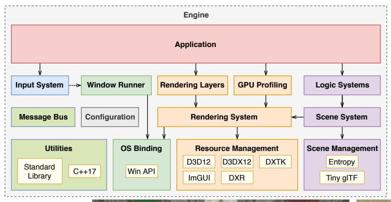
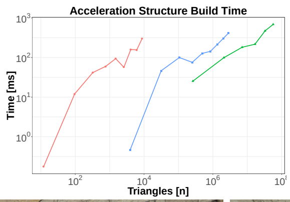
BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

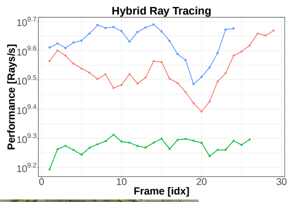
Excel@FIT 2019
http://excel.fit.vutbr.cz

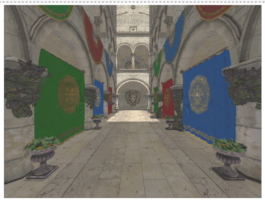
Hybrid Raytracing in DXR

Tomáš Polášek*









Abstract
The goal of this paper is to assess the usability of hardware accelerated ray tracing in near-future rendering engines. Specifically, the *DirectX Ray Tracing API* and Nvidia Turing GPU architecture are being examined.
The assessment is accomplished by designing and implementing a hybrid rendering engine with support for hardware accelerated ray tracing. This engine is then used in implementing frequently used graphical effects, such as shadows, reflections and Ambient Occlusion. Second part of the evaluation is made in terms of difficulty of integration into a regular game engine - complexity of implementation and performance of the resulting system.
There are two main contributions of this thesis, the first one being *Hybrid Rendering* engine called *Quark*, which uses hardware accelerated ray tracing to implement above-mentioned graphical effects. The hybrid-rendering approach uses rasterization to perform the bulk of the computation-intensive operations, while allowing ray tracing to add additional information to the synthesized image. The second important contribution are the performance measurements of the final system, which include time spent on the ray tracing operations and number of rays cast for different input models.
Presented system shows one possible way of using the *Nvidia Turing Ray Tracing cores* in generating more realistic images. Preliminary measurements of the rendering system show great potential of this new technology, with results of 5 to 12 *GigaRays* per second on *RTX 2080 Ti*. The largest problem so far is the integration of this technology into rasterization-based engines. Data needs to be prepared for ray tracing and manually accessed from ray tracing shaders. The second problem is the build-time of acceleration structures, which is in order of milliseconds, even for smaller models with around 50 thousand triangles.

Keywords: Hybrid Ray Tracing — DirectX Ray Tracing — Hardware Accelerated Ray Tracing

Supplementary Material: Demonstration Video — Reflection Comparison — Downloadable Code — Automated Testing Video

*xpolas34@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Ray tracing has always been the more intuitive and straightforward way of solving many of the rendering problems – primary ones being visibility, lighting and shadows. One problem which always comes up with the use of ray tracing is its computational complexity, which is still higher, when compared to rasterization. It is the goal of hardware accelerators, such as the new *Nvidia Turing* [1], to allow us to implement real-time ray tracing algorithms. According to the official reports, the *Ray Tracing cores* should allow for casting of up to 10 GigaRays per second [2].

The goal of this thesis is to evaluate the potential of this new technology and try to solve some of the most common rendering problems with it. While the primary concern is certainly the computational efficiency of the solution, which can be measured in rays per second, there are also other matters which are of interest. One of them is the pre-processing time, which is required in order to prepare the scene data for ray tracing. There are also problems with the nature of ray tracing, which requires to have the whole scene available, since any single ray cast may hit practically any part of the scene.

A big part of the evaluation is the practical use of the ray tracing cores, using *DirectX 12* and *DirectX Ray Tracing*, in implementing some graphical effects, which are difficult or impossible to achieve with rasterization. First of these effects are hard shadows, which are relatively well solved with rasterization, with many methods such as *Shadow Mapping* [3] or *Shadow Volumes* [4]. Both of these methods have their problems, while *Shadow Mapping* can be implemented very efficiently, there are many artifacts which need solving – e.g. *Shadow Acne* or *Peter Panning*. In comparison, *Shadow Volumes* method generates much more precise shadow, however it has serious performance problems.

Another effect which is quite problematic for rasterization methods to get right are reflections. There are two main ways of solving reflections when using pure rasterization – resolving the image in screen space [5] or placing secondary cameras behind reflective surfaces. The first method is most commonly used for fast reflections, which do not require precision. The biggest disadvantage is the missing information about objects not currently visible on-screen, which means their reflections will be missing. The other approach, by placing hidden cameras, comes short when trying to display reflective surfaces which are not perfect planes, or some other well-defined primitives.

