Parallel Ray Tracing

Alexander Peterson, Austin Smith, Jose Mendoza

Abstract—We study the impact of parallelization of CPU-based ray traced rendering methods. Our goal was to decrease computational run-time by rendering rays in parallel. We define and construct a ray tracing renderer capable of simulating various materials and reflection, refraction, and absorption behaviors of light.

Our parallelization efforts were motivated by the fact that in ray tracing renderers, the majority (if not all) of the computation of simulating rays can be done in parallel. We parallelized the render by splitting the image into a number of chunks and giving one chunk to each thread to render on their own.

Further, we investigated GPU based rendering, but have yet to implement it. Utilizing GPU hardware, which is specialized to run parallel programs very quickly, rendering times should be improved substantially. We will look into using the Nvidia CUDA architecture to achieve this.

Index Terms—Computer Society, IEEE, journal, LaTeX, paper, parallelism, computer graphics, ray tracing.

GitHub Repository

1 Current State of Ray Tracing

The current state of ray tracing is very promising. The idea of ray tracers can be traced back to as early as the 1970's, and has been a staple in high fidelity computer graphics ever since.

One of the earliest examples of using ray tracing to render images for films is The Compleat Angler, a short 38-second film rendered in 1979, by Turner Whitted [1]. It included light refraction and reflection, which was groundbreaking for the time.

Since then, the computational power that our technology hold has increased exponentially. We now have the ability to use ray tracing to render large scenes for movies and television, and year by year we are getting closer to real-time ray tracing.

Real-time ray tracing produces a major challenge to overcome: speed. To render a real-time scene with even just 30 frames per second, each frame must finish rendering in just under 34 milliseconds. This is not much time for a ray tracer to produce an image, which can often take millions of calculations to produce at higher resolutions. As hardware has progressed, it's clear that the only way to surpass this speed requirement would be to utilize parallelization.

Graphics hardware has been progressing to the point where real-time ray tracing is now possible with a few caveats. Clever tricks to drastically reduce the computation required have been implemented to help reach the necessary speed goals. One such trick is to render at a lower sample count and using a denoiser to polish the image. Another trick is to render the frames at a lower framerate and use artificial intelligence to generate in-between frames.

NVIDIA has been a major player in helping to achieve real-time ray tracing. Their RTX series graphics cards have special hardware that can run ray tracing computations with high efficiency. They also have spent significant resources on developing technology to implement the aforementioned "tricks."

NVIDIA calls their AI frame generation tool DLSS (Deep Learning Super Sampling) and the result has been high-framerate real-time ray tracing with surprisingly good visual quality. They also are developing NRD (NVIDIA Real-Time Denoisers) which is their version of a denoiser. [2]

In combination of highly specialized graphics hardware that supports thousands of cores to parallelize the computations of ray tracing and the specialized tricks to make the computation needed much less, real-time, high-fidelity ray tracing is not a far off prospect.

Our goal with this study was to explore ray tracing and apply our own CPU-based parallelization method to see just how much speedup could be obtained from parallelization on the CPU.

2 Introduction

Ray tracing is a method of rendering images by simulating rays of light and their interactions with objects in a scene. To compute the color of the pixel at x, y in the rendered image, multiple rays of light are sent travelling through that pixel into the scene. The interactions of each ray are simulated and a color is set accordingly. Thus, to render an image, many rays of light need to be simulated for each pixel.

Ray tracing is computationally expensive. Each ray needs to check what objects it will intersect with and how

• *Alexander Peterson, Austin Smith, and Jose Mendoza are with the Department of Computer Science, University of Central Florida, FL, 32816.*

Manuscript received March 10, 2023.

it will interact with those objects. This often involves some vector math to simulate reflections and refraction and absorption.

On the CPU, these computations occur sequentially. However, simulated rays do not interact with each other nor do they mutate the scene. This suggests that rays can be simulated in parallel. In fact, all of ray tracing can be done in parallel. We wish to experiment with the use of C++'s multi-threading libraries to be able to accelerate this process.

It's often beneficial to study how ray tracing differs from the traditional mode of rendering: rasterization. To study it's differences is to study it's benefits and drawbacks, which is important in understanding why one would use ray tracing in the first place.

