are (localized-) Environment Maps (also Cube Maps). In this case the scene is rendered from six view angles into a cube map at a localized position. A cube map internally consists of six texture layers, one for each cube face. This captures a 360 degree view in a single texture. These cube maps are typically placed by an artist within a world editor and generated during the world building process this process is also called baking. Such a cube map stores the entire light influence for the point where the cube map

was rendered. Usually the cube maps are additionally pre-filtered (or blurred) to efficiently simulate a basic glossiness for all surfaces [8]. The blurring simulates a wider distribution of reflection rays, which causes a glossy appearance. As already mentioned: the reflections from each cube map are only correct for a single 3D point. Whenever a cube map is used for other points, the reflections are only correct for infinitely distant light. Such a candidate is the (nearly) infinitely distant sun. To overcome this restriction several cube maps are generated in a scene, between which the application must choose at runtime. To avoid the popping effect (discrete switching between textures), some applications interpolate between the N nearest cube maps (the popping effect is visible in games like Half-Life 2 and Portal using the Source Engine). A further improvement are parallax corrected cube maps [13], which can adjust the singular location to a cubic environment. However, even this method is severely limited due to its cubic nature and needs further adaptation by artists. This technique is used in the game Remember Me (2013) and the Source Engine (2004).

Advantages: (1) Easy to implement and (2) Very fast. Disadvantages: (1) Must be placed by artist, (2) Limited to fixed count, and (3) Only correct for infinitely distant light.

2.4 Image Based Reflections

We now continue with ray tracing like methods: Image Based Reflections (IBR). In this method several IBR proxies are placed by an artist in the world editor, whereat an IBR proxy is a box which captures a small volume in the scene which is then rendered into a 2D texture. For each reflection ray an intersection test against the plane, which is spanned by that box, is computed. These ray against plane tests are very fast but this reflection model is only useful for nearly planar reflectors such as building facades or streets for instance. Although a single plane intersection test is fast some form of hierarchy is required if many IBR proxies are used (e.g. 50+) to avoid testing against all planes for every pixel. In addition the method is inappropriate for perfect specular reflections since the geometry inside an IBR box is approximated by a plane. For glossy reflections the results are reasonable because the distorted planar reflector can not be perceived exactly by the viewer. This technique is used in the game Thief 4 (2014) from 2014 and the Unreal Engine 3 (UE3) (2006).

Advantages: (1) Good visual approximation and (2) Very fast. Disadvantages: (1) Must be placed by artist and (2) Nearly planar reflectors only.

2.5 Screen Space Local Reflections

Finally we look at a pixel based ray tracing method: Screen Space Local Reflections (SSLR). In a nutshell: we transform the reflection ray from view space into screen space, and then move along this ray until we step through the depth buffer. By this algorithm, we hope to find the intersection of a ray against the scene geometry, which is stored in form of the depth buffer. That means, in particular, we can only find intersections with geometry, which is already visible

on the screen. This is why it is called a screen space effect. Many SSLR implementations perform this simple ray hit search only, which is commonly called a linear ray march. Once the first hit is found, a binary search for refinement can be done. To simulate glossy reflections most applications apply subsequent blur passes to the reflection color buffer. But there are also alternatives where several rays are casted and the average color forms the result. A frequently used optimization, to find the ray intersections, is to render this pass only at half resolution, which yields to acceptable results when a blur pass is applied anyway. The advantage of SSLR is that it allows reflections of arbitrary geometry (assumed the geometry is visible on screen). It can additionally be implemented as a pure post-process or a set of consecutive post-processes, which alleviates the effort to integrate it into a present 3D engine. It is moreover independent of the scene complexity because the reflections are fetched from the color buffer of a previous render pass. The calculations are computed for every pixel which makes the algorithms effort proportional to the number of pixels on the screen. Disadvantages are primarily the limitations of the screen boundary and the hidden geometry problem. This technique is used in UE3 and the game Killzone Shadow Fall (2013). Such ray tracing approximations are state of the art in the field of post-processing [7,11].

Advantages: (1) Reflection of arbitrary geometry and (2) Pure post process. Disadvantages: (1) Hiden geometry problem and (2) Limited to screen space.