Last of the tested effects is *Ambient Occlusion*, which is in itself an approximation of the way in which enclosed spaces are usually darker, since less light gets to them. Among the most used rasterization approaches are methods *Screen-Space Ambient Occlusion* or *Horizon Based Ambient Occlusion* [6]. All of these methods work in screen space, which is quite fast, but comes with a multitude of artifacts.

To fairly assess the potential of the *Ray Tracing cores*, this thesis proposes a new type of rendering system, called *Hybrid rendering*, which combines advantages of both rasterization and ray tracing. One of the contributions of this thesis is the design and implementation of a *Hybrid rendering* engine called *Quark*, which is built using *DirectX 12* and *DirectX Ray Tracing API* [7]. This engine is then used in implementation of above-mentioned graphical effects.

The results presented by this thesis show the potential of this new technology, which could lead to radically different types of effects, in contrast to currently used rasterization techniques. There are however some challenges, in terms in integration of this system into existing engines, which are yet to be resolved.

2. Principles of Hybrid Rendering

The basic idea of every renderer is to, more or less precisely, solve the *Rendering equation* [8]. The *Rendering equation* is an integral equation which, when solved for a given scene, results in *radiance* of each point in that scene, as viewed from some given direction.

Currently there are two main approaches used for solving the *Rendering equation* and synthesizing the final image. The first one is through the use of rasterization, which takes the input polygons – usually triangles – and fills the pixel grid correspondingly. Rasterization is used for most real-time graphics, since it is very fast and easily accelerated, however the output is not very precise in respect to solving the *Rendering equation*.

The second commonly used approach is to use *Ray Casting*, which leads to the *Ray Tracing* techniques. The main idea of these methods is to repeatedly cast rays into the scene, through the camera plane or lens, which allows them to solve the visibility between parts of the scene. This maps very well to solving the *Rendering equation*, since we are essentially searching the scene for reflected light-sources.

The *Hybrid rendering* technique, presented as a part of this thesis, is simply a combination of these two approaches, taking advantage of their respective benefits. Its carrying idea is very similar to rasterization technique called *Deferred rendering*, the rendering process can be surmised as follows:

1. Target scene is prepared for rasterization.
2. The scene is rendered using a *Deferred pass*, which generates a so called *G-Buffer*. This *G-Buffer* has several layer – position, normal, albedo, material, depth etc. Example of a *G-Buffer* content can be seen in figure 1.
3. Scene is prepared for ray tracing. This includes the building of all acceleration structures.
4. The scene is rendering using the *Ray tracing pass*. The *G-Buffer* is used to provide some basic information, which reduces the number of necessary rays – there is no need to cast the primary rays, since rasterization pass already generated information for the visible geometry.
5. Final image is resolved by using the information generated by the rasterization and ray tracing passes and is displayed on the screen.

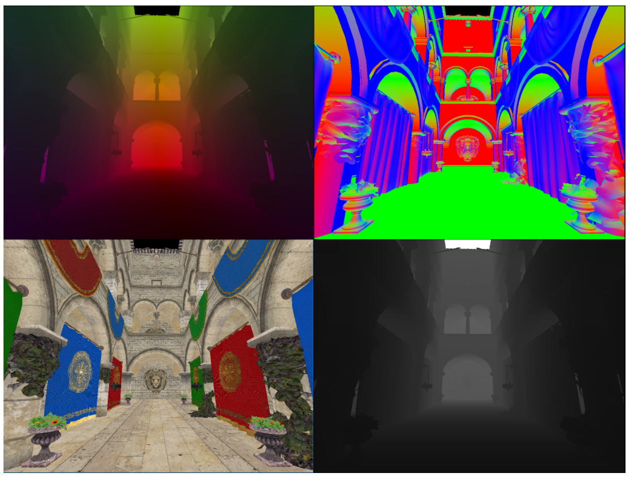


Figure 1. Example content of a G-Buffer, which contains information about position in the scene (**top-left**), normal (**top-right**), albedo (**bottom-left**) and depth (**bottom-right**).

The goal of this approach is to minimize the number of rays cast in each frame, keeping the heavy lifting operations on the rasterization pass, which solves the basic visibility and shading. Ray tracing is then used to fill in more precise information – like reflections and shadows.

3. Hardware Accelerated Ray Tracing

The main problem of *Ray Tracing* methods is their computational complexity. Casting a ray through a scene and finding the closes hit is not a trivial task. There are several ways of accelerating this operation, mainly through the use of *bounding volumes* and *acceleration structures* [9]. Another problem is that – since each ray can essentially hit any part of the scene – all of the data has to be accessible, which requires random access to most of the resources. This is in contrast to rasterization, where there is usually well-defined set of parameters for each triangle and even seemingly random access resources, like textures, are still accessed in a predictable pattern.