Rasterization colors pixels based on a projection from the world onto a two-dimensional screen. The color of each pixel is dictated by the color of whatever object was projected onto that pixel. Lighting effects, like shadows, are possible through a variety of techniques. One method, called *shadow mapping* calculates the scene from a light sources point-of-view, which can then be used to determine if a pixel is visible to the light or not. Specular and diffuse lighting are able to be implemented with effects through the *fragment shader*. Rendering mirrors in a rasterization system often uses simple illusions such as rendering the skybox on the mirror-like object rather than actually reflecting incoming light.

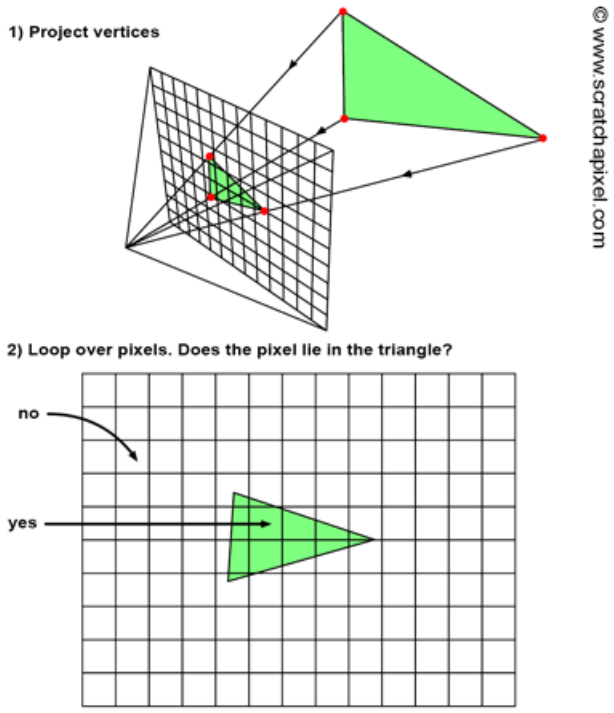

Fig. 1. Visual of a traditional rasterization pipeline projecting three-dimensional objects onto a two-dimensional screen.[3]

In summary, in the traditional rasterization rendering pipeline, lighting effects are often difficult to implement and lacking in realism. This approach is fine for most applications, but when realism and visual fidelity matter, rasterization often falls just short.

Ray tracing, as discussed, takes a much different approach. Rays of light are simulated and their interactions dictate the color of each pixel. The main benefit of this is that lighting effects are naturally built into the renderer. There are no extra computations needed for shadows, lighting, or reflections because the light simulation takes care of all of those effects. And because the light simulation is based on light in the real world, the resulting renders are nearly photo-realistic.

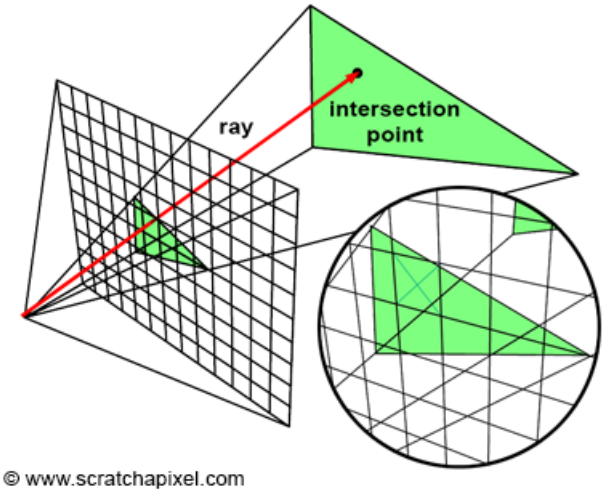

Fig. 2. Visual of a ray tracing renderer coloring a pixel based on a ray of light. [3]

Thus, when photorealistic lighting and high-fidelity quality is needed at the expense of longer render times, ray tracing is a great choice.

3 Methodology

Any ray tracing renderer needs to have a few key things:

- A vector, color, and point class.
- A ray class.
- A camera class.
- A way to create an image output.

3.1 Vectors, Colors, and Points

A vector consists of a list of numbers. There are an infinite way to interpret vectors, but we care just three interpretations.

The first interpretation is position. In three-dimensional space, we can specify a point with three values, corresponding to it's location as defined by the coordinate space we are using. This is useful for giving objects a position, such as where in the world-space should a sphere be, or where a point light is emitting from.

The second interpretation is a direction. With three values, we can specify a three-dimensional direction. This is useful for our Ray class, as well as specifying *directional lighting* which could be used to specify sunlight.

```
class Vec3 {
  public:
    double entries[3];

    double x() { return e[0]; }
    double y() { return e[1]; }
    double z() { return e[2]; }

    Vec3() {
        for (int i = 0; i < 3; i++)
            e[i] = 0.0;
    }
}
```

Fig. 3. A simplified version of our C++ Vec3 class.

