



# RAY TRACING OPTIMIZATION

CASTOR • GADOR • GALANG • VALENZUELA

COE 163 THV



# OUTLINE

## METHODOLOGY

Optimizing the Codebase

## CONCLUSION

Synthesis of Main Ideas

## INTRODUCTION

Rendering

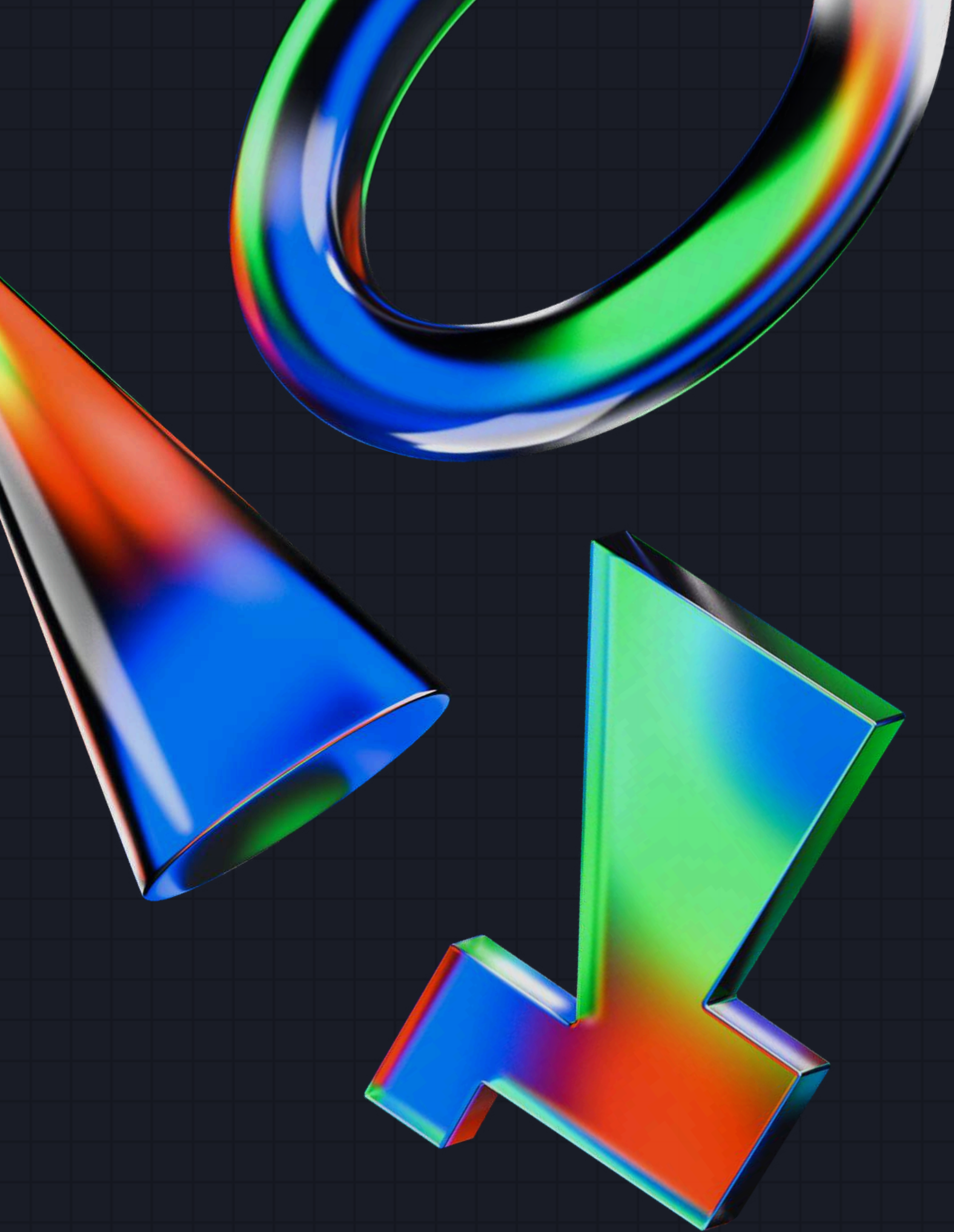
What is Ray Tracing?

The Need for Optimization

## RESULTS AND DISCUSSION

Analysis and Comparison of Runtime and  
Memory Requirements of the Unoptimized Code  
and the Proposed Optimization  
Additional Optimization Proposals

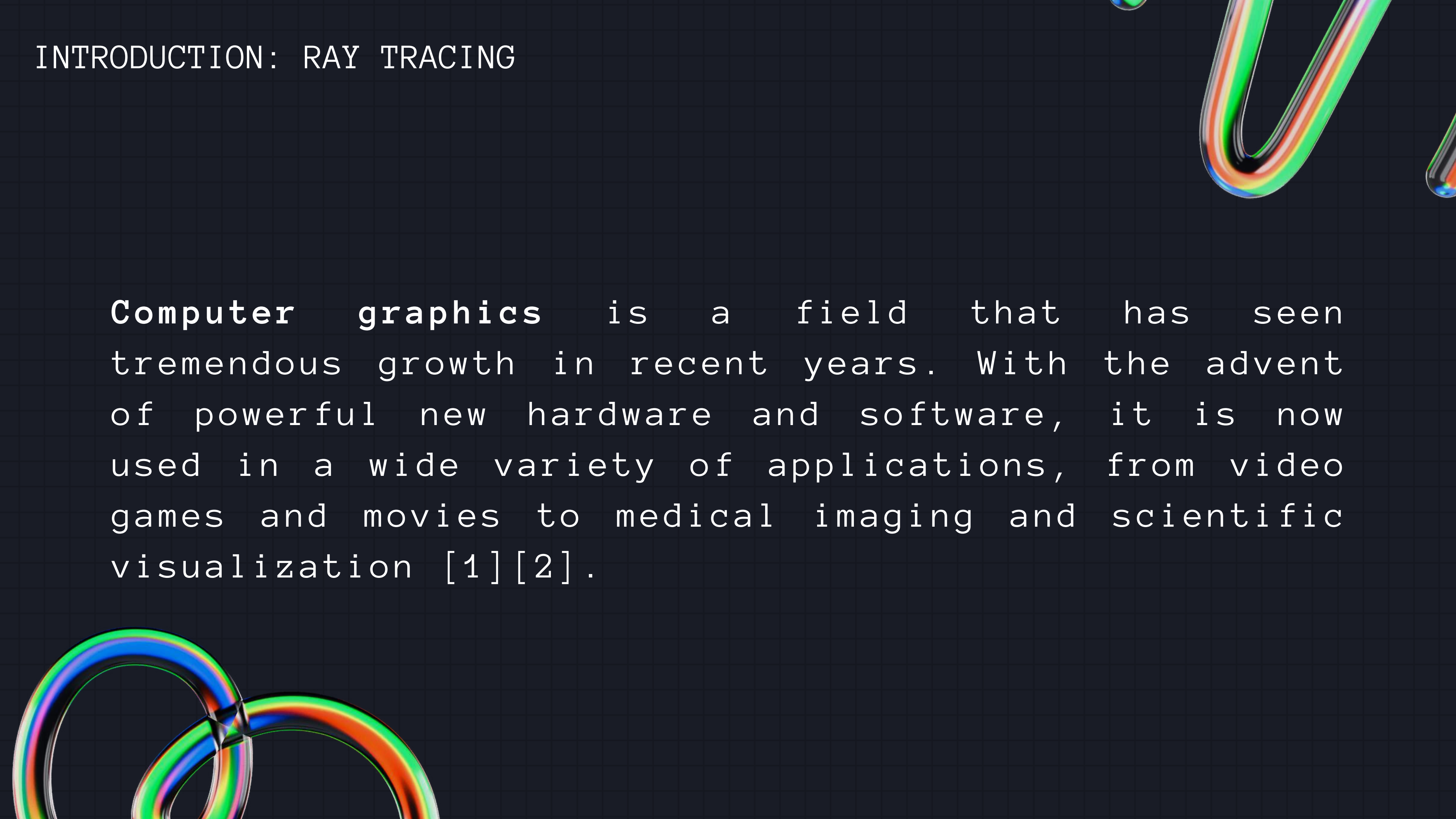




# INTRODUCTION

# INTRODUCTION: RAY TRACING

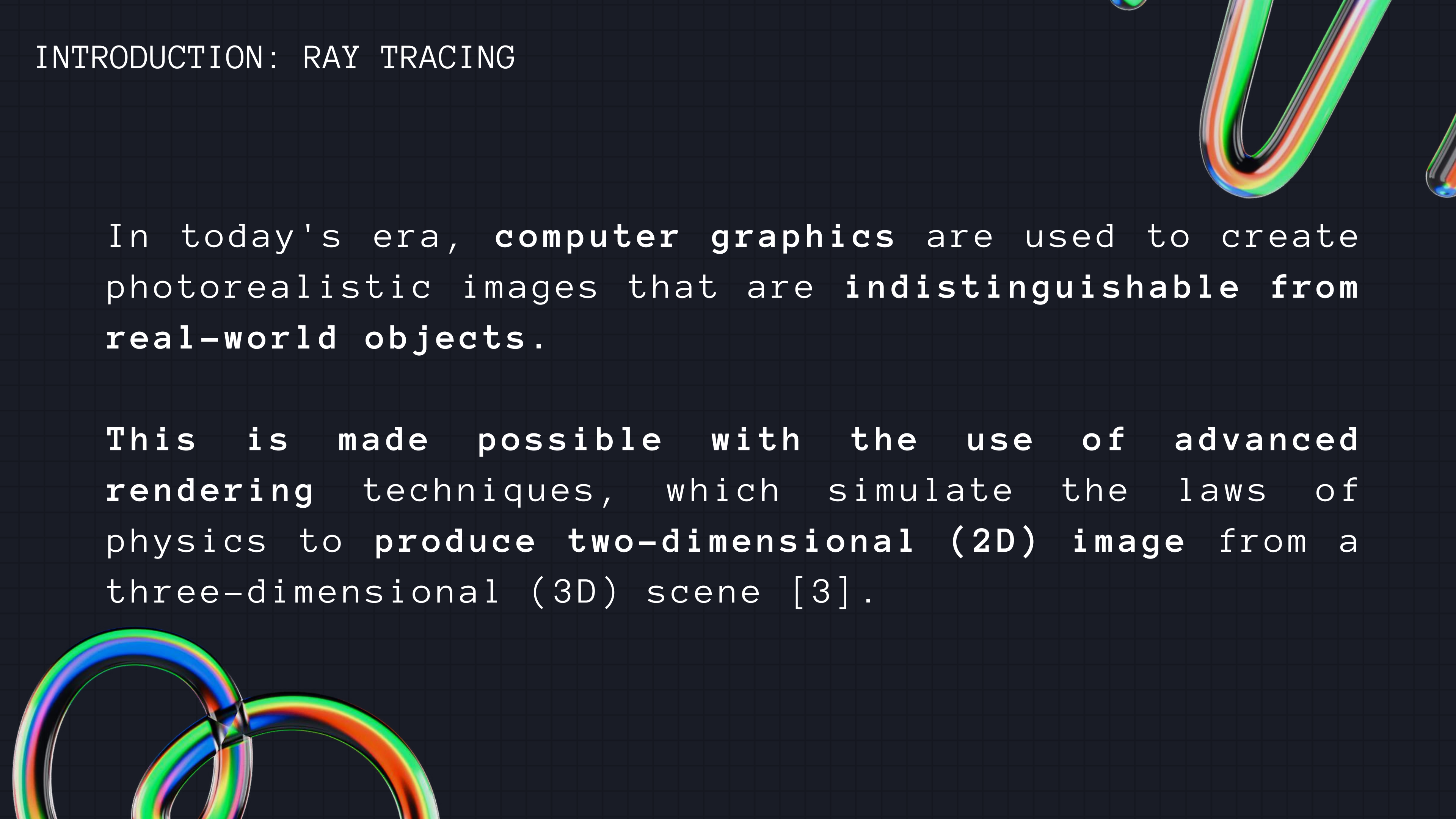
**Computer graphics** is a field that has seen tremendous growth in recent years. With the advent of powerful new hardware and software, it is now used in a wide variety of applications, from video games and movies to medical imaging and scientific visualization [1][2].