Rasterization is commonly accelerated using *GPUs*, which allows to push out millions of pixels per second. There were already many attempts to build hardware acceleration for ray tracing [10], however most of them failed outright, or were slowly forgotten.

Among the ways of accelerating ray tracing, is either through computation of collisions on CPUs, or through the use of GPGPU calculations, using compute shaders. The new *Nvidia Turing* [2] GPU architecture contains specialized hardware cores – *Ray Tracing cores* – which allow the acceleration of ray tracing. The acceleration consists of two parts: traversal through the acceleration structure and intersection evaluation. Diagram of this process can be seen in figure 2.

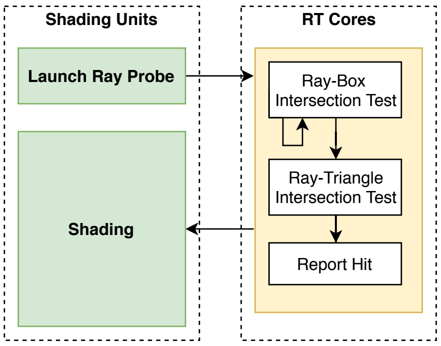


Figure 2. Diagram displaying the process of ray tracing acceleration using the *Ray Tracing cores* [2]. The operation starts by launching a ray probe from the shading cores. This request is passed to the Ray Tracing cores, which asynchronously [1] perform the operation. This allows the shading units to perform other work in the mean time.

4. Design of Hybrid Rendering Engine

One of the crucial parts of this thesis is the *Hybrid rendering* engine, which is later used for experiments with the new hardware accelerated ray tracing system.

The main goal of this engine was to allow the use of hardware accelerated ray tracing, which also shaped its design and chosen libraries. The whole rendering engine can be thematically divided into sub-systems – which can be seen in figure 3 – the most important of which is the *rendering sub-system*. Its foundation is built on *DirectX 12* and *Direct3D API*, which is a lower-level alternative to *OpenGL* – similar to *Vulkan*.

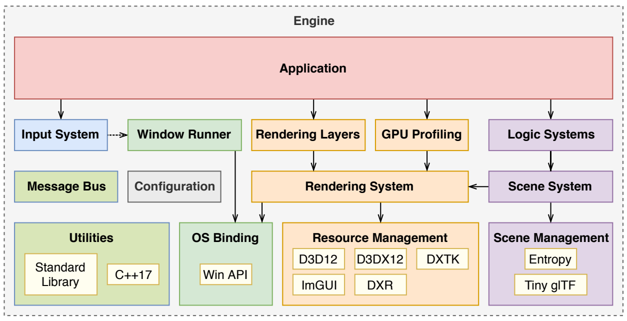


Figure 3. Diagram containing the rough partitioning of the engine, into sub-systems. Libraries used can be discerned by the **white** background.

DirectX 12 has been chosen because it was the only[1] rendering API which supported hardware accelerated ray tracing through the use of *DirectX Ray Tracing* [7].

4.1 DirectX Ray Tracing

DirectX Ray Tracing, or *DXR* in short, is an extension of *DirectX 12*, which consists of the following three parts: *Shader Tables*, *Acceleration Structures* and the *Ray Tracing Pipeline*. First of these parts – the *Shader Table* – is a concept, which is not necessary when using the standard rasterization pipeline. Each rasterization shader type can only contain one implementation, while some of the ray tracing shaders may have multiple definitions. Rays cast into the scene may hit any geometry, which means all of the shaders have to be available – chosen shader depends on which geometry has been hit by the ray. Each table may contain several shader programs and their *local* parameters, example table can be seen in figure 4.

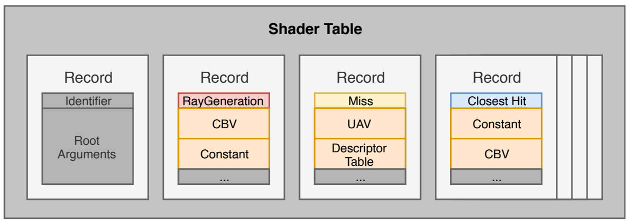


Figure 4. Example shader table, which contains multiple shader types [11] – *Ray Generation*, *Miss* and *Closest Hit*. Each shader program is packed together with its local parameters.

Included with these *Shader Tables* are the new shader types, the original rasterization shaders cannot be used with ray tracing pipeline. The new shaders can be seen in figure 5.

The second, and the most important part, is the acceleration structure, which is used for accelerated ray-scene computation. There are two parts to acceleration structures in *DXR* – top-level and bottom-level.

[1]Currently accelerated ray tracing is also supported by an extension of the *Vulkan API*.

