IMPLEMENTING FAST VECTORIZED SINGLE-CORE SPHERE TRACING

Cliff Hodel, Véronique Kaufmann, Altin Alickaj, Max Mathys

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

With the growing demand for fast high-resolution rendering of photo-realistic images, having high-performance ray-tracers at hand has become fundamental [1, 2]. Implementing fast ray tracing algorithms is challenging for various reasons. The most critical one being that every scene, which we wish to render, is unique and exhibits different computational characteristics. This paper presents a fast single-core implementation of the sphere tracing algorithm, a technique to render implicit surfaces [3]. Performance is increased through various optimizations, including the vectorization of distance-computations and vectorization of rays to be traced. We show that when combining these separate optimization steps, a remarkable speedup of up to 40x can be achieved compared to traditional implementations.

1. INTRODUCTION

Motivation. The growing demand for photo-realistic renderings of complex environments in recent years has led to a growing interest in high-performance ray tracers [1, 2]. To achieve that, it is crucial to find fast algorithms to compute ray-scene intersections.

The difficulty to come up with fast implementations is a result of the high dependency of the algorithm on the scene to be rendered. Depending on the position, number, size, and type of objects in the scene, rendering can require a large amount of computing power. Different object characteristics, such as reflecting surfaces, will lead to even more expensive computations.

There are various algorithms to perform ray tracing and the choice depends highly on the application and the internal representation of the scene. One way of representing objects is through implicit surfaces (see section 2). Sphere tracing [3] is an algorithm that performs fast rendering of scenes represented by such implicit surfaces.

Contribution. We propose a fast single-core implementation of the sphere tracing algorithm [3]. The sphere tracer supports specular and mirror reflections, anti-aliasing, multiple point light sources, and five different objects: boxes, spheres, planes, tori, and boxframes. We apply SIMD vectorization (using AVX2 instructions) in two different variants depending on the nature of the input scene. On one hand, we implement parallel computations of multiple objects in the scene. On the other hand, we implement the computation of eight pixels in parallel. We finally combine these two parts to get a fast sphere tracing algorithm that performs well for any input scene.

Related work. To vectorize ray-tracing, [4] proposes an algorithm using SIMD vectorization for the rendering of shapes represented using a polygonal mesh. The foundation for sphere tracing has been laid by Hart [3]. It is also applicable for use on GPUs, as is shown in [5].

There are multiple ways of rendering implicit surfaces apart from sphere tracing. Another approach is to transform the implicit shapes into polygon shapes [6], and then apply polygon rendering techniques.

2. BACKGROUND ON THE ALGORITHM

In this section, we first introduce the mathematical and algorithmic background of the sphere tracing algorithm [3] and then define a suitable cost measure to analyze our implementations.

Implicit surfaces. Sphere tracing is a method for rendering an image of a scene, where the objects in the scene are given by implicit surfaces. These implicit surfaces are defined by signed distance bounds. A function $f : \mathbb{R}^3 \to \mathbb{R}$ is a signed distance bound for an implicit surface A if and only if

$$f(x) \leq d(x, A) \quad \text{for all } x \in \mathbb{R}^3, \tag{1}$$

where $d(x, A)$ defines the geometric distance of $x \in \mathbb{R}^3$ to an implicit surface A in 3D space. If (1) holds with equality, then f is called a signed distance function.

Sphere tracing. In order to render a pixel of the final image, sphere tracing marches along the corresponding ray

$$r(t) = c + t \cdot v\,,$$

where c denotes the ray origin and v the normalized ray direction, until it intersects with one of the objects in the scene or we have reached the maximal distance D (in case the ray does not intersect with any object in the scene). At

the beginning, we have $t = 0$. At each step, we compute the distance bounds d_i to all objects o_i in the scene using the associated signed distance bounds f_i. If the smallest distance d_{min} is smaller than some threshold ϵ, then we have found our intersecting object and we can proceed with computing the shade of this pixel. Else, we travel along ray r for d_{min} units and repeat the procedure. The following pseudocode describes the algorithm in more detail:

```
t = 0
while t < D:
    d_min = ∞
    foreach object o_i:
        d_i = f_i(r(t))
        if d_i < d_min:
            d_min = d_i
        if d_min < ε:
            intersection found with object o_i
            return
    t = t + d_min
```

Listing 1. Sphere tracing pseudocode

Reflections. In addition to the basic sphere tracing functionality described in the previous paragraph, we also implement mirror and specular reflections. To compute the specular reflections, we use a simplified version of the Phong reflection model [7]. For each intersection point, the illumination is determined using the following formula:

$$I = \sum_{m \in lights} k_d \cdot \underbrace{N \cdot (-L_m)}_{diffuse} \cdot I_m + k_s \cdot \underbrace{V \cdot R_m}_{specular} \cdot I_m$$

with N being the surface normal at the intersection point and L_m the direction of the ray coming from the point light m, hitting the intersection point. I_m is the intensity of the respective point light. R_m is the reflected direction of the light-direction computed as

$$R_m = 2(N \cdot L_m) \cdot L_m - L_m$$

For the mirror reflections, we shoot secondary rays from the intersection point in the reflected direction R as computed for the Phong shading. These reflection rays are then traced recursively to determine the shading color of the intersection point. To avoid infinite recursive rays, we only trace secondary rays up to a depth of 3 and then cut them off.