The final interpretation is color. Using RGB encoding, a vector can represent the amount of red, green, and blue, in a specific color.

We use the same implementation of a vector for each of these purposes. We also define common operations between vectors, such as the dot product, cross product, adding or subtracting, and scaling.

3.2 Rays

A ray, in it's simplest form, consists of an origin point and a direction. Luckily, we just defined how to implement points and directions.

It's convenient to be able to find points that belong to a ray. For this, we introduce the parameterized form of a ray with parameter t.

$$\vec{P}(t) = \vec{O} + t\vec{R}$$

where:

$\vec{O}$ is the origin of the ray
$\vec{R}$ is the direction of the ray
t is how far along the ray the point is

Increase t and the point on the ray moves further from the origin $\vec{O}$, in the direction of $\vec{R}$. This way of representing rays is especially useful for detecting intersections between rays and objects.

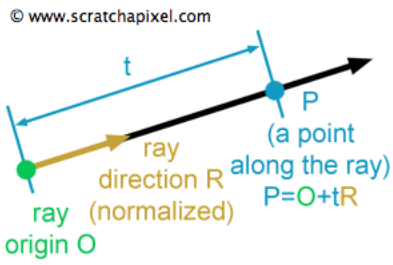

Fig. 4. Visual of a ray in parameterized form. [4]

To calculate ray intersections with objects, there must be a method of testing what value of t in the ray's parametric form does the ray meet the geometry of the object. Perhaps the simplest object to implement is a sphere.

The equation for a sphere is as follows:

$$(\vec{P} - \vec{C}) \cdot (\vec{P} - \vec{C}) = r^2$$

where:

$\vec{P}$ is the vector $\{x, y, z\}$.
$\vec{C}$ is the center of the sphere.
r is the radius of the sphere.

Plugging in the point $\vec{P}(t) = \vec{O} + t\vec{R}$ results in a quadratic in t.

$$t^2\vec{D} \cdot \vec{D} + 2t\vec{D} \cdot (\vec{O} - \vec{C}) + (\vec{O} - \vec{C}) \cdot (\vec{O} - \vec{C}) - r^2 = 0$$

With this we can solve for the values of t that satisfy the equation, which will give us the exact values of t that correspond to what point along the ray intersects with the sphere.

Note that the equation is quadratic and therefore will always have two roots. But not every ray intersects every sphere, so some roots may be imaginary.

3.3 Camera

A camera is useful for positioning the point of view in your scene. The camera consists of two main parts, the eye and viewport.

The eye is the source of all rays. The viewport is the canvas in which rays will pass through to determine pixel colors.

The eye and camera have a few important parameters that alter a few characteristics of the final image:

- The aspect ratio of the viewport. This is the width of the viewport divided by the height of the viewport.
- The viewport height. The viewport width is inferred from the aspect ratio and the viewport height.
- The focal length. This is the distance from the eye to the closest point on the viewport.
- The aperture, which controls the amount of blur on objects outside of focus.

By having these parameters, one can change the position, orientation, and focus distance of the camera.

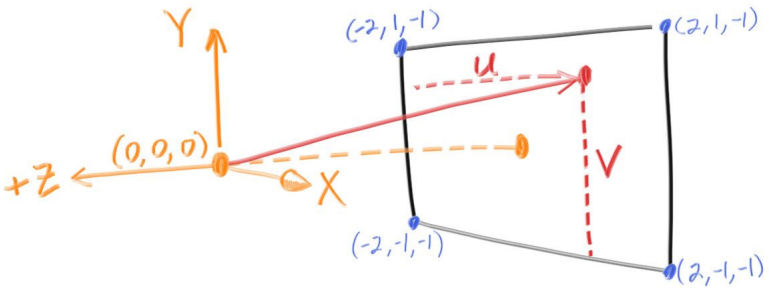

Fig. 5. Visual of the eye and viewport, showing how to reference a pixel (u, v) on the viewport.[5]

Depth of field is created by adding a small, random, circular offset controlled by the aperture to the position of the eye when sending rays into the scene. This means rays that hit something directly at the focal length will converge around the same spot, but at any other distance will be out of focus. This is highly representative of a small lens within a real camera, as light is captured at all points along the lens, rather than at a single point.