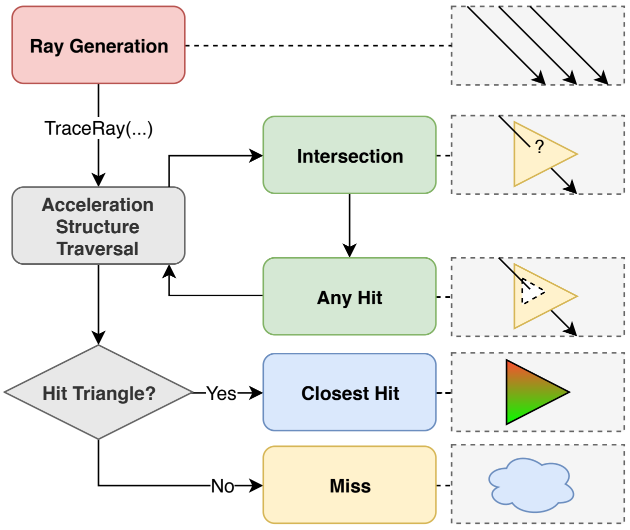


Figure 5. New types of shaders, used in *DirectX Ray Tracing* [11]. *Closest Hit* shaders are roughly equal to the *Pixel* or *Fragment* shaders, and may be specified separately for each geometry.

The bottom-level is generated from provided geometry – vertex position data and, optionally, indices. At the leaf level, the bottom-level acceleration structure contains triangles, which are wrapped in hierarchy of bounding volumes, the upper-most of which contains the whole model[2]. There are usually many bottom-level acceleration structures per one scene.

The top-level acceleration structure is used to instantiate the bottom-level geometry, placing it into the virtual scene. There are several parameters, which can be specified for each of the instances – transformation within the scene, offset within the shader table and identifier of the bottom-level structure. The number of top-level acceleration structures is not limited, however there is usually only one per scene. Connection between these two acceleration structures can be seen in figure 6.

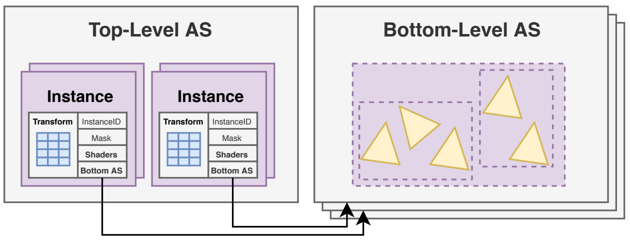


Figure 6. Diagram of the acceleration structures used in *DXR* [11]. Bottom-level is used to store the geometry, while the top-level instantiates this geometry and places it within the virtual scene.

The final part of *DXR* is the ray tracing pipeline, which connects all of the above parts together. Apart

[2]It is possible to pack several models into a single bottom-level acceleration structure, which has performance benefits.

from the *Shader Tables*, it is also possible to provide other parameters, such as maximal recursion depth of the ray casts and size of the ray payload, which is a user-defined structure carried by each ray.

5. Implementation Details

The most important part of the engine is the rendering sub-system, which has been implemented using the *DirectX 12 API*. In order to allow easy comparison of ray tracing and rasterization outputs, all of the rendering effects are implemented in *Rendering Layers*, each of which contain one or more passes. The application allows switching between 3 primary modes – *Rasterization*, *Full Ray Tracing* and *Hybrid Ray Tracing*. There are also two ray tracing acceleration back-ends implemented, one of which is automatically chosen at application startup:

- Hardware acceleration using *Ray Tracing cores*.
- Fallback using compute shaders [12].

The rasterization rendering layer is very rudimentary and allows rendering of textured scenes for comparison. Most of the ray tracing operations have been implemented as a part of the ray tracing rendering layer. It contains 3 passes, which are called *Deferred*, *Ray Tracing* and *Resolve*. First of these generates the necessary G-Buffers (position, normal, albedo and material), which are used in the next layers. *Ray Tracing* rendering layer contains implementation of the ray tracing effects described above. The last layer – the *Resolve* layer – takes input from the *Deferred* and *Ray Tracing* layers and combines their output into the final image.

The ray tracing rendering layer requires access to the geometry and textures of the scene, which required implementation of caching mechanism, which prepares these resources for further use. Since the ray tracing shaders do not receive any other attributes than the world-space position of the hit geometry, it is necessary to access the other vertex attributes directly, through the use of raw byte-buffers. These buffers contain the same vertex data, which is passed to the rasterization. Indexation of these buffers is performed manually, through the use parameters specified in the top-level acceleration structure.

