STAG: Smart Tools and Apps in computer Graphics (2016)
Giovanni Pintore and Filippo Stanco (Editors)

Selective rasterized ray-traced reflections on the GPU

Mattias Frid Kastrati † Prashant Goswami ‡

Blekinge Institute of Technology, Sweden

Abstract

Ray-tracing achieves impressive effects such as realistic reflections on complex surfaces but is also more computationally expensive than classic rasterization. Rasterized ray-tracing methods can accelerate ray-tracing by taking advantage of the massive parallelization available in the rasterization pipeline on the GPU. In this paper, we propose a selective rasterized ray-tracing method that optimizes the rasterized ray-tracing by selectively allocating computational resources to reflective regions in the image. Our experiments suggest that the method can speed-up the computation by up to 4 times and also reduce the memory footprint by almost 66% without affecting the image quality. We demonstrate the effectiveness of our method using complex scenes and animations.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms

1. Introduction

Ray-tracing is a method that can be used to convert vectorized graphics into screen pixels [App68, Whi80]. This is done by tracing the path of light through pixels on the image plane. The technique is able to render highly detailed scenes and complex light interactions with high fidelity. Although it can simulate a wide array of illumination effects, such as transparency, detailed shadows and reflections, ray-tracing is computationally expensive. Even though ray-tracing can very easily be parallelized both on the GPU and CPU, the main challenge is with the time-consuming ray-object intersection tests, especially in highly complex scenes.

To circumvent the intersection cost, several acceleration structures have been proposed and implemented both on the CPU and GPU. The most well-known ones include the BSP-tree and kd-tree [CCI13, Hav00], octree and voxels [CG12, LR13, WIK*06]. Recently Hu et al. [HHZ*14] proposed a rasterized ray-tracing method that achieves ray-tracing via built-in rasterization on the GPU, thereby reducing the ray-scene intersection cost.

In this paper, the proposed rasterized ray-tracing method for the GPU by Hu et al. [HHZ*14] is improved and merged with simple deferred rendering to achieve selective ray-tracing. By taking advantage of the fact that [HHZ*14] is completely rasterized and works inside the GPU's rasterization pipeline, the rendering performance can be significantly improved by using a simple selective ray-tracing scheme. In our approach, while the entire image is rendered with computationally inexpensive deferred rendering, only the parts of the final image containing reflective details are ray-traced. The results from both renderings are then blended to create diffuse or clear reflections. The idea is to apply computational efforts on rasterized ray-tracing only on those parts of image where there are possibilities for great fidelity gains from ray-tracing. Our experiments demonstrate that the fidelity losses caused by selective ray-tracing are insignificant when compared to the performance gains when our method is compared to a full ray-tracing one. We also introduce an optimization to [HHZ*14] that makes the detailed scene representation occupy significantly less memory. As a proof of concept, we chose to implement mirror-like reflections in our selective method, however all global illumination effects that are supported in [HHZ*14] could also be implemented.

2. Related Work

[App68] pioneered ray-tracing and [Whi80] later extended it. Today ray-tracing is able to render detailed shadows, ambient occlusion, reflections, transparency with refractions and other illumination effects. Traditional research to improve performance of ray-tracing mainly focuses on improving the performance of the intersection tests between rays and scene geometry. Lately the strong growth of the processing power of the GPU has made it another interesting research area for improving the performance of ray-tracing [HHZ*14]. However early methods used the same acceleration structures as on the CPU and treated the GPU only as a

† fridh.mattias@gmail.com
‡ prashant.goswami@bth.se

DOI: 10.2312/stag.20161363

high-performance multiprocessor. This makes it nontrivial to integrate these techniques into the traditional rasterization pipeline.

[HHZ*14] propose a framework for ray-tracing that is meant to be easier to integrate into the traditional rasterization pipeline. Their method utilizes a two layer data structure to represent scene geometry. The first layer is a low resolution voxel structure, which is fast to build. The second layer has higher resolution and contains more information about the actual geometry stored as a linked-list A-buffer, based on the method proposed by [YHGT10]. Because of the layering less time is spent tracing rays in the highly detailed scene representation. Their scene representation is completely rasterized. Ray-tracing models such as ray casting, Whitted ray-tracing, ambient occlusion and path tracing can be implemented in the framework.

Some of the earliest examples of using voxels to improve the performance of ray-tracing are [FI85] and [FTI86]. They describe voxels as an extension of a classic raster grid, with each pixel becoming many voxels. A voxel is in other words the 3D representation of a pixel. Typically geometric data is stored in each of the voxels that contain geometry.

A-buffers were invented by [Car84] who suggested and implemented this new data structure for usage in the REYES engine (and later in Pixar's Photorealistic Renderman engine). The idea is that a fragment generated by a triangle may not cover an entire pixel and because of that it should be blended with other fragments behind it for a more realistic result, something which could not be done with the classic Z-buffer rendering [Cat74]. To solve this all fragments generated are saved in per pixel bins. A-buffers also make it possible to render order-independent transparency.