Anti-aliasing. Anti-aliasing is a technique used to remove artifacts resulting from the pixel-granularity of the image. It is used to smooth edges and get the illusion of a higher resolution image without increasing image size. There are various approaches to anti-aliasing. We choose a rather simple yet effective method, where we shoot four rays for each pixel and then average over the four rays to determine the final pixel color.

Cost analysis. We use pixels/cycle as a cost measure for our optimizations. The reason for this choice of cost measure is that throughout our optimizations the flop count of our implementations changes significantly. Comparing different implementations through flops/cycle is therefore not feasible. The number of pixels stays constant between different implementations (for fixed image size), which is why using pixels/cycle as a cost measure is much more appropriate.

3. FAST VECTORIZED SPHERE TRACING

This section presents the optimizations we performed on the sphere tracing algorithm. The baseline algorithm is presented in Section 3.1. In Section 3.2 we start with relatively small optimizations, such as precomputing expensive rotation matrices and introducing appropriate compiler flags. Then, we introduce two big improvements using SIMD vectorization. In particular, we show that it is possible to either vectorize over objects (*object vectorization*) or over rays (*ray vectorization*). These methods are described in Section 3.3 and 3.4. Further we also propose methods to combine these two vectorization approaches (*mixed vectorization*) in Section 3.5. Finally, we provide functionality to decide the most appropriate ray tracing method for a particular scene (Section 3.6).

3.1. Baseline Algorithm

As our starting point we use the sphere tracing implementation proposed in the tutorial [8]. This is a non-optimized implementation of the algorithm, and it does in particular not use any vector instructions. Since this implementation was written in C++, the first step consisted of writing the code in C. This C implementation was then extended by multiple shapes and all the other additional features described in Section 1 (reflections, anti-aliasing) and will be used as the reference throughout the rest of the paper. Listing 2 presents a high-level overview of our implementation, explaining the structure of our code.

```
spheretrace() {
    find_intersection()
    shade() {
        shadow_spheretrace() // pixel illuminated?
        // calculate shade of pixel
    }
    spheretrace() // calculate reflections
}
```

Listing 2. An overview explaining our code and functions: `spheretrace()` gets called for each pixel. First, we need to find the object with which the corresponding ray intersects (is done as shown in Listing 1). Afterwards, `shade()` determines the coloring of this pixel. `shadow_spheretrace()`, which is functionally very similar to `find_intersection()`,

determines whether this pixel lies in the shadow or not. In the end, we call `spheretrace()` recursively to shoot the secondary rays for the mirror reflections (we allow a depth of up to 3 recursive calls for one ray).

Anti-aliasing is implemented outside of this main loop: `spheretrace()` is called once for each of the four rays we will average over. This means that for each pixel in the image, `spheretrace` actually gets called four times.

Sphere tracer characteristics. A sphere tracer is usually very compute-bound. Running our sphere-tracer on a i7-8650U CPU with L1i/d cache size 128 KiB, L2 cache 1 MiB, and L3 cache 8 MiB, all scenes fit into the L1 and L2 caches. For a single object in the scene, we store its position, color, reflection parameters, and object-specific characteristics like orientation or sizes. A single object takes up to 84 bytes of memory. Even for very large scenes, everything except the output image usually fits inside the three caches. Since the pixels of the output image are written sequentially and exactly once, the image size will not interfere with cache accesses noticeably. In fact, by profiling with Vtune [9], we can see that there are usually no L3 misses at all. Since I/O accesses are usually no performance bottlenecks, we exploit the very parallel nature of sphere tracing via ILP to get performance gains. We also perform various precomputation optimizations to reduce work.

3.2. Preliminary Improvements

First, we describe the improvements of the overall algorithm before applying vector instructions.

Precompute rotation matrix. Shapes that allow for rotation (such as boxes, tori, or boxframes) suffer from the expensive computations of these rotations. In particular, multiple computations of $\sin$ and $\cos$ make these operations slow. These rotation matrices had to be re-computed for each distance computation (i.e. line 5 of Listing 1), wasting cycles. The first optimization is therefore to precompute these rotation matrices once at the beginning, such that a rotation consists of only one matrix-vector multiplication.

Compiler optimizations. The next step is to choose appropriate compiler options. This includes two things in particular: first, we use link-time optimization (`-flto` flag), since our implementation is distributed over multiple files. Among other optimizations, this enables more efficient inlining of functions. Then, we also add architecture-specific compiling for Skylake (`-march=skylake` flag).

3.3. Object Vectorization

A large portion of the computation performed in the sphere tracing algorithm consists of computing the distances between a point on the ray and the objects. In every step of the algorithm, we have to compute the distance to all the objects in the scene in order to find the closest one and with it the size of the next step on the ray (as shown in Listing 1). If a scene contains many objects, these computations become quite time-consuming. We, therefore, propose to compute multiple distance functions at the same time. This is obtained using vectorization through SIMD instructions, namely Intel's AVX2 instruction set.

Scene configuration. First we rearrange the way a specific scene and its objects are stored. Most of the operations are vector-matrix multiplications or vector-scalar multiplications, which is why we focus the scene representation on component-wise operations on coordinates of vectors or points (x, y, z). In particular, this means that we switch from storing arrays of coordinates to storing a separate array for each of the three coordinates. Since all of our vectors contain floats and we use AVX2, we can perform operations on eight coordinates (i.e. eight objects) simultaneously.

