Interactive Ray Tracing Using Hardware Accelerated Image-Space Methods

Philippe C.D. Robert
robert@iam.unibe.ch

Severin Schoepke
severin.schoepke@students.unibe.ch

Research Group on Computational Geometry and Graphics
Institute of Computer Science and Applied Mathematics
University of Bern, Neubrückstrasse 10, 3012 Bern, Switzerland

Abstract

In recent years, interactive ray tracing has become realisable, albeit mainly using clustered workstations and sophisticated acceleration structures. On non-clustered computer architectures this is still not an easy task to achieve, especially when rendering animated scenes, even though the computation power of modern workstations is increasing rapidly.

In this paper we propose commonly known image-space rendering techniques to be used in the context of ray tracing. We describe a visibility preprocessing algorithm to perform interactive ray tracing based on the standard depth testing capability of graphics processing units. This method – item buffer rendering – is particularly suitable for rendering animated scenes, as it completely avoids the necessity of creating and updating any kind of spatial acceleration structure in order to achieve high frame rates. The item buffer stores indices referencing those primitives which are visible in screen-space. Primary ray intersection testing can therefore be executed very efficiently by performing only one primitive lookup operation and one intersection test. As a consequence, this approach reduces the total number of primary ray intersection tests to a minimum. Additionally we integrate shadow rendering into our ray tracer using the shadow mapping technique to avoid computationally expensive shadow rays. We compare CPU and GPU based implementations of our ray tracer and analyse the advantages and disadvantages of both approaches in terms of visual quality and rendering performance.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Picture-Image Generation - Display Algorithms; I.3.7 [Computer Graphics]: 3D Graphics And Realism - Ray Tracing

Keywords: Programmable Graphics Hardware, GPGPU, Intersection Testing

1 Introduction

Image-space rendering techniques have been used to accelerate ray tracing since the late 1980ies. Unfortunately at that time graphics hardware was not yet as powerful as today, so there was no advantage in using hardware accelerated methods to speedup ray tracing. In the last few years this has changed substantially, though.

In this paper we utilise an algorithm to perform interactive ray tracing which avoids the necessity of creating and updating any kind of spatial acceleration structure in order to achieve high frame rates. Instead we use screen-space visibility processing to accelerate primary ray intersection testing. Therefore, this approach is particularly suitable for ray tracing animated scenes. Both our implementations are largely based on Weghorst, Hopper and Greenberg's *item buffer* algorithm [30]. As opposed to their work we are using graphics hardware accelerated depth testing capabilities of up-to-date graphics processing units (GPU) to perform interactive ray tracing. Based on this method we describe ways to further speedup rendering using interleaved sampling and colour interpolation as well as to preserve image quality even under difficult conditions.

At the same time we integrate shadow rendering into our ray tracer using the shadow mapping technique, which was introduced by Lance Williams in 1978 [32]. This allows us to render shadows without using computationally expensive shadow rays.

Finally, as it has become feasible to perform general-purpose computations on programmable graphics hardware [19], we describe an implementation of our ray tracer which is done entirely on the GPU using the OpenGL Shading Language [11].

The remainder of this paper is organised as follows. In Section 2 we outline previous work in the fields of ray tracing, image-space rendering techniques and general purpose computation using graphics hardware (GPGPU) in the context of ray tracing. Section 3 discusses the investigated image-space rendering techniques, the item buffer and shadow mapping algorithms. In Section 4 we outline our implementation and in Section 5 we present the results and discuss its advantages and disadvantages of our implementation. Finally, we state our conclusions in Section 6 and outline future work in Section 7.

2 Related Work

The idea of using screen-space coherence to accelerate ray tracing came up in the late 1980ies.

Weghorst *et al.* [30] introduced a method based on *item buffers* to reduce the total number of intersection tests for primary rays. Item buffers simply store the indices which reference those objects within the scene which are visible at the given location on the image plane. As a consequence, a trivial lookup operation can be used to determine the visible object for a specific pixel, hence, no acceleration structure is needed to produce high frame rates. This is especially fortunate when rendering animated scenes ([22, 29, 28]). Salesin and Stolfi [24] introduced the *zz-buffer* algorithm which is based on screen-space object indexing similar to the item buffer algorithm. The zz-buffer is used to detect the objects intersected by primary rays and by rays to the light sources. It is also used to identify the pixels which do not contain small features and as such can be sampled by a single ray only. Lamparter *et al.* [13] proposed a related method called *ray-z-buffer*, which is based on breadth-first ray tracing. The ray-z-buffer is a generalisation of the depth buffer, based on nested, adapted quadtrees. It is used to render arbitrarily large scenes. Unlike the item buffer the ray-z-buffer stores tuples describing the set of rays and the visible primitives. Kim *et al.* [12] introduced the *zf-buffer*, another variant of the item buffer algorithm. Instead of storing object indices the zf-buffer records pointers to those objects which are visible as determined by the z-buffer algorithm. Kim *et al.* also applied this method to render reflective objects.

