KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

First Cycle, 15 credits

A Ray Tracing Implementation Performance Comparison between the CPU and the GPU

ROBIN NORDMARK

TIM OLSÉN

Stockholm, Sweden 2022

A Ray Tracing Implementation Performance Comparison between the CPU and the GPU

ROBIN NORDMARK
TIM OLSÉN

Degree Project in Computer Science and Engineering
Date: June 8, 2022
Supervisor: Stefano Markidis
Examiner: Pawel Herman
School of Electrical Engineering and Computer Science
Swedish title: En prestandajämförelse av ray tracing-algoritmen implementerad på både CPU och GPU

Abstract

Ray tracing has gained recent popularity due to the advancement of computer hardware capabilities. The algorithm is used as a rendering technique for computer graphics by tracing rays of light to determine the color of a single pixel, thus simulating the physical behavior of light. This study explores the performance differences between the ray tracing algorithm on the CPU and the GPU processor units. By using CUDA, NVIDIA's platform for parallel programming, general-purpose programming could be utilized on the GPU and C++ for the CPU counterpart. By rendering different numbers of spheres in varying resolutions, the performance difference could be measured on the two devices and put against each other. From the data gathered, we could conclude that the GPU, in most measurements, could finish its execution up to 1000–10000 times quicker than the CPU. However, there were instances, in lower resolutions, where the CPU would outperform the GPU. The performance on the GPU would in these lower resolutions be more unpredictable due to memory latency. The results of this study highlight the performance capabilities of the GPU, but also certain use cases on the CPU for lower pixel counts.

Sammanfattning

Ray tracing har på senare tid ökat i popularitet på grund av utvecklingen av datorhårdvara. Algoritmen används som en renderingsteknik för datorgrafik genom att spåra ljusstrålar för att korrekt färga enskilda pixlar, och på så sätt simulera hur ljus rör sig i verkligheten. Denna studie undersöker prestandaskillnaderna för ray tracing-algoritmen på CPU- och GPU-processorenheterna. Genom att nyttja CUDA, NVIDIA:s plattform för parallellprogrammering, kunde programmering nyttjas på GPU:n, och C++ på CPU:n. Genom att rendera en uppsättning sfärer i varierande upplösningar kunde prestandaskillnader mätas på de två enheterna och därefter jämföras. Från den insamlade datan kunde slutsatsen dras att GPU:n, i de flesta mätningarna, avslutade sin exekvering upp till 1000–10000 gånger snabbare än CPU:n. Det förekom dock fall, vid lägre upplösningar, där CPU:n kunde prestera bättre än GPU:n på grund av latens i minneshantering. Resultaten av denna studie understryker potentialen hos GPU:n, men också vissa specifika användningsområden för CPU:n.

Contents

Chapter 1

Introduction

During the past couple of decades, computer graphics have become an integral part of everyday life in all parts of society. Important scientific work needs to be visualized, simulations are using graphics to imitate real-world scenarios, and computer-generated imagery (CGI) is used in multiple multi-million-dollar industries. In such a wide field as computer graphics, it is bound to arise numerous different techniques in pursuit of an even more true-to-life render. One such technique is Ray Tracing. Ray Tracing was first conceptualized during the 16th century, but it took nearly 400 years for the first implementations to arise. The algorithm involves tracing rays throughout a scene, bouncing them off of object surfaces towards sources of light, and calculating color values accordingly. The first implementations during the 1960s included various 3D visual renders where Ray Tracing was used to calculate realistic reflections of light [1]. Even though the interest in the technique fell off during these times as computers of this era were too slow and other computationally cheaper techniques were just as good, the interest in Ray Tracing saw a resurgence a couple of decades later [2]. Today Ray Tracing is predominantly used for 3D visualization, such as in animated movies, but new uses also include real-time graphics rendering in computer games, as the performance in computer hardware continues to improve.

General-purpose computing on graphics processing units (GPGPU) is a powerful alternative to the more common approach of programming using the Central Processing Unit (CPU). Ray Tracing implementations can make use of either one of these techniques, and both are used for different reasons in practice. Real-time Ray Tracing is made possible by GPGPU as calculations can be made in parallel, while animation studios often use clusters of CPUs

for rendering ray traced scenes.

1.1 Research question

The purpose of this thesis study is to investigate the performance of two implementations of the Ray Tracing algorithm as our research question states:

How does a CPU versus a GPU implementation of Ray Tracing differ in performance?

This problem will be explored and attempted to be answered by implementing the Ray Tracing algorithm first on the CPU using C++, and then port that implementation into the CUDA framework to utilize the GPU. The performance will be benchmarked by rendering 3D images and the elapsed time measured.

It is very much anticipated that the GPU should gain an advantage over the CPU in such a comparison due to calculations being made in parallel. The interesting part, however, is exploring the scale on which both performances differ and what these differences might imply. This is done in hope of providing a deeper understanding of how both hardware devices perform for the Ray Tracing algorithm.

1.2 Scope

Since using Ray Tracing in real-time is computationally heavy and therefore taxing on the limited hardware we have available we will not be examining animations of any kind. We will instead render 3D still images and compare the performance of each implementation based on render times. This study will only focus on NVIDIA GPU devices as CUDA is utilized for the GPU implementation. Other GPUs would require different frameworks, such as OpenCL.

Chapter 2

Background

2.1 Terminology

Pixel: A single building block of a digital picture.
Scene: A composition of objects.
A Render: A digitally produced picture.
Ray: A vector representing light.
Space complexity: The amount of memory that is needed to calculate and complete a problem.
Time complexity: The amount of time it takes for a function to complete a problem.
Thread: The smallest type of instruction that the operating system can schedule.

2.2 Ray Tracing

The Ray Tracing algorithm use case in the field of Computer Graphics is to recreate true-to-life renderings of various scenes from the real world. To describe the process of Ray Tracing one can use the pinhole camera model as seen in Figure 2.1. A light-proof box with a flat piece of photographic film inside has a pinhole cut out which is covered. When uncovering the pinhole rays of light can pierce the film and a chemical reaction occurs. This process is equivalent to how a simple camera works. A modified model, as seen in Figure 2.2, which suits computer graphics better fully disregards the box and replaces the pinhole with a camera and the film with an image plane or screen.

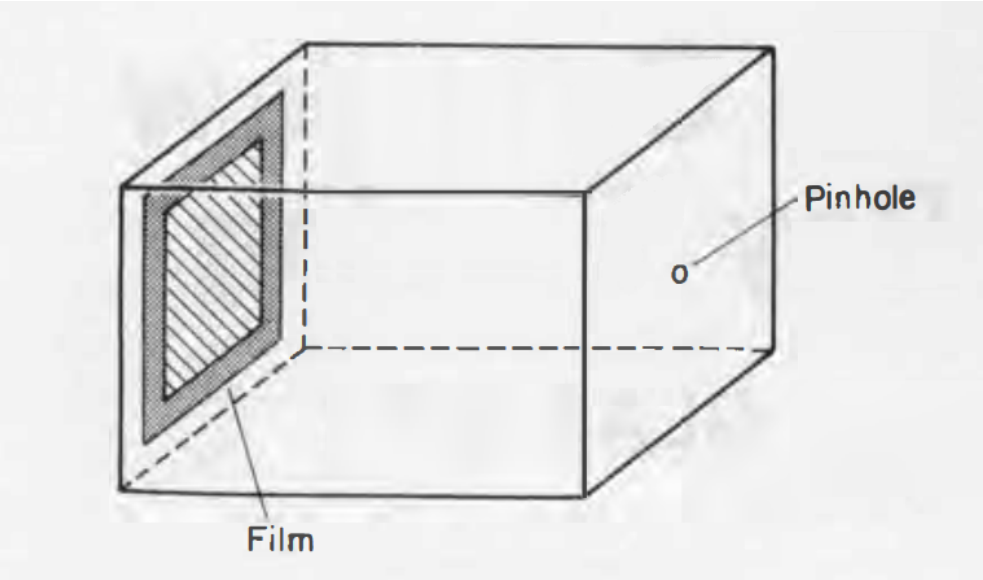


Figure 2.1: Pinhole camera model [**2**].

A fundamental question to be answered in computer graphics is; if a single color is to represent a pixel, what color is the correct one to choose? There are various algorithmic approaches to answering this question. The Ray Tracing algorithm sets out to compute the color of each pixel in a scene according to how a light source bounces rays within this scene against every object. Each object can have different properties as to what material these are supposed to represent. These properties will decide if a ray is wholly absorbed, reflected, or a combination of both. A single pixel is then given a color according to how the light has been reflected throughout the scene. The idea this describes is called Forward Ray Tracing. With the modified pinhole model in mind a ray from the light source is followed within the scene, bouncing on various objects, passing through the image plane, and finally arriving at our eye. Only rays that hit the eye will contribute to the color of the pixel that is found on the image plane. A problem with this approach is that it is computationally heavy. This is because most rays that we follow throughout the scene never hit the image plane. Another more practical solution is to use Backward Ray Tracing which is the same as the previously described forward algorithm but in reverse. By starting at the eye, we can follow only the essential rays, now called shadow rays, through the image plane, the scene, and to a light source [2].

The simplicity of Ray Tracing makes the technique quite elegant. As the al-

gorithm describes how light moves throughout a scene, akin to how photons move in real life, it is an excellent tool for calculating reflections and shadows in a render. These parts come naturally to the algorithm, though the technique can be expanded upon to further increase its photorealism with some additional shading techniques. One of these will be introduced in subsection 2.2.2.