2.6 Screen Space Cone Tracing

Based on local reflections in screen space, we move on to a method which traces cones instead of rays: SSCT [6]. Again, the cone is an approximation for many reflection rays. The process is very similar to SSLR but in each iteration of the ray march we sample from a certain MIPmap of the depth texture, which is a further approximation of the actual cone. By relying on MIP-maps certain integration errors are unavoidable. This is due to solid angles that can subtend either flat spaces or multiple pieces of geometry. It can manifest as alias or temporal inconsistency when moving the camera view. Such errors will increase notably as the cone angle size increases to simulate more glossy appearances. Thus for SSCT using a proper texture filter is crucial. Next to nave MIP-mapping, manual filtering by using slices of a 3D texture, where each slice is a Gaussian blurred version of the original texture [9], has also been tested. The results are significantly better in quality but loose the fast texture accesses since we sample from a high resolution 3D texture.

Advantages: (1) Reflection of arbitrary geometry, (2) Pure post process, and (3) Glossy reflections with arbitrary roughness. Disadvantages: (1) Hiden geometry problem, (2) Limited to screen space, and (3) Artifact avoidance is very slow.

2.7 Hi-Z Cone Tracing

The last presented method is from the book GPU Pro 5: HZCT [15]. The remarkable concept with this method is on the one hand that the ray tracing and cone

tracing parts are clearly separated and on the other hand that the ray tracing process is accelerated with hierarchical buffers. These hierarchical buffers will be generated during the post-process in each frame and allow a faster, stable, and precise ray tracing in screen space. While SSLR and SSCT use a binary search to refine the intersection point, HZCT is much more target-oriented. However, a disadvantage is that HZCT does not allow tracing rays which point towards the camera. This is due to the ray setup in combination with the hierarchical buffers. As soon as an intersection has been detected, the cone tracing pass integrates all incident radiance from the intersection point to the ray origin using the visibility buffer. The cone approximation is quite similar to that in SSCT but it is combined with a visibility buffer to circumvent invalid integration over a large solid cone angle.

Advantages: (1) Reflection of arbitrary geometry, (2) Pure post process, (3) Glossy reflections with arbitrary roughness, (4) High stability and precision, and (5) Acceleration with hierarchical buffers. Disadvantages: (1) Hiden geometry problem, (2) Limited to screen space, (3) Unable to ray trace towards the camera, and (4) Complex to implement.

2.8 Comparison

The only method which provides perfect reconstruction of reflected geometry here are planar reflections. All the other techniques merely approximate the reflections in a more or less coarse manner. Unfortunately, planar reflections are inappropriate for every reflective geometry which has not a planar shape. Moreover the necessity of re-rendering the scene makes it unfeasible for post-processing. Nevertheless, planar reflection is still a valuable fallback method, especially when others fail with receiving scene information. A very efficient method here are environment maps, which is at least beneficial as fallback, too. Primarily because they can be pre-computed. But either with or without parallax correction, using environment maps requires some adaptation by artists, i.e. the reflections can not be computed as a pure post-process. At least the localization within the scene must be managed with a world editor. This also applies to IBR. From the screen space ray- and cone tracing methods, HZCT seems to be the most advanced technique. The major benefits over SSLR and SSCT are the stability, the precision, and the acceleration. However, like most ray tracing algorithms in screen space, it is limited to the screen boundary and we have the hidden geometry problem. Even HZCT does not solve these restrictions, but this is where fallback methods are considered.

3 Screen Space Cone Tracing

The core idea of this work is based on SSCT [6]. However, the implementation is based on HZCT and will be slightly augmented with a fallback for rays pointing towards the camera, to maximize the extent of reflection rays in screen space. We will then provide a quality comparison to the original HZCT and a method from the field of SSLR. The previous section presented benefits and failings of related work. The most advanced technique for screen space reflections seems to be HZCT. However, it does not cover the maximum extent of the screen space, because rays pointing towards the camera can not be gathered. Fortunately, this method separates ray tracing and cone tracing, thus the ray tracing part can be augmented with linear ray marching for exceptional cases. After we review the algorithm in detail we will take a closer look at a couple of SSLR methods. One of them is a technique used in Killzone Shadow Fall, in which a mask buffer is generated alongside the ray tracing process. This is later used to enhance the blurring of the ray trace color buffer, which is required for glossy reflections in this SSLR method. The blurred variants are stored inside the MIP-chain.