To render shadows we use the well-known shadow mapping algorithm which was

first described by Lance Williams [32]. An alternative solution to accelerate shadow rendering is Haines and Greenberg's *light buffers* introduced in 1986 [9]. Instead of using one buffer per light source they use six buffers to spatially subdivide the object space. Shadowed regions can then be identified by shooting shadow feelers. This method is slower but better adapted to a wide range of lighting scenarios.

With the advent of programmable graphics hardware it has become feasible to offload arbitrary computational tasks to the GPU using a stream processing model [1, 27, 2, 19, 14]. Unfortunatly this is not always as trivial as one might expect, technically and performance-wise [7]. Regardlessly, in 2002 Carr *et al.* [3] were able to implement a fixed-point ray-triangle intersection testing engine on an ATI R200. Around the same time Purcell *et al.* [20, 21] developed a complete ray tracing pipeline on a GPU simulator. Since then others have implemented classical ray tracers on the GPU using the extended feature set of current generation hardware [5, 10]. In 2004 Weiskopf *et al.* [31] implemented a nonlinear ray tracer on the GPU using several acceleration techniques, such as early ray termination and adaptive ray integration. In 2005 Simonsen and Thrane [25] compared various ray tracing acceleration structures on the GPU. Foley and Sugerman [8] used the kd-tree acceleration structure to perform ray tracing on the GPU, whereas Carr *et al.* [4] introduced a method for quick intersection of dynamic triangular meshes on the GPU, based on a threaded bounding volume hierarchy built from a geometry image.

3 Rendering Algorithms

In the following sections we will outline the item buffer and shadow mapping methods and provide an insight into important implementation aspects of our interactive ray tracer. For the sake of simplicity and performance we thereby concentrate on rendering triangles only.

3.1 Visibility Processing

One of the biggest challenges when performing interactive ray tracing is how to reduce the total number of ray-object intersection tests to a minimum [23]. Usually this is done by segmenting the 3D space into spatial subsets containing a certain number of objects using some sort of acceleration structure – e.g., kd-trees, grids or bounding volumes. Fast traversal routines can then be used to decide which subsets of the scene to pick in order to compute the proper ray-object intersection. Unfortunately, creating and updating acceleration structures is memory and time consuming, especially when rendering dynamic scenes. Instead of using a sophisticated acceleration structure we therefore use an *item buffer* to determine which objects need to be tested for intersection.

The item buffer is a data structure which stores for every pixel on screen a reference to the triangle which is visible at that position – this is visualised in Figure 2. Primary ray intersection testing can thus be executed very efficiently by performing only one triangle lookup operation and one ray-object intersection test. The final rendering performance thus heavily depends on an efficient and reliable visibility processing. Commonly this is done using the z-buffer hidden-surface algorithm [26]. Unlike Weghorst *et al.* we use the hardware accelerated OpenGL depth testing for this purpose, which promises high rendering performance.

To build the item buffer we perform the following steps: every triangle of the scene is rendered to a framebuffer using standard OpenGL; in this process we encode the

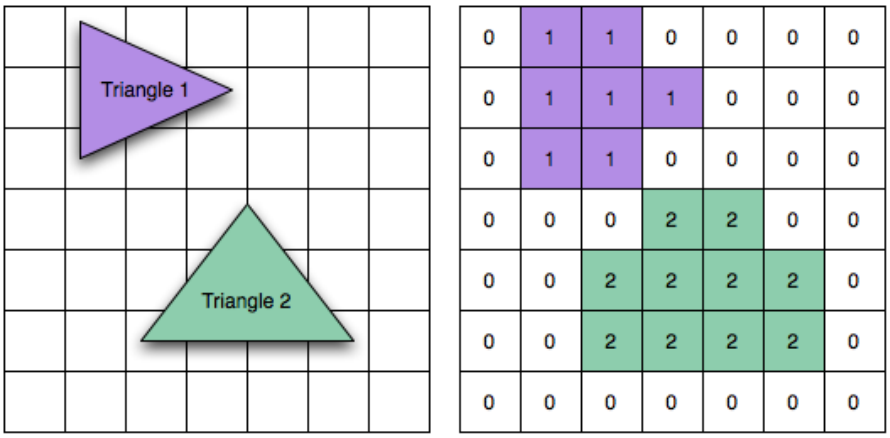

Figure 2: The item buffer data structure (right) contains the indices of the visible triangles in the framebuffer (left).

triangle indices into RGB colour values. Using this approach we are able to address up to 2^{24} triangles. While rendering the triangles we enable OpenGL's depth testing to make sure that only the visible triangles are stored in the framebuffer. The item buffer is then created by reading back the framebuffer data to host memory and decoding the RGB colour values into triangle indices. If more than 2^{24} triangles have to be visualised, RGBA values in conjunction with the appropriate OpenGL blending mode can be applied to address up to 2^{32} triangles. Alternatively, bounding objects may be rendered instead of the individual triangles. This approach is also helpful to keep the total number of triangles at a minimum when creating the item buffer, such as when visualising large scenes. Of course this will then result in a higher number of ray-triangle intersection tests per primary ray to compute the correct intersection.