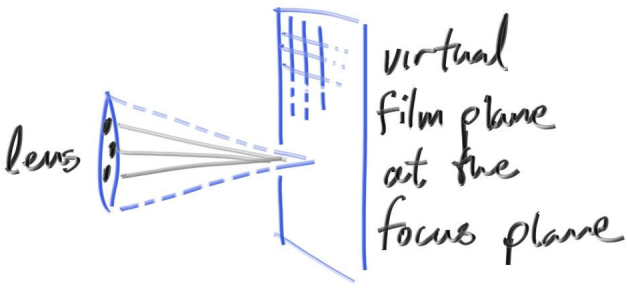

Fig. 6. Visual of the effect of aperture on the origin of rays. The aperture is precisely the diameter of the lens. [4]

3.4 Image Output

Our renderer produces images in the PPM format, which simply outputs the RGB values for each pixel. This was the most straightforward way to produce an image from our renderer, as it already computes the RGB values for each pixel during the rendering process.

3.5 Materials

A material is a property of an object that dictates how an incoming ray of light should interact with it. We call this the material's *scatter*, because it may include both reflection and refraction. A material may also have a color that will impact the color of the incoming ray.

Two materials of interest to us are metals and diffuse materials. Metals are easy to implement as they simply reflect the incoming ray along the normal of the surface. By adding a small amount of randomness to where the incoming ray gets reflected, you can add roughness to the metal.

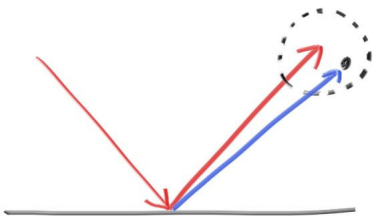

Fig. 7. Visual representing the reflection of incoming rays along a metallic surface. The circle represents the slight amount of randomness added to create a rough appearance. [5]

Diffuse materials simply reflect incoming rays of light in a random direction. This creates a smooth appearance.

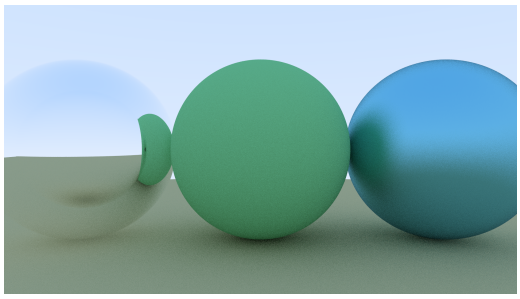

Fig. 8. A render with (left to right) a perfectly smooth metal ball, a diffuse ball, and a rough metal ball.

3.6 Render Parameters

Two major parameters in our ray tracer are the number of samples per pixel and how many ray bounces one ray can travel.

The number of samples per pixel controls how many rays are sent through each individual pixel during the rendering process. A higher sample per pixel count will result in an image with less noise, as there are more opportunities for the true components of the scene to be rendered. On the downside, more samples per pixels means more rays to simulate, which increases the computation time significantly.

The maximum number of bounces controls how many times the ray will recursively bounce through the scene. Each bounce contributes half as much color as the bounce before, so more bounces will equate to a slightly brighter image. With a maximum number of bounces at one, only the initial rays from the eye will be used to color the image, resulting in no *global illumination* (the ability for objects to reflect small amounts of light onto their surroundings) or reflections. As it turns out, only around five or six bounces is needed to create realistic looking global illumination and reflections.

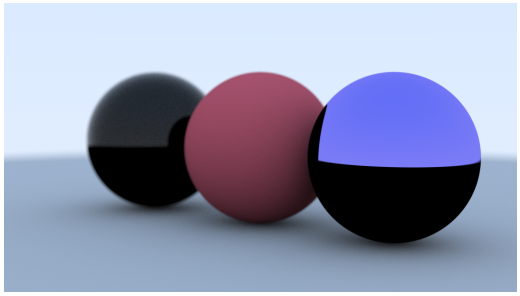

Fig. 9. A render with only one ray bounce allowed. Note how the mirror ball on the right contributes color in places only where it does not reflect onto other objects.

3.7 Parallelization

We called the method of parallelization that we implemented the *chunking method*.

This method groups pixels into n disjoint sets which are then given to n different threads to compute.