[YHGT10] introduced the first real implementation of linked-list A-buffers on the GPU. It differs from previous methods in that the A-buffers are generated in a single pass and that the per-pixel lists are of arbitrary sizes. This was doable because of recent advancements in GPU APIs and hardware features which allowed for atomic memory operations and arbitrary array sizes. The article also discusses and summarizes previous multi-pass and fixed-size methods.

[Cra10a] and [Cra10b] also propose two different versions of a single-pass A-buffer generation on the GPU. The first version [Cra10a] did not have support for dynamic depth, however, the second version [Cra10b] allowed this in a similar manner as the previously mentioned method by [YHGT10]. The main difference between [Cra10b] and [YHGT10] implementations is the paging system introduced by [Cra10b] to increase cache coherence. The paging also allows for decreased storage cost for links and the concurrent accesses to the shared memory pool also benefit from it.

[ZHS08] also use layered data structures to ray-trace reflections. Their method use precomputed layered depth textures. They store six different layered depth textures, two for each coordinate axis. Among those two, one is used for front-face triangles and one is used for back-face triangles. Their method is very similar to the relief mapping method presented by [POC05] and the search algorithms presented in both papers can be used to search in layered data structures (e.g. A-buffers).

Deferred rendering was first introduced by [DWS*88]. They introduced the key concept of only shading each pixel once depending on depth. It was later expanded by [ST90] who introduced the G-buffers. G-buffers are essentially geometrical data stored in multiple render targets for later use in the lighting pass. After the introduction of G-buffers modern deferred rendering began taking shape. In 2008 the usage of this method started to become mainstream in real-time rendering applications [SAGC*12].

The idea to merge ray-tracing and deferred rendering is not new. In fact it is possible to do it in Nvidia's OptiX engine. [PBD*10] go through the functionalities of OptiX in detail. [SAGC*12] use OptiX to implement a selective ray-tracing approach to almost double the performance compared to full ray-tracing in their scenarios. Deferred rendering is used to compute primary rays and then it is heuristically decided what parts of the picture that give the most fidelity gains to ray-trace. Three different ray-traced effects are supported: reflections, refractions and shadows. All of their results are based on static scenes.

[GAMR12] propose a ray-tracer using DirectCompute. Their method is not bounded by the same restrictions as a method implemented in the standard rasterization pipeline, and, it is as stated by [HHZ*14] hard to integrate into an engine that relies on said pipeline. However, they present some ideas that are useful in both cases. Particularly the way they split the computation into three steps, primary rays, intersection and color stages.

Inspired by [HHZ*14] and [SAGC*12], we present a selective rasterized ray-tracing method that integrates a rasterized ray-tracing method with a simple selective ray-tracing scheme. By rendering only reflective areas with ray-tracing computational resources can be spent where they most efficiently increase the fidelity of the final images. The reflections are then blended into an image rendered with a computationally cheaper rendering technique. Since [HHZ*14] is completely rasterized and integrated into the rasterization pipeline, it is well suited for selective rasterized ray-tracing. We utilize this fact and use deferred rendering to render the non-reflective areas.

3. Method

3.1. Rasterized Ray-Tracing

Our method is largely inspired by the rasterized ray-tracing method presented in Hu et al. [HHZ*14]. In their method they achieve rasterized ray-tracing inside the GPU-pipeline by representing the scene in rasterized data structures with two layers: a coarse voxel structure on top and sparse A-buffers in the second layer. These data structures are generated simultaneously at the beginning of every frame. The purpose is to avoid expensive tracing in the sparse A-buffers and instead save computations by making educated guesses by first tracing the coarse voxel structure. [HHZ*14] also introduce a ray-scene intersection method to go with this two-level scene representation.

3.1.1. Data Structures

For the coarse voxel map in [HHZ*14], the voxelization method suggested by [CG12] is used. The method was suggested before the recent additions of API supported conservative rasterization and

voxelization, which are available in both DirectX and OpenGL on the latest GPUs. As a result the proposed method still uses the edge shifting technique from [HAMO05].

The resolution of coarse voxels affects the performance and memory usage [HHZ*14]. A high resolution requires more memory and gives more sparsely populated areas which takes more time to trace. A low resolution on the other hand will give a less accurate guess in preparation of tracing the finer representation, giving more false positives and requiring more steps in the finer data structure. The actual trade-off depends on the distribution of the geometry in the scene and should be measured in order to determine a proper resolution of the voxel map.

For the A-buffers [HHZ*14] use the previously discussed method suggested by [YHGT10]. [HHZ*14] project the geometry along all three axes to produce a voxel-like result. The A-buffers remove the empty spaces between the geometry and therefore waste less memory and are faster to traverse than a voxel structure with the same resolution. However, for simplicity, an A-buffer structure with the resolution 1024^2 can be viewed as having the same precision as a 1024^3 resolution voxel structure.

3.1.2. Ray/Scene Intersection

[HHZ*14] use the geometry step from deferred rendering to skip the primary rays in the ray-tracing. This effectively removes a whole iteration of the intersection tests and with the exact same result as the primary rays would have in a pure ray-tracer. The results in the G-buffers can then be used to bounce the rays into the voxels and A-buffers to render ray-traced global illumination effects. To gain the advantage of the two-level data structure, the coarse data structure is searched first. Only when a voxel is marked as containing geometry will the more fine-grained data structure be accessed [HHZ*14].