Since item buffer rendering is a screen-space based method it is possible to scale the item buffer creation using distributed sort-first parallel rendering [18] devoting multiple GPUs. Using the GPU's depth buffer to create the item buffer exhibits one serious drawback, though. On current generation hardware it is limited to max. 24 bit precision. Because the depth buffer is on a log scale, with more precision essentially being allocated for things that are "closer" to the viewport, it is crucial to use an optimal projection setup. Nevertheless, in some situations this limitation causes rendering artefacts. This happens whenever the item buffer contains references to "wrong" triangles due to incorrect depth test results. This case is depicted on the right of Figure 3. A similar problem may appear at the borders of neighboured triangles where the index of a wrong triangle may be written to the item buffer due to the limited resolution of the framebuffer. This is depicted on the left of Figure 3. Even worse, both problems can appear altogether. In our implementation we try to overcome these shortcomings by using *item buffer multisampling* in order to compute the correct intersections for all primary rays. The results are visually appealing by all means; a typical showcase is given in Figure 4.

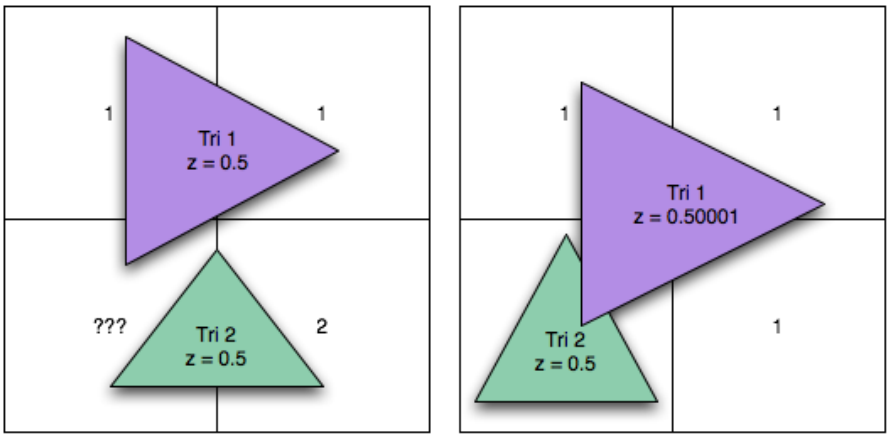

Figure 3: Two triangles with the same z-value $z = 0.5$ (left) and two overlapping triangles with almost the same z-value where the depth test fails (right).

3.2 Shadow Mapping

Because it is not possible to shoot shadow feelers efficiently using only the item buffer method, we perform shadow rendering using the image-based shadow mapping technique [32]. Although the visual quality is not as good as with real shadow feelers it is a good rendering technique for dynamic applications. Shadow mapping is based on the depth buffer hidden-surface algorithm as well. The most important aspect of this technique is that the depth buffer generated by rendering the scene from the light's point of view is the same as a visibility test over the light source's viewing volume. It can thus be used directly as a shadow map which partitions the view in both lit and shadowed regions. The algorithm itself is as simple as follows. First, the scene is rendered from the light's point of view. Consequently, the z-values for the objects closest to the light source are stored in the depth buffer: the shadow map. Then, the scene is rendered from the camera's point of view and as each fragment is generated it is transformed into the light sources coordinate system and tested for visibility. If the distance to the fragment is greater than the value stored in the shadow map, there is some object in front of it and thus it lies in the shadow, otherwise the fragment is not shadowed.

The proper size of the shadow map has to be evaluated using the the camera frustum and the type of the light source in order to make sure that no aliasing and quantisation issues occur. And because shadow mapping is an image-based technique it is fully accelerated on the GPU through its texture mapping functionality. Furthermore the shadow map only needs to be updated when the scene is subject to changes during the rendering – e.g., moving light sources or altering geometry.

4 Implementation

Our implementation is kept very basic; e.g., we do not use any SIMD functionality of modern CPUs or parallel rendering strategies. As a consequence it is certainly possible to achieve even better performance results when using more advanced implementations.

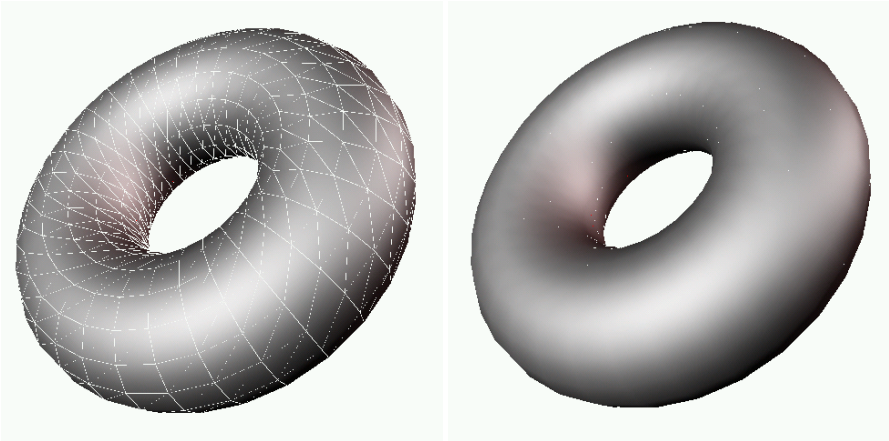

Figure 4: Rendering artefacts in case of no multisampling (left), almost no artefacts in case of 3x3 multisampling (right).