Distance computations. At each point on the ray we now compute the distances to up to eight objects of the same kind in parallel, instead of performing all these computations sequentially. Additionally, we implemented masking of objects if there are fewer than eight left. If we have five or more objects of the same kind, we still process them as vectors. If there are fewer than five objects of the same kind left, we process them sequentially. The cutoff value of five was chosen experimentally.

Drawbacks. Object vectorization allows for efficient parallelization in scenes that contain many objects of the same type. If the scene does not contain enough objects, this method is not able to exploit parallelism and therefore cannot achieve any speedups.

3.4. Ray Vectorization

To overcome the drawbacks mentioned in Section 3.3, we introduce a different way to make use of vectorization. Instead of vectorizing the computations along one ray, we shoot rays for eight pixels simultaneously and follow the traces of these rays in parallel.

Distance computations. At each step of the algorithm, the points following each of the eight rays will be at different distances $r_i(t_i) = c_i + t_i \cdot v_i$. From each of these points, we then compute the closest object in the scene to decide the size of the next step for that particular ray. In other words, when with object vectorization we compute the distances from one point on one ray to eight different objects, we now compute the distances from eight points on eight different rays to one object.

Shade function. Another advantage of ray vectorization is that we can vectorize a larger part of the computation. While in object vectorization, shading had to be performed sequentially for each pixel, ray vectorization allows vectorizing this part of the code as well.

Drawbacks. Ray vectorization works well, independently of the number of objects in the scene. Due to a more extensive ILP throughout the whole render process, it generally achieves much better performance than object vectorization. However, there are also drawbacks. Since all of the eight rays we compute in parallel might take different paths, some of the rays may already be finished (either by intersecting with an object or by reaching the maximal trace distance) and have to wait for the other rays to terminate as well. This happens especially around the edges of a surface and gets even more severe with reflections involved. A visual example of this can be found in Figure 4.

This results in a severe loss of performance, in particular for small image sizes. The reason for this is that with a lower number of pixels, the probability of eight rays taking very different paths is much higher than in a large image, where usually all eight rays intersect at similar positions.

Depending on the scene, low utilization leads to ray vectorization being even worse than object vectorization.

3.5. Mixed Vectorization

To alleviate the problem of low vector utilization, we can combine the strengths of ray and object vectorization in an operation mode we call *mixed vectorization*. In this computation, we start following eight rays in parallel as for standard ray vectorization, but continuously monitor the vector utilization of the current eight rays. Whenever we experience performance loss due to low vector utilization, we finish computing the remaining rays sequentially using object vectorization. The utilization signaling the point where using object vectorization is superior has been chosen experimentally to be at most 25%, i.e. at most two of the eight rays in the vector have to be active to consider switching to object vectorization to be beneficial. The ray utilization for different scenes can be seen in Table 1.

Drawbacks. Using mixed vectorization, we noticeably improve performance for most scenes. However, the drawbacks are now the same as for object vectorization. Since switching to object vectorization only makes sense when there are enough objects in the scene to make vectorization possible, mixed vectorization will not be better than standard ray vectorization in all the other cases. Note also that switching vectorization techniques usually involves a small additional overhead.

3.6. Deciding the Optimal Render Mode

As we have seen in the previous sections, all vectorization approaches have their advantages and disadvantages. Therefore we now present several ways of combining different approaches to get the benefits of all of them.

For this purpose, we came up with different decision functions that decide which render mode to use based on the given input scene using heuristics. We propose different methods for making these decisions.

Micro-image. The first approach consists of rendering a low-resolution version of the image with each approach and measuring their runtime. Finally, render the high-resolution image with the vectorization strategy that performed best on the micro-image. However, this is problematic as we saw that ray vectorization performs better for large images than for small ones. Further, it also requires computational power to compute those micro-images, which might impose a sizeable overhead. We have run an experiment with the micro-image approach and either exclusively used object vectorization or plain ray vectorization to render the images. For small images (640x480), this decision function achieved 100% accuracy for our 13 test scenes. But as already mentioned, for bigger image sizes, this accuracy will drop significantly.

Static analysis: object vs ray. Next, we have a static decision function, which incorporates information such as the number and types of objects or number of reflections into its decision. Using a single run through the scene configuration, we decide whether to use object vectorization or ray vectorization. Our experimental results show that object vectorization is better if there are many objects inside a scene and if there are also many reflections to be done (as is the case for the *massive* scene (d) for example). We define a decision function to choose object vectorization if there are many objects (i.e. at least 16 objects of the same kind) and many reflections (i.e. at least 90% of objects are reflecting). Otherwise, render using ray vectorization. This method worked 90% of the time for our test scenes in Figure 1 for images of size 640x480. However, accuracy drops for larger image sizes.