# INTRODUCTION: RAY TRACING

In today's era, **computer graphics** are used to create photorealistic images that are **indistinguishable from real-world objects**.

This is made possible with the use of advanced **rendering** techniques, which simulate the laws of physics to **produce two-dimensional (2D) image** from a three-dimensional (3D) scene [3].





# INTRODUCTION: GRAPHICS RENDERING METHODS

There are different ways to render graphics, these include:

- **Rasterization** – uses vectors to represent polygons, and distance between the polygons' points and the screen, then transforms it to groups of pixels; most popular until today since very fast.
- **Ray Casting** – uses rays connecting each pixel to scene to get texture/color (no reflections and shadows)

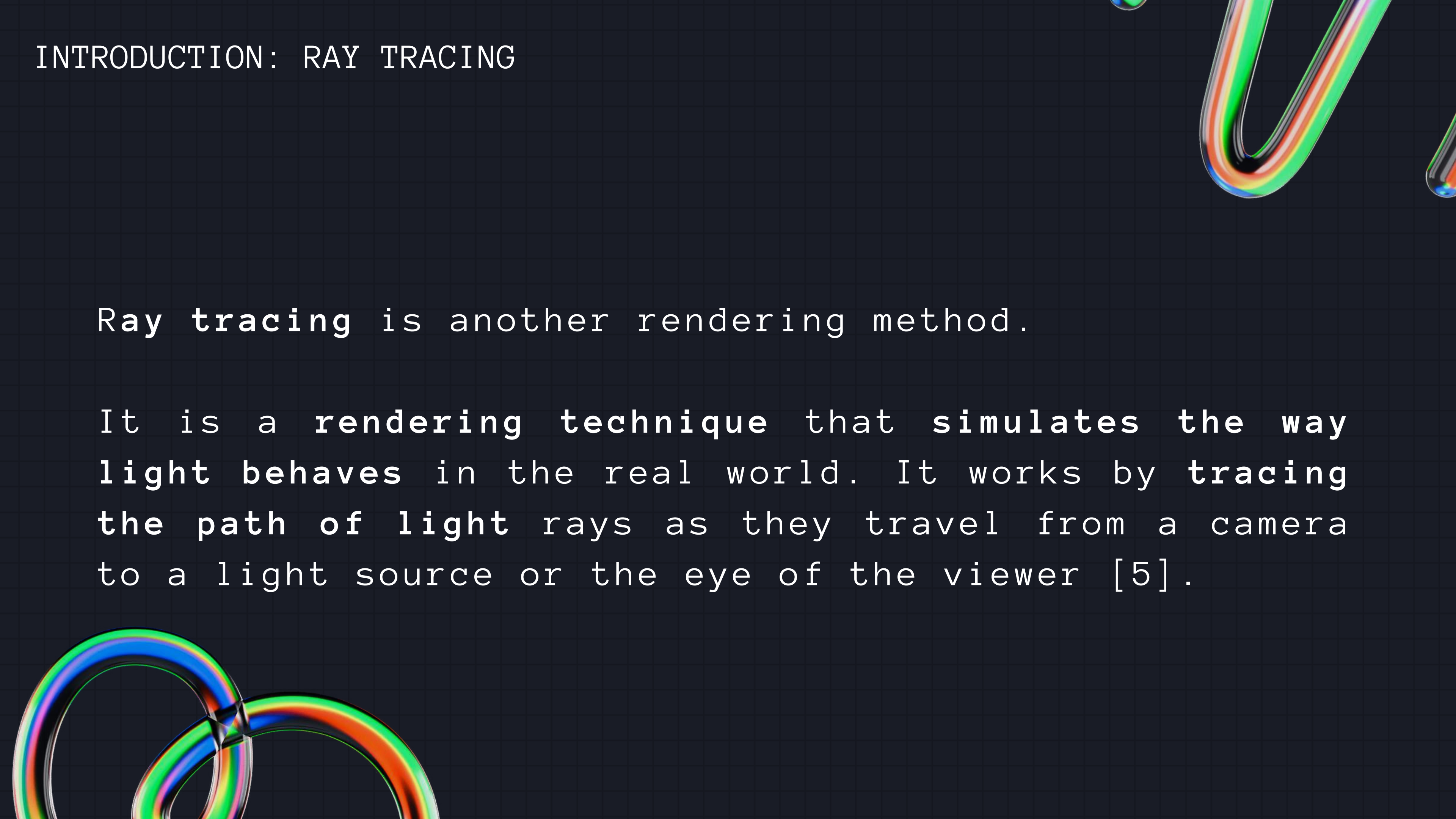
# INTRODUCTION: GRAPHICS RENDERING METHODS

- **Path Tracing** – uses a **ray** connecting the camera and the scene, and **continuously bounces the ray** instead of tracing new rays to light sources (better real-world light simulation) [4]

# INTRODUCTION: RAY TRACING

Ray tracing is another rendering method.

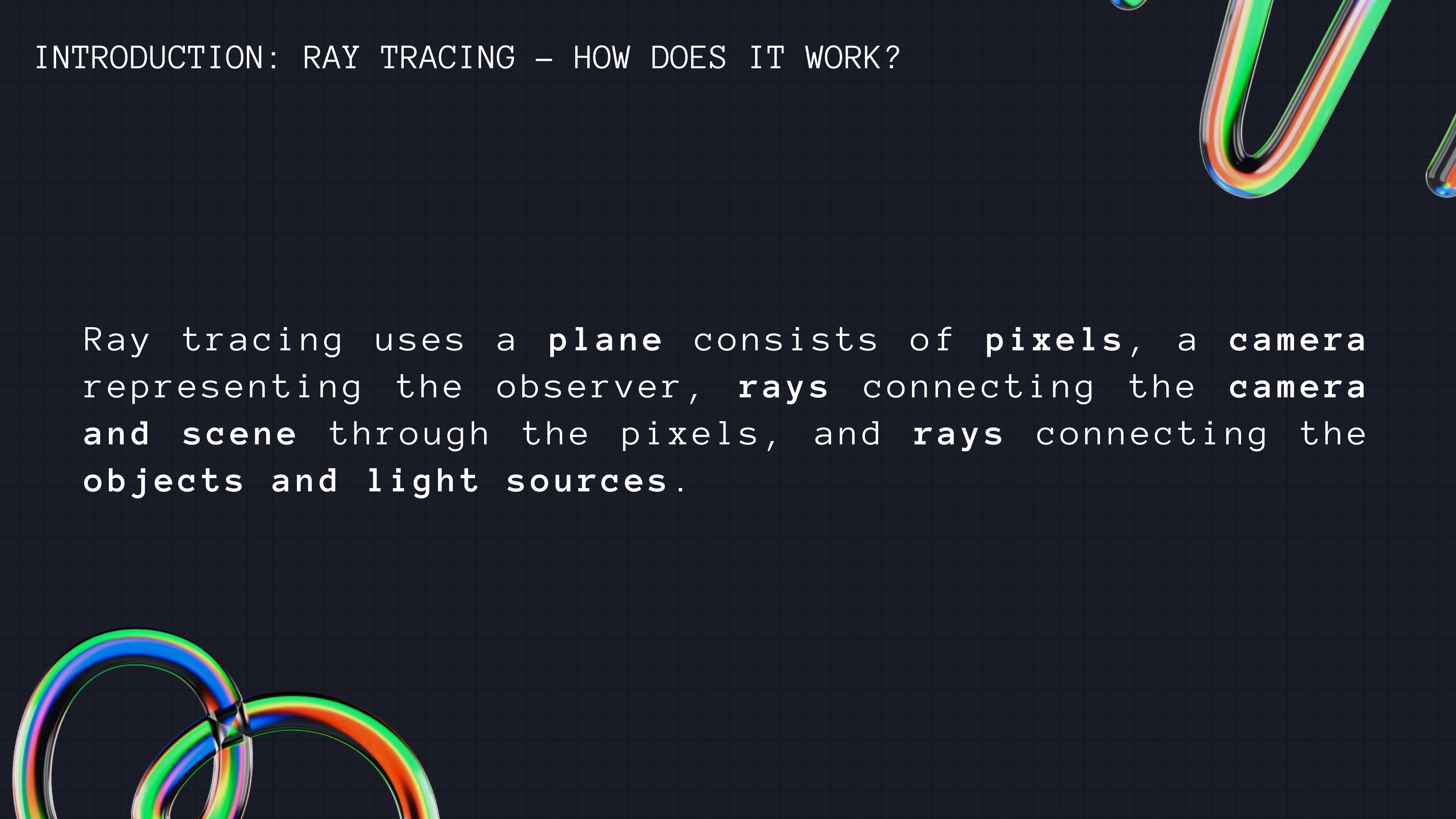
It is a rendering technique that simulates the way light behaves in the real world. It works by tracing the path of light rays as they travel from a camera to a light source or the eye of the viewer [5].