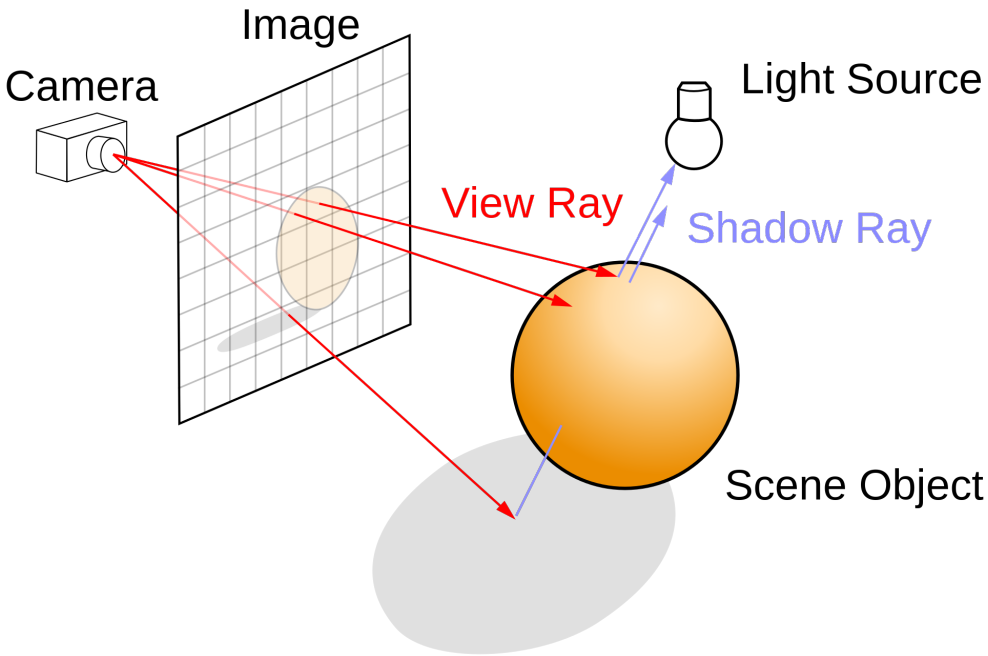


Figure 2.2: An established model for describing Ray Tracing.
Credit goes to Wikipedia user Henrik.

2.2.1 Pseudocode

The Ray Tracing algorithm relies heavily on linear algebra, and therefore there are some vector arithmetics in the pseudocode shown in Algorithm 1. The algorithm iterates over each pixel, creating a ray vector that starts from a fixed camera position and points in the direction of the current pixel. The ray will then be followed throughout the scene, taking notice of any intersections with objects, and finally, return a color depending on how and where the ray intersected each object within the scene.

Algorithm 1 Ray Tracing

```
1: procedure RUN( )
2:     for x ← 0 to width do
3:         for y ← 0 to height do
4:             ray ← norm((x, y, 0) − CAMERA)
5:             print TRACE_RAY(ray)
6:         end for
7:     end for
8: end procedure

9: procedure TRACE_RAY(ray)
10:    for spheres in scene do
11:        hits ← intersects_sphere(ray)
12:    end for
13:    return calculated_color(min(hits))
14: end procedure
```

2.2.2 Blinn-Phong

The Blinn-Phong reflection model delivers a method of creating ambient light, diffuse light, and specular highlights to an object. The model is based on Phong shading with some slight adjustments to how lightning is calculated [3]. Blinn-Phong is often to be preferred as it creates a subjectively more realistic shading texture. By adding such a shading model to a ray tracer an improvement in realism is achieved.

2.3 CPU

A central processing unit (CPU) is the primary unit used to fetch and execute instructions on a computer. The instructions are generally fetched from the main memory and passed to the arithmetic/logic unit (ALU) which is the electronic circuitry responsible for executing logical and arithmetic operations. The communication done within the different units of the CPU is organized with a control unit (CU) that uses electrical signals to direct the different parts of the system [4]. Thus the CPU architecture design philosophy enables a diverse set of workloads to be executed and lends itself especially well to performing serial computing tasks.

Modern CPUs use several processing cores to increase performance. These cores each have their own ALUs and CUs which means that they can execute instructions simultaneously and independently of each other. However, these cores can communicate with each other through the use of a shared address and memory space using different hierarchical structures of the cache system [4]. While the operating system can use the multi-core architecture to optimize performance, it is oftentimes up to the software developer to utilize parallel programming to further increase the performance of individual software [5].

2.4 GPU

The graphics processing unit (GPU) is a highly parallel processor making it suitable for computationally heavy programs. While it started as a means to help accelerate 3D graphics, it has over time expanded its capabilities and thus found itself in the center of a wide range of areas in computing. The most prevalent of these include high-performance computing (HPC) and deep learning [6].

The GPU is composed of hundreds of lightweight cores capable of handling a lot of hardware threads. These threads are designed primarily for floating-point computations rather than more advanced instruction-level parallelism that can be found on CPUs. As a consequence, the hardware threads on the GPU have a very high latency when fetching data from memory. The GPU makes up for this latency by rapidly switching between the available threads; therefore, it can be a challenging task to construct code that utilizes the hardware most optimally but it is essential to use enough threads to hide this latency [7].

2.5 CUDA

The Compute Unified Device Architecture (CUDA) is a platform that opens the possibility of general-purpose programming on NVIDIA GPU devices. CUDA allows for the graphics application programming interface (API) to be overlooked, which provides a layer of abstraction due to the ability to disregard the underlying graphical concepts [8].

The CUDA source code can be written in a general-purpose programming language, such as C++, extended with annotations to distinguish it from the stan-

dard C++ syntax. The source code can be compiled using the CUDA compiler (nvcc), which passes a CUDA binary to a C++ host compiler, which provides an executable that can be run on the CUDA runtime [9].
Because the code is executed on two physically different hardware devices, CUDA distinguishes them through the terms host and device, the first referring to the CPU and its memory and the latter to the GPU and its memory [9].

2.5.1 Thread blocks

CUDA uses the abstraction of a grid consisting of thread blocks to represent sets of threads that within the same block can cooperate with fast shared memory. Each block is run on a single multiprocessor within the GPU responsible for scheduling the threads as shown in Figure 2.3. Each block, along with every thread within these blocks, possesses a unique identifier available to the programmer through the use of built-in variables.

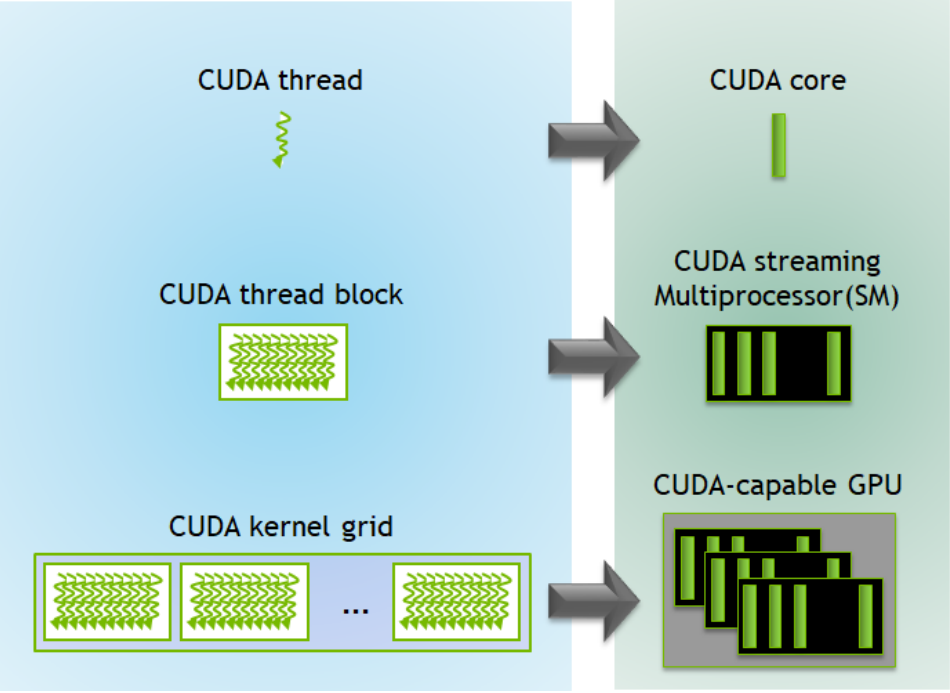


Figure 2.3: CUDA abstractions mapped to the GPU hardware [10].

2.5.2 Kernel

A CUDA program is initially run on the host. To start executing code on the device, a kernel call is invoked along with a grid of thread blocks. A kernel works as a normal C++ function, extended with annotations to decide how many CUDA threads should run the function in parallel - meaning that the function will be run once per thread. When the kernel call has been invoked, the host will continue to run any remaining code, which may cause the program to terminate prematurely. To prevent this, the function *cudaDeviceSynchronize* can be called to stall the host and wait for every thread on the device to finish its execution [9].

2.5.3 Automatic memory management

Memory allocation in CUDA works similarly to how it is done in C/C++. It is possible to allocate memory accessible both from the host and device, this memory address space is referred to as the Unified Memory [11]. The function *cudaMallocManaged* will allocate memory that is automatically managed by the Unified Memory system. The function *cudaFree* will free the allocated memory, regardless of where the memory is handled [9].

2.6 Profiling

Profiling tools are used to analyze the performance of a program. The information will either be presented as blocks of text or visualized within some graphical interface. The profiler will measure a wide variety of parts during execution, including the space and time complexity of the program, the amount of executed instructions and which functions are being called, and how much time elapses during their respective execution. With this knowledge about the execution of the program, ways to optimize its run-time can be explored more easily if needed. This can be achieved by finding bottlenecks in the execution of the program and restructuring accordingly thereafter. An example of profiling done during one of the renders can be seen in Figure A.1 in the Appendix.

2.7 Previous work

There is not an abundance of performance comparisons for GPU/CPU implementations related to Ray Tracing, but there are at least two bachelor thesis studies from Mälardalen University. The work presented by Norgren [12] sets

out to evaluate real-time Ray Tracing solutions on GPU/CPU, which is then later built upon by Liljeqvist [13] who attempts to perform a similar study, exploring performance drops from the aforementioned study using a hybrid GPU/CPU solution. Both of these studies evaluate a Ray Tracing solution in real-time, using several Ray Tracing engines and graphics libraries. Norgren concluded that Ray Tracing on the GPU almost always can be assumed to perform better than that of a CPU implementation. They also point out the fact that large companies such as Intel research CPU implementations for Ray Tracing, and the possibility of equal performance can be achieved, but the cost for this is much higher compared to that of GPU solutions. Liljeqvist concluded that it was infeasible to implement a hybrid (both utilizing the GPU and the CPU) solution which could outperform a pure solution.

This study seeks to evaluate pre-rendered scenes using Ray Tracing, meaning that no real-time evaluations are done. It is also worth mentioning that, unlike the previous work, no graphics engines or libraries are utilized – placing the implementation closer to the actual GPU/CPU hardware.
While this thesis is by many means different, the previous work can still act as a reference point when comparing benchmarks on the different hardware devices.

Chapter 3

Method

3.1 Renders

Every execution of the Ray Tracers would produce color values for each pixel. These values would later be saved in an external PPM (Portable Pixel Map) file. The PPM file could then be visualized in any optional image editor, such as GIMP.