Static analysis: mixed vs ray. Now we present a decision function, whose job is to decide between plain ray vectorization and mixed vectorization. Remember that mixed vectorization is good whenever object vectorization is. This is the case whenever there are many objects in the scene. Also note that whenever object vectorization is good, mixed vectorization is usually significantly better than ray vectorization. By choosing to render the image with mixed vectorization whenever there are enough objects in the scene, the decision function chooses the right method most of the time. More concretely, except for *planes scene* (see Figure 1), which contains 9 planes in total, the decision function always chooses the faster version. The heuristic of choosing between mixed and ray vectorization is very accurate and promising since for most of our test scenes one of these two render modes achieved the best speedup. The accuracy can also be explained by noting that for small images, mixed vectorization fixes the problem of having low vector utilization and that mixed vectorization converges to ray vectorization for image sizes going to infinity, since uti-

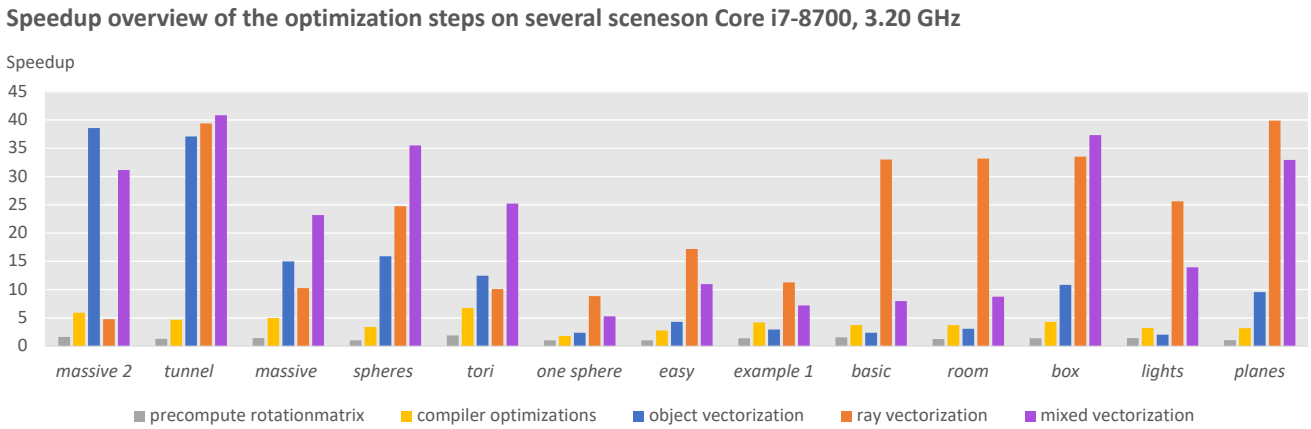


Fig. 1. Speedup plot showing the relative speedup of the optimization steps described in Section 3. The speedup reports the relative speedup as compared to the baseline implementation (Section 3.1) for image size 160x120. It can be observed that object vectorization and ray vectorization complement each other depending on the scene. In particular, object vectorization achieved better performance when there are many objects in the scene. This is the case in all scenes except *one sphere, easy, example 1, basic, room, lights*. Further, the plot shows how mixed vectorization attempts to fill the gap between the two or even outperform both. Visual representations of all scenes can be found in Appendix I.

lization increases for larger images. It is a fast method with very low overhead. However, as for every heuristic, there are many examples where they fail.

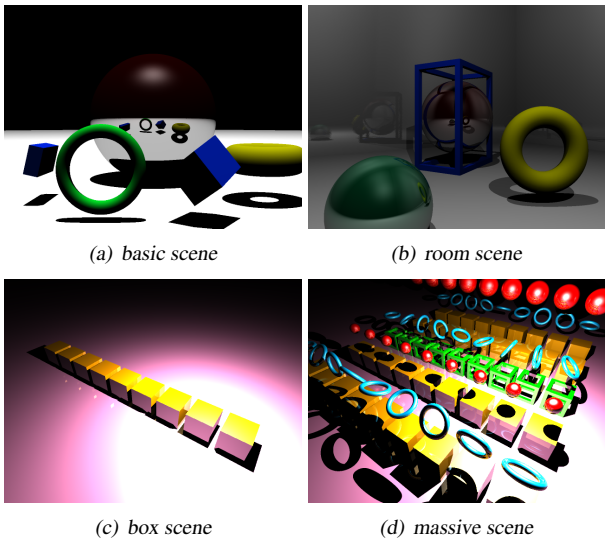
4. EXPERIMENTAL RESULTS

Experimental setup. All experiments were performed on an Intel i7-8700 (Coffee Lake) CPU with Skylake microarchitecture. The base frequency is 3.20 GHz, with Turbo Boost disabled. The cache sizes for L1i/d, L2, and L3 are 64KiB, 256KiB, and 12MiB. For compiling, we used GCC 10.3.0 with the following flags (except for the reference implementation, since the flags already represent our first optimization step): `-O3 -lm -flto -march=skylake` .

To report performance of our implementations we use four example scenes which we call *basic*, *room*, *box* and *massive*. They can be found in Figure 2. We chose scenes with different characteristics and rendering complexity to get representative results. The scenes are ordered from (a) - (d) with ascending complexity. Each scene contains at least one plane. Additionally:

- The *basic* scene contains one sphere, two tori and two boxes.
- The *room* scene contains two spheres, one boxframe, one torus and two additional planes acting as walls.
- The *box* scene contains nine boxes.
- The *massive* scene contains a large number of objects with at least nine of every shape (box, torus, sphere, boxframe).

The aspect ratio of the rendered image is fixed to 4:3. The resolution is parameterized by an integer N and results in a resolution of $4N \times 3N$ pixels. The runtime is computed for each scene and we consider $10 \leq N \leq 400$ with a step-size of 10. This results in a resolution range from 40x30 up to 1600x1200 pixels. As discussed in Section 2, the performance metric used is pixels/cycle.