# INTRODUCTION: RAY TRACING – HOW DOES IT WORK?

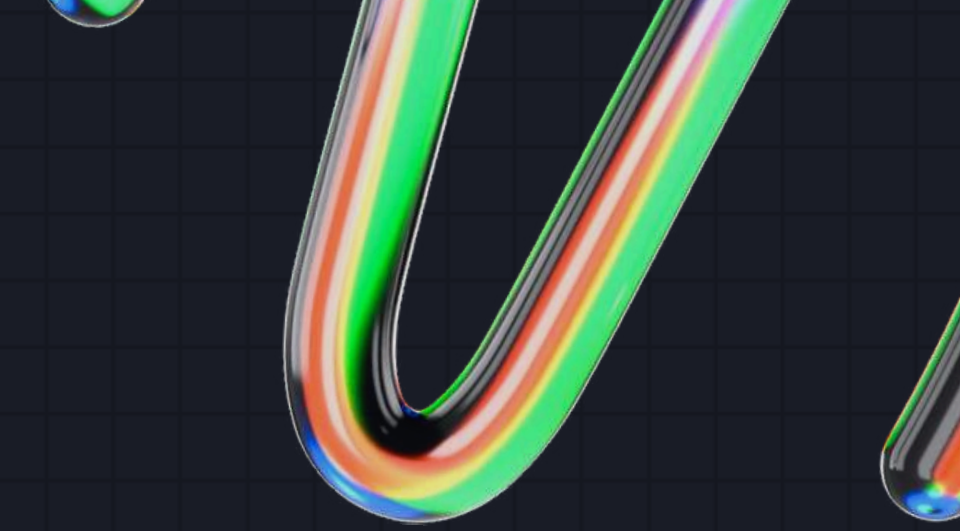
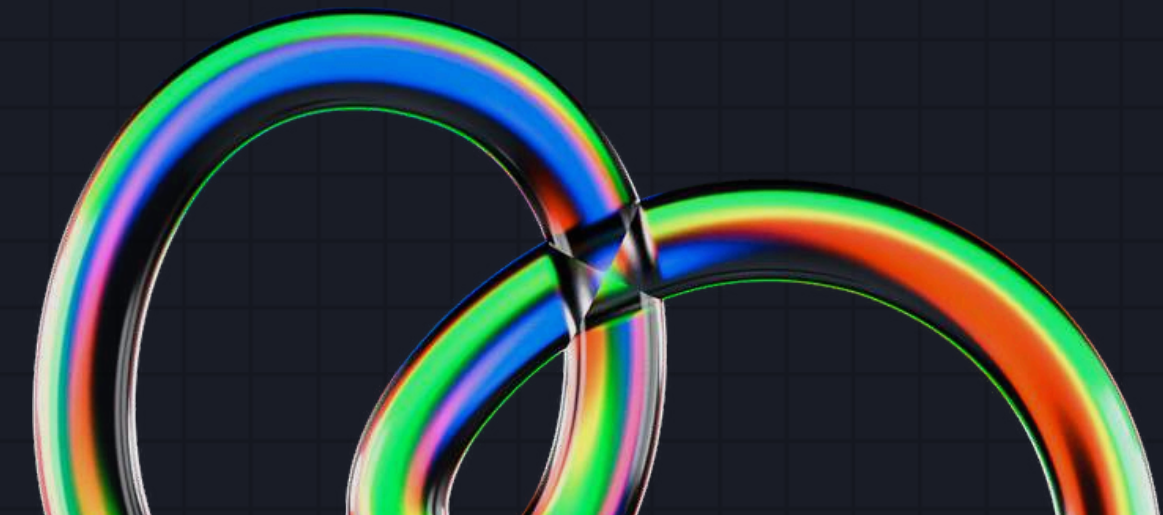
Ray tracing uses a plane consists of **pixels**, a **camera** representing the observer, **rays** connecting the **camera** and **scene** through the pixels, and **rays** connecting the **objects** and **light sources**.



# INTRODUCTION: RAY TRACING – HOW DOES IT WORK?

**Pixel Colors** are determined by the other objects intersected by the ray connecting the objects and light sources, this is **Shading**.

**Reflection** and **refraction** are also taken into account, which are present due to the **properties** of the materials such as **Diffuse** and **Specular** reflections.

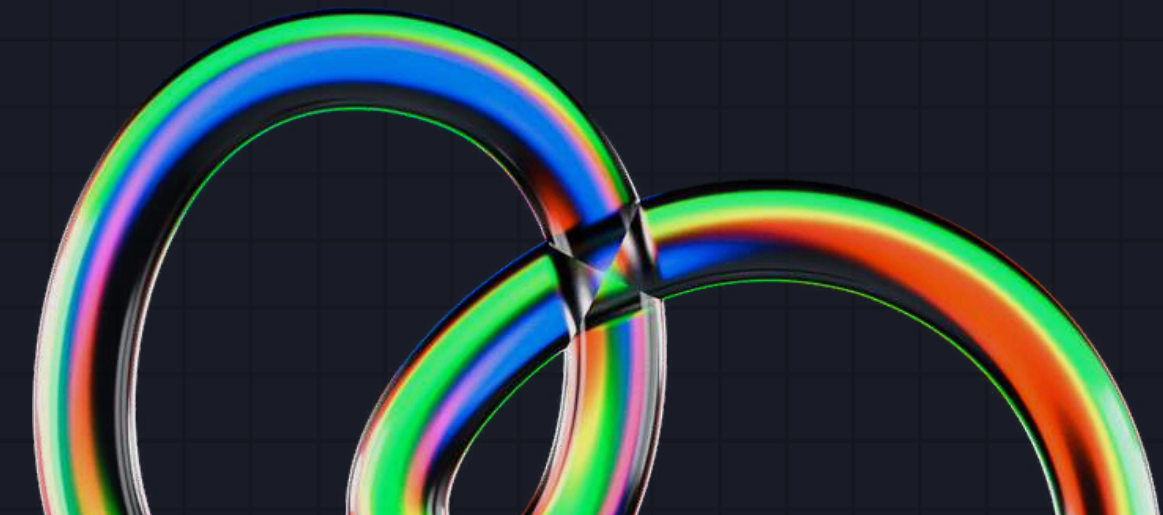
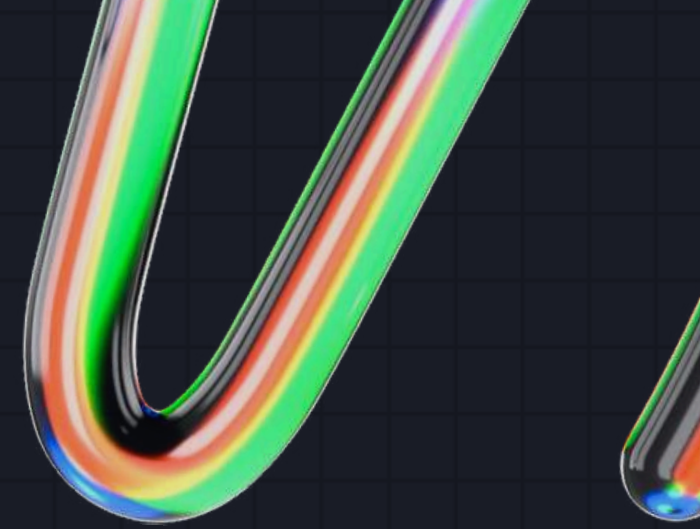


# INTRODUCTION: RAY TRACING – HOW IS IT IMPLEMENTED?

It uses **Vectors** and **Matrices** to represent and operate on the rays.

**Shadows** are computed by multiplying **Shadow Attenuations**, **soft and colored shadows** are computed by **summing** over the light sources.

**Lights** also involve **distance attenuation**.

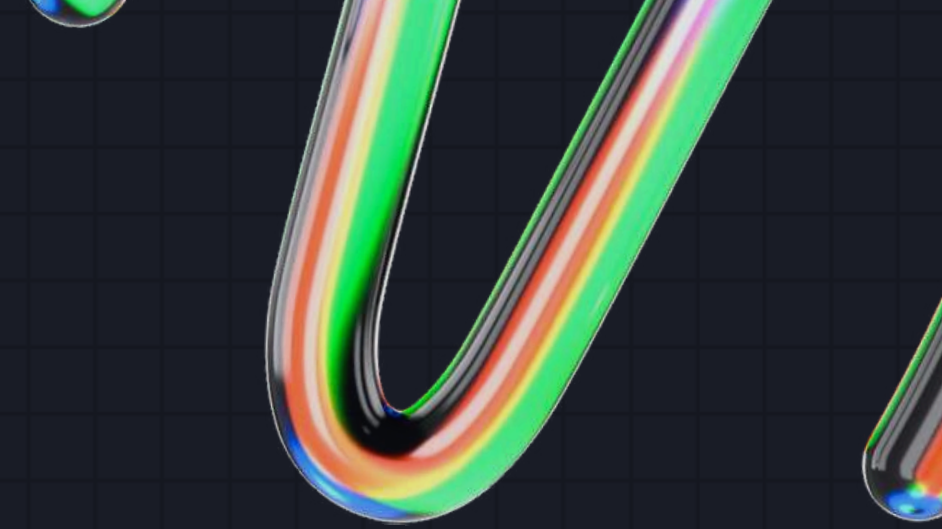
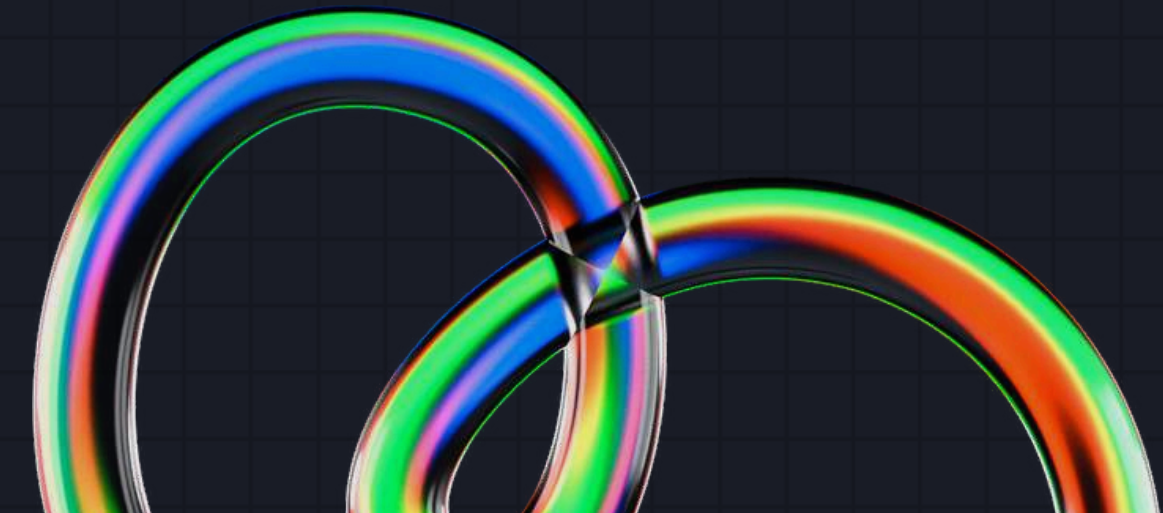