The search method used to detect voxels containing geometry is based on the method presented in [AW87], a 3D-DDA (3D Digital Differential Analyzer) method. An intersection is considered a hit if the voxel that is intersected by the ray contains a normal. This normal is sent to the A-buffer search and if that search step returns a miss, the algorithm moves to the next voxel. The A-buffer search is performed with a 2D-DDA linear search inspired by [ZHS08]. It is initiated to search for actual geometry in the A-buffers. The number of search steps is kept low, two or three, and it is assumed that only one surface is contained in each of the voxels. To further pin-point any detected geometry hits in the A-buffers, a binary 2D-DDA search based on the method presented by [POC05] is also performed.

3.2. Selective Rasterized Ray-Tracing

The approach presented in this paper merges deferred rendering together with selective rasterized ray-tracing for dynamic scenes. An overview of the framework is shown in Figure 1. The geometry in the first row in the image is sent to three geometry shaders. The results from these three shaders can be seen in the second row, the geometry stage in the framework. The left branch in the figure shows the result from a shader that is the same as the geometry step in regular deferred rendering. The right branch shows the result from

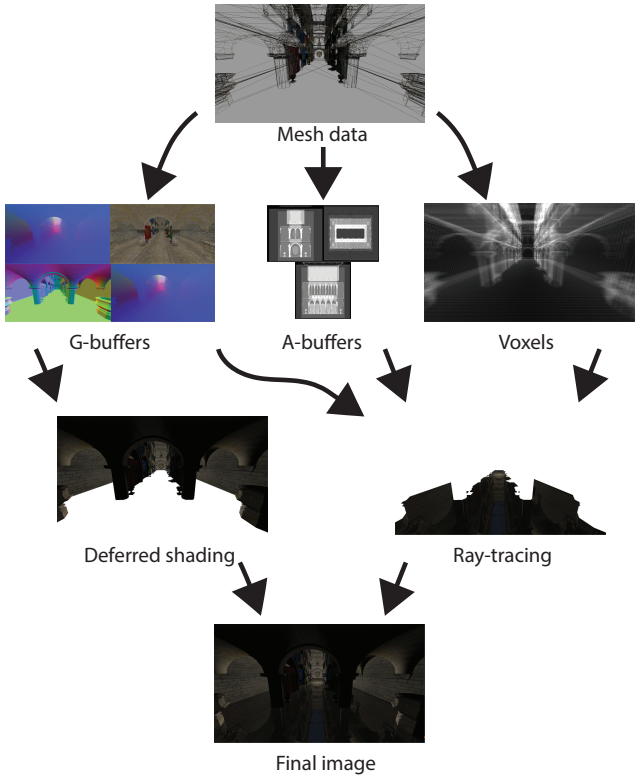


Figure 1: An overview of the proposed pipeline. After the data structures in layer two have been generated, all of the geometry is 100% rasterized and represented as texels of various kinds.

a shader that performs a low resolution voxelization and the branch in the middle results from a shader that generates three high resolution A-buffers. With the results from this geometry stage, time effective ray-tracing can be performed and used during the lighting step on the third row together with deferred rendering. The final result can be viewed on the fourth row.

To the previously discussed method [HHZ*14], we add the extra step of deferred shading on row three in Figure 1. [HHZ*14] also use parts of deferred rendering but only to perform the geometry step for the primary rays, the leftmost branch on the second row in Figure 1. We also introduce an optimization of the high resolution A-buffers that necessitate the splitting of the voxel and A-buffer generation, the rightmost and middle branches on the second row in Figure 1. In the following sections we discuss these additions and other smaller differences in detail.

3.2.1. Data Structures

Similar to [HHZ*14] the moving average method presented by [CG12] is used to store a state in the voxels. However, the spin-locks discussed in [VF14] are used instead of the method with `imageAtomicCompSwap` used by [CG12]. This was done to improve precision and avoid flickering. The normals are later used to decide in what A-buffer to store the geometry in the next step and also what A-buffer to look for geometry in during the rendering. However, this moving average method is not perfect. As pointed

out by [CG12], all triangles are weighted the same. If a voxel contains a large triangle that is dominant in the z-axis and two very small triangles that are x-axis dominant, the voxel will be marked as dominant in the x-axis. This might later lead to that the ray/scene intersection misses the larger, perhaps more important triangle entirely since it is only visible as a line from the x-axis. Most of the errors that result from this flaw in the moving average method are mitigated by [HHZ*14] since they generate the A-buffer and voxel structures in the same pass. This can be seen as a way of anti-aliasing the voxel structure, as it truly is a lower resolution version of the A-buffers. Since both data structures are necessary for the rasterized ray-tracing, it is also a computationally cheap way to implement coverage based normal averaging in the voxels, since the contribution of smaller triangles will be coming from less A-buffer entries than that of bigger triangles. However since the method proposed here splits the generation of the scene representation into two passes, it does not benefit from this.