Each chunk consists of pixels that will be rendered. In each pixel multiple rays are sampled and their colors are computed. If the image size does not evenly divide n, the

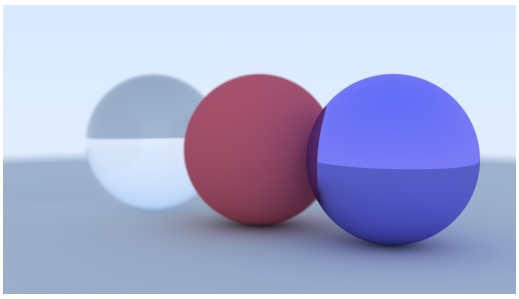

Fig. 10. The same render with up to five bounces permitted.

number of threads, then the remainder is given to the last thread. The size of the remainder is strictly less than n, so this poses no significant performance penalty for any reasonable use case.

Each thread maintains a list of the pixels it computes. Once computation is complete, each list is merged into one and the image is rendered.

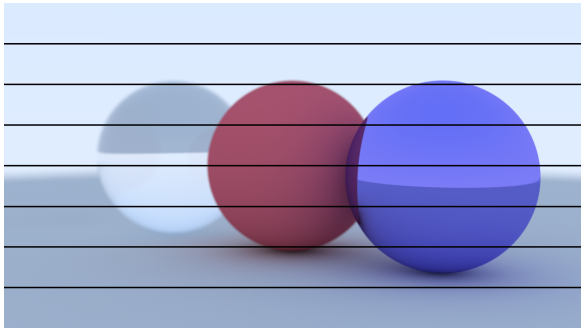

Fig. 11. Visual showing the chunks each thread will get.

This method's main benefit is that it is simple to understand and compute. With evenly split chunks, multi-threading comes with little difficulty. However, chunks in a scene *may* have vastly different amounts of computation associated with them. Consider a scene in which one chunk has a complex shape that requires multiple ray bounces, and the rest of the scene is empty. The threads responsible for the empty chunks will finish their computation much faster than the thread assigned the complex chunk. Thus a lot of time will be wasted under-utilizing the treads.

As the main requirement for a good boost in computation time through parallelization is the even distribution of work, this problem lead us to think about other alternatives for future improvements.

One idea we had was to have a shared variable that indicates which ray needs to be rendered next. When a thread isn't working, it checks that variable and sets it to the next ray appropriately. This way, all threads work for roughly the same amount of time, and an even distribution of work is achieved.

As we were happy with the performance of the chunking method, we decided to leave the implementation of this method for our future goals.

4 Outcome

Our renderer is producing images with all of the materials we set out to include.

With the chunking method, we noticed significant speedups. We collected data on a fixed scene where the only thing we varied was the number of threads the program was given. The program was run on a 2019 MacBook Pro with a 1.4 GHz Intel i5 processor.

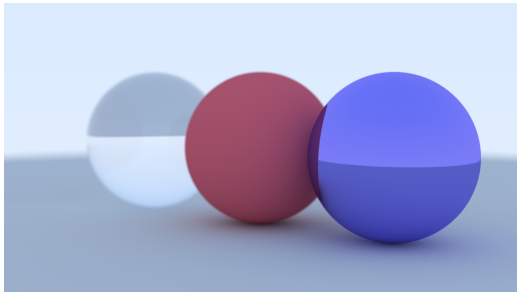

Fig. 12. The scene we used to test parallelization run-times.

	8 cores	1 core
Render time (seconds)	158.43	369.33

Our data suggests a speedup of around 60%.

Appendix A
Challenges

Our initial issue was realizing the difficulty of parallel ray-tracing, since it is even a currently active field of research in the Computer Graphics community, and thus we knew that our approaches might be naive at first, but we also have a lot of room for optimization.

After we were able to implement parallelization, we noticed that the performance on a native system had better throughput than the one when this project would run on a virtual machine, such as using an Ubuntu distribution in Windows Subsystem for Linux. Our development environment had a lot of variation amongst the team, as our individual work-spaces were different by two operating systems, and three different C++ compilers. After doing all kinds of testing and debugging, from machine code analysis provided by the IDEs that we were using, to tuning the allocated CPU cores that the virtual machine used by locating a file that the virtual machine program looks at when initializing the environment. Despite extensive testing and debugging, we determined that the added layer of virtualization was the primary cause of the performance slowdown.

Appendix B
Completed Tasks

We first were able to find a course that explained the concepts of basic ray-tracing, along with sample code that we could emulate [5]. While this initially implemented a single-threaded approach, the premise of this study was not to create a ray tracer, but to *parallelize* one.