# INTRODUCTION: RAY TRACING – HOW IS IT IMPLEMENTED?

**Ambient** reflection can be added, which adds **light everywhere**.

**Emission** can be added, which is the **property** of materials that **emit light**



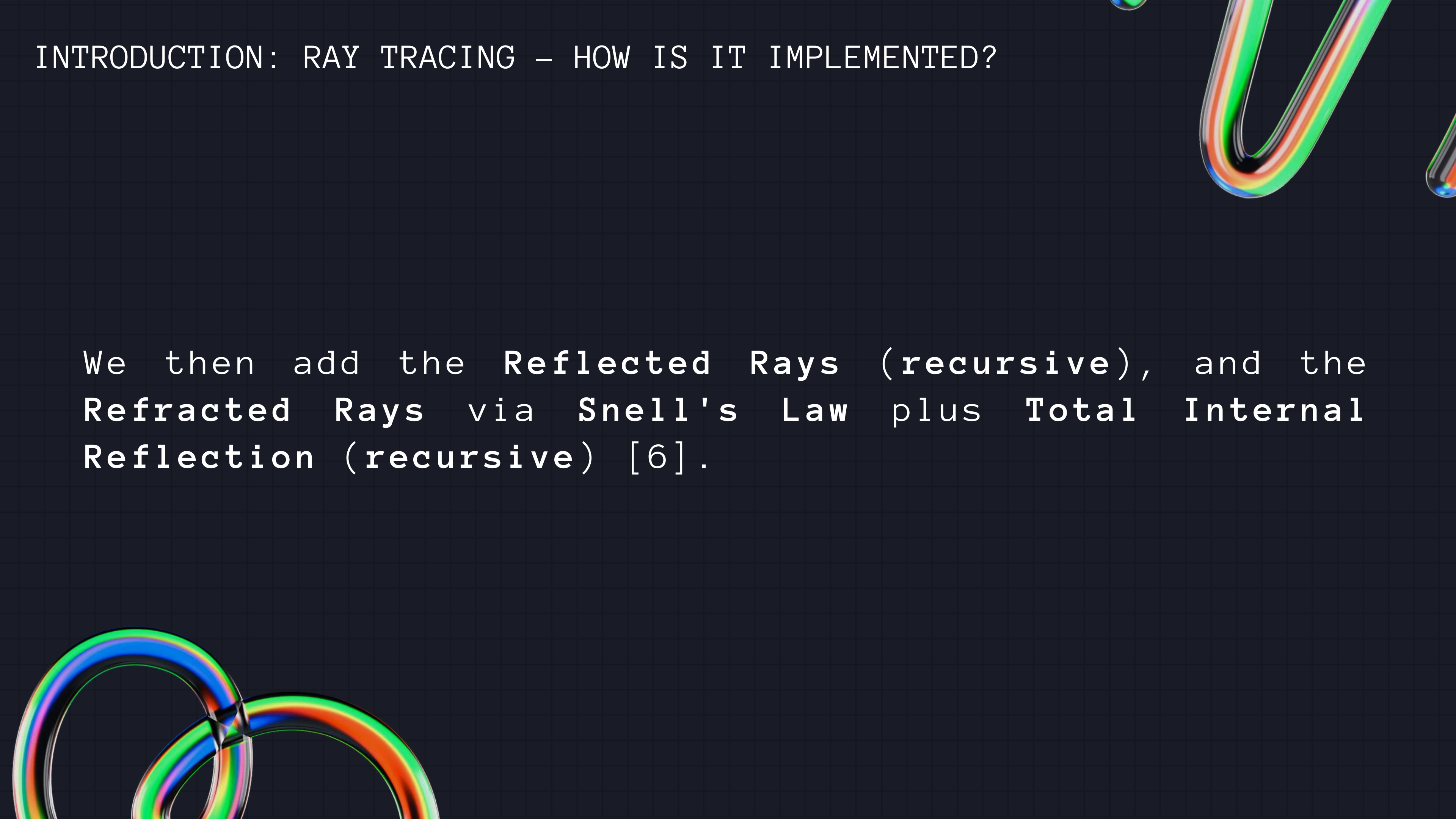
# INTRODUCTION: RAY TRACING – HOW IS IT IMPLEMENTED?

## Blinn-Phong Illumination Model

$$\sum_j A_j^d A_j^s \circ [\mathbf{k}_d I_{L_j} (\mathbf{N} \cdot \mathbf{L}_j)_+ + \mathbf{k}_s I_{L_j} (\mathbf{H} \cdot \mathbf{N})_+^\alpha] \\ + \mathbf{k}_d I_a \\ + \mathbf{k}_e$$

# INTRODUCTION: RAY TRACING – HOW IS IT IMPLEMENTED?

We then add the **Reflected Rays (recursive)**, and the **Refracted Rays** via **Snell's Law** plus **Total Internal Reflection (recursive)** [6].





# INTRODUCTION: RAY TRACING – HOW IS IT IMPLEMENTED?

## Overall Recursive Ray Tracing Pseudocode

```
function traceImage (scene)
```

```
  for each pixel (i, j) in image:
```

```
     $\mathbf{A} = \text{pixelToWorld}(i, j)$ 
```

```
     $\mathbf{d} = (\mathbf{A} - \mathbf{C}) / \|\mathbf{A} - \mathbf{C}\|$ 
```

```
     $I(i, j) = \text{traceRay}(\mathbf{C}, \mathbf{d}, \text{scene})$ 
```

```
function shade ( $\mathbf{Q}, \mathbf{N}, \mathbf{M}, \mathbf{d}$ , scene)
```

```
   $I = \mathbf{M}.k_e$ 
```

```
  for each light:
```

```
     $\text{atten} = \text{light} \rightarrow \text{distanceAtten}(\mathbf{Q}) * \text{light} \rightarrow \text{shadowAtten}(\mathbf{Q}, \text{scene})$ 
```

```
     $\mathbf{L} = \text{light} \rightarrow \text{getDirection}(\mathbf{Q})$ 
```

```
     $I = I + \text{ambient} + \text{atten} * (\text{diffuse} + \text{specular})$ 
```

```
  return I
```

```
function traceRay ( $\mathbf{P}, \mathbf{d}$ , scene)
```

```
  ( $\mathbf{Q}, \mathbf{N}, \mathbf{M}$ )  $\leftarrow$  scene.intersect ( $\mathbf{P}, \mathbf{d}$ )
```

```
   $\mathbf{r} = -2(\mathbf{d} \cdot \mathbf{N})\mathbf{N} + \mathbf{d}$ 
```

```
   $\theta_t = \sin^{-1}\left(\frac{n_i}{n_t} \sin \theta_i\right)$ 
```

```
   $I = \text{shade}(\mathbf{Q}, \mathbf{N}, \mathbf{M}, \mathbf{d}, \text{scene})$ 
```

```
    +  $\mathbf{M}.k_s \text{traceRay}(\mathbf{Q}, \mathbf{r}, \text{scene})$ 
```

```
    +  $\mathbf{M}.k_t \text{traceRay}(\mathbf{Q}, \mathbf{t}, \text{scene})$ 
```

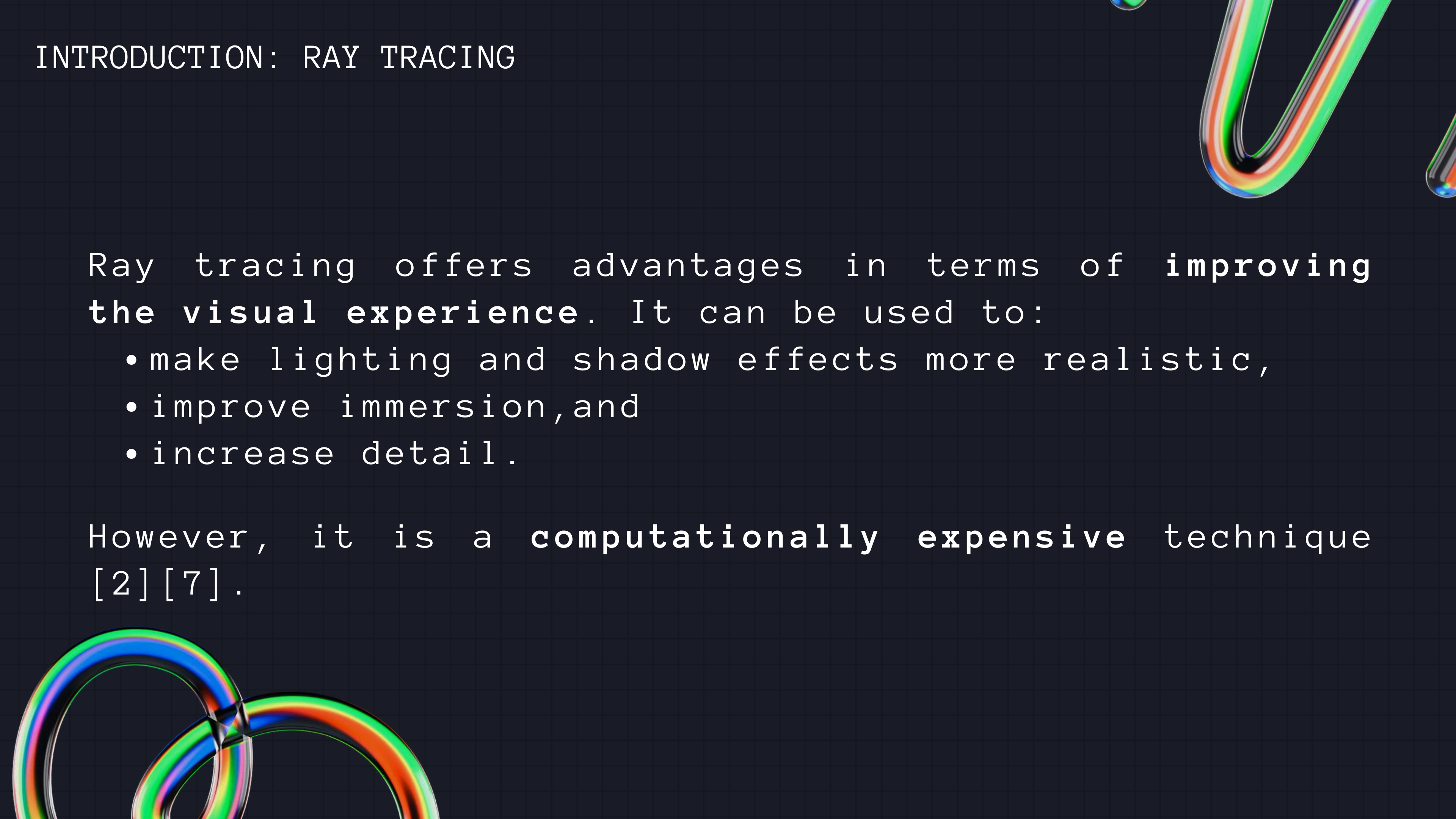
```
  return I
```

# INTRODUCTION: RAY TRACING

Ray tracing offers advantages in terms of **improving the visual experience**. It can be used to:

- make lighting and shadow effects more realistic,
- improve immersion, and
- increase detail.

However, it is a **computationally expensive** technique [2][7].