The method proposed by [Cra10b] is used for the A-buffer generation. The method is inspired by [YHGT10], with the addition of *paging*. To avoid congesting the global head pointer and to increase memory coalescing, a number of A-buffer fragments are allocated at a time. This global head pointer is guarded with an atomic add, while the local head pointers are guarded with the same spin-lock mechanism used for the voxels [VF14]. In the current implementation four entries are allocated each time an A-buffer bin requests more space. The global memory pool can also be grown. If necessary this is done between rendering loops, based on the number of excess fragments in the previous frame.

Like [HHZ*14] the geometry is projected in all three axes to produce a voxel-like end result. However, unlike [HHZ*14], the geometry is only stored in the A-buffer which it is decided to be dominant in based on the averaged normal stored in the coarse voxel structure. In theory this cuts off about two-thirds of the memory usage compared to their method. In [HHZ*14], in the intersection stage, only the A-buffers that are decided to be dominant through looking at the normal stored in the voxels are ever accessed. This does not interfere with the accuracy of the algorithm since the other A-buffer fragments were unused. Generating all the structures at the same time effectively removes the option to cull unused fragments from the A-buffers. It cannot be known during the voxel generation which axis the voxel will be dominant in when finished. However, as previously discussed it also removes some problems with conservative voxelization and coverage based normal averaging.

Two texels are used to represent the values of each A-buffer entry. As in [Cra10b] they are stored in texture buffer memory, which is very limited spacewise on most modern graphics cards. As a result, they are more optimized than the G-buffers shown in Figure 3. The layout of the A-buffers is shown in Figure 2. As in [Cra10b] the diffuse color and the z-depth in local A-buffer space is saved in the first A-buffer. The other two coordinates are indirectly saved as the texture coordinates for the fragment that is written. In the second A-buffer, a reflectivity constant is stored in the first fragment channel and after that the normal's y- and z-values followed by a triangle ID which is used to avoid self-reflection in the ray-tracing. The normal's x-value is stored by exploiting the fact that the sign bit in the first channel of the second A-buffer is unused, since the

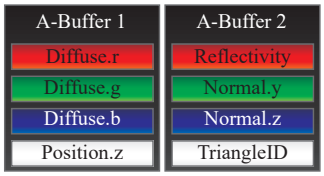


Figure 2: The layout of the A-buffers used in our method.

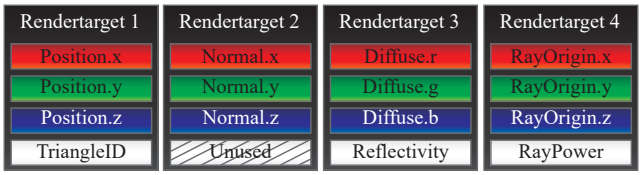


Figure 3: The layout of the G-buffers used in our method.

reflectivity constant itself is unsigned. In the end the x-value is calculated with Equation 1.

$$normal_x = sgn(reflectivity) * \sqrt{1 - normal_y{}^2 - normal_z{}^2} \quad (1)$$

3.2.2. Intersection

The presented method uses the geometry step from deferred rendering to avoid ray-tracing the initial rays. For the deferred rendering used in the presented method, the minimal method presented by [Wol13] was used, with some additions to the geometry step. To keep the integrity of the lighting intact, various information about the surface closest to the eye is stored in G-buffers [SAGC*12]. For both the ray-tracing and the deferred rendering presented to work, diffuse color, surface reflectivity, surface normal, surface position and eye position are saved in G-buffers as shown in Figure 3. The eye position is later used to represent the old surface position in the recursive ray-tracing. The data generated from this step is used both as the primary rays in the ray-tracing for reflective pixels and in the deferred lighting step. The intersection stage of the ray-tracing is performed in the same way as in [HHZ*14].

3.2.3. Lighting

In the lighting stage, all the pixels are split into two categories: pixels that should be ray-traced and pixels that should only be shaded with standard deferred rendering. All the pixels with reflectivity values greater than zero in the G-buffers are selected for ray-tracing. To enable diffuse reflections, the pixels that are ray-traced are also deferred shaded and the result is blended. The deferred shading is done as described by [Wol13]. To further increase realism of the non-ray-traced pixels, standard methods such as shadow maps could be applied to simulate further global illumination effects.

If a pixel is marked as reflective by the initial deferred geometry step, it is ray-traced using the intersection scheme presented by [HHZ*14]. Inspired by [GAMR12], the initial intersection stage and the lighting stage are separated. This is to improve performance when tracing shadows, as the code flow gets more uniform

in length. However, currently, no shadows are traced in the reflections as this would create a realism gap between the reflections and the purely deferred rendered pixels. Instead an illumination model that is conceptually close to the illumination described in [Wol13] is used. The result is alpha blended with the result from the deferred shading to create diffuse reflections.

4. Results

4.1. Experiment Setup

We implemented our method in C++ and OpenGL 4.3 which provides API methods that enable atomic operations in rasterization shaders on the GPU. All the tests in the engine were done on a PC equipped with an Intel Xeon E5-1650 3.20GHz and an Nvidia Quadro K4000 GPU. In the experiments three scenes were used: the sphere and the teapot scenes in Figure 4 and the Crytek Sponza scene in Figure 8. For all the rendering results with the teapot and sphere models, a voxel resolution of 64^3 and an A-buffer resolution of 1024^2 was used. For the Sponza scene the same A-buffer resolution was used, but because of the complexity of the scene geometry, 256^3 voxel resolution was used to maintain the assumption that only one surface is contained in each voxel true. The size of the screen buffer in all experiments was kept at 1920×1080.