3.1 Overview of HZCT

Since the method is based on HZCT we continue with an overview of the different passes as proposed in GPU Pro 5. The core algorithm can be divided into five steps: **Hi-Z Pass**: Generates the entire MIP chain of the Hi-Z buffer. This hierarchical buffer is required to accelerate the ray tracing. **Pre-Integration Pass**: Generates the entire MIP chain of visibility buffer. This hierarchical buffer is required for accurate radiance integration during cone tracing. **Pre-Convolution Pass**: Pre-filters the color buffer. A blur pass of the default MIP-map generation can be used here. **Ray-Tracing Pass**: Finds the ray intersections. This pass requires the Hi-Z buffer. **Cone-Tracing Pass**: Integrates incident radiance for a solid cone angle. This pass requires the Hi-Z-, the color-, and the visibility buffers.

The last two steps can be combined into a single shader. It is also possible to split them up, but this should only be taken into account when it really matters, because particularly state changes to the framebuffer binding are very time consuming and it would also increase the memory footprint.

3.2 Competitor: SSLR

Before we compare SSCT with SSLR methods we first take a closer look at glossy reflections in SSLR, such as Killzone (- Shadow Fall). Although SSLR only uses ray tracing there are several methods to implement glossiness. The nave approach is to cast multiple rays and take the mean value of the color samples. A smarter approach is to split the algorithm into multiple passes again and blur the ray trace color buffer. This is how it is done in Killzone. A mask buffer is used to enhance the blurring of the ray trace color buffer. We can divide this algorithm into the following three steps: 1. **The Ray-Tracing Pass** generates the ray

trace color buffer and mask buffer. 2. **The Blur Pass** generates the MIP levels for both the color- and mask buffer with a Gaussian blur. 3. **The Reflection Pass** draws final reflections on the screen.

3.3 Optimizations

There are many ways for optimizations in post-processing effects. Particular for ray tracing effects with multiple stages. As already mentioned, a frequently used optimization is to render the respective effects only at half resolution. However, this is usually only justifiable when blur passes are involved, which hide the lower texture quality. Another issue is memory efficiency. Incoherent memory access stalls the graphics pipeline, due to wasted memory bursts (larger blocks of data for better cache utilization). The game Thief 4 approximates the normal vectors with (0, 0, 1)T for better memory coalescing. The bump mapping (effect to enhance shading appearance on textures) is implemented supplementary as a post-process, i.e. the normal deviation is applied after the ray tracing. Furthermore gradient-based texture operations should be avoided within dynamic branching, or they should be at least moved out of flow control to prevent divergence. They may force the pipeline to load texture data for program paths where they are not needed, due to the massive parallelism on GPUs. In practice, this means that the intrinsics textureLod/textureGrad should be preferred over texture (in GLSL) and SampleLevel/SampleGrad should be preferred over Sample (in DirectX High Level Shading Language (HLSL)) respectively.

4 Results and Discussion

The example images compare our method SSCT with the original method HZCT, then with the related method SSLR. All rendering times are determined with a hardware timer query, which is very precise and they only reflect the rendering duration of the effect, excluding the scene rendering. Figure 1 compares SSCT with HZCT on a wooden cylinder on the floor which reflects the surrounding tiles. In Fig. 2 the outside of a window is reflected on the floor. Here, HZCT only captures a small amount of that window, but SSCT extends the range of the reflective area. Moreover HZCT ignores the reflections on the front wall completely, whereas SSCT additionally reflects the floor underneath the window.

In Fig. 3 glossy reflections are visible on the floor and the walls of the Sponza Atrium. The scene itself is rendered neither with shadow mapping nor complex BRDF models. Only a single directional light source is embedded and only the stone floor reflects indirect light. Here, SSCT produces much better results than SSLR for long ray traversals. Artifacts in SSLR are visible on the walls, due to a constant step size in the ray marching. See also Table 1. The roughness factor is also used for the normal deviation, to increase the rough appearance.