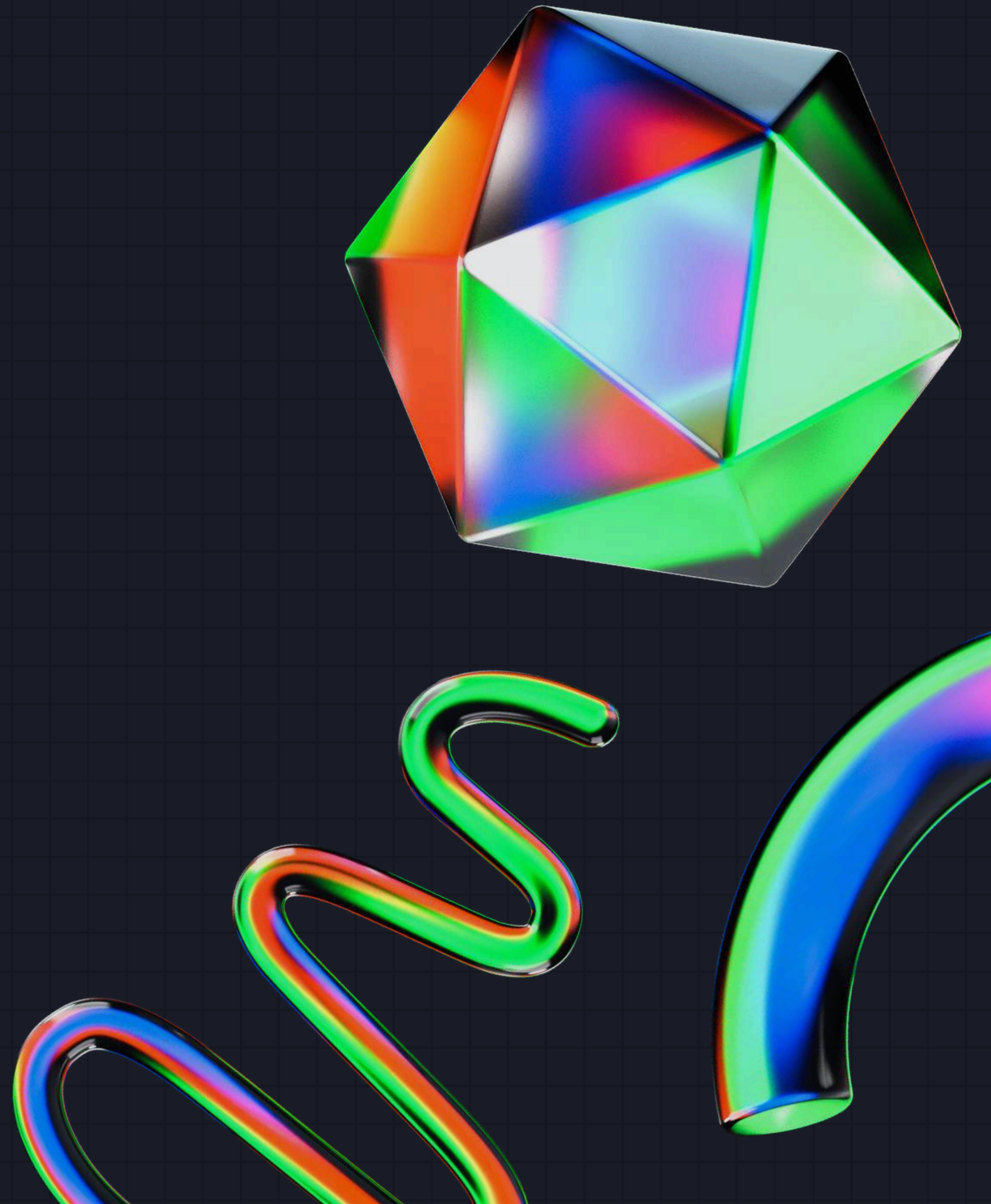


## INTRODUCTION: OPTIMIZATION

In order to effectively harness the advantages of ray tracing, code optimizations may be done. This not only improves the performance of the program, but also ensures that computation resources are efficiently used.



# METHODOLOGY





## METHODOLOGY: MACHINES USED FOR TESTING

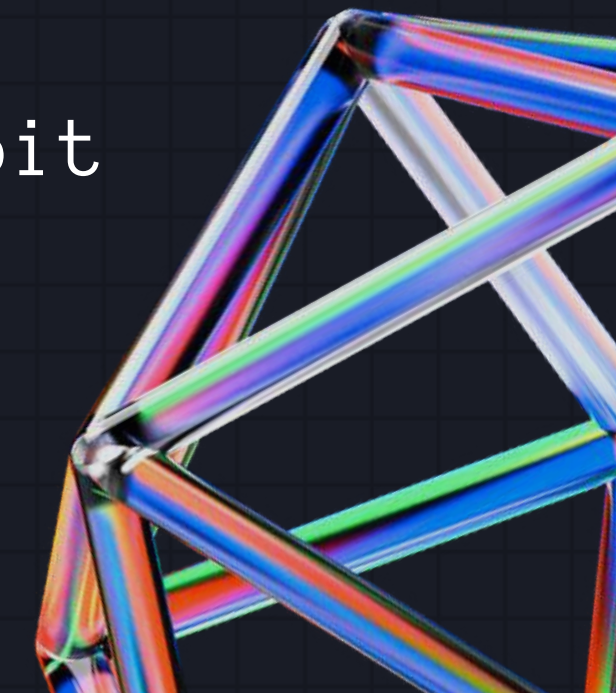
The following machines were used to run the ray tracing program:

### Machine A

CPU: AMD Ryzen 5 5600H  
GPU: NVIDIA GeForce RTX 3050Ti  
RAM: 8GB + 8GB DDR4  
HDD: NONE  
SSD: 500GB NVMe M.2  
OS: Windows 11 Home 64-bit

### Machine B

CPU: AMD Ryzen 5 3550H  
GPU: NVIDIA GeForce GTX 1650  
RAM: 8GB + 4GB DDR4  
HDD: 1TB 5600 RPM  
SSD: 256GB NVMe M.2  
OS: Windows 10 Home 64-bit





## METHODOLOGY: MACHINES USED FOR TESTING

The following machines were used to run the ray tracing program:

### Machine C

CPU: AMD Ryzen 5 3550H  
GPU: NVIDIA GeForce GTX 1650  
RAM: 8GB DDR4  
HDD: 1TB 5600 RPM  
SSD: 256GB NVMe M.2  
OS: Windows 10 Home 64-bit

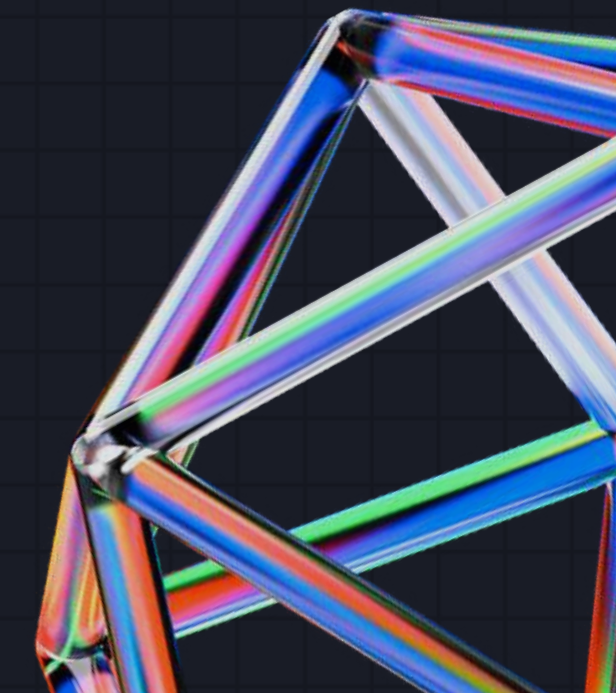
### Machine D

CPU: Intel i3-1005G1  
GPU: NVIDIA GeForce MX350  
RAM: 4GB DDR4  
HDD: 1TB 5600 RPM  
SSD: 128GB NVMe M.2  
OS: Windows 10 Home 64-bit






Both the unoptimized and the optimized program were used to run the test scenes on four different machines. The runtime was measured through time profiling using the time library in Rust. The codes were run three (3) times everytime a major optimization was implemented in order to measure the average runtime per scene. The output image was then checked to verify the correctness of the implementation.






## METHODOLOGY: PRELIMINARY OPTIMIZATION ANALYSIS

- In order to proceed with optimization, the code to be improved is first analyzed to understand their underlying processes.
  - Since the objective of the project is **Optimization**, utilizing **Parallel** and **GPU Programming** is considered.
  - Upon studying Ray Tracing and initially scanning the provided code, it was observed that the code involved lots of **independent data computations** that are candidates for Parallelism, alongside **vectors** and **matrices** whose operations can be sped up.
- 



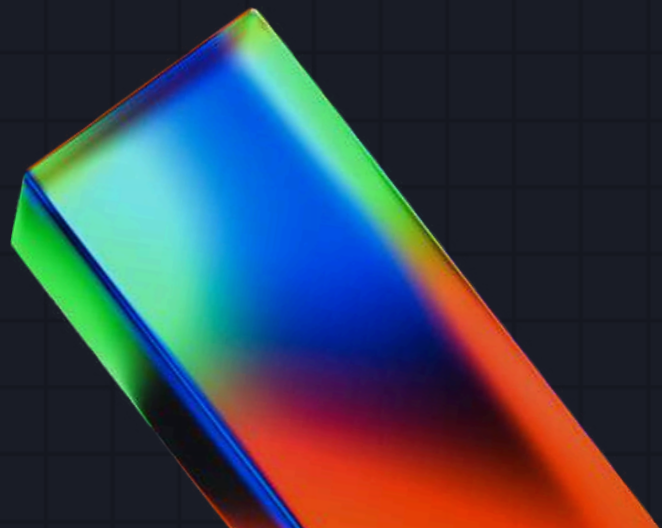
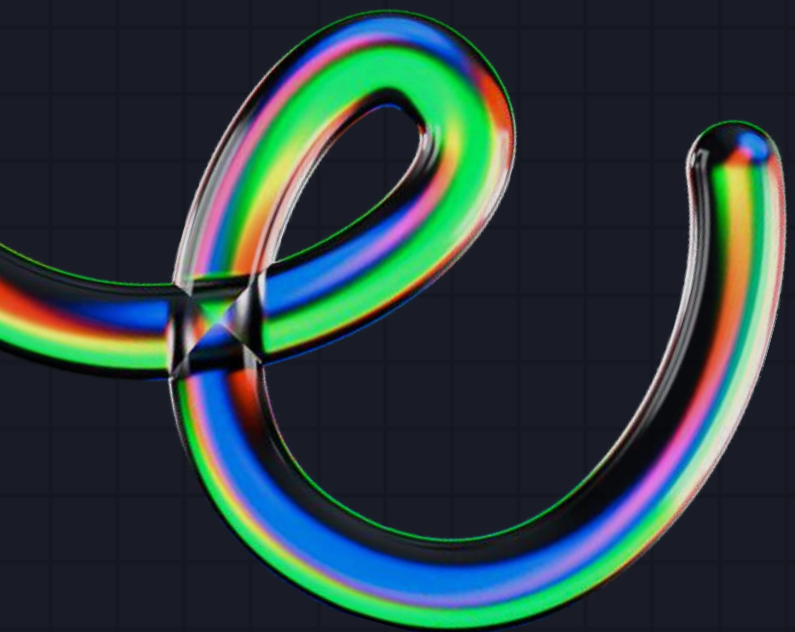
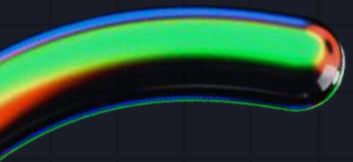
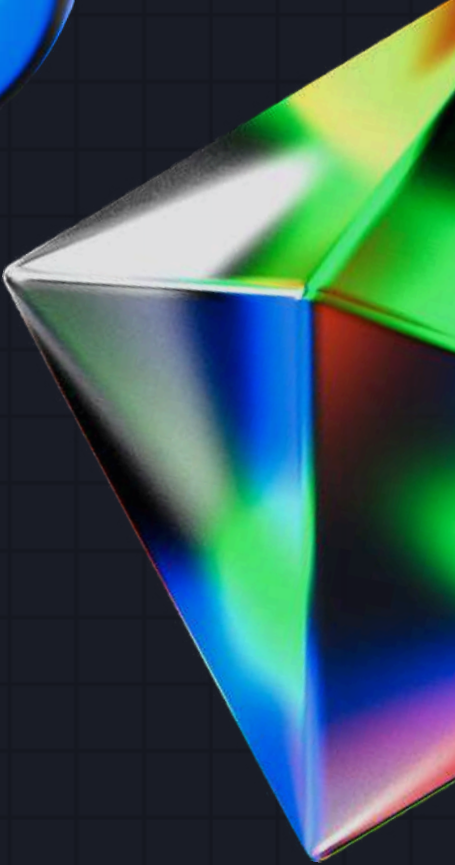
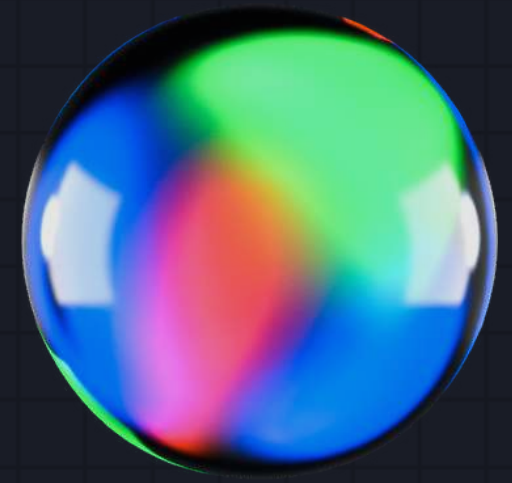
## METHODOLOGY: PRELIMINARY OPTIMIZATION ANALYSIS