(a) basic scene *(b) room scene*

(c) box scene *(d) massive scene*

Fig. 2. Scenes that were used to report performance

Results. The five optimizations as described in Section 3 (precomputed rotation matrix, compiler optimizations, object vectorization, ray vectorization, mixed vectorization)

will be analyzed. The reported speedup for each optimization is always computed relative to the reference implementation described in Section 3.1.

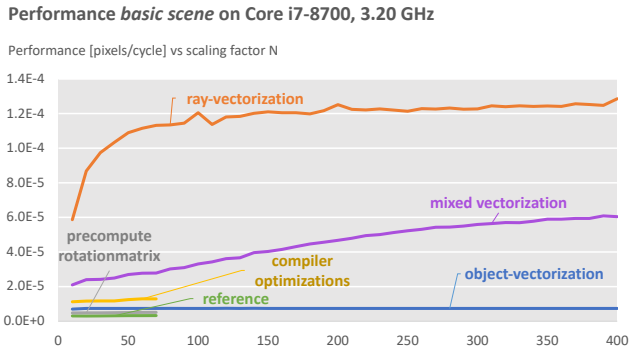
All additional features such as reflections, rotations, or shadows were enabled. Over all tested scenes, the final version reached a speedup of **8x - 40x**. Figure 1 provides a general overview of the measured speedups over all our test scenes. The final version reached up to **31%** of the processor's peak flop performance. Compared to the maximum utilization of the reference implementation (0.6%), this is an improvement of more than 51x. Note that the code versions *reference, compiler optimizations* and *precompute rotation matrix* could only be benchmarked up to $N = 70$ due to exploding runtimes. Their performance is however constant for $10 \leq N \leq 70$ and we highly expect them to stay constant for larger $N > 70$.

Precompute rotation matrix. This optimization reduced work inside distance functions by precomputing rotation matrices from Euler angles which could then be used for faster rotation calculations. A speedup between 1.4 and 1.5 can be observed. The speedup is relatively consistent for all tested scenes.

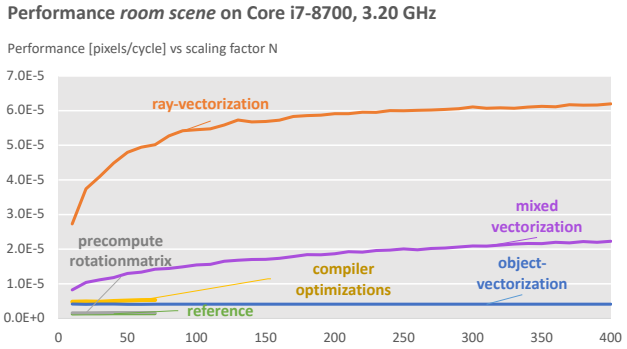
Compiler optimizations. Recall that this optimization enabled link-time optimizations, allowing the compiler to inline a lot of our geometry-related functions needed in distance function calculations. In addition, we compiled for Skylake, allowing the compiler to perform further optimizations. In total, a speedup between 3.6 and 4.3 was achieved.

Object vectorization. As mentioned, object vectorization is able to exploit parallelism for scenes where there are a lot of objects of the same kind. This is the case in the (c) *box* and (d) *massive* scene. Since the box scene has nine boxes, the distance to eight of nine boxes can be computed fully in parallel, reaching a speedup of 10.6. In the massive scene, there are also many possibilities to parallelize; a speedup of 15.2 can be observed. Object vectorization is not able to exploit parallelism for scenes with a low number of objects. This is the case in the (a) *basic* and (b) *room* scene. For both scenes, object vectorization is even slower than the previous optimization, which is due to increased overhead and the inability to exploit ILP. Other than ray and mixed vectorization, the performance is constant for all image sizes. It is determined by the scene configuration and mostly independent of the image size. This makes sense since we only follow one ray at a time and objects in larger images are scaled proportionally.

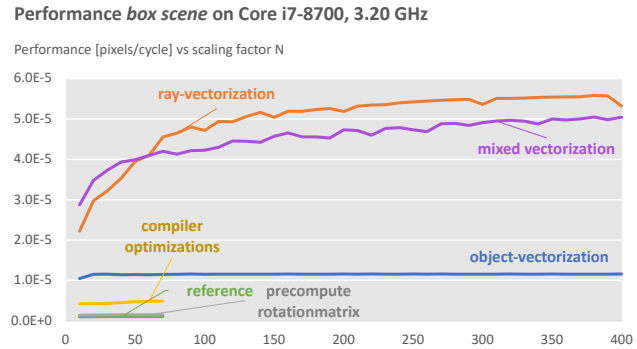
Ray vectorization. Ray vectorization allows utilizing parallelism efficiently even for scenes with a small number of objects. This can be seen in scenes (a), (b), (c), where ray vectorization is much better than any previous optimization. Speedups of 19.3, 21.1, and 22.6 can be observed. For scenes with high complexity (i.e. many reflecting surfaces and/or a lot of objects), ray vectorization suffers from