3.2 Benchmarking

A variety of different scenes were constructed for the data collection phase. Each scene consisted of a number of spheres, ranging between one to nine, which were lit by a single light source bouncing reflections between each sphere. A sphere is a basic geometric figure and could easily be defined in the scene without any external graphics libraries, which suited both Ray Tracing implementations due to them being close to the hardware. An example of a render with four spheres can be seen in Figure 3.1. The complexity of the scene could be modified by changing the active amount of spheres being rendered, in other words, the complexity corresponds to how much should be rendered in a given scene. The decision to limit each scene to a fixed number of spheres was made so that if any trend revealed itself it would be easily spotted when later comparing the benchmarks. The number of spheres could have exceeded that of nine without any problem, but a set of max 3×3 spheres fit the render quite nicely, as each sphere was neither too small nor too large. The last parameter for varying each render was selected to be the resolution, as the pixel count of a render is important for both workload and details in the rendered picture. An aspect ratio of 1:1 was chosen for each render and the res-

olution ranged between 1×1 and 1000×1000 pixels for both the GPU and the CPU. The upper limit was dependent on the limitation of the CPU's capability to render larger pictures. The amount of work having to be executed during each render could then easily be changed using the resolution parameter.

When running the benchmarks, a scene from the dataset was chosen and then executed on the CPU implementation and then the GPU implementation. For both implementations, a C++ timer function using the standard library Chrono [14] was used to measure the elapsed time right before starting to cast the first ray until after all computations were done. The timer would only be started after all necessary preparations had been done, such as memory allocations in the case of the GPU implementation which is further mentioned in section 3.5. A total of 50 runs were executed for every single scene and resolution, thus providing data with a solid enough margin of error.

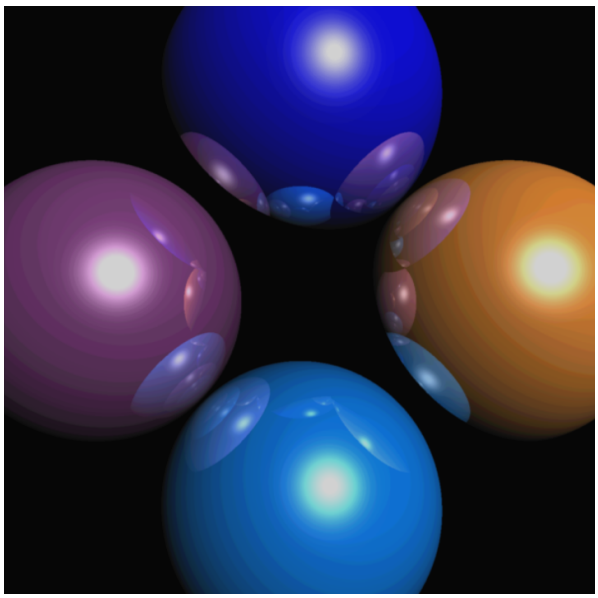


Figure 3.1: An example render made using the GPU implementation.

3.3 Data sets

The data sets were generated by writing the data from the benchmarks into CSV files. The variables represented were the number of spheres, resolution, and time in seconds. For each combination of spheres and resolution, there would exist 50 separate time entries. Since the benchmarks were done on two

different hardware devices, thus creating separate files, CSV files could later be merged to also possess an entry representing the hardware as shown in Table 3.1. Merging CSV files and visualizing the data sets in different types of graphs were done using a python script.

Spheres	Resolution	Time	Type
6	200x200	0.000171	GPU
6	200x200	0.000112	GPU
...	...	...	...
6	200x200	0.084363	CPU
6	200x200	0.078270	CPU

Table 3.1: Example data set.

3.4 Implementations

The following subsection deals with the implementations of the Ray Tracing algorithm. The CPU code was written in C++, whereas the GPU counterpart was written in CUDA C++. Some inspiration was gathered from an implementation done in python, an implementation that is rather slow due to the high-level nature of the python programming language [15].
The entire project code for the implementation can be found in the GitHub repository associated with this thesis [16].

3.4.1 CPU Implementation

Both the CPU and GPU implementations were constructed with the attempt of consistency in mind, meaning that as little as possible had to be changed when moving on from the CPU to the parallelization on the GPU. Classes describing mathematical constructs such as three-dimensional vectors, spheres, and rays were defined in separate header files to have more clearly defined objects in the code.

The idea for the algorithm on the CPU consisted of tracing rays through each pixel of the image into the scene in a sequential manner. The main function contains the initial setup before running the Ray Tracing algorithm. This involves defining a light source vector, and the spheres to be rendered to the scene. The number of spheres and their placements was predetermined in a *makeScene* function.

The *run* function defines the camera vector and invokes the *trace_ray* function for every pixel in the scene, much like the pseudocode presented in subsection 2.2.1. To take care of reflection, a ray is traced through the scene a finite amount of times, defined as five times as shown in Listing 1. This means that light that keeps reflecting infinitely between surfaces will be caught.

```
while(depth < 5) {
    Ray OD = Ray(rayO, rayD);
    Traced traced = trace_ray(OD);
    if(traced.m_col_ray.x() == -1 &&
       traced.m_col_ray.y() == -1 &&
       traced.m_col_ray.z() == -1) {
        break;
    }
    Sphere object = traced.m_sphere;
    Vec M = traced.m_M;
    Vec N = traced.m_N;
    Vec col_ray = traced.m_col_ray;
    rayO = M + 0.0001 * N;
    rayD = norm(rayD - 2 * dot(rayD, N) * N);
    col += reflection * col_ray;
    reflection *= traced.m_sphere.ref();
    ++depth;
}
```

Listing 1: Calculation of reflection and color.

A crucial part of the code is the *trace_ray* function. Following the structure from the pseudocode, every object in the scene is inspected for intersection with the ray to determine whether the ray will bounce off an object as shown in Listing 2. In the case of no intersection, infinity is returned resulting in plain black color, representing nothingness. Lastly, the color is calculated with Blinn-Phong shading as described in subsection 2.2.2.

```
Traced trace_ray(Ray & r) {
    float t = INFINITY;
    float t_object;
    int object_i = 0;
    int i = 0;
    for(Sphere object : scene) {
        t_object = intersect_sphere(object, r);
        if(t_object < t) {
            t = t_object;
            object_i = i;
        }
        ++i;
    }
    if(t == INFINITY) {
        return Traced();
    }
    // Intersect with other objects
    // along with shading left out
}
```

Listing 2: The *trace_ray* function.

3.4.2 GPU Implementation

The main differences when moving to the implementation on the GPU were parallelization and memory management. Since rendering on the CPU was done sequentially, every pixel could be written to a file directly. However, due to the unpredictability of the parallelism, results needed to be stored in device memory to later be written to a file. Initial setup before invoking the run kernel consists of allocating memory on the device as well as defining the number of threads per block as described in subsection 2.5.1.
Memory allocation on the device was done with *cudaMallocManaged*, as shown in Listing 3, where the size allocated is determined by N multiplied by the size of *Vec*, where N represents the total amount of pixels.

```
Vec * res;
int N = HEIGHT * WIDTH;
cudaMallocManaged(&res, N*sizeof(Vec));
```

Listing 3: Allocation of memory on the device.

Using *dim3* variables, defined in the CUDA standard library, the block sizes along with the number of threads could be declared. The threads are defined as two-dimensional 8×8 grids. The blocks are also two-dimensional and depend on the resolution, meaning that the number of blocks is defined as $\frac{width}{8} \times \frac{width}{8}$ grids, each added by one to make up for non-integer quotients as shown in Listing 4.

```
dim3 blocks(c_WIDTH/blocks_x+1, c_HEIGHT/blocks_y+1);
dim3 threads(blocks_x, blocks_y);
```

Listing 4: Declaration of thread and block partitions.

A kernel call is invoked to the run function, as depicted in Listing 5, which leads the GPU to execute the *run* function in parallel. The *run* function has the same key functionality as described in subsection 3.4.1. Parameters passed are mainly used to access essential data when moving from the host to the device.

```
run<<<blocks, threads>>>(
    res, scene, LIGHT, number_of_spheres, x
);
```

Listing 5: Kernel launch for initializing computations on the GPU.

3.5 NVIDIA Visual Profiler

The NVIDIA visual profiler is a CUDA-compatible profiler with a graphical interface. By running the compiled executable file of the GPU implementation through the visual profile some observations could be made. Regardless

of the resolution being rendered, a majority of the total time was spent allocating memory as the GPU needed to cooperate with the CPU as explained in subsection 2.5.3. This can be seen in Figure A.1 found in the Appendix. It is first when the *cudaMallocManaged* function has completed its setup that the kernel run is launched.

3.6 Verification

Verifying the correctness of the implementation is an integral part when ensuring that correct results are provided. One of the most essential verifications in this study was observing the rendered images. Even though unoptimized code may lead to slower than possible performance, the rendered images provide the groundwork for ensuring an at least correct implementation. Having consistency between both implementations is an attempt to assert that performance differences are on the hardware, rather than optimization issues on the implementation.

3.7 Hardware limitation

The amount of benchmarking performed in this study was limited by the hardware that was available at the time. Using an almost seven-year-old CPU (released 2015) may have had an impact on the max resolution which could be rendered in a reasonable amount of time compared to using hardware released more recently. As the pixel count grew in size, each benchmark would have taken entire days to complete.

Listed below is the hardware used during the benchmarking. Please observe the generous amount of CUDA cores the GPU has access to compared to the CPUs counterpart in cores.

CPU: Intel(R) Core(TM) i7-6700K CPU @ 4.00 GHz, 4 cores, 8 logical processors.
GPU: NVIDIA GeForce GTX 1070, 1607 MHz, 8GB GDDR5 (256-bit), 1920 CUDA Cores.

Chapter 4

Results

This section presents several graphs produced from the data collected through the previously mentioned benchmarking. Subsections are divided into resolution and scene complexity to depict time with respect to these different variables.

4.1 Resolution complexity

The following subsection presents data, represented as graphs, with execution time in relation to varied resolutions, thus making the complexity of the scene itself constant. The scene that was rendered consisted of one sphere, which can be viewed in Figure B.1 in the Appendix. The image was the same regardless of which hardware it was rendered on.