- Matrix–Matrix Multiplication (MMM) and Addition are operations that greatly benefit from optimization. The general idea is to improve the temporal and spatial locality to avoid cache misses with techniques like loop reordering and the use of BLAS.
  - The code also contains lots of overhead functions that can be replaced by simpler and more straightforward operations.
  - Some computations are repeated multiple times inside the loop so rearrangements can help speed up the program.
- 



CODEBASE OPTIMIZATION I

# render and build\_image function optimization



## the render function

- The render function takes a scene struct as parameter and outputs a vector of pixels of type u8.
- First, it creates a vector of pixels, which is the same size as the scene's height and width, initialized to black.
- It then iterates over the pixels in the image. For each pixel, it casts a ray from the camera through the center of the pixel. The ray is then intersected with the scene, and the color of the pixel is calculated based on the intersection point.
- If the ray does not intersect with the scene, the pixel is colored black.

## the build\_image function

- The build\_image function takes a tuple of image dimension of type usize and a vector of pixels of type u8 as parameter and outputs a DynamicImage struct.
- It first checks to make sure that the number of pixels is divisible by 3. This is because each pixel in an RGB image is represented by three bytes, one for each of the red, green, and blue channels. If the number of pixels is not divisible by 3, then the function will panic.



## the build\_image function

- It then initializes a DynamicImage struct with the specified dimensions.
- It then iterates over the pixels in column major order (height, then width). For each pixel, it gets the corresponding pixel value from the vector of pixels and sets the pixel value in the DynamicImage struct.

## render and build\_image Optimization

- The inner and outer loops in the render and build\_image function are independent of each other and do not rely on the results of past or future iterations.
- Each iteration operates on a specific pixel in the image. Hence, there is an opportunity for parallelism.

## render and build\_image Optimization

- The rayon library is used to convert the iterators into parallel iterators which split the work into smaller chunks and distribute them across available threads to be processed in parallel.
- Additionally, a mutex is used to ensure thread safety when accessing and modifying the pixels vector and image struct. It provides a synchronization primitive that allows only one thread to access the protected data at a time.



# METHODOLOGY: CODEBASE OPTIMIZATION I

```
pub fn render(scene: &Scene) -> Vec <u8> {
    let mut pixels: Vec <u8> = Vec::new();
    pixels.resize(scene.img_width * scene.img_height * 3, 0u8);

    for i in 0..scene.img_height {
        for j in 0..scene.img_width {
            // Pass through ray in center of (i, j) pixel
            let ray = make_ray(scene, (i, j));
            let start_idx = (i * scene.img_width + j) * 3;

            // Intersection test with scene
            if let Some(id) = intersect_scene_from_view(ray, scene) {
                // Use get_color_recursive to get reflections
                //let pix_color = get_color(ray, scene, id, &scene.lights);
                let pix_color = get_color_recursive(ray, &scene, id, 0);

                pixels[start_idx + 0] = (255.0 * pix_color[0]) as u8;
                pixels[start_idx + 1] = (255.0 * pix_color[1]) as u8;
                pixels[start_idx + 2] = (255.0 * pix_color[2]) as u8;
            }
            else {
                // Color all pixels black
                pixels[start_idx + 0] = 0u8;
                pixels[start_idx + 1] = 0u8;
                pixels[start_idx + 2] = 0u8;
            }
        }
    }

    pixels
}
```

Fig 1. Unoptimized render function

```
pub fn render(scene: &Scene) -> Vec <u8> {
    let mut pixels: Vec <u8> = Vec::new();
    pixels.resize(scene.img_width * scene.img_height * 3, 0u8);

    let pixels_mutex = Mutex::new(pixels);

    (0..scene.img_height).into_par_iter().for_each(|i| {
        (0..scene.img_width).into_par_iter().for_each(|j| {
            // Pass through ray in center of (i, j) pixel
            let ray = make_ray(scene, (i, j));
            let start_idx = (i * scene.img_width + j) * 3;

            // Intersection test with scene
            if let Some(id) = intersect_scene_from_view(ray, scene) {
                // Use get_color_recursive to get reflections
                //let pix_color = get_color(ray, scene, id, &scene.lights);
                let mut pixels = pixels_mutex.lock().unwrap();

                let pix_color = get_color_recursive(ray, &scene, id, 0);

                pixels[start_idx + 0] = (255.0 * pix_color[0]) as u8;
                pixels[start_idx + 1] = (255.0 * pix_color[1]) as u8;
                pixels[start_idx + 2] = (255.0 * pix_color[2]) as u8;
            }
            else {
                // Color all pixels black
                let mut pixels = pixels_mutex.lock().unwrap();

                pixels[start_idx + 0] = 0u8;
                pixels[start_idx + 1] = 0u8;
                pixels[start_idx + 2] = 0u8;
            }
        });
    });

    let pixels = pixels_mutex.lock().unwrap().to_owned();
    pixels
}
```

Fig 2. Optimized render function



```

pub fn build_image(image_dim: (usize, usize), pixels: &Vec<u8>) -> DynamicImage {
    if pixels.len() % 3 != 0 {
        panic!("Number of pixel values ({{}}) provided is not divisible by 3!", pixels.len());
    }

    if (pixels.len() / 3) % image_dim.0 != 0 || (pixels.len() / 3) % image_dim.1 != 0 {
        panic!("Number of pixel values ({{}}) provided is not divisible by the dimensions!", pixels.len());
    }

    let mut image = DynamicImage::new_rgb8(image_dim.0 as u32, image_dim.1 as u32);

    // Write in column major (height, then width)
    for y in 0..image_dim.1 {
        for x in 0..image_dim.0 {
            let start_idx = (y * image_dim.0 + x) * 3;

            image.put_pixel(x as u32, y as u32, image::Rgb([pixels[start_idx], pixels[start_idx + 1],
pixels[start_idx + 2], 0]));
        }
    }

    image
}

```

*Fig 3. Unoptimized build\_image function*



```

pub fn build_image(image_dim: (usize, usize), pixels: &Vec<u8>) -> DynamicImage {
    if pixels.len() % 3 != 0 {
        panic!("Number of pixel values ({{}}) provided is not divisible by 3!", pixels.len());
    }

    if (pixels.len() / 3) % image_dim.0 != 0 || (pixels.len() / 3) % image_dim.1 != 0 {
        panic!("Number of pixel values ({{}}) provided is not divisible by the dimensions!", pixels.len());
    }

    let mut image = DynamicImage::new_rgb8(image_dim.0 as u32, image_dim.1 as u32);

    let image_mutex = Mutex::new(image);

    // Write in column major (height, then width)
    (0..image_dim.1).into_par_iter().for_each(|y| {
        (0..image_dim.0).into_par_iter().for_each(|x| {
            let start_idx = (y * image_dim.0 + x) * 3;

            let mut image = image_mutex.lock().unwrap();

            image.put_pixel(x as u32, y as u32, image::Rgba([pixels[start_idx], pixels[start_idx + 1],
pixels[start_idx + 2], 0]));
        });
    });

    let image = image_mutex.lock().unwrap().to_owned();
    image
}

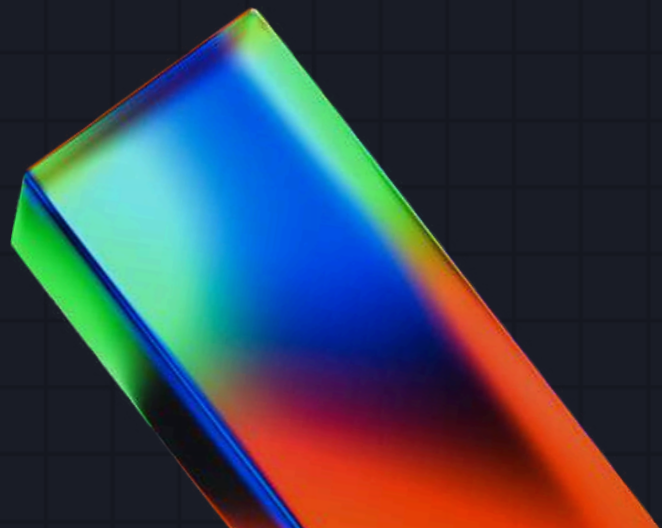
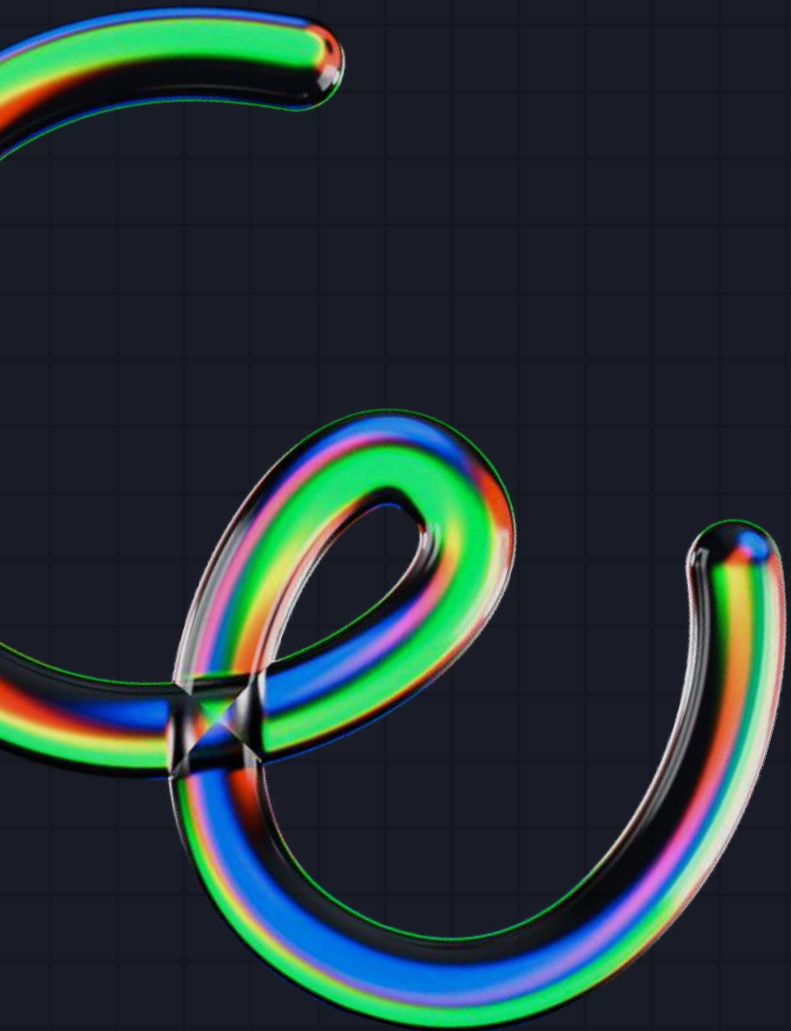
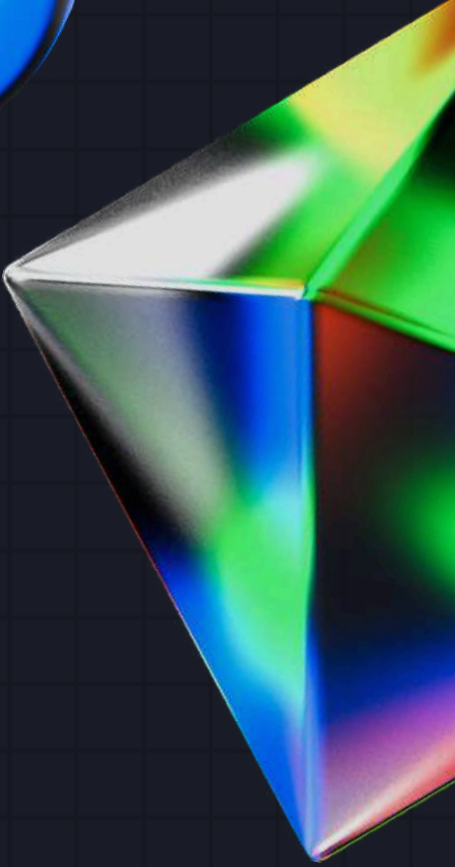
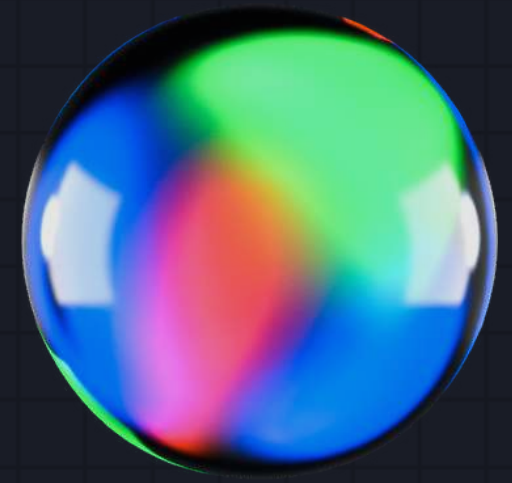
```