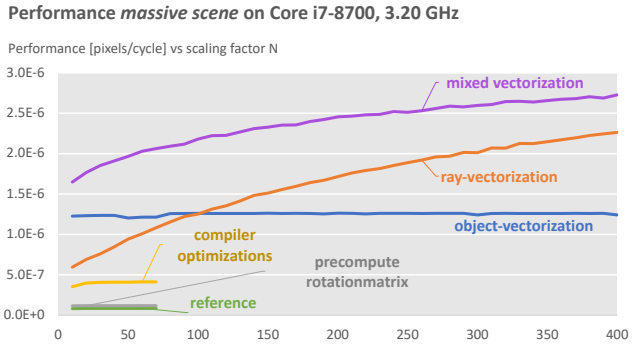
(a) basic scene



(b) room scene



(c) box scene



(d) massive scene

Fig. 3. Performance plots of chosen scenes

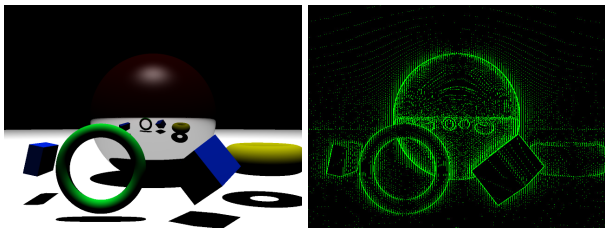
	(a)	(b)	(c)	(d)
1	3.60%	5.65%	0.93%	15.48%
2	1.63%	2.88%	0.69%	6.93%
3	1.23%	2.19%	0.62%	4.76%
4	1.01%	2.00%	0.60%	3.79%
5	0.95%	2.20%	0.58%	3.19%
6	0.94%	1.31%	0.58%	2.89%
7	0.90%	1.40%	0.62%	2.97%
8	89.76%	82.36%	95.39%	59.98%

Table 1. Ray utilization: time spent for scenes (a) - (d) and number of concurrently processed rays (1 - 8). The columns denote the four scenes (a) - (d). The rows denote the number of rays which are processed concurrently (1 - 8). The entries denote the fraction of sphere tracing events for a given scene and number of concurrently processed rays. For example, scene (d) (massive) ends up performing useful computations on only one ray in 15.48% and on two rays in 6.93% of the cases.

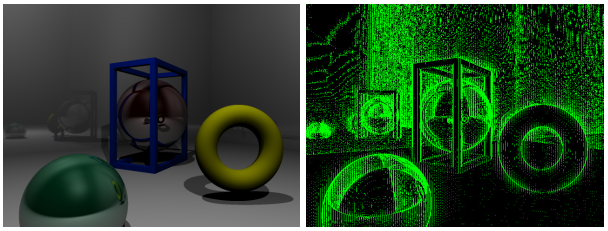
low utilization. This is the case for scene (d), where only a speedup of 7.3 is achieved. Table 1 denotes the ray utilization for scenes (a)-(d). Ray utilization is measured by the fraction of time spent where 1-8 rays are utilized. Figure 4 shows instances where the ray utilization was low for each scene. If at one point the ray utilization was $\leq 25\%$, the pixel is painted green. It can be observed that in scene (d) there are many occurrences where the ray utilization is low. This is because of the high number of objects in the scene, where all of them have a strictly positive reflection parameter.

The performance is better for images with a higher resolution because ray utilization is better. In Figure 3 it can be seen for all scenes (regardless of scene complexity) that the performance improves for higher resolutions. This can be explained by the fact that for images with high resolution, rays calculated in a fixed-size batch of 8 behave more "similarly". The rays shot in a batch are closer to each other in images with larger resolution than rays shot in an image with lower resolution. In return, their intersection points are closer to each other. This leads to better ray utilization because rays don't stall each other as much.

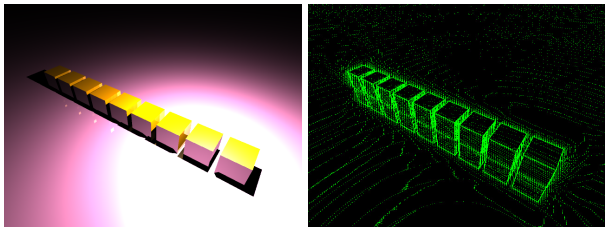
In Figure 5, this effect is demonstrated. The *spheres* scene is rendered in two different resolutions. The left image is rendered with a resolution of 640x480. The right image is rendered with a resolution of 12800x9600. We show a zoomed-in part of the rendered images. For both images, areas with low utilization are marked as green pixels. It can be seen that even though both images render the same scene, there are many more instances of low ray utilization in the image with the smaller resolution (left). Low utilization usually occurs near the edges of an object. A part of the rays within a batch intersects with the object. Another part



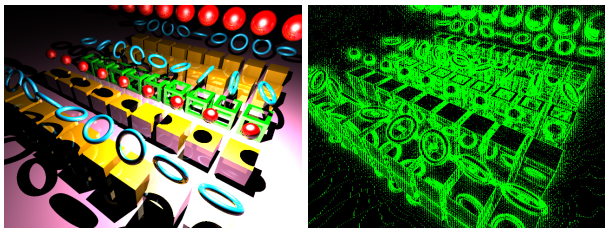
(a) basic scene



(b) room scene

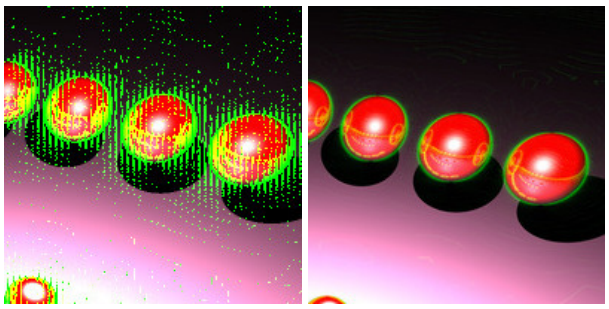


(c) box scene



(d) massive scene

Fig. 4. Areas where low vector lanes utilization ($\leq 25\%$) is detected; left: original, right: low vector lanes utilization.