Our ray-triangle intersector code is based on the original implementation of the algorithm introduced by Moller and Trumbore [17]. Various tests have shown the efficiency of this algorithm, for example in [16]. For the shading we use the Phong lighting model.

4.1 Ray Tracing on the CPU

Integrating the item buffer based visibility processing and shadow mapping into a ray tracer leads to the following multi-pass rendering algorithm – the ray tracing related part of the algorithm is outlined in Figure 5. First, we create an item buffer which holds the indices to all currently visible triangles. For this purpose we use a 32bit floating point off-screen `framebuffer object` (FBO) with a depth attachement. Then, for every light source we create a shadow map by rendering the entire scene from the light's point of view. Again we use an FBO to accomplish this task. The item buffer and shadow maps can now be transferred from the GPU to host memory; to hide the delay caused by this operation we perform an asynchronous memory transfer using the OpenGL `pixel buffer object` extension (PBO). This enables us to start the intersection testing before the entire item buffer is downloaded completely. For every primary ray we perform one ray-triangle intersection test, the correct triangle is thereby quickly determined using a trivial lookup operation based on the index stored in the item buffer. If the intersection succeeds we shade the pixel according to the shadow map and lighting model in use, if the test fails we perform additional intersection tests to avoid rendering artefacts. To do so we employ the following "lazy multisampling" strategy: if the first intersection test fails we loop over the local $n \times n$ neighbourhood in the item buffer and perform $n^2 - 1$ ray-triangle intersection tests with the triangles referenced by the indices of the surrounding buffer entries. In most cases we are able to find correct intersections using this approach, if not we interpolate the pixel colour using the surrounding pixels. Using this multisampling strategy the item buffer has exactly the same dimensions as the framebuffer. Another approach to avoid these rendering artefacts is to supersample the scene. The obvious drawback of this method is that many more primary rays have to be shot in order to produce the final image.

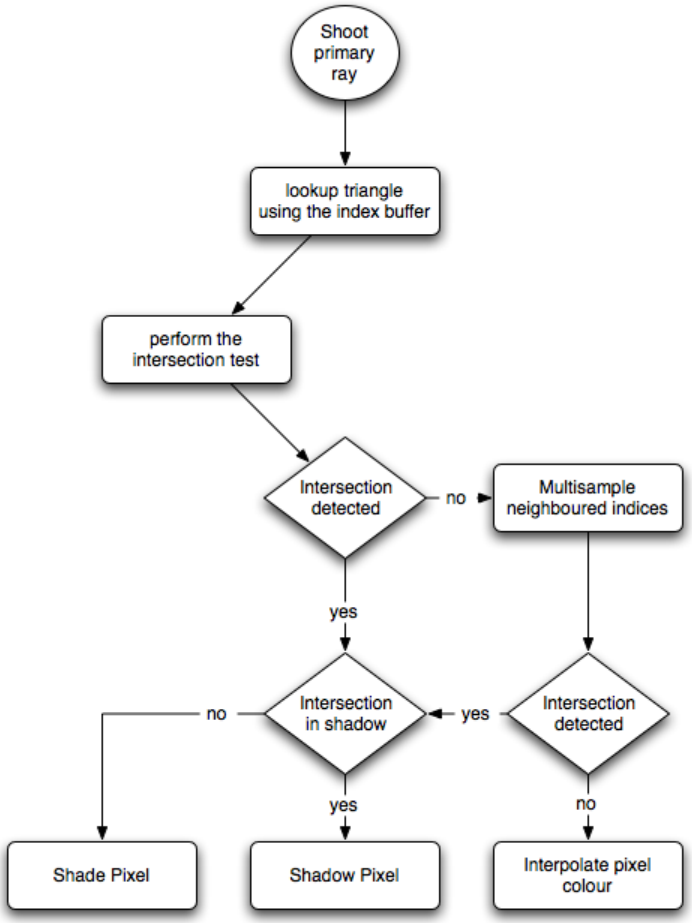

Figure 5: The ray tracing algorithm using an item buffer and shadow mapping.

To further accelerate the rendering we also implemented *interleaved rendering*. In this mode we only shoot every n^{th} primary ray, the remaining pixels are then coloured using interpolation in a second rendering pass. Because this technique leads to higher frame rates than normal rendering but also to a blurred image it is especially useful for camera animations. In Figure 7 we show a comparison between interleaved and normal rendering.

4.2 Ray Tracing on the GPU

To achieve a higher rendering performance we rewrote our ray tracer to run completely on the GPU using a single fragment shader program. Among other improvements this allows us to get rid of the item buffer and shadow map readback operation. Furthermore, we do not only gain from the superior performance and parallelism of modern GPUs, but we also may use the free CPU cycles for other, non rendering related tasks.