*Fig 4. Optimized build\_image function*



CODEBASE OPTIMIZATION II

`make_ray` optimization  
+  
render refactoring



## the make\_ray function

- The make\_ray function first creates a coordinate frame for the camera.
- This frame consists of the  $w$ ,  $u$ , and  $v$  vectors.
- the  $w$  vector points from the camera's center to its eye.
- the  $u$  vector points in the direction of the camera's up vector.
- the  $v$  vector points in the direction that is perpendicular to both  $w$  and  $u$

## the make\_ray function

- Two weights are calculated to determine the direction of the ray which is a vector that starts at the camera's eye and points towards the pixel coordinate. The direction of the ray is then normalized to unit length.
- The function returns a Ray object.
  - A ray object has two properties: a position and a direction
  - The position of the ray is the location of the camera's eye.
  - The direction of the ray is the direction that the ray is pointing.



## the make\_ray function

- To summarize, the make\_ray function takes a Scene as an input and a pixel coordinate tuple and returns a Ray object that represents the ray that intersects the pixel.

## make\_ray\_optimization and render refactoring

- Reuse computed values
  - The calculations for `fov_y_rad` and vectors `w`, `u`, `v` are constant for each pixel.
  - These calculations can be moved outside the loop of the render function.
  - `fov_y_rad` and vectors `w`, `u`, `v` are now passed as parameters to the `make_ray` function instead of being computed inside the `make_ray` function

## make\_ray\_optimization and render refactoring

- Eliminate locking mechanism
  - A Mutex and a separate lock are used to protect the shared 'pixels' vector. Each thread acquires the lock for each pixel, which introduces locking overhead.
  - Initialize 'pixels' vector directly with the computed pixel values.
- Other Rayon Functions
  - Utilize 'into\_par\_iter', 'flat\_map' and 'collect' functions from the Rayon crate for efficient concurrent execution of 'pixel' computations



```
fn make_ray(scene: &Scene, pixel_coords: (usize, usize)) -> Ray {  
    // Create coordinate frame  
    let w = (scene.camera.eye - &scene.camera.center).norm();  
    let u = scene.camera.up.cross(&w).norm();  
    let v = w.cross(&u);  
  
    let fov_y_rad = scene.camera.fovy.to_radians();  
    let weight_a = ((0.5 * fov_y_rad).tan() / (0.5 * (scene.img_height as f64))) * (((pixel_coords.1 as f64) + 0.5) - (0.5 * (scene.img_width as f64)));  
    let weight_b = ((0.5 * fov_y_rad).tan() / (0.5 * (scene.img_height as f64))) * ((0.5 * (scene.img_height as f64)) - (0.5 + (pixel_coords.0 as f64)));  
  
    let ray_dir = (u * weight_a + &(v * weight_b) - &w).norm();  
  
    Ray {  
        position: Point3 { point: scene.camera.eye.vec },  
        direction: ray_dir,  
    }  
}
```

Fig 5. Unoptimized make\_ray function

```
fn make_ray(scene: &Scene, pixel_coords: (usize, usize), u: &Vector3, v: &Vector3, w: &Vector3, fov_y_rad: f64) -> Ray {  
    let weight_a = ((0.5 * fov_y_rad).tan() / (0.5 * (scene.img_height as f64))) * (((pixel_coords.1 as f64) + 0.5) - (0.5 * (scene.img_width as f64)));  
    let weight_b = ((0.5 * fov_y_rad).tan() / (0.5 * (scene.img_height as f64))) * ((0.5 * (scene.img_height as f64)) - (0.5 + (pixel_coords.0 as f64)));  
  
    let ray_dir = (*u * weight_a + &(*v * weight_b) - &w).norm();  
  
    Ray {  
        position: Point3 { point: scene.camera.eye.vec },  
        direction: ray_dir,  
    }  
}
```

*Fig 6. Optimized make\_ray function*



# METHODOLOGY: CODEBASE OPTIMIZATION II

```
pub fn render(scene: &Scene) -> Vec<u8> {
    let mut pixels: Vec<u8> = Vec::new();
    pixels.resize(scene.img_width * scene.img_height * 3, 0u8);

    let pixels_mutex = Mutex::new(pixels);

    (0..scene.img_height).into_par_iter().for_each(|i| {
        (0..scene.img_width).into_par_iter().for_each(|j| {
            // Pass through ray in center of (i, j) pixel
            let ray = make_ray(scene, (i, j));
            let start_idx = (i * scene.img_width + j) * 3;

            // Intersection test with scene
            if let Some(id) = intersect_scene_from_view(ray, scene) {
                // Use get_color_recursive to get reflections
                //let pix_color = get_color(ray, scene, id, &scene.lights);
                let mut pixels = pixels_mutex.lock().unwrap();

                let pix_color = get_color_recursive(ray, &scene, id, 0);

                pixels[start_idx + 0] = (255.0 * pix_color[0]) as u8;
                pixels[start_idx + 1] = (255.0 * pix_color[1]) as u8;
                pixels[start_idx + 2] = (255.0 * pix_color[2]) as u8;
            }
            else {
                // Color all pixels black
                let mut pixels = pixels_mutex.lock().unwrap();

                pixels[start_idx + 0] = 0u8;
                pixels[start_idx + 1] = 0u8;
                pixels[start_idx + 2] = 0u8;
            }
        });
    });

    let pixels = pixels_mutex.lock().unwrap().to_owned();
    pixels
}
```