Implementation of the above-mentioned graphical effects is contained within the ray tracing shaders. Hard shadows use a single ray, which is cast from the world position recovered from the G-Buffer in direction of the light. These rays are cast with special flags, which do not trigger the execution of *Closest Hit* shaders, which increases the performance. Reflections are calculated by casting rays very similar to the primary rays used in full ray tracing. These rays require access to the vertex and texture buffers, which means they are much more expensive.

Ambient occlusion can be efficiently solved using ray tracing, by casting short-range rays around the tested point in space and calculating the ratio of occluded rays. Diagram of this method can be seen in figure 7. The rays are similar to the shadow rays, however their length is reduced to the desired ambient occlusion range. The shorter range allows for better performance, which allows casting many more rays – 8, 16 or more.

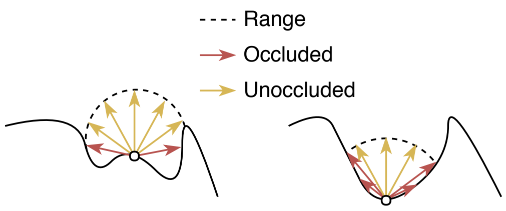


Figure 7. Ambient occlusion methods approximate detail shadows by calculating ratio of occluded rays in a specified position. Created effect allows the human eye to differentiate object detail, which can be seen in the *Demonstration Video*.

Among other interesting parts of the implementation are sub-systems for profiling and scene management. The profiling sub-system allows simultaneous profiling of CPU and GPU code, through the use of GPU queries. Scenes can be loaded from internal format or *glTF* [13] files, which is widely supported. Much more thorough description of the implementation can be found in the full text of this thesis.

6. Experiments and Results

There are many performance parameters, which are worth investigating. The first metric I have chosen to measure, is the maximum number of rays cast per second – *GigaRays/s* as presented in the official specification [2]. Since the performance could be radially different based on scene complexity and number of rays which actually hit geometry, there are multiple testing scenarios presented.

Another interesting parameter is the necessary preparation time, which is mainly the build time of the acceleration structures. Since these structures are not guaranteed to be compatible between devices [11], it will be necessary to generate them at run-time. This parameter can change the way this technology is used, since long build times may prohibit their just in time building.

In order to make the measurements, multiple suitable scenes had to be found, which is where the *glTF* scene support is very helpful. There are many testing scenes available in the official examples, from which the following were chosen: *Textured Cube*, *Suzanne* and *Sponza*. Information about each scene can be found in table 1. Each of these scenes tests a different scenario, for example the cube has a very simple geometry, which allows to stress test the *Ray Tracing cores*. *Sponza* on the other hand has many sub-objects, which require more complex acceleration structures.

Table 1. Table contains information about tested scenes. Each mesh is processed seperately, but contained within the same bottom-level acceleration structure. Texture count includes textures with diffuse, metallic-roughness and normals information.

Scene	Triangles	Meshes	Textures
Box	12	1	2
Suzanne	3936	1	2
Sponza	262267	103	69

After some initial experiments, it became clear that there is one more parameter to control for – the position and direction of the camera. Performance can be radically different, when only part of the screen is covered with geometry and most of the rays miss completely; For this reason, the first batch of testing scenarios is performed with static camera locations.

The second profiling category is focused on acceleration structure build times. There are several variables to account for:

- Number of meshes within a single bottom-level acceleration structure.
- Number of bottom-level acceleration structures.
- Number of instances within the top-level acceleration structure.

In order to test how each of these affect the build times, I have created automatic testing utilities, which take a single *glTF* scene and duplicate it in controlled manner, while measuring how long the build takes.

The last category measured category [3] attempts to re-create movement in the scene, by using automated camera tracks. These tracks can be recorded in the application, saved to hard drive and then used – example of the run can be seen in the *Automated Testing* video in the abstract.

In order to automate these tasks, the application accepts command line arguments, which enable the profiling modes. After starting up, the testing is automatically performed and profiling information is saved into a log. Testing was performed on two systems whose specifications can be found in table 2.

[3]The final thesis will contain at least one more, which will test how the recursion depth affects the number of rays per second.

Table 2. Specifications of the testing systems. **PC1** uses hardware acceleration, while **PC2** uses compute fallback [12].

	CPU	GPU
PC1	Intel Xeon W 2135	Nvidia RTX 2080Ti
PC2	Intel i5 4670k	Nvidia GTX 970

6.1 Ray Tracing Performance

Pure ray tracing performance profiling is performed by placing the camera into specific locations of the scene, which can be seen in figure 8. For each of these locations and each tested resolution, the number of *hit pixels* and *miss pixels* has been manually calculated[4], allowing precise calculation of the number of rays cast per second.