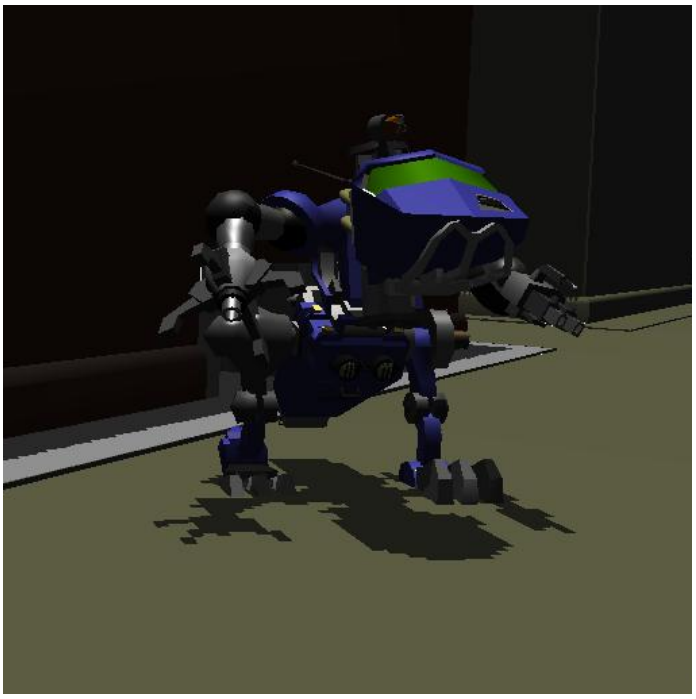

Figure 6: The BART robots scene ray traced on the CPU using the item buffer and shadow mapping.

4.2.1 Single-Pass Rendering

The first step of the GPU based implementation – item buffer creation – is done exactly as in the CPU based implementation. But instead of reading the data back to host memory it remains on the GPU as a texture. The same applies for the shadow maps. We then activate our fragment shader and draw a screen-sized quad enforcing one fragment shader pass per pixel. For every pixel the shader now reads the index from the item buffer texture and – if it is valid – performs the intersection test. Finally, if a hit has been detected the pixel is shaded according to the information stored in the shadow maps. To be able to access the scene data in the shader – such as normals, colours and so on – we have to store it into multiple 32bit floating point RGBA textures. To find out whether a point in 3D is shadowed or not we use the following function:

```
bool isShadowed(const vec4 pos, mat4 lightMVP)
{
    vec4 smtc = pos * lightMVP;  // sh_map texcoords
    float shadowDepth = texture2D(sh_map, smtc.xy).x;
    float posDepth    = smtc.z - EPSILON;

    if(shadowDepth < posDepth) return true;
    else return false;
}
```

Here `lightMVP` is the model-view-projection matrix for the current light source as described in [6], `sh_map` the shadow map and `pos` the position in the 3D scene.

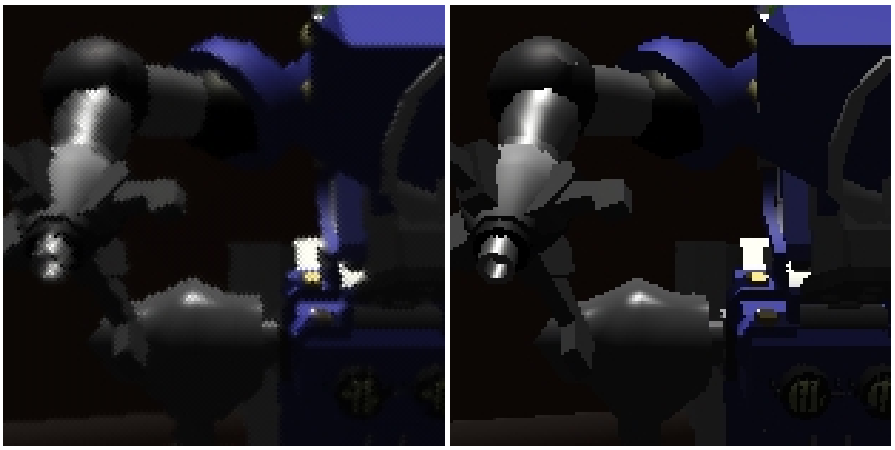

Figure 7: Interleaved rendering (left) leads to higher frame rates than normal rendering (right), but also to a blurred image [extract from the robots scene].

4.2.2 Multi-Pass Rendering

To accelerate the GPU based ray tracing implementation even more we tried to discard the pixels which do not contribute to the final image. To avoid costly shader executions for these pixels we mask them out in a separate rendering pass right after item buffer creation. In the following passes only the pixels which were not masked out are further processed. This technique is widely known as *early-z culling*. For this purpose we had to split the GLSL implementation into multiple shaders – an early-z tester, ray-triangle intersection tester and pixel shader – and perform the rendering in multiple passes. Unfortunately this optimisation did not lead to better results because the current OpenGL drivers are not yet capable of performing early-z culling on 32bit floating point FBOs. This will hopefully change in a near future.