(a) 640x480 *(b) 12800x9600*

Fig. 5. Zoomed-in part of *spheres* scene rendered with two different resolutions, green pixels indicate low ray utilization ($\leq 25\%$).

travels further until it intersects with the underlying plane or the maximum distance is reached. This difference in ray length leads to low utilization. Shorter rays take less time to compute than longer rays.

In the left image, vertical artifacts can be seen near the edges of the spheres. Rays are calculated in batches of eight pixels in horizontal order. If a green pixel can be seen, it means that for the ray computed for this pixel, there was low utilization. These vertical artifacts occur because the rays at one side of the batch finish earlier. And then the ray on the other side of the batch will see low utilization.

The effect that images with smaller resolutions suffer from low ray utilization is especially important for the (d) massive scene. This makes the performance of ray vectorization for the massive scene worse than object vectorization for small N. Ray utilization only improves for larger images (where $N > 100$) to a point where ray vectorization is able to outperform object vectorization.

Mixed vectorization. Mixed vectorization generally uses ray vectorization and switches to object vectorization when low ray utilization is detected. The performance plots (Figure 3) show that mixed vectorization is worse than ray vectorization if there are too few objects in the scene. This can be seen both in *basic* scene and *room* scene. However, with an increasing number of objects in a scene, mixed vectorization achieves significant performance gains. For scenes like *massive* scene, mixed vectorization is therefore always better than ray vectorization. Note that in *box* scene, mixed vectorization starts better than ray vectorization but is overtaken by it with increasing N. In general, it's reasonable that mixed and ray vectorization converge to the same performance when N goes to infinity. With large N ray utilization goes towards 100%, thus mixed vectorization converges to standard ray vectorization.

5. CONCLUSION AND FUTURE WORK

We have implemented a highly optimized single-core implementation of the sphere tracing algorithm. Our starting point was a straightforward C implementation, which used a mere 0.6% of the machine's peak flop performance. By marching along eight rays in parallel using AVX2 instructions, and switching to object vectorization under low ray utilization (if the scene allows for it) our approach achieved a speedup of 28x on average over all our test scenes. The maximal speedup observed for a single scene was a striking 40x and the maximal achieved performance was 31% of the machine's peak flop performance. Essentially, *ray vectorization* is the superior approach compared to *object vectorization*, since ray vectorization is able to parallelize the work regardless of the input scene and also allows us to parallelize larger portions of the code. But object vectorization still finds its place in the mixed approach. By statically choosing either ray vectorization or the mixed approach (depending on the input scene) we presented an image renderer, which works well on a wide range of scenes, making it universally applicable.

As a next step, it would be interesting to expand the implementation to multi-core. Rendering an image is an embarrassingly parallel workload since each pixel is independent of all the others. We hope that we then can render images in real-time, which should be possible for the smaller scenes and/or lower resolutions.

6. CONTRIBUTIONS OF TEAM MEMBERS

Cliff. Implemented the distance functions of the additional shapes for the basic C implementation and the rotation matrix pre-computation. Together with Véronique, vectorized the geometry functions. Vectorized the distance function of the boxframe shape and also implemented the masking for object vectorization. Implemented the shading and shadows for ray vectorization in tight collaboration with Max. Implemented the decision function (ray vs object vectorization) based on micro-images.

Véronique. Implemented reflections (mirror and specular) and anti-aliasing through supersampling for the baseline algorithm. Implemented vectorized geometry functions (vector operations etc) in collaboration with Cliff. Object-vectorized distance functions for a part of the shapes. Implemented the rendering and spheretracing for ray vectorization including reflections and improving performance of vectorized reflections in close collaboration with Altin.

Altin. Designed the basic C implementation. Restructured scene representations in memory to be usable in object and ray vectorization. Worked on the object vectorized spheretrace workflow and some of the vectorized distance functions. Implemented the rendering and spheretracing for ray vectorization in close collaboration with Véronique. Did compiler optimizations and designed static decision functions together with Max.

Max. Worked on object vectorization: implemented a part of the vectorized distance functions as well as general functionality. Worked on ray vectorization: implemented a part of the vectorized distance functions, vectorized shading, vectorized shadow tracing with Cliff as well as general functionality. Worked on decision functions with Altin.

7. REFERENCES

[1] Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, Brenton Rayner, Jonathan Brouillat, and Max Liani, "Renderman: An advanced path-tracing ar-

chitecture for movie rendering," *ACM Trans. Graph.*, vol. 37, no. 3, Aug. 2018.

[2] Mark Lee, Brian Green, Feng Xie, and Eric Tabellion, "Vectorized production path tracing," in *Proceedings of High Performance Graphics*, New York, NY, USA, 2017, HPG '17, Association for Computing Machinery.

[3] John C. Hart, "Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces," *The Visual Computer*, vol. 12, no. 10, pp. 527–545, 1996.

[4] José Noguera, Carlos Ureña, and Rubén García Hernandez, "A Vectorized Traversal Algorithm for Ray Tracing.," 01 2009, pp. 58–63.