Resulting GigaRays per second values were calculated using the following procedure. Frame time values, in milliseconds, were aggregated into one second blocks and averaged. From frame times, the total number of frames per second was calculated. Other used parameters are width and height of the render area and number of rays cast per pixel (*rpp*). The final value of GigaRays per second is then:

$$gigarays = fps \cdot width \cdot height \cdot rpp \tag{1}$$

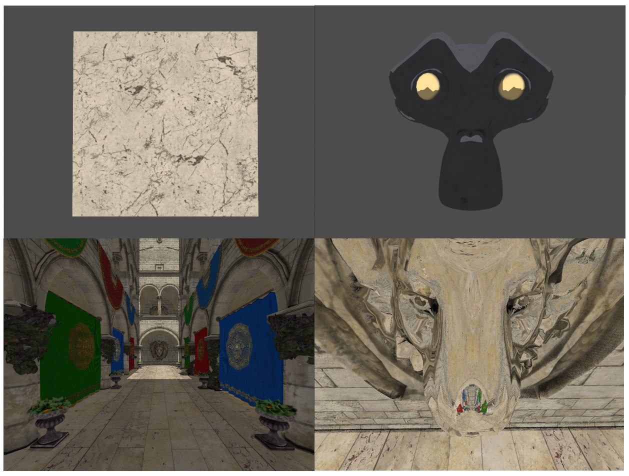


Figure 8. Camera locations used for measuring the performance. The scenes used are: *Cube* (**top-left**), *Suzanne* (**top-right**) and *Sponza* (**bottom-left**). Extra image (**bottom-right**) shows the possibilities of reflections for arbitrary geometry.

[4]Specific data can be found in the project repository **/prof/hardware/results.txt**

Profiling scenarios can be divided into two categories – standard scene rendering and stress testing the ray tracing cores. Scene rendering scenarios attempt to assess what real-life performance may look like. The three described scenes are rendered using pure ray tracing – without generating the deferred buffers. Primary rays are cast for each pixel. Upon hitting geometry one shadow ray, for light visibility, and 64 short-range ambient occlusion rays are cast. If the primary ray misses all geometry, no other ray casting occurs.

The stress testing scenarios are performed on the *Sponza* scene with 64 primary rays per pixel. The **Miss** test measures performance when all of the primary rays miss. For **Stress** test, all of the 64 primary rays hit geometry, however no additional shading is performed.

Results for the hardware accelerated ray tracing, using both testing computers, can be found in table 3. The first testing system – **PC1** with *RTX 2080 Ti* – shows performance corresponding with the official sources of up to 10 GigaRays per second. Second system, which does not have support for hardware accelerated ray tracing and uses the fallback layer, shows much lower performance.

There is a clear difference between ray tracing performance of each model, which is reduced with growing number of triangles. One of the interesting parts of the measured data is the difference between real rays per second for *Sponza* and *Suzanne* models. With much smaller number of triangles, the performance should be much higher. This result can be explained by taking into consideration the code divergence [14] between the shader units, since the position in the *Sponza* scene is chosen to have 0% miss rate.

Table 3. Table containing profiling results, taken using the static camera positions. All of these tests were performed in 1440p. First column specifies which scene/test is ran. **Rays/pixel** value is corrected for the camera position and any ray misses. **RFPS** is reference frames per second, taken using the rasterization pass.

PC1	FPS	Rays/pixel	GigaRays/s	RFPS
Box	142	25.7	12.82	2134
Suzanne	123	12.7	5.50	1879
Sponza	31	66.0	7.09	537
Miss	56	64.0	12.52	537
Stress	28	64.0	6.25	537

PC2	FPS	Rays/pixel	GigaRays/s	RFPS
Box	2.3	25.7	0.55	1524
Suzanne	2.5	12.7	0.59	1182
Sponza	0.2	66.0	0.04	291
Miss	3.8	64.0	0.85	291
Stress	0.3	64.0	0.08	291

6.2 Acceleration Structure Build Times

Ray tracing acceleration structures, which need to be built before initiation of ray tracing operations, are divided into 2 parts. The bottom structure contains the geometry, while the top acceleration structure contains instances of the bottom levels. Profiling of build times of these structures is performed by taking the base scenes – *Cube*, *Suzanne* and *Sponza* – and duplicating them multiple times in the rendered scene. The duplication is performed in a cube, so for a duplication factor two, there are 8 models present. Bottom level acceleration structures are then built for each of these models [5].

The goal of the following tests is to measure the performance of the automated build system, which is provided by the ray tracing acceleration back-ends. All of the following tests are performed with default settings, which means the acceleration structures are built for maximum ray tracing performance. During the tests, the duplication factor is gradually increased, up to total of 10. For *Sponza* model, this value had to be lowered, since the GPU was removed due to operation timeout.