Second, and with most importance, we implemented a parallelized way of creating our output image; by splitting the image generation into an amount of rows that can be handle the user's available threads, each of them will be in charge of calculating parts of the image, to which they are all later stitched together to display the final image.

We have also created a build system that allows our code to be compiled and ran easily on many different types of hardware.

Appendix C
Future Studies

Our main focus is to do traditional parallelization through the CPU, but if we wanted to continue on improving this project, we would want to do so through the GPU. Our initial plan of attack is to use NVidia's CUDA architecture, which is a specialized way of doing calculation through NVidia GPUs' specialized cores. To compare, a commercial CPU can have 8 cores and 16 threads, while commercial NVidia GPU's have *hundreds* of CUDA cores, which we believe will speed up our calculation by many folds. Lastly, our magnum opus would be to implement an extension to CUDA, which is NVidia's OptiX engine; newer commercial NVidia GPU's from the RTX series now also have hardware-accelerated RT cores, which are specialized for doing ray-tracing calculations, if we were able to implement this, we believe we would have an iota of run time.

One additional feature that would increase run time is to use AI de-noising. Traditionally, ray tracing creates noisy images if one does not have enough rays to be initialized; what the tensor processing units solve is being doing real-time adjustments to the final image as to how it should look if it were completely clean, which allows for less ray calculations, while still achieving native-looking visual fidelity.

Acknowledgments

The authors would like to thank our Professor, Juan Parra, for allowing us to try this project out, since this a field that all three of us have always been interested in.

Appendix D
Renders

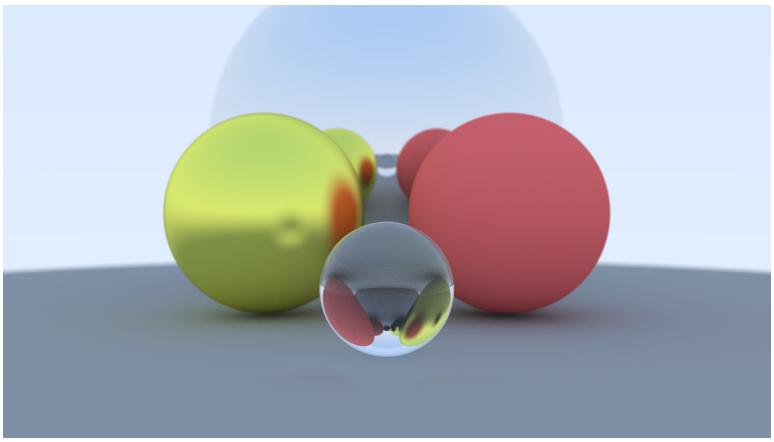

Fig. 13. A render highlighting the effects of refraction.

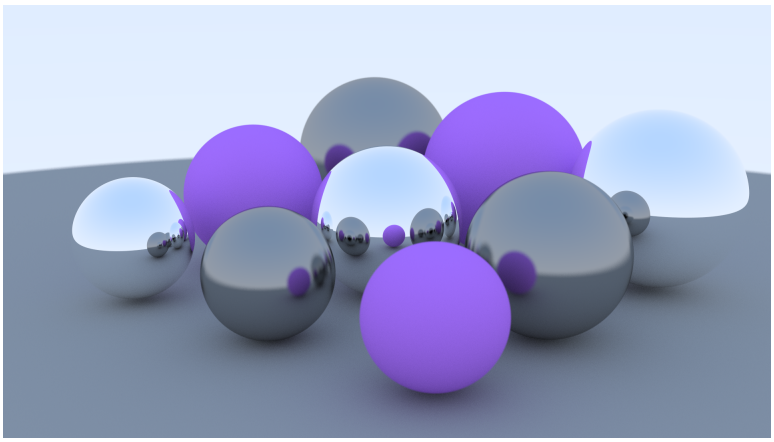

Fig. 14. A grid of spheres with various materials.

References

[1] T. Whitted. "The compleat angler." (1978), [Online]. Available: https://archive.org/details/thecompleatangler1978.

[2] NVIDIA. "Rtx technology." (2023), [Online]. Available: https://developer.nvidia.com/rtx/ray-tracing.

[3] Stratchapixel. "Rasterization: A practical implementation." (2023), [Online]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/overview-rasterization-algorithm.html.

[4] Scratchapixel. "Ray-tracing: Generating camera rays." (2023), [Online]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/definition-ray.html.

[5] P. Shirley. "Ray tracing in one weekend." (Dec. 2020), [Online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html.