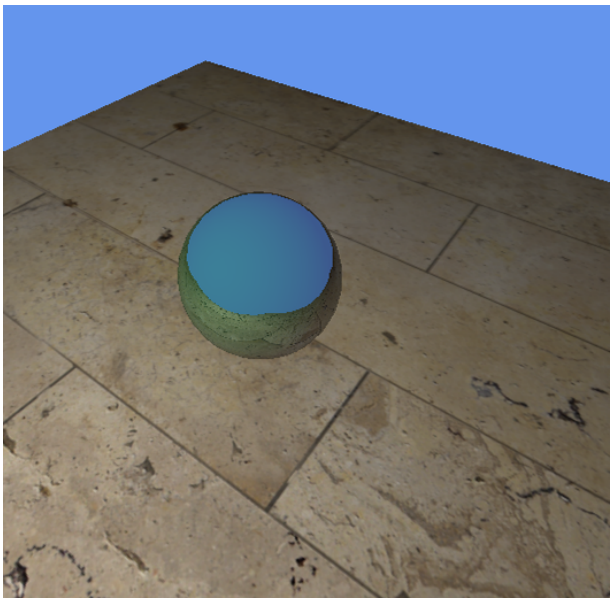
4.2. Memory Savings in the A-buffers

As previously mentioned, the proposed method uses a paging scheme in the A-buffers to avoid congesting the global head-pointer and to increase memory coalescing. The paging of the A-buffers allows four entries to be allocated to each A-buffer bin each time it requests more space, which causes some over-allocation. Both the waste caused by this and the total buffer size can be viewed in Figure 5. For the smaller scenes the waste is larger than the actual memory used to store information. This is because the A-buffer pages contain less entries than the four that are allowed in each page, i.e. very few or no parts of the scene have four in triangle depth when viewed from the coordinate system axes.

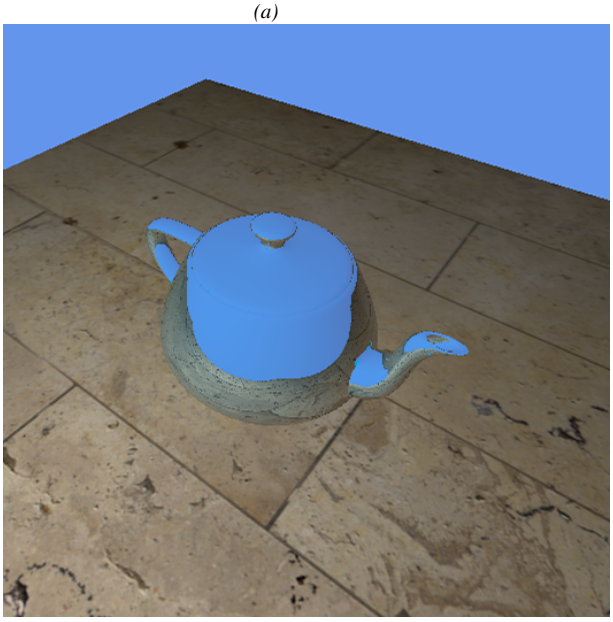
To demonstrate the memory savings that result from splitting the voxel A-buffer generations, two scenarios have been tracked. In the first the method described by [HHZ*14] was used and in the second the method proposed in this paper was used. The amount of memory used in each case and the ratio of saved memory with our method is given in Table 1. As can be seen, our method uses only about a third of the amount of A-buffer memory as compared to the original method. If perfect A-buffer growth and no paging were used, it would have been a perfect third, which was the expected result from doing the optimization. The paging causes a small amount of over-allocation, which can be viewed in Figure 5.

Scene	Hu et al. (MB)	Ours (MB)	Gain
Sponza	1079.55	380.74	64.73%
Teapot	210.18	71.85	65.82%
Sphere	182.37	65.46	64.11%

Table 1: Comparison between per A-buffer memory usage in Hu et al. vs. our method.



(a)



(b)

Figure 4: Hybrid renderings of the (a) sphere, and (b) teapot scenes. They were rendered in 1920×1080 but are cropped to highlight the reflective areas.

4.3. Performance

4.3.1. Execution Time

The results from the three scenarios no reflections, full ray-tracing and our selective ray-tracing for the sphere, teapot and Sponza scenes can be viewed in Table 2. During the experiments the camera was not moving and the A-buffers had been pre-grown to an

Scene	Scenario	Constuction(ms)	Deferred geometry step (ms)	Ray-tracing (ms)	Others(ms)	Total(ms)
Sphere	No refl.	1.26	0.56	0	0.22	2.04
	Full RT	13.15	0.58	100.14	0.47	114.35
Figure 4(a)	Selective	13.17	0.58	13.09	0.42	27.27
Teapot	No refl.	1.37	0.55	0	0.22	2.16
	Full RT	25.69	0.6	108.41	0.49	135.18
Figure 4(b)	Selective	25.71	0.6	27.48	0.44	54.22
Sponza	No refl.	7.02	1.09	0	0.6	8.7
	Full RT	483.4	1.07	2129.61	3.65	2617.73
Figure 8	Selective	482.87	1.07	383.49	3.72	871.14

Table 2: Comparison of execution times for all scenes for the three scenarios no reflections, full ray-tracing and selective ray-tracing. The construction phase contains G-buffer, voxel and A-buffer generation. In the no reflections scenario, no voxels or A-buffers where generated. The angles for the renderings can be seen in the figures mentioned before the selective ray-tracing results.