Results from one of the tests, which can be seen in figure 9, display the dependence of build times on the total number of triangles in the bottom level acceleration structures. The results show roughly linear complexity in respect to the number of triangles, when taking into consideration the logarithmic axes. With growing number of triangles, the total number of bottom-level acceleration structures also increases. This leads to worse relative performace for smaller models – e.g. the *Cube*.

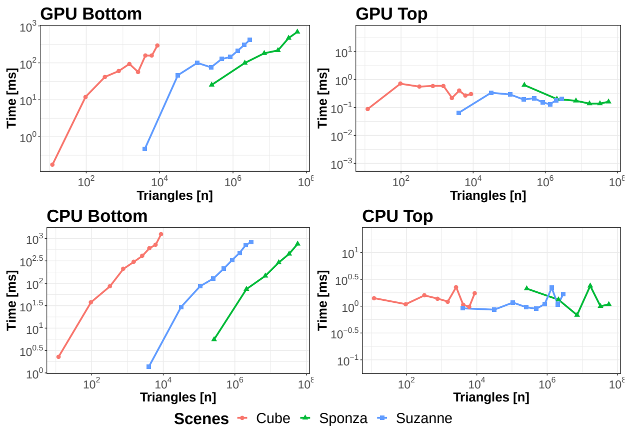


Figure 9. Graph showing build times of ray tracing acceleration structures. Base models were duplicated both in bottom-level acceleration structures and in the scene, which simulates higher number of triangles. Profiling was performed on **PC1**.

[5] De-duplication is disabled for these tests, normally duplicate meshes just reuse the same bottom level acceleration structure.

Compared to the bottom level, the top level shows almost no change with increases in the number of instances. This can be caused by relatively small number of instances, which is around 1000, for the highest duplication. Other graphs, including dependance on number of meshes and profiling for **PC1**, can be found in the project repository (folder **prof/automated/Graphs**).

The measured build times show, that even for relatively small models of around 50 thousand triangles, the resulting build times would be in the order of milliseconds. This performance prohibits building of these acceleration structures in a just-in-time fashion and necessitates a smarter building schemes. For future development of this technology, it would be beneficial to unify format of these structures and allow developers to ship pre-built structures straight with other resources.

6.3 Hybrid Ray Tracing

Since most applications or games, which would use this technology, do not intend to use it for full ray tracing and use it just for some additional effects, it would be probably used as a part of larger pipeline with many other passes. To simulate this behavior, the hybrid ray tracing approach first generates deferred G-Buffers, which allows it to skip casting primary rays. In order to simulate the camera movement, the application allows running pre-recorded camera tracks, which automatically control the camera – one of these can be seen in the *Automated Testing Video*.

In contrast to previous tests, the following tests also use different shaders, which additionally calculate simplified physically-based lighting. For each pixel one shadow ray and 8 ambient occlusion rays are cast. In case of pure ray tracing, the primary ray is also still used. Following tests were all performed on **PC1** in 1440p resolution.

Results of the first test, which can be seen in figure 10, show the difference between using pure ray tracing with primary rays and the hybrid ray tracing approach. Performance of both systems seems to be very similar, with a little higher GigaRays per second values for the pure ray tracing approach. This can be explained by the additional passes in case of hybrid ray tracing approach, where the ray tracing pass needs to wait for the deferred pass to finish rendering. In real-life scenarious, this shouldn't be a problem, since deferred buffers are generated even without using ray tracing.

Second test, shown in figure 11, shows frames per second comparison between the two ray tracing approaches and a pure rasterization solution. As expected, the rasterization is much faster, at around 15

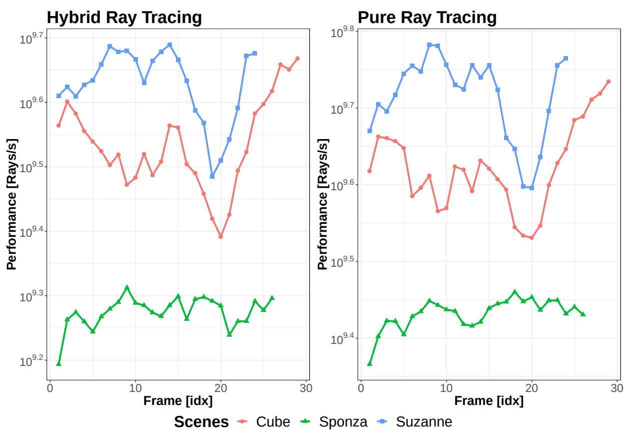


Figure 10. Graph showing the GigaRays per second values for automated camera paths in 1440p resolution. Profiling was performed on **PC1**. GigaRays per second were calculated according to the equation 1

times the number of frames per second. One interesting part of this test is the lowering of frames per second on both of the ray tracing test, with rasterization being unaffected. This is caused by the camera going closer to the object – *Cube* and *Suzanne* in this case – which results in many more primary ray hits and calculation of shadows and ambient occlusion.