5 Discussion and Results

We benchmarked our implementations on an AMD Athlon64 3500+, GeForce 7800GT workstation running Windows XP. The installed driver had revision 8421. Tests on a dual 2.8 GHz Xeon CPU workstation running Fedora Core 4 Linux lead to comparable results. For the benchmarks we used three different scenes of various complexity, a simple torus, the well-known Stanford bunny and the animated robots scene from the BART ray tracing benchmark [15]. Their main characteristics are listed in Table 1. Furthermore, one light source is turned on while rendering.

Name	Triangles	Notes
Torus	1024	A simple scene
Bunny	69451	The Stanford bunny
Robots	71708	An animated scene

Table 1: The three sample scenes.

Since the base number of primary rays is the same for all three scenes we expected other

factors to be significant with respect to differing frame rates; e.g., the time to create the item buffer and shadow maps, differences in shading or a varying number of item buffer multisampling as well as dynamic scene updates. In order to access the item buffer on the CPU we have to perform a framebuffer readback operation. Fortunately, by using the high bandwidth of the new PCI Express for Graphics (PEG) 16x standard and asynchronous memory transfer this is not so much of a performance problem anymore. Although, in Table 2 we see that asychronous memory transfer is only a little faster than the synchronous readback. This is mostly because the memory transfer itself only takes about *2–3ms*, and as such only uses a fraction of the total spent rendering time. Moreover we did not use a very fine-grained memory transfer strategy, the performance would clearly benefit from transferring more but smaller memory chunks per frame. In addition, item buffer creation only takes between *3ms* and *15ms*. This could even be further optimised using advanced rendering techniques instead of *immediate mode rendering*, as it is done here.

	Robots	Bunny	Torus
synchronous	2.8	6.1	4.4
asynchronous	2.9	6.2	4.5
item buffer creation	73	71	850

Table 2: Frames per second, measured without shadow mapping and item buffer multisampling. The last row shows the performance achieved when computing the item buffer exclusively [without transfer to host memory].

	Robots	Bunny	Torus
CPU, no shadows	3.2	6.1	4.5
CPU, shadowed	1.4	5.5	3.8
GPU, no shadows	19	36	117
GPU, shadowed	19	36	106

Table 3: Frames per second with and without shadow rendering, no multisampling.

	Robots	Bunny	Torus
CPU, interleaved	3.4	5.8	5.0
CPU, no multisampling	2.6	4.9	4.5
CPU, lazy 3×3	2.5	3.9	4.3
CPU, lazy 5×5	2.4	2.5	4.1
CPU, 3×3	1.6	2.2	2.3
CPU, 5×5	1.2	1.2	1.4
GPU, no multisampling	19	36	117

Table 4: Frames per second with and without multisampling, no shadow mapping.

Please note that the reason for the higher frame rates of the bunny is that less intersections occur and thus less shading operations have to be performed – the torus simply fills the viewport to a higher extent than the bunny. This is the case in all our benchmarks.

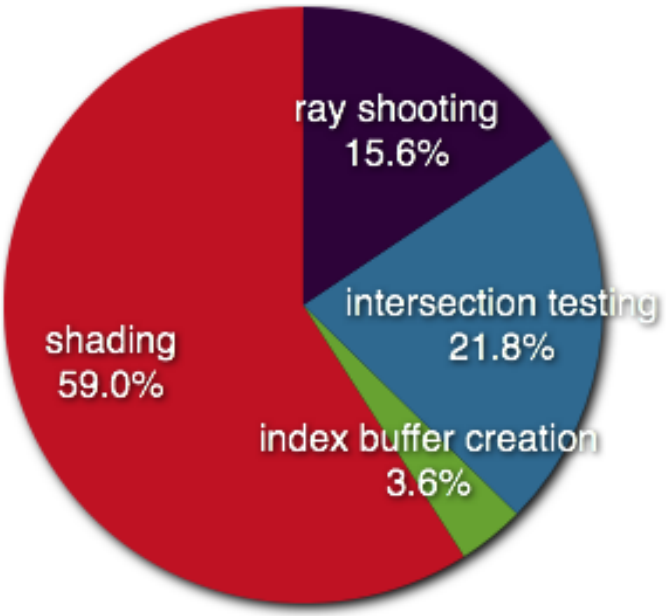

Figure 8: The cost distribution of the CPU cycles when rendering the robots scene on the CPU.

Comparing the rendering times between the CPU and the GPU version reveals that the latter is faster almost by an order of magnitude – see Table 3. This can be explained by the fact that we did not use parallel execution to perform the ray shooting on the CPU whereas we gain from the parallelism on the GPU. Moreover, we did not have to perform memory transfers between the GPU and the host memory.

It is obvious that when performing item buffer multisampling the achieved rendering performance drops noticeably, as can be seen in Table 4. In most cases the ”lazy multisampling” strategy represents the best compromise between performance and quality. Most of the time is spent in ray shooting, intersection testing and especially shading operations. Figure 8 shows the average cost distribution when rendering the robots scene on the CPU. Here, item buffer creation is almost negligible. The values for the GPU variant on the other hand are much more influenced by the item buffer creation costs as well as the scene graph updates – in case of the dynamic robots scene – which enforce expensive data texture updates once per frame.