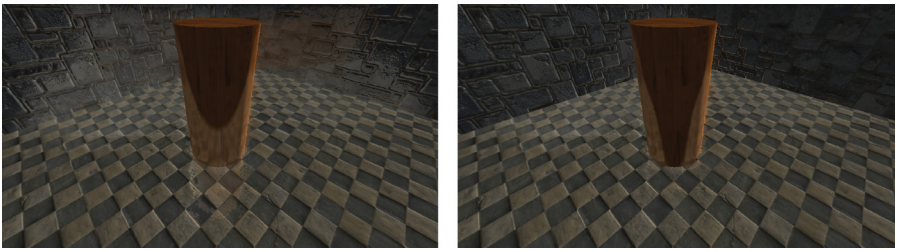

Fig. 1. Looking down to the floor at a cylinder, which reflects the tiles (960×540 resolution. Left: SSCT rendering time ≈ 3.49 ms Right: HZCT rendering time ≈ 1.04 ms

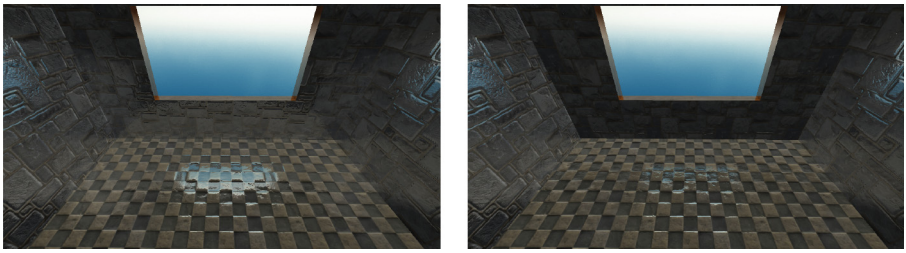

Fig. 2. Looking down to the floor, which reflects the outside of a window (960×540 resolution). Top: SSCT rendering time ≈ 2.25 ms. HZCT rendering time ≈ 1.12 ms

Table 1. Performance results for various resolutions and roughness factors.

Resolution	Roughness	SSCT performance	HZCT performance	SSLR performance
640×480	0.0	0.76 ms	0.69 ms	1.14 ms
640×480	0.1	0.81 ms	0.74 ms	0.68 ms
800×600	0.0	1.12 ms	1.02 ms	1.69 ms
800×600	0.1	1.17 ms	1.10 ms	0.99 ms
960×540	0.0	1.50 ms	1.10 ms	1.86 ms
960×540	0.1	1.54 ms	1.19 ms	1.09 ms
1280×768	0.0	2.45 ms	1.92 ms	3.31 ms
1280×768	0.1	2.66 ms	2.07 ms	1.95 ms
1920×1080	0.0	4.92 ms	4.07 ms	6.09 ms
1920×1080	0.1	5.30 ms	4.49 ms	4.02 ms

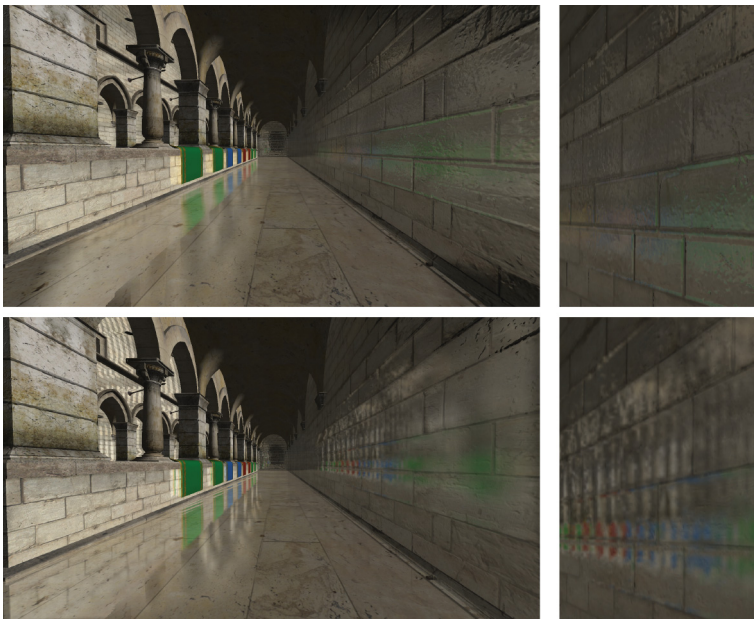

Fig. 3. Glossy reflections are visible on the floor and the walls of the Sponza Atrium (1920×1080 resolution). Top: SSCT rendering time ≈ 5.30 ms, enlarged image detail $\approx$. Bottom: SSLR rendering time ≈ 5.19 ms, enlarged image detail