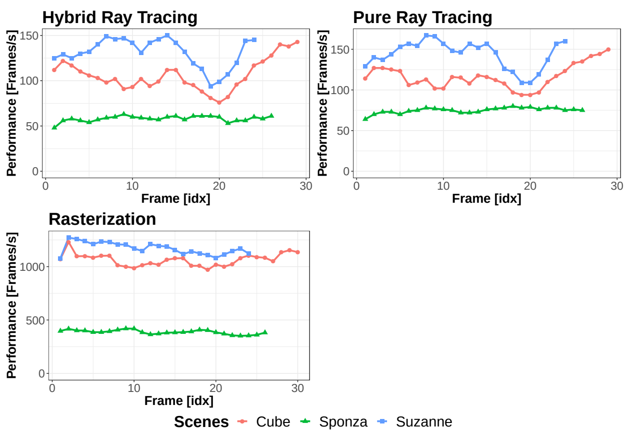


Figure 11. Graph showing the frames per second values for automated camera paths in 1440p resolution. Profiling was performed on **PC1**.

7. Conclusions

This paper presents a hybrid rendering approach, which takes advantage of both rasterization and ray tracing acceleration. This technique is then used in designing and implementing a *DirectX 12* powered hybrid rendering engine, named *Quark*, which takes advantage of the hardware accelerated ray tracing provided by the new *Nvidia Turing* GPUs. The engine is subsequently used in solving several rendering problems, which are difficult for rasterization approaches – shadows, ambient occlusion and reflections. Finally, the resulting application is profiled and the measurements are used to assess the potential of hardware accelerated ray tracing on several different devices.

Tentative results, presented in this paper, show a great potential of this new technology. For relatively performance heavy scenarios – primary ray, shadow ray and 64 ambient occlusion rays – the *RTX 2080 Ti* GPU allows to cast from 5 to 12 *GigaRays* per second. The second tested device – *GTX 970* – does not present such great performance, with around 0.04 to 0.9 *GigaRays* per second. This is however stil great result, when taking into consideration the *GTX 970* does not have the *Ray Tracing cores*.

The results presented in this thesis can be used in deciding whether the initial time investment of implementing this new technology is worth it for a given project. The design and implementation of the hybrid rendering engine is freely available and can be used as a base for other implementations.

I will be continuing to work on the hybrid rendering engine, with future plans of implementing the *Vulkan* ray tracing extension and transforming the engine into a platform for further experiments. There are also few more parameters, which I would like to measure and present in the final paper - e.g. how does the length of the ray affect the performance or comparison of iterative and recursive ray tracing. The final goal I would like to accomplish is to implement some basic global illumination technique with lighting accumulation from multiple frames.

Acknowledgements

I would like to thank my supervisor Ing. Jozef Kobrtek for guidance and Martin Sobek from 2K for helping me with choosing this topic.

References

[1] NVIDIA. NVIDIA Turing GPU architecture, 2018.

[2] Emmett Kilgariff, Henry Moreton, Nick Stam, and Brandon Bell. NVIDIA Turing Architecture In-Depth. Online, September 2018.

[3] Lance Williams. Casting curved shadows on curved surfaces. *ACM SIGGRAPH Computer Graphics*, 12(3):270–274, aug 1978.

[4] Franklin C. Crow. Shadow algorithms for computer graphics. *ACM SIGGRAPH Computer Graphics*, 11(2):242–248, aug 1977.

[5] Tomasz Stachowiak. Stochastic Screen-Space Reflections. SIGGRAPH 2015, 2015.

[6] Daniel Kvarfordt and Benjamin Lillandt. Screen Space Ambient Occlusion. Online, 2017.

[7] Martin Stich. Introduction to NVIDIA RTX and DirectX Ray Tracing. Online, March 2018.

[8] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques - SIGGRAPH 86*. ACM Press, 1986.

[9] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *ACM SIGGRAPH Computer Graphics*, 20(4):269–278, aug 1986.

[10] Jae-Ho Nah, Jeong-Soo Park, Chanmin Park, Jin-Woo Kim, Yun-Hye Jung, Woo-Chan Park, and Tack-Don Han. T&i engine. In *Proceedings of the 2011 SIGGRAPH Asia Conference on - SA '11*. ACM Press, 2011.

[11] Microsoft. *DXR Functional Spec*.

[12] D3D12 Raytracing Fallback Layer. Online, October 2018.

[13] Khronos. glTF 2.0 Overview. Online, 2018.

[14] Alex Dunn. Tips and tricks: Ray tracing best practices. online, March 2019.