[5] G. Liktor, "Ray Tracing Implicit Surfaces on the GPU," *Computer Graphics and Geometry*, vol. 10, no. 3, pp. 36–53, 2008.

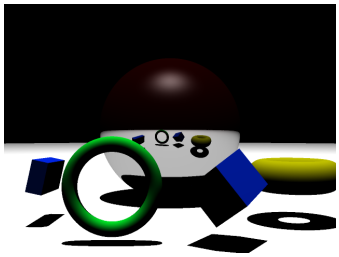
[6] Jules Bloomenthal, "Polygonization of Implicit Surfaces," *Computer Aided Geometric Design*, vol. 5, no. 4, pp. 341–355, 1988.

[7] Bui Tuong Phong, "Illumination for computer generated pictures," vol. 18, no. 6, 1975.

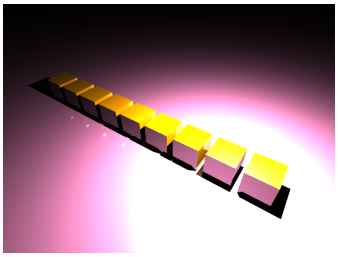
[8] Scratchapixel, "Rendering Implicit Surfaces and Distance Fields: Sphere Tracing," *https://www.scratchapixel.com/lessons/advanced-rendering/rendering-distance-fields*.

[9] Intel, "Intel Vtune Profiler," *https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html#gs.41o2ot*.

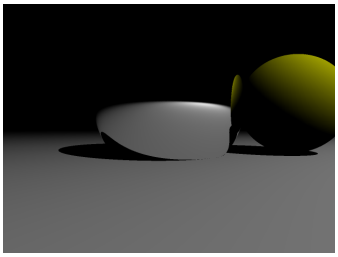
Appendix I



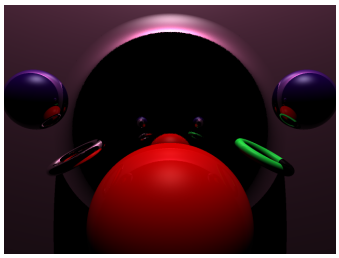
(a) basic



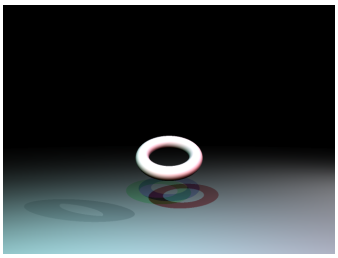
(b) boxes



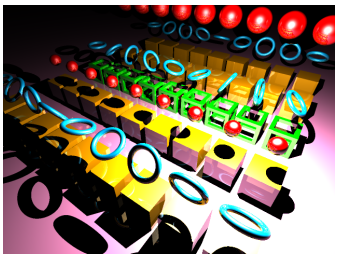
(c) easy



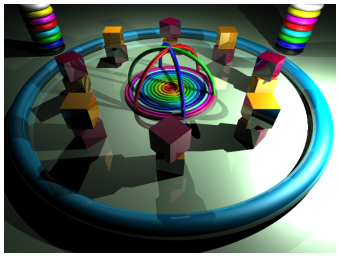
(d) example 1



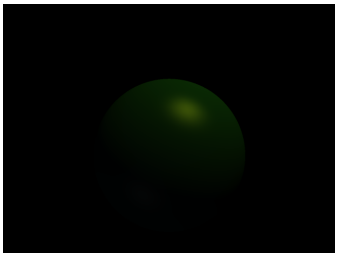
(e) lights



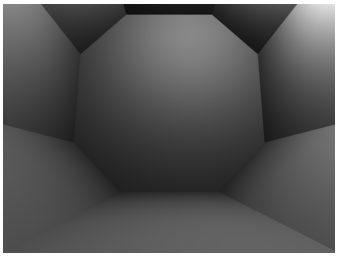
(f) massive



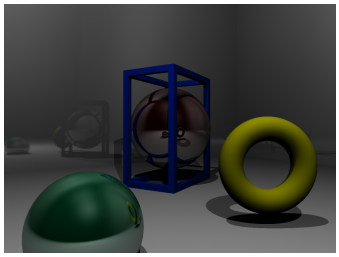
(g) massive 2



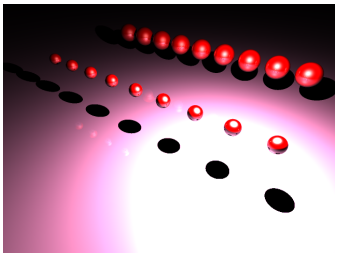
(h) one sphere



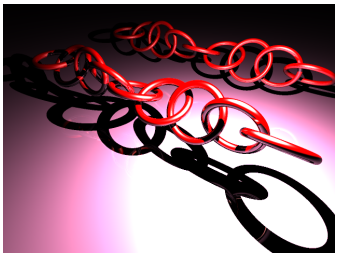
(i) planes



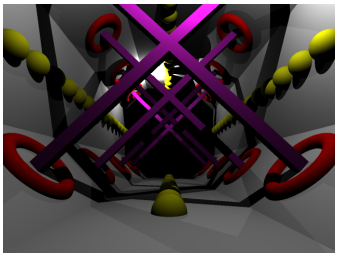
(j) room



(k) spheres



(l) tori



(m) tunnel

Fig. 6. Visual Representation of Scenes (Alphabetic order)