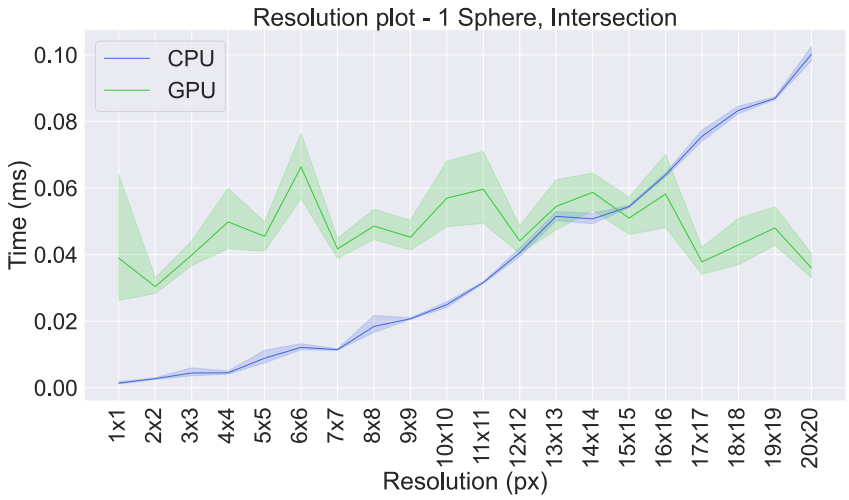


Figure 4.1: Intersection of the CPU/GPU render times.

The line graph shown in Figure 4.1 depicts time (in milliseconds) along with the resolution for one rendered sphere on the different hardware. The resolution ranged from 1×1 to 20×20 pixels, with an intersection between 14×14 and 15×15 pixels. Each resolution was rendered 50 times and the thickened lines represent the average of these 50 runs. On average, the execution time for the CPU was faster than that of the GPU for ranges of 1×1 to ca. 15×15 pixels. All 50 runs show somewhat the same execution time for the CPU. In contrast, the 50 runs for each resolution on the GPU show a far wider spread, thus having a more unpredictable execution time.

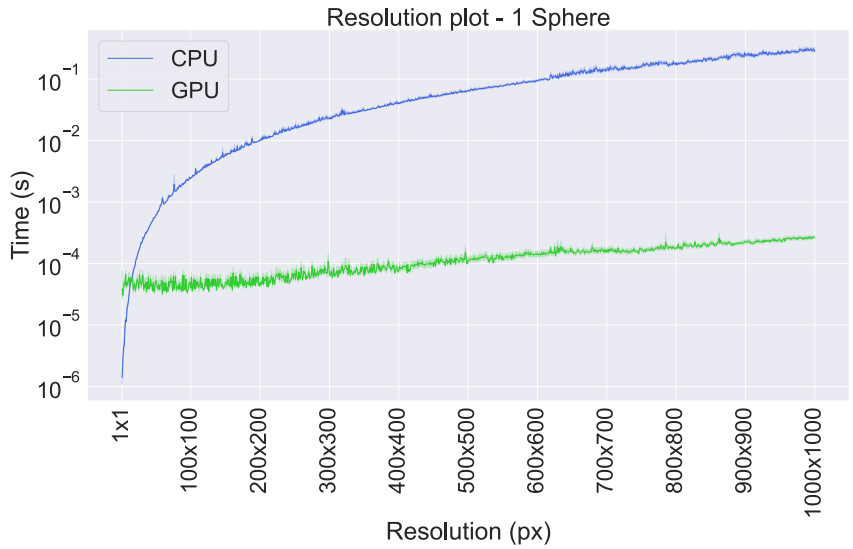
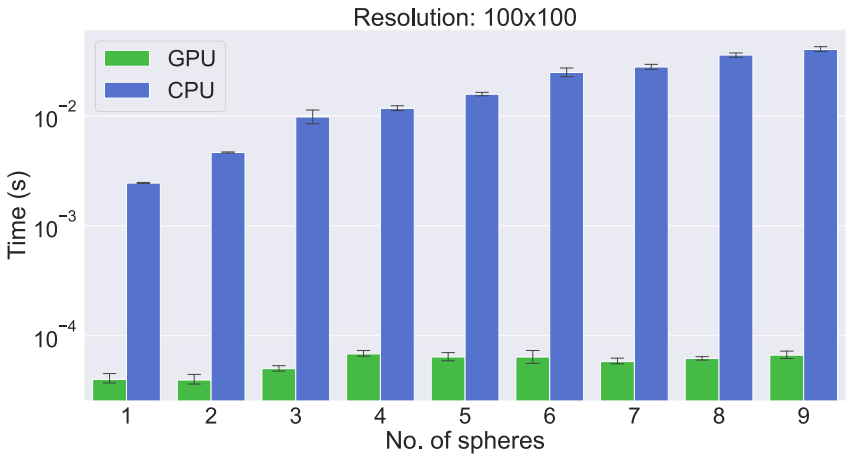


Figure 4.2: A wide range of resolutions.

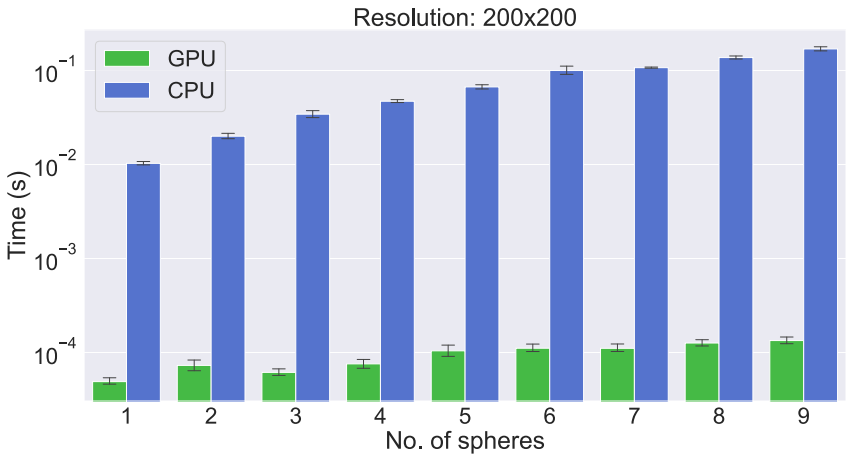
The graph represents the same line graph as Figure 4.1, although altered with a logarithmic time axis, in seconds, with resolutions ranging from 1×1 to 1000×1000 pixels. As stated before, the execution time for the GPU shows a far wider spread initially, which proceeds to stabilize as resolution grows higher, thus having a more predictable execution time. The CPU execution time grows very quickly as the resolution gets higher. Although the execution time for the CPU grows more quickly than the GPU, it continues to have a more predictable execution time than the GPU, however with a few spikes, up until around 600×600 pixels.

4.2 Scene complexity

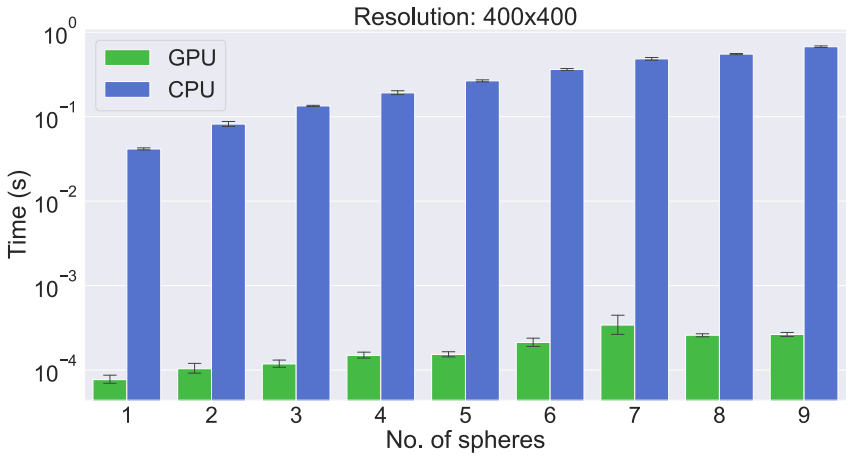
The data presented in this subsection depicts execution time along with the number of spheres being rendered, thus altering the complexity of the scene. This makes the resolution a constant parameter, where bar graphs are shown representing the different resolutions. The different scenes rendered can be found in Appendix B.



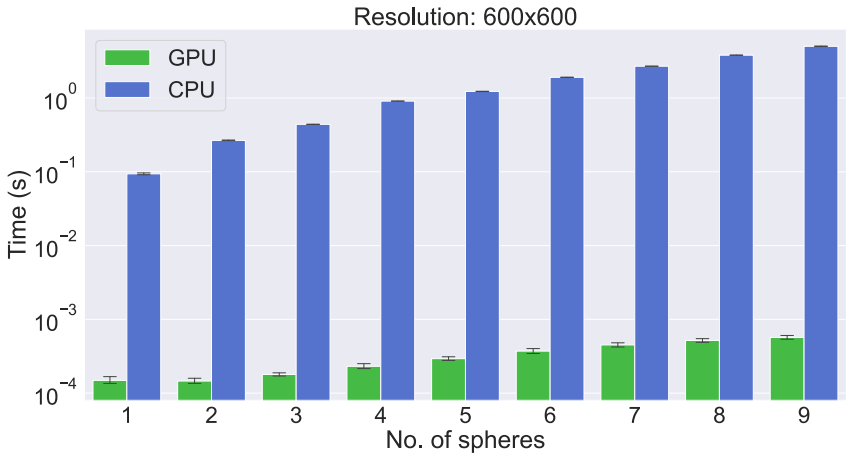
(a) Execution time for 100×100 px resolution.



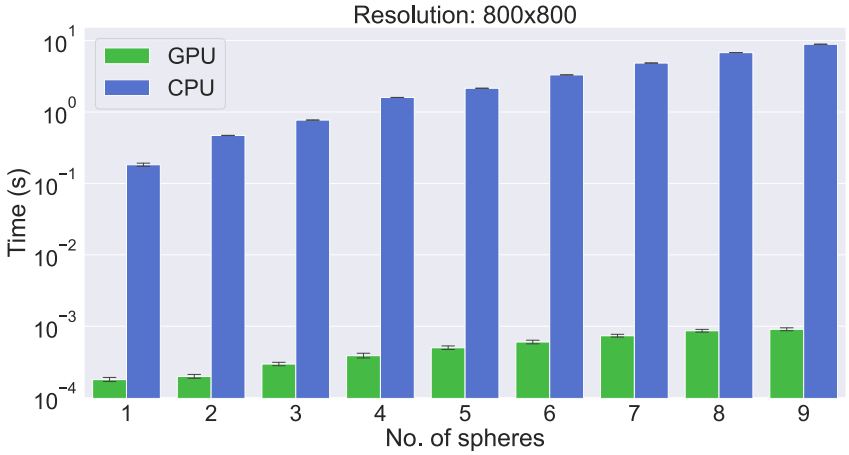
(b) Execution time for 200×200 px resolution.