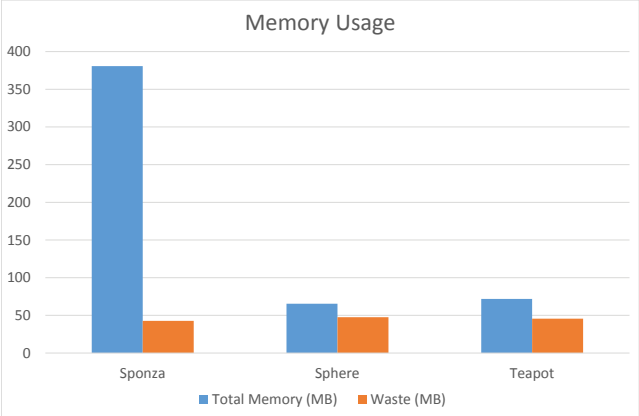


Figure 5: Memory usage of A-buffers per axis for the scenes used in the experiments. Note how much memory is wasted on the less complicated scenes compared to the total memory used.

efficient size to avoid biasing the results with some initial slower frames.

From Table 3 shows that the selective method is between 2.49 and 4.19 times faster than the fully ray-traced method for the tested scenes. The fully ray-traced results should be seen as an equivalent to if they had been generated using the method proposed by [HHZ*14]. The no reflections scenario examined is between 13.37 and a 100.13 times faster than the implemented selective ray-tracing method, as no computation is used to capture ray-tracing effects or to build the scene representation data structures. All the numbers presented here are measured with only a single ray bounce, none in the deferred rendering case.

The results in Table 3 indicate that the method spends about the same time constructing the scene representations as it spends ray-tracing them for the selected angles. To be able to reach the real-time frame rates for large scenes, both phases have to decrease their computation times considerably. Also it is evident that a lot of performance is gained in the ray-tracing stage (as expected). For the selected angles the Sponza scene, which is the most complex scene in the experiments, has the second largest performance gain in both the ray-tracing stage and in the total summary. However more angles would have to be studied before any conclusion can be drawn regarding this.

Scene	Selective/Full RT	No reflections/Selective
Sphere	4.19x	13.37x
Teapot	2.49x	25.1x
Sponza	3.0x	100x

Table 3: Speed-ups between the different scenarios examined. The execution time results can be found in Table 2.

4.3.2. Screen Coverage

The amount of pixels that are ray-traced affect the performance gain obtained by using selective ray-tracing. If an excessive amount of pixels are ray-traced it affects the performance negatively and vice versa. Because of that experiments that study the execution time as a function of screen coverage have been conducted. In this section only execution times from the ray-tracing stage are presented, as the data structure generation etc. remain at a constant execution time over the different scenarios, this can be seen in Table 2.

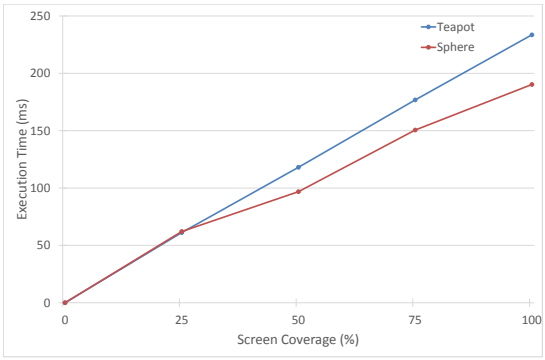


Figure 6: Screen coverage experiment results for the sphere and teapot scenes. Only the execution time for the ray-tracing stage are included.

As shown in Figure 6, the execution time for the simpler scenes grow linearly with screen coverage percentage. The Sponza scene

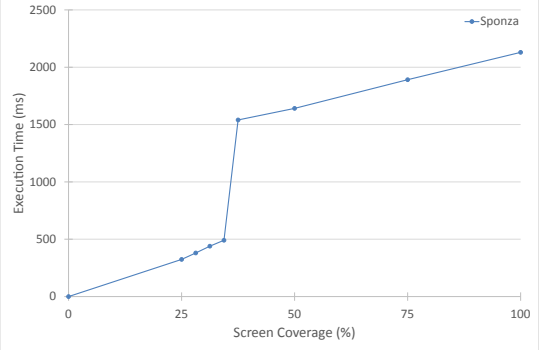


Figure 7: Screen coverage experiment results for the Sponza scene. Only the execution time for the ray-tracing stage are included. The jump in the between 34.375% and 37.5% is believed to be caused by cache thrashing or that pixels with very expensive reflection paths entered the image.

behaves slightly different, as viewed in Figure 7. A big jump happens in the execution time between the screen coverage of 34.375% and 37.5%. This is believed to be caused by cache thrashing or that pixels with very expensive reflection paths entered the image. If the jump is disregarded the expected linear relationship can also be seen for the Sponza scene.