Another advantage of our ray tracer is that the achieved frame rate remains more or less constant when rendering animations or fly-throughs, no matter how the triangles are distributed in space – as long as the entire viewport is covered by the scene. The curve in Figure 9 depicts the frame rate over time for the animated robots scene, rendered on the CPU without multisampling.

Last but not least we comment on the visual quality of the rendering. The results are in most cases engaging, although if the scene contains many tiny triangles, such as the Stanford bunny, a higher item buffer multisampling rate is a necessity to produce good results. If there are too many such tiny-sized triangles close to each other visual artefacts are inevitable. We try to overcome these faults by shading the missing pixels

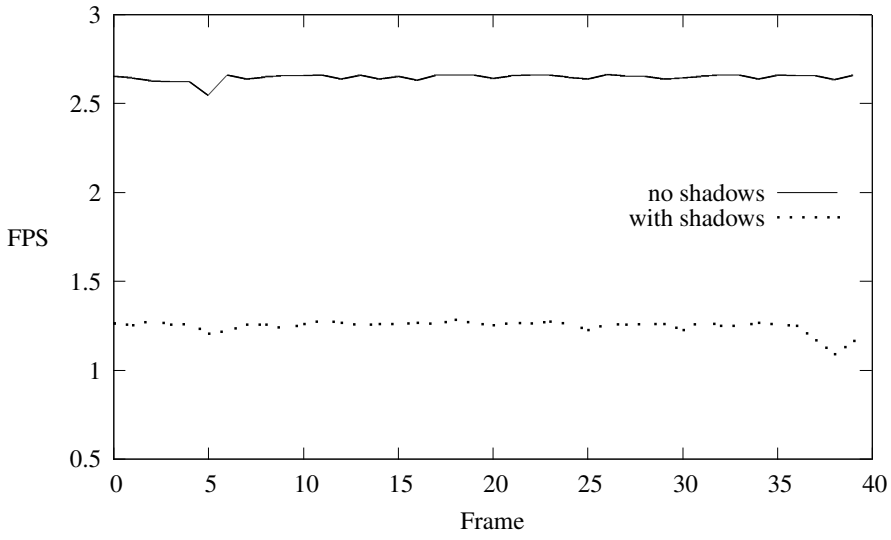

Figure 9: Frames per second over a period of time for the robots scene.

using colour interpolation. We believe that in this case the achieved quality is satisfactory for dynamic scenarios by all means, but less for static renderings – see Figure 10. The visual differences between the GPU and the CPU versions of our ray tracer are of minor importance. It seems that the 32bit floating point implementation on Nvidia GPUs is indeed very close to the IEEE-754 standard.

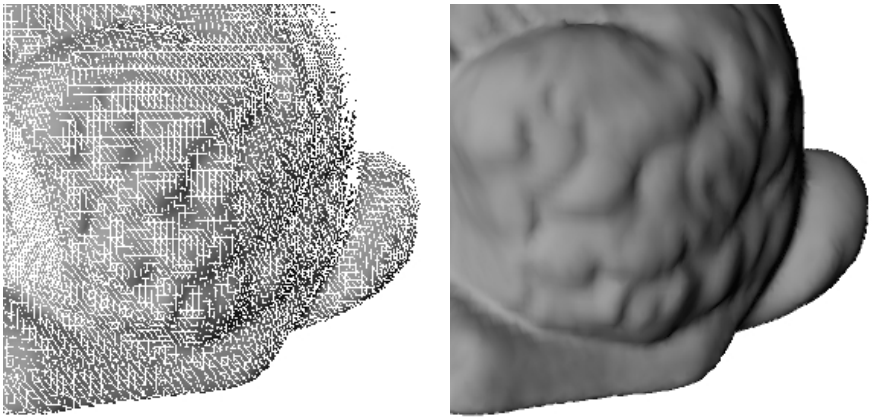

Figure 10: Artefacts without multisampling and interpolation (left), no visible artefacts with 3×3 multisampling and interpolation (right).

6 Conclusion

In this paper we presented a ray tracer which uses an item buffer to achieve high frame rates instead of a more common, spatial acceleration structure. We showed that it is

feasible to combine the raw power of GPUs and screen- and image-space rendering techniques to accelerate ray tracing. This facilitates ray tracing of animated scenes at interactive frame rates. In addition, we are using shadow mapping instead of shooting shadow feelers to perform the shadowing computations.

We describe two implementations, both accelerated by the graphics hardware. The first implementation only uses the graphics hardware to build the item buffer and shadow maps, the second one is done entirely on the GPU using GLSL. We demonstrated some advantages of this approach as well as the superior performance of the GPU based implementation. On the downside, secondary rays cannot be performed reasonably fast without integrating a spatial acceleration structure, more work is required to address this issue. Also, shadow mapping often produces hard shadows. This could be softened by filtering the borders of the shadowed regions.

In order to achieve higher frame rates we could further optimise our code using the SIMD functionality of the CPU and scalable rendering techniques. E.g., we could make use of primary ray coherence and apply parallel rendering strategies using multiple GPUs to create the item buffer and the shadow maps as well as to perform the ray tracing.