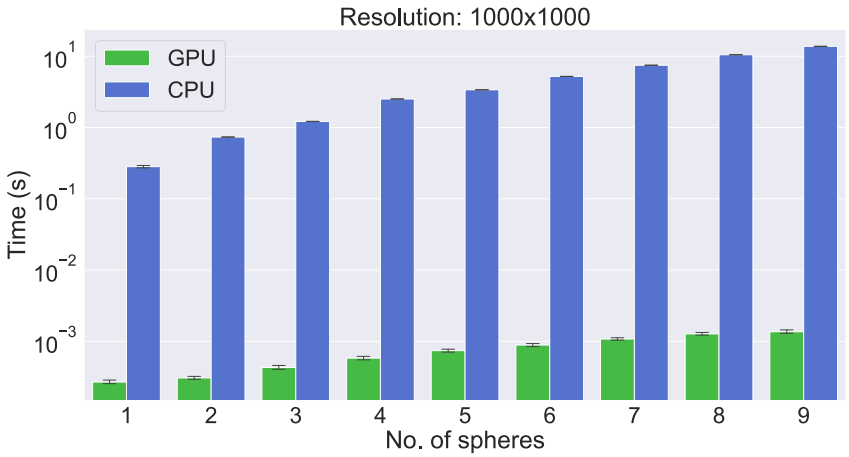
(c) Execution time for 400×400 px resolution.



(d) Execution time for 600×600 px resolution.



(e) Execution time for 800×800 px resolution.



(f) Execution time for 1000×1000 px resolution.

Figure 4.3: Bar graphs depicting execution times for different numbers of resolutions.

The bar graphs show average run-times for rendering spheres, ranging from one to nine spheres, in varying resolutions. Each scene was rendered 50 times. The color of the bars represents the different hardware, where shorter bars mean less execution time, thus better performance. The line on top of each bar is an error bar which represents the variation of the execution time. The greatest time differences for the CPU are observed when there are less total spheres present in the scene, and the more spheres that are added to the scene the less the difference becomes which is seen as the bars level out. The CPU bars are strictly growing throughout each resolution, as oppose to the GPU bars which vary in total time execution up until a resolution of 600×600 when the GPU also begin to follow an increasing trend.

Chapter 5

Discussion

The results from the benchmarking mostly confirm what was expected of this comparison going into the study. Compared to the GPU, the CPU lacked the advantage of parallelization as no attempt was made to utilize more than one core during the renders, making it an unfair comparison. However, what is interesting to evaluate is the magnitude of which the outperformance differed and also how the two hardware handled smaller resolutions when rendering.

When rendering resolutions ranging between 1×1 to 20×20 pixels, as seen in Figure 4.1, we can observe a clear intersection of the two lines around 14×14 and 15×15 pixels. The CPU overtakes the GPU in the amount of time to render at that point and continues to grow thereafter. Before this, the CPU has lower total rendering times than the GPU, and the variance is far less than that of the GPU. When also taking Figure 4.2 into account we can see that the GPUs time variance will eventually pan out, but not until after the 400×400 pixel count. This is also clearly seen if comparing Figure 4.3a, Figure 4.3b, Figure 4.3c and Figure 4.3d, Figure 4.3e, Figure 4.3f where the former GPU plots do not follow any specific pattern whereas in the latter a strictly growing pattern can be seen. As explained in section 2.4, there is a very high latency for GPU threads when fetching data from memory. Due to the undersized pixel counts in the initial resolutions rendered, the amount of threads utilized by the GPU is far less than optimal, making GPU rendering times worse than when using the CPU due to unexpected memory latencies.

The scale on which the GPU outperformed the CPU can be observed in Figure 4.3. A clear trend is seen where the time difference is around 1000–10000 times faster for each of the renders compared. A remark about how the Chrono

timer is set up when benchmarking the GPU should be made here. The GPU, compared to the CPU, has a much more intricate setup which is not included when timing a render, as stated in section 3.5. If this setup would be taken into consideration when comparing performances the GPU implementation would increase a bit in total execution time, depending on how much memory is allocated. This is not the fault of the algorithm, but a side effect of the parallelization on the GPU, which is worth mentioning in such a comparison context.

When observing the CPU bars in Figure 4.3 the fastest-growing times, bars standing beside each other with the greatest difference, seem to be those with fewer total spheres in the scene, around 1–3. This should be the case, as a single sphere does not have to reflect itself between any other objects. But as soon as one or more spheres are introduced into the scene the total number of reflections drastically increases and thus also the execution time. The more the scene is filled with spheres the less the time grows, which may indicate that the jump in computations is not as great when the scene gets cramped. Though, none of this is clearly seen in the GPU bars as the time still varies more than the CPU equivalent.

The findings of this study confirms some of the previous research conclusions, as written about in section 2.7. In almost all scenarios of benchmarking the GPU outperformed the CPU, as concluded by Norgren. This was to be expected, but as Norgren also mentions there is potential in a CPU implementation of the Ray Tracing algorithm, as companies such as Intel spend both time and effort to research this. If optimizations were explored for the CPU implementation the potential for a higher breaking point between the two implementations could exist.

5.1 Relevance

In the field of game development, there has been a resurgence of pixel art graphics in the last couple of years, which has been especially prevalent for indie developers. These smaller teams, typically composed of employees in the single digits, often have a much tighter economic budget when developing new titles [17], than that of bigger established studios. A case could be made for using a CPU for rendering smaller pixel counts, such as sprite work in games. The performance difference in these insignificant pixel counts is not huge by any means, but the alternative is available.

5.2 Future work

The results gained from this research could be extended from different perspectives. The CPU implementation was done on a multi-core processor, however, programmed sequentially. Potential further study suggests exploring parallelization on the CPU in multi-core, or even manycore, architectures. Since there is an initial breaking point for lower pixels where the CPU gains an advantage, the parallelization on the CPU could potentially extend that breaking point. This could result in significant resolutions with relevant use cases.

The work presented by Norgren [12] suggests working with implementations closer to the hardware, rather than already existing graphics engines. While our study does just that, it is not done on a real-time ray tracer. Therefore, evaluating differences in real-time in CUDA/C++ is an interesting point of view for further research.

Chapter 6

Conclusion

This study set out to explore performance differences in a CPU and a GPU implementation of the Ray Tracing algorithm. The results show that the differences in performance are immense due to the highly parallelized GPU. However, the benefits of the massive number of cores on the GPU become irrelevant when lower resolutions lead to higher memory latency. This is where the CPU gains its advantage as memory latency is not an issue due to the much more advanced processor core found on the CPU.

Bibliography

[1] Jon Peddie. *Ray Tracing: A Tool for All*. Springer Nature Switzerland AG, 2019. ISBN: 9783030174897.

[2] Andrew S. Glassner. *An introduction to ray tracing*. London: Academic, 1989. ISBN: 0122861604.

[3] Bui Tuong Phong. "Illumination for Computer Generated Pictures". In: *Commun. ACM* 18.6 (June 1975), pp. 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839. URL: https://doi.org/10.1145/360825.360839.

[4] John L. Hennessy and David A. Patterson. *Computer architecture : a quantitative approach*. 5th ed. Waltham, MA: Morgan Kaufmann, 2011. ISBN: 9780123838728.

[5] Thomas. Rauber and Gudula. Rünger. *Parallel Programming For Multicore and Cluster Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN: 9783642048180.

[6] Intel. *What is a GPU?* 2022. URL: https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html (visited on 01/06/2022).

[7] André R. Brodtkorb, Trond R. Hagen and Martin L. Sætra. "Graphics processing unit (GPU) programming strategies and trends in GPU computing". In: *Journal of Parallel and Distributed Computing* 73.1 (2013). Metaheuristics on GPUs, pp. 4–13. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2012.04.003. URL: https://www.sciencedirect.com/science/article/pii/S0743731512000998.

[8] Jason. Sanders and Edward. Kandrot. *CUDA by example : an introduction to general-purpose GPU programming*. Boston, Mass.: Addison-Wesley, 2010. ISBN: 9780131387683.

[9] Nvidia. *CUDA Toolkit Documentation v11.7.0*. 2022. URL: https://docs.nvidia.com/cuda/index.html (visited on 01/06/2022).

[10] Pradeep Gupta. *CUDA Refresher: The CUDA Programming Model*. 2020. URL: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/ (visited on 01/06/2022).

[11] Steven Chien, Ivy Peng and Stefano Markidis. "Performance Evaluation of Advanced Features in CUDA Unified Memory". In: *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 2019, pp. 50–57. DOI: 10.1109/MCHPC49590.2019.00014.

[12] Daniel Norgren. "Implementing and Evaluating CPU/GPU Real-Time Ray Tracing Solutions". Mälardalen University, 2016.

[13] Erik Liljeqvist. "Evaluating a CPU/GPU Implementation for Real-Time Ray Tracing". Mäldardalen University, 2017.

[14] Cppreference.com. *Date and time utilities*. 2022. URL: https://en.cppreference.com/w/cpp/chrono (visited on 01/06/2022).

[15] Cyrille Rossant. *Very simple ray tracing engine*. https://gist.github.com/rossant/6046463. 2017.

[16] Robin Nordmark and Tim Olsén. *CPU/GPU Performance Analysis of Ray Tracing*. https://github.com/skvarre/raytracing. 2022.

[17] Guo Freeman et al. ""Pro-Amateur"-Driven Technological Innovation: Participation and Challenges in Indie Game Development". In: *Proc. ACM Hum.-Comput. Interact.* 4.GROUP (Jan. 2020). DOI: 10.1145/3375184. URL: https://doi.org/10.1145/3375184.