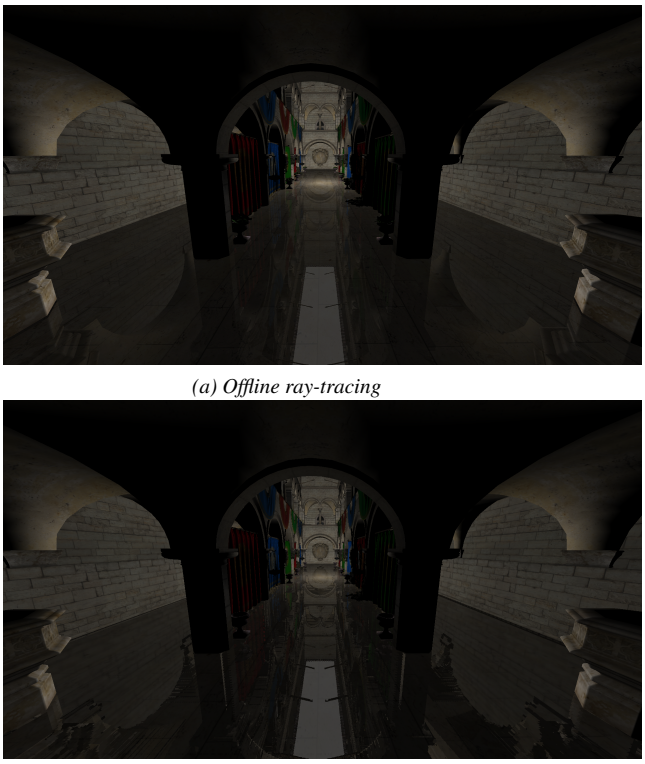
4.4. Image Quality

To compare the image quality of the methods, the Sponza renderings have been chosen as it is the most complex scene used in the experiments. As a reference point, the ground truth image has been ray-traced offline in Autodesk Maya (Figure 8(a)). The method proposed by [HHZ*14] is believed to be able to generate images of the same visual quality as in Figure 8(a). In the experiment the offline rendered image was compared with the proposed selective rasterized ray-tracing method (Figure 8(b)). To the human eye, the images look very similar with only some minor visible differences. A tool called *PerceptualDiff* [YMT] based on [YPG01] is used to measure the perceptual difference. According to the tool only 0.55% pixels differ in a way that is perceivable by the human eye.

From the image comparison experiments it is apparent that the selective ray-tracing method can compete with the original method [HHZ*14] in terms of image quality, add to this the fact that our proposed method is about three times faster than the original method for the studied scene and angle.

5. Conclusions and Future Work

In this paper we showed that the rasterized ray-tracing method proposed by [HHZ*14] can be improved by integrating it into a deferred rendering pipeline and performing selective ray-tracing for dynamic scenes at interactive frame rates. Real-time frame rates could be more easily maintained for less complex scenes, while for the studied complex scene too much time was spent building and traversing the scene representations. The proposed selective method showed promise in terms of performance gain when compared to full ray-tracing if the screen coverage ratio was kept at a



(a) Offline ray-tracing

(b) Selective rasterized ray-tracing

Figure 8: Comparison: (a) is the ground truth rendering of the Crytek Sponza rendered with Autodesk Maya's software renderer, this is thought to be of equal quality as the original rasterized ray-tracing method [HHZ*14], (b) was rendered with the proposed selective rasterized ray-tracing method method.

reasonable level. It has also been shown that there is a linear relationship between the performance gained from the selective method and screen coverage ratio. The proposed method also reduced the memory footprint by at least 60% for each scene without affecting the image quality. The selective method is able to compete with offline ray-tracers with similar settings in terms of visual quality.

In the future it would be interesting to implement a smart selection scheme for the ray-tracing that would allow renderings of more complex illumination effects with similar performance gains as the current method. Another interesting path of future research would be selective refreshing of the data structures. As it is now, a lot of time is spent computing voxels and A-buffer fragments that are identical in every frame. Since the results point in the direction that the data structure construction is about as expensive as the shading in the current method, this could possibly cut the rendering times with almost 50% and thus getting the method even closer to real-time frame rates for complex scenes.

6. Acknowledgements

We thank the anonymous reviewers for their feedback. We also thank Dr. Nicolas Holzschuch from INRIA, Grenoble for his constructive suggestions.