7 Future Work

Numerous possibilities exist for follow-up research work. First, we would like to integrate reflection and refraction rays into our interactive ray tracer. We see a good chance in achieving this using a multi-level ray tracing approach based on more traditional acceleration structures [23]. Second, we would like to scale the rendering using parallel rendering strategies devoting multiple GPUs and CPUs. One particular advantage of ray tracing over z-buffer based rendering is its built in occlusion culling which enables efficient rendering of complex scenes. Unfortunately, since we are using OpenGL to create the item buffer this advantage diminishes. In order to address this issue we are investigating a temporally adaptive algorithm to create the item buffer, resulting in much less OpenGL rendering. The concept is closely related to the so called *frameless rendering*. The idea is to exploit the coherence in imagery across time in order to restrict updates only to those parts of the buffer which are subject to changes.

8 Acknowledgements

The authors would like to thank H. Bieri, S. Schär (University of Bern), S. Eilemann (University of Zurich), J. Hutchison and all other reviewers for their valuable assistance and discussions.

References

[1] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. The GPU as Numerical Simulation Engine.

[2] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[3] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46. Eurographics Association, 2002.

[4] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast GPU Ray Tracing of Dynamic Meshes using Geometry Images. In *Proceedings of Graphics Interface 2006*, 2006.

[5] Martin Christen. Ray Tracing on GPU. Master's thesis, University of Applied Sciences Basel, 2005.

[6] Cass Everitt, Ashu Rege, and Cam Cebenoyan. Hardware Shadow Mapping. In ACM SIGGRAPH 2002, 2002.

[7] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. In *Graphics Hardware 2004*, 2004.

[8] Tim Foley and Jeremy Sugerman. KD-tree acceleration structures for a GPU raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[9] Eric A. Haines and Donald P. Greenberg. The Light Buffer: a Shadow Testing Accelerator. *IEEE Computer Graphics and Applications*, 6(9):6–16, 1986.

[10] Filip Karlsson and Carl Johan Ljungstedt. Ray tracing fully implemented on programmable graphics hardware. Master's thesis, Chalmers University of Technology, 2004.

[11] John Kessenich, Dave Baldwin, and Randi Rost. OpenGL 2.0 Shading Language, 2003.

[12] Sehyun Kim, Sung ye Kim, and Kyung hyun Yoon. A Study on the Ray-Tracing Acceleration Technique Based on the ZF-Buffer Algorithm. In *IV '00: Proceedings of the International Conference on Information Visualisation*, page 393, Washington, DC, USA, 2000. IEEE Computer Society.

[13] Bernd Lamparter, Heinrich Müller, and Jörg Winckler. The Ray-z-Buffer—An Approach for Ray Tracing Arbitrarily Large Scenes. Technical Report report00021, 1990.

[14] Aaron Lefohn, Joe M. Kniss, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Glift: Generic, Efficient, Random-Access GPU Data Structures. *ACM Transactions on Graphics*, 25(1):60–99, jan 2006.

[15] Jonas Lext, Ulf Assarsson, and Thomas Moeller. BART: A benchmark for animated ray tracing. Technical Report 00-14, Chalmers University of Technology, Goeteborg, Sweden, 2000.

[16] Tomas Möller. Practical Analysis of Optimized Ray-Triangle Intersection.

[17] Tomas Möller and Ben Trumbore. Fast, Minimum Storage Ray-Triangle Intersection. *J. Graph. Tools*, 2(1):21–28, 1997.

[18] Steve Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Algorithms*, pages 23–32, July 1994.

[19] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[20] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[21] Timothy John Purcell. *Ray tracing on a stream processor*. PhD thesis, Stanford University, 2004.

[22] Erik Reinhard, Brian Smits, and Charles Hansen. Dynamic acceleration structures for interactive ray tracing. *Proceedings Eurographics Workshop on Rendering*, pages 299–306, June 2000.

[23] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3):1176–1185, 2005.

[24] David Salesin and Jorge Stolfi. The ZZ-Buffer: A Simple and Efficient Rendering Algorithm with Reliable Antialiasing. In *Proceedings of the PIXIM '89 Conference*, pages 451–66, 1989.

[25] Lars Ole Simonsen and Niels Thrane. A Comparison of Acceleration Structures for GPU Assisted Ray Tracing. Master's thesis, University of Aarhus, 2005.

[26] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *ACM Comput. Surv.*, 6(1):1–55, 1974.

[27] Suresh Venkatasubramanian. The Graphics Card as a stream computer. In *SIGMOD-DIMACS Workshop on Management and Processing of Data Streams*, 2003.

[28] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[29] Ingo Wald and Philipp Slusallek. State-of-the-Art in Interactive RayTracing. EUROGRAPHICS 2001, 2001.

[30] Hank Weghorst, Gary Hooper, and Donald P. Greenberg. Improved Computational Methods for Ray Tracing. *ACM Trans. Graph.*, 3(1):52–69, 1984.

[31] Daniel Weiskopf, Tobias Schafhitzel, and Thomas Ertl. GPU-Based Nonlinear Ray Tracing. In *Eurographics 2004*, 2004.

[32] Lance Williams. Casting curved shadows on curved surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274, New York, NY, USA, 1978. ACM Press.