Appendix A

Profiling

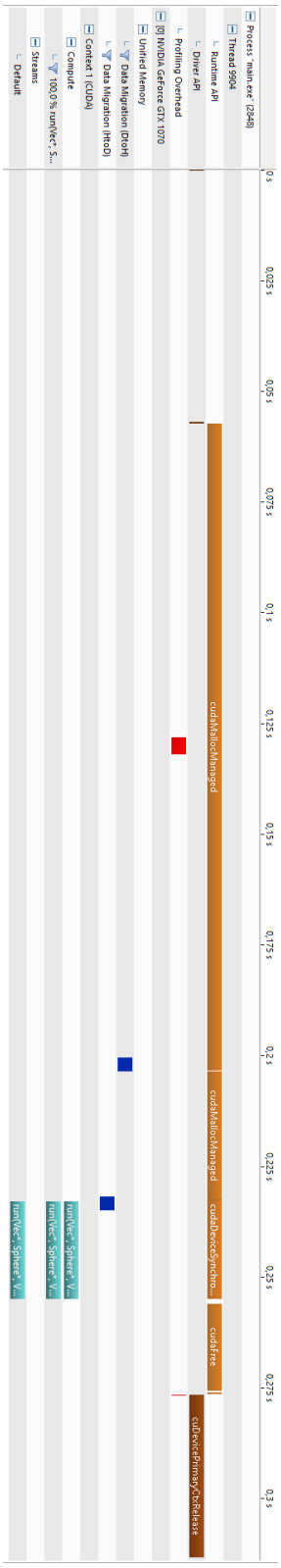


Figure A.1: Snapshot from visual profiler for the GPU.

Appendix B

Renders

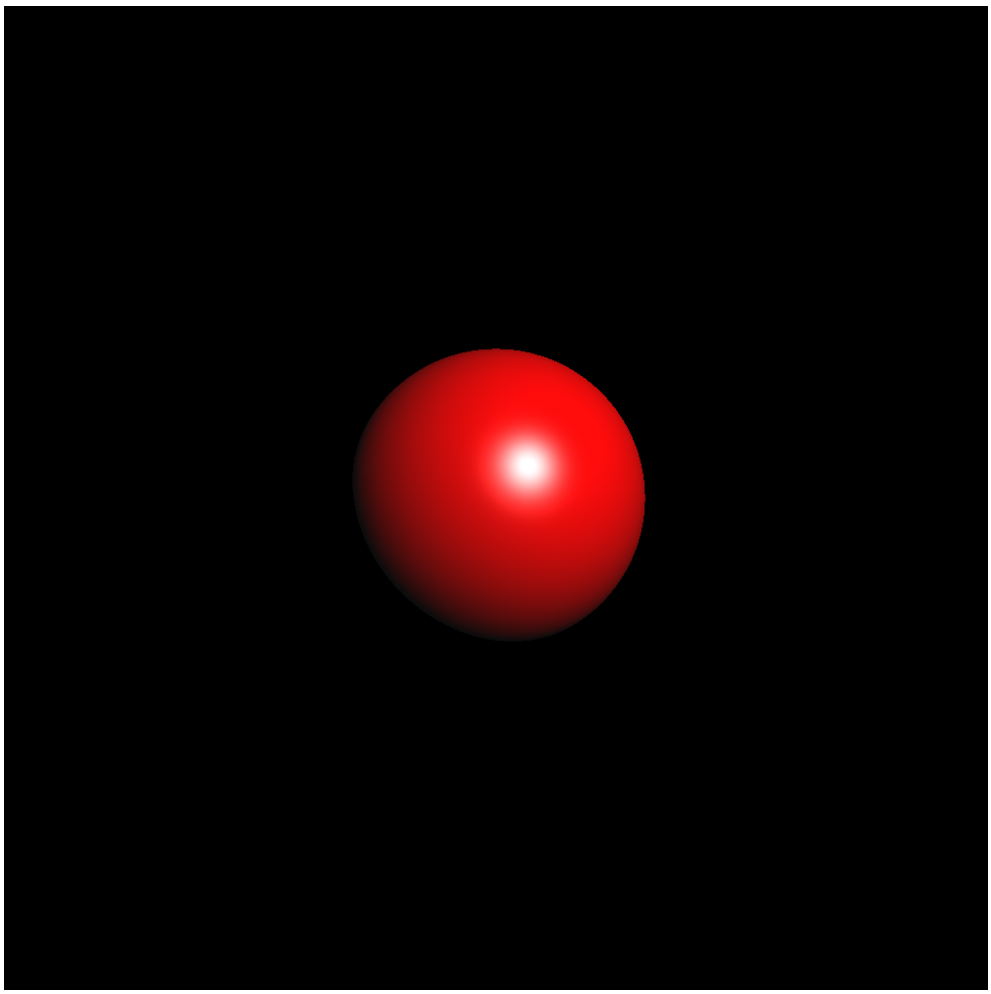


Figure B.1: 1 Sphere rendered on the GPU with a resolution of 1000×1000 px.

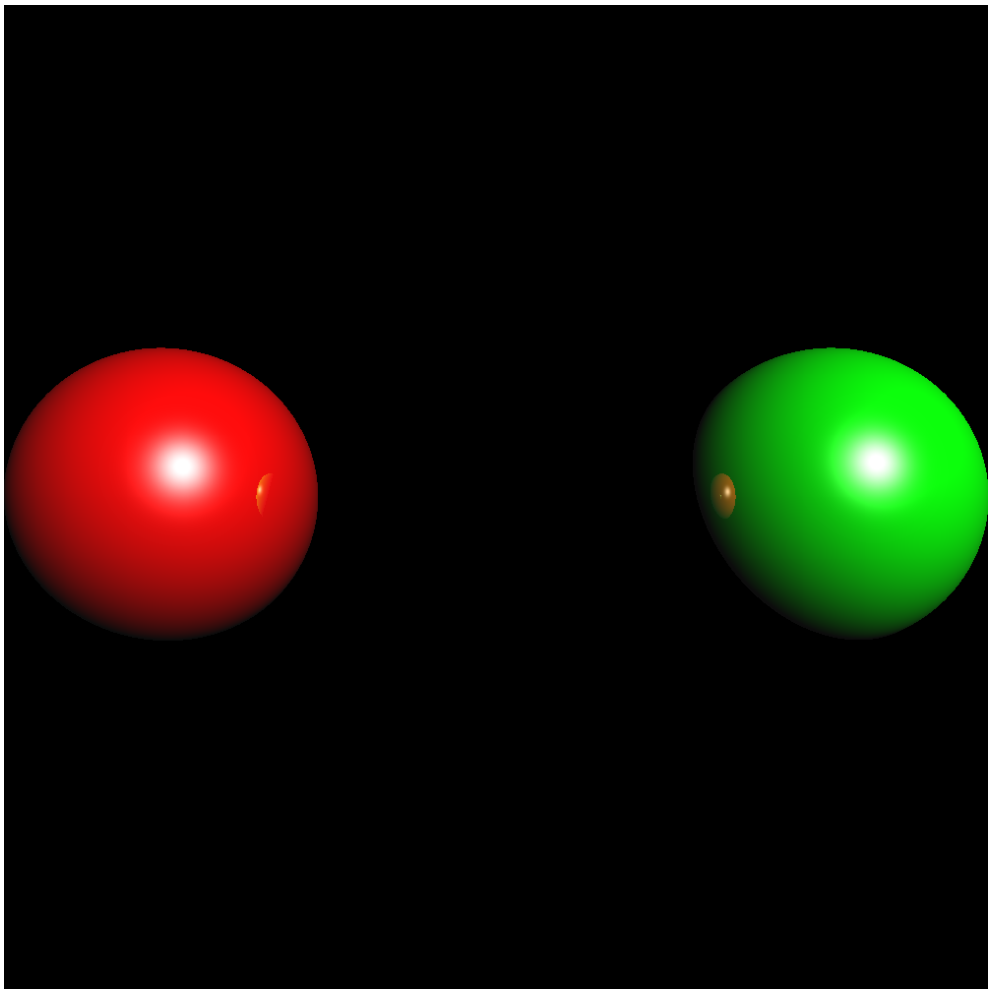


Figure B.2: 2 Spheres rendered on the GPU with a resolution of 1000×1000 px.

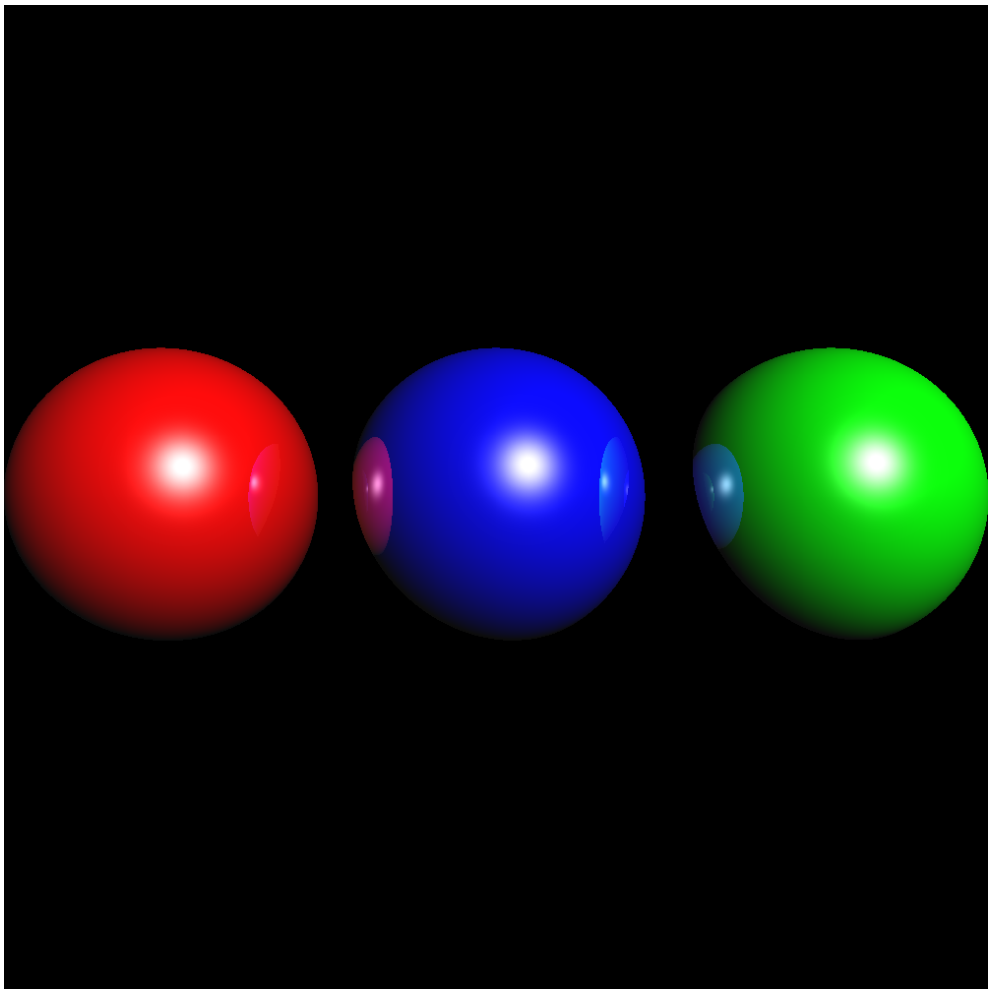


Figure B.3: 3 Spheres rendered on the GPU with a resolution of 1000×1000 px.