References

[App68] APPEL A.: Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference* (New York, NY, USA, 1968), AFIPS '68 (Spring), ACM, pp. 37–45. 1

[AW87] AMANATIDES J., WOO A.: A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proceedings of EuroGraphics '87* (Amsterdam, The Netherlands, 1987), Elsevier Science, pp. 3–10. 3

[Car84] CARPENTER L.: The A-buffer, an Antialiased Hidden Surface Method. *SIGGRAPH Computer Graphics 18*, 3 (Jan. 1984), 103–108. 2

[Cat74] CATMULL E. E.: *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Dissertation thesis, University of Utah, Salt Lake City, UT, USA, Dec. 1974. 2

[CCI13] CHOI B., CHANG B., IHM I.: Improving Memory Space Efficiency of Kd tree for Real time Ray Tracing. *Computer Graphics Forum 32*, 7 (2013), 335–344. 1

[CG12] CRASSIN C., GREEN S.: Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer. In *OpenGL Insights*, Cozzi P., Riccio C., (Eds.). CRC Press, Boca Raton, London, New York, 2012. 1, 2, 3, 4

[Cra10a] CRASSIN C.: Fast and Accurate Single-Pass A-Buffer using OpenGL 4.0+. http://blog.icare3d.org/2010/06/fast-and-accurate-single-pass-buffer.html, 2010. Last accessed on 7 September 2016. 2

[Cra10b] CRASSIN C.: OpenGL 4.0+ ABuffer V2.0: Linked lists of fragment pages. http://blog.icare3d.org/2010/07/opengl-40-abuffer-v20-linked-lists-of.html, 2010. Last accessed on 7 September 2016. 2, 4

[DWS*88] DEERING M., WINNER S., SCHEDIWY B., DUFFY C., HUNT N.: The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics. *SIGGRAPH Computer Graphics 22*, 4 (June 1988), 21–30. 2

[FI85] FUJIMOTO A., IWATA K.: Accelerated Ray Tracing. In *Computer Graphics*. Springer Japan, 1985, pp. 41–65. 2

[FTI86] FUJIMOTO A., TANAKA T., IWATA K.: ARTS: Accelerated Ray-Tracing System. *Computer Graphics and Applications, IEEE 6*, 4 (April 1986), 16–26. 2

[GAMR12] GARCÍA A., ÁVILA F., MURGUÍA S., REYES L.: Interactive Ray Tracing Using the Compute Shader in DirectX 11. In *GPU Pro 3*. CRC Press, Boca Raton, FL, USA, 2012, pp. 353–376. 2, 4

[HAMO05] HASSELGREN J., AKENINE-MÖLLER T., OHLSSON L.: Conservative Rasterization. In *GPU Gems 2*. Addison Wesley, Reading, MA, USA, 2005. 3

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. Dissertation thesis, Czech Technical University, Prague, Czech Republic, Nov. 2000. 1

[HHZ*14] HU W., HUANG Y., ZHANG F., YUAN G., LI W.: Ray tracing via GPU rasterization. *The Visual Computer 30*, 6 (2014), 697–706. 1, 2, 3, 4, 5, 6, 7

[LR13] LIU X., ROKNE J. G.: A Micro 64-tree Structure for Accelerating Ray Tracing on a GPU. In *Proceedings of Graphics Interface 2013* (Toronto, Ont., Canada, 2013), GI '13, Canadian Information Processing Society, pp. 165–172. 1

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics 29*, 4 (July 2010), 66:1–66:13. 2

[POC05] POLICARPO F., OLIVEIRA M. M., COMBA J. A. L. D.: Real-time Relief Mapping on Arbitrary Polygonal Surfaces. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2005), I3D '05, ACM, pp. 155–162. 2, 3

[SAGC*12] SABINO T. L., ANDRADE P., GONZALES CLUA E. W., MONTENEGRO A., PAGLIOSA P.: A Hybrid GPU Rasterized and Ray Traced Rendering Pipeline for Real Time Rendering of Per Pixel Effects. In *Proceedings of the 11th International Conference on Entertainment Computing* (Berlin, Heidelberg, Germany, 2012), ICEC'12, Springer-Verlag, pp. 292–305. 2, 4

[ST90] SAITO T., TAKAHASHI T.: Comprehensible Rendering of 3-D Shapes. *SIGGRAPH Computer Graphics 24*, 4 (Sept. 1990), 197–206. 2

[VF14] VASILAKIS A. A., FUDOS I.: K+-buffer: Fragment Synchronized K-buffer. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2014), I3D '14, ACM, pp. 143–150. 3, 4

[Whi80] WHITTED T.: An Improved Illumination Model for Shaded Display. *Communications of the ACM 23*, 6 (June 1980), 343–349. 1

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray Tracing Animated Scenes Using Coherent Grid Traversal. *ACM Transactions on Graphics 25*, 3 (July 2006), 485–493. 1

[Wol13] WOLFF D.: *OpenGL 4.0 Shading Language Cookbook*, 2 ed. Packt Publishing, Birmingham, UK, 2013, pp. 196–202. 4, 5

[YHGT10] YANG J. C., HENSLEY J., GRÜN H., THIBIEROZ N.: Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum 29*, 4 (2010), 1297–1304. 2, 3, 4

[YMT] YEE H. Y., MYINT S., TERRACE J.: perceptualdiff. https://github.com/myint/perceptualdiff. Last accessed on 7 September 2016. 7

[YPG01] YEE H., PATTANAIK S., GREENBERG D. P.: Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments. *ACM Transactions on Graphics 20*, 1 (Jan. 2001), 39–65. 7

[ZHS08] ZHANG C., HSIEH H.-H., SHEN H.-W.: Real-time Reflections on Curved Objects Using Layered Depth Textures. In *Proceedings of IADIS International Conference on Computer Graphics and Visualization* (2008), IADIS. 2, 3