5 Conclusion

We have seen a novel method for local glossy reflections called SSCT. The implementation has been presented in detail and a comparison to other state of the art methods has been shown as well. Our method is based on HZCT and is augmented with a fallback for special cases. The advantages of our method over other SSLR methods are, that the cone tracing produces more plausible looking glossy reflections and it can be clearly separated from the ray tracing process. Additionally the input parameters for SSCT are more correlated with the material configuration of a BRDF. This is due to the cone tracing, which is derived from multiple ray samples. That makes it much easy for artists to create plausible effects in 3D scenes. In contrast, most SSLR implementations merely blur the entire ray trace color buffer, which is out of proportion to BRDF and material parameters. Moreover the cone tracing in SSCT considers the amount of cone intersection with the scene and takes several texture samples, while glossiness in SSLR is usually based on a single and unweighted texture sample from the blurred ray trace color buffer. We can, though, make use of ideas implemented in SSLR: because of the modular nature of SSCT, we can further enhance the image quality by using a mask buffer in the blur pass for the color buffer. However, our method still lacks solutions for the hidden geometry problem and the screen boundary limitations. Only workarounds do exist to circumvent these

restrictions. We can summarize, therefore, local reflections in pure screen space effects are still an unsolved area of indirect lighting. Nevertheless, in prepared scenes and in combination with other reflection techniques it can be very useful with satisfying frame rates.

References

1. Bastos, R., Stürzlinger., W.: Forward mapped planar mirror reflections. Technical report, University of North Carolina at Chapel Hill (1998)
2. Bauer, F., Knuth, M., Kuijper, A., Bender, J.: Screen-space ambient occlusion using a-buffer techniques. In: Computer-Aided Design and Computer Graphics, CAD/Graphics 2013, pp. 140–147. IEEE (2013)
3. Campbell, III, A.T., Fussell, D.S.: An analytic approach to illumination with area light sources. Technical report, University of Texas at Austin, Austin, TX, USA (1991)
4. Engelke, T., Becker, M., Wuest, H., Keil, J., Kuijper, A.: MobileAR browser - a generic architecture for rapid AR-multi-level development. Expert Syst. Appl. **40**(7), 2704–2714 (2013)
5. Henne, M., Hickel, H.: The making of "toy story". In: Proceedings of the 41st IEEE International Computer Conference, COMPCON 1996, pp. 463–468 (1996)
6. Hermanns, L., Franke, T.A.: Screen space cone tracing for glossy reflections. In: ACM SIGGRAPH 2014 Posters, SIGGRAPH 2014, p. 102:1 (2014)
7. Johnsson, M.: Approximating ray traced reflections using screen-space data (2012)
8. Kautz, J., McCool, M.D.: Approximation of glossy reflection with prefiltered environment maps. In: Proceedings of the Graphics Interface 2000 Conference, pp. 119–126 (2000)
9. Kuijper, A., Florack, L.: The relevance of non-generic events in scale space models. Int. J. Comput. Vis. **1**(57), 67–84 (2004)
10. Limper, M., Jung, Y., Behr, J., Sturm, T., Franke, T.A., Schwenk, K., Kuijper, A.: Fast, progressive loading of binary-encoded declarative-3D web content. IEEE Comput. Graphics Appl. **33**(5), 26–36 (2013)
11. McGuire, M., Mara, M.: Efficient GPU screen-space ray tracing. J. Comput. Graphics Tech. (JCGT) **3**(4), 73–85 (2014)
12. Schwenk, K., Voß, G., Behr, J., Jung, Y., Limper, M., Herzig, P., Kuijper, A.: Extending a distributed virtual reality system with exchangeable rendering backends - techniques, applications, experiences. Vis. Comput. **29**(10), 1039–1049 (2013)
13. Sébastien, L., Zanuttini, A.: Local image-based lighting with parallax-corrected cubemaps. In: ACM SIGGRAPH 2012 Talks, SIGGRAPH 2012, p. 36:1 (2012)
14. Stein, C., Limper, M., Kuijper, A.: Spatial data structures to accelerate the visibility determination for large model visualization on the web. In: Web3D14, pp. 53–61 (2014)
15. Uludag, Y.: Hi-Z screen-space cone-traced reflections. In: Engel, W. (ed.) GPU Pro 5, Chap. 4, pp. 149–192. CRC Press (2014)
16. Wientapper, F., Wuest, H., Kuijper, A.: Reconstruction and accurate alignment of feature maps for augmented reality. In: 3DIMPVT 2011: The First Joint 3DIM/3DPVT Conference, pp. 140–147. IEEE (2011)