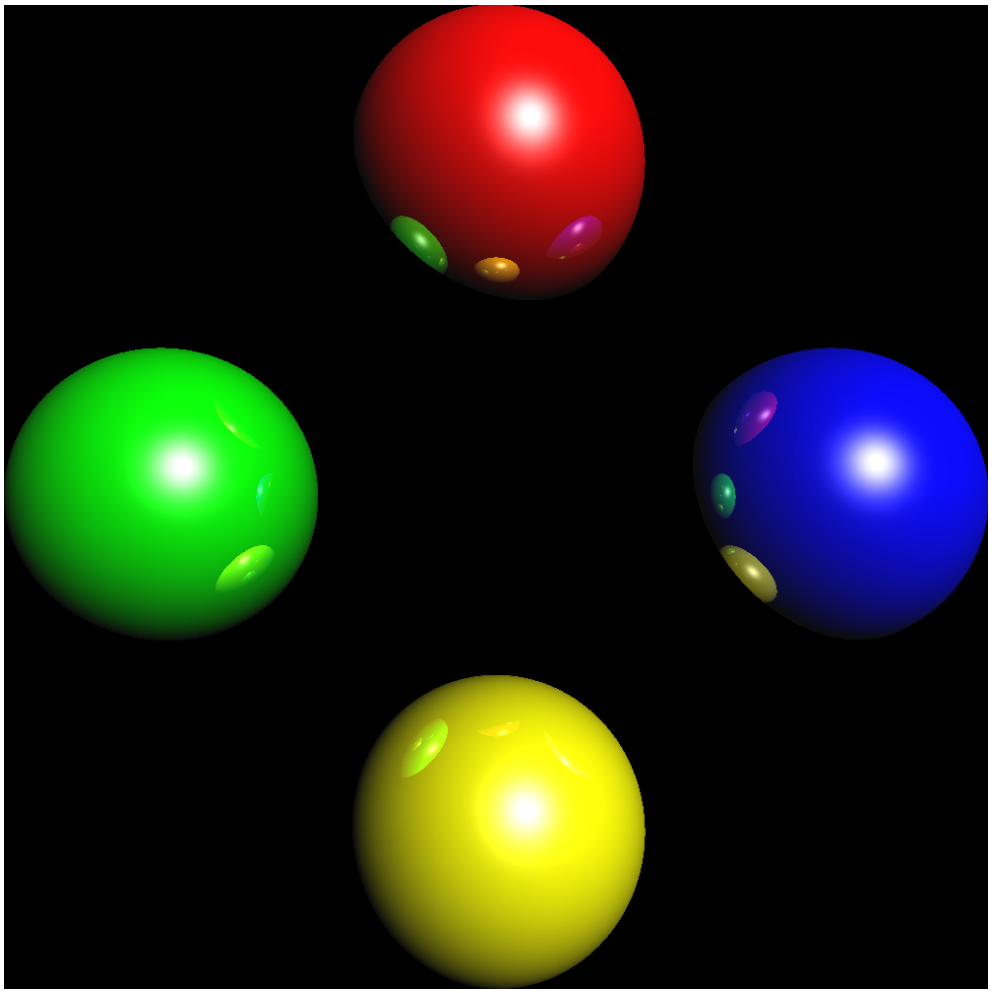


Figure B.4: 4 Spheres rendered on the GPU with a resolution of 1000×1000 px.

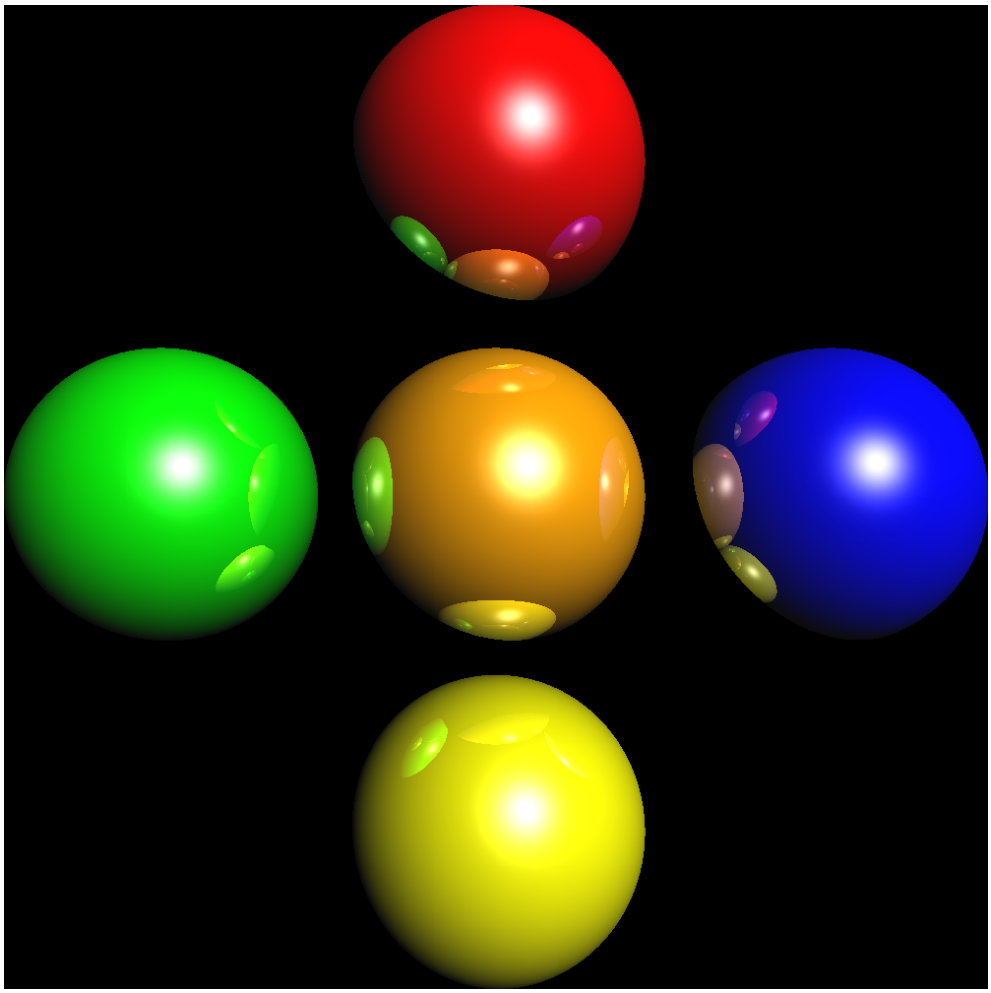


Figure B.5: 5 Spheres rendered on the GPU with a resolution of 1000×1000 px.

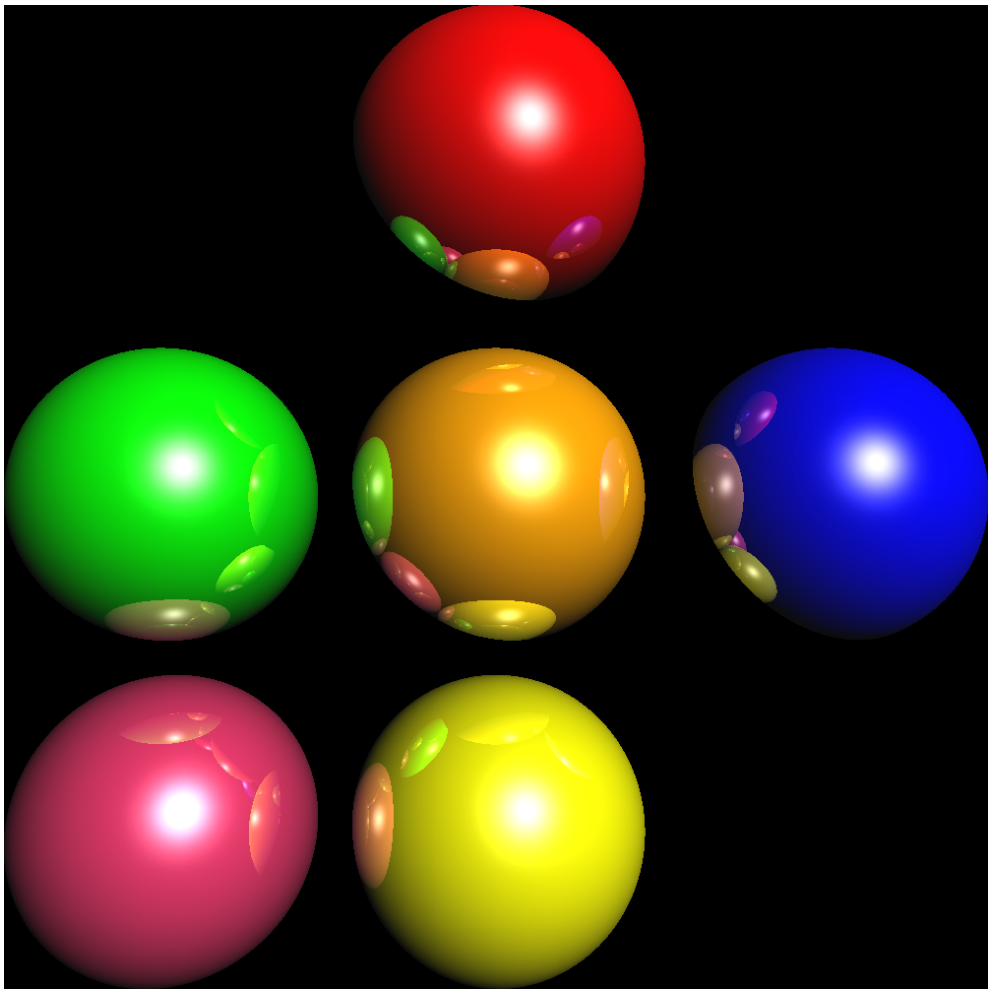


Figure B.6: 6 Spheres rendered on the GPU with a resolution of 1000×1000 px.

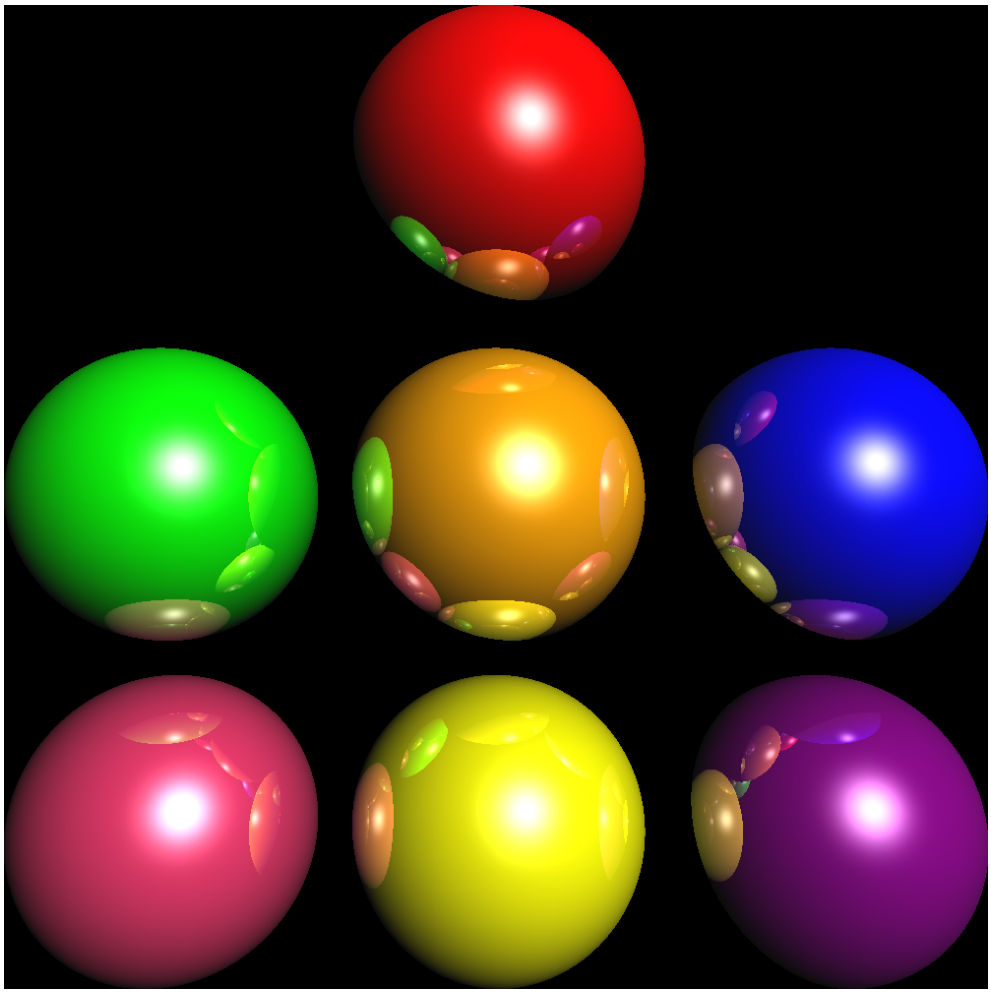


Figure B.7: 7 Spheres rendered on the GPU with a resolution of 1000×1000 px.

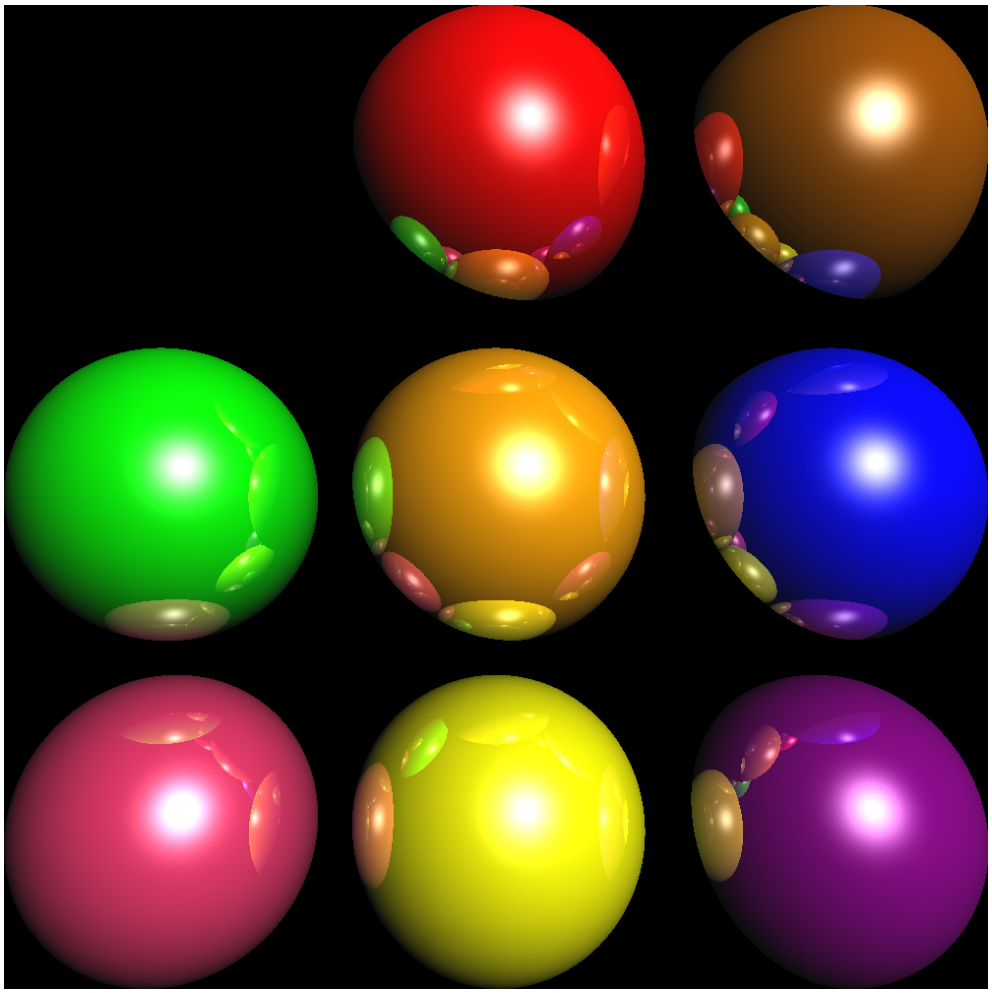


Figure B.8: 8 Spheres rendered on the GPU with a resolution of 1000×1000 px.

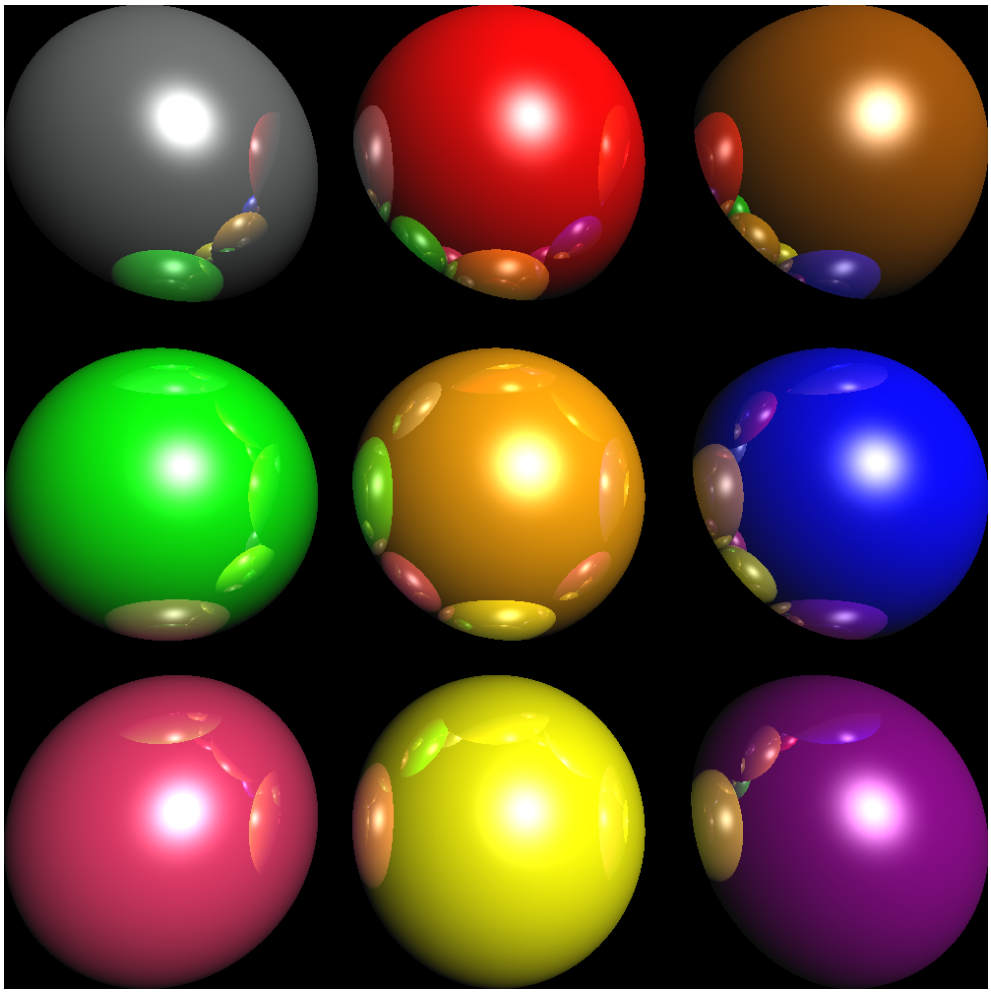


Figure B.9: 9 Spheres rendered on the GPU with a resolution of 1000×1000 px.

TRITA-EECS-EX-2022:519