*Recall Fig 2. Optimized render function*

```
pub fn render(scene: &Scene) -> Vec<u8> {
    let w = (scene.camera.eye - &scene.camera.center).norm();
    let u = scene.camera.up.cross(&w).norm();
    let v = w.cross(&u);

    let fov_y_rad = scene.camera.fovy.to_radians();

    let pixels: Vec<u8> = (0..scene.img_width * scene.img_height)
        .into_par_iter()
        .flat_map(|idx| {
            let i = idx / scene.img_width;
            let j = idx % scene.img_width;
            let ray = make_ray(scene, (i, j), &u, &v, &w, fov_y_rad)
            let start_idx = idx * 3;

            if let Some(id) = intersect_scene_from_view(ray, scene) {
                let pix_color = get_color_recursive(ray, &scene, id,
0);
                vec![
                    (255.0 * pix_color[0]) as u8,
                    (255.0 * pix_color[1]) as u8,
                    (255.0 * pix_color[2]) as u8,
                ]
            } else {
                vec![0u8, 0u8, 0u8]
            }
        })
        .collect();

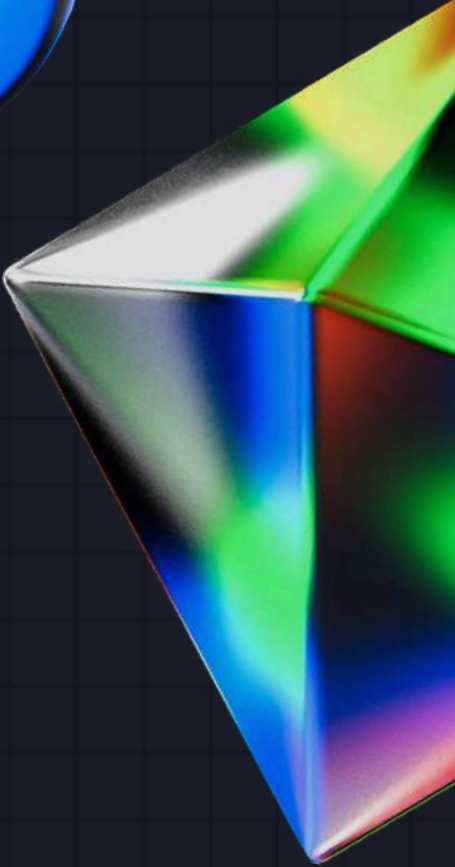
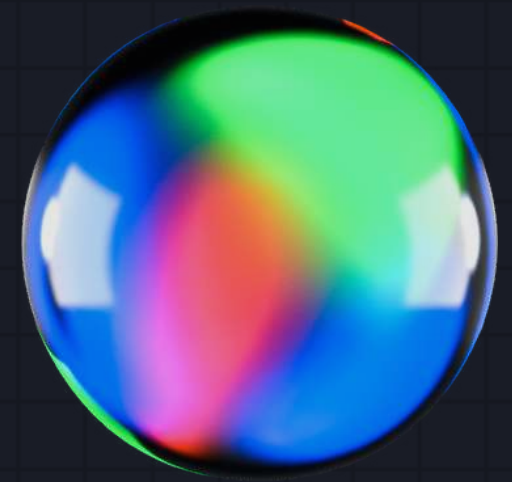
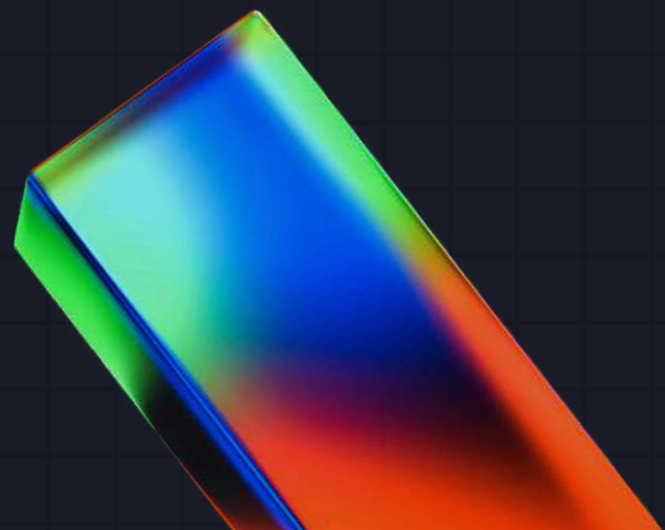
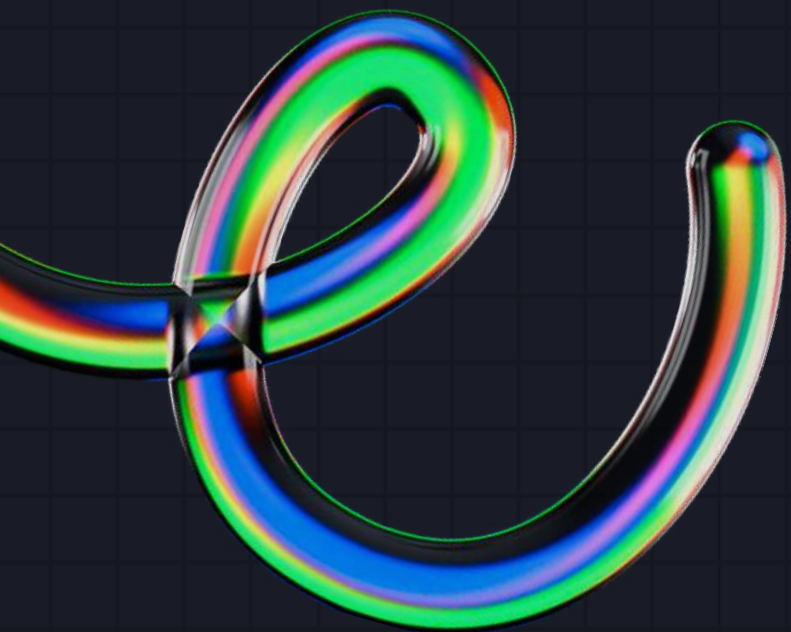
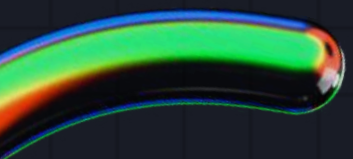
    pixels
}
```

*Fig 7. Refactored render function*



CODEBASE OPTIMIZATION III

# refactored functions involving iterating over maps



## iterating over maps

- The initial code starts with implementing a specific trait for the "ops" module, and the output type is set accordingly.
- The method is then added, which involves `.map()` and/or `.zip()` functions, and the output type is also set.
  - `.map()`– transforms elements of an iterator/sequence, and is stored until the transformed iterator is consumed
  - `.zip()`– combines two iterators
- Note that there are several implementations in the code that utilized iterating over maps

## iterating over maps optimization

- The optimized code begins similarly as the initial.
- The body of the methods is optimized by **avoiding iterating over maps and combining iterators**, which are both relatively slow.
- Instead, the outputs are obtained by **explicitly constructing them via list comprehensions**, thus eliminating the need for iterator traversals since elements can be directly accessed via indexing.



## iterating over maps optimization

- Other reasons for the significant improvements are the reduced function calls and memory allocations and deallocations (from `.map()`).

```
impl ops::Add<&Point3> for Point3 {  
    type Output = Vector3;  
  
    fn add(self, other: &Point3) -> Vector3 {  
        Vector3 {  
            vec: [  
                self.point[0] + other.point[0],  
                self.point[1] + other.point[1],  
                self.point[2] + other.point[2],  
            ],  
        }  
    }  
}
```

Fig 8. Unoptimized code snippet from point3.rs (with mapping)

```
impl std::ops::Mul<f64> for Point3 {  
    type Output = Point3;  
  
    fn mul(self, other: f64) -> Point3 {  
        let point = [  
            self.point[0] * other,  
            self.point[1] * other,  
            self.point[2] * other,  
        ];  
  
        Point3 { point }  
    }  
}
```

Fig 9. Optimized code snippet from point3.rs (without mapping)



## METHODOLOGY: CODEBASE OPTIMIZATION III

```
impl Vector3 {
    pub fn dot(&self, other: &Vector3) -> f64 {
        let mut result = 0.0;
        result += self.vec[0] * other.vec[0];
        result += self.vec[1] * other.vec[1];
        result += self.vec[2] * other.vec[2];
        result
    }

    pub fn cross(&self, other: &Vector3) -> Vector3 {
        Vector3 {
            vec: [
                self.vec[1] * other.vec[2] - self.vec[2] * other.vec[1],
                self.vec[2] * other.vec[0] - self.vec[0] * other.vec[2],
                self.vec[0] * other.vec[1] - self.vec[1] * other.vec[0],
            ],
        }
    }

    pub fn len(&self) -> f64 {
        let squared_sum = self.vec[0] * self.vec[0]
            + self.vec[1] * self.vec[1]
            + self.vec[2] * self.vec[2];
        squared_sum.sqrt()
    }

    pub fn norm(&self) -> Vector3 {
        let vec_len = self.len();
        let normalized = Vector3 {
            vec: [
                self.vec[0] / vec_len,
                self.vec[1] / vec_len,
                self.vec[2] / vec_len,
            ],
        };
        normalized
    }
}
```

Fig 10. Unoptimized code snippet from vector3.rs (with mapping)

```
impl ops::Add<&Vector3> for Vector3 {
    type Output = Vector3;

    fn add(self, other: &Vector3) -> Vector3 {
        Vector3 {
            vec: [
                self.vec[0] + other.vec[0],
                self.vec[1] + other.vec[1],
                self.vec[2] + other.vec[2],
            ],
        }
    }
}
```

Fig 11. Optimized code snippet from vector3.rs (without mapping)

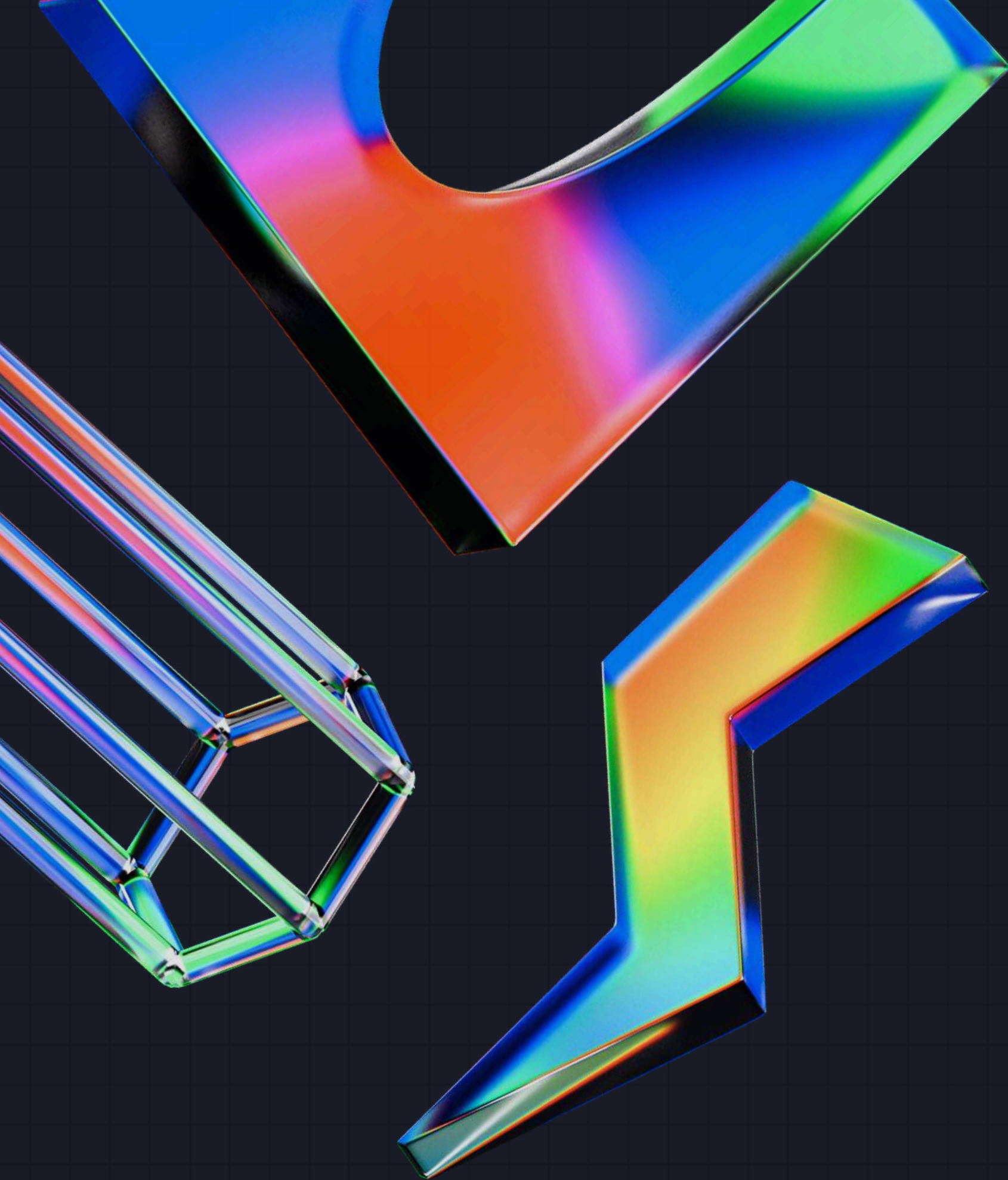


```
impl ops::Mul<&Vector3> for Vector3 {  
    type Output = Vector3;  
  
    fn mul(self, other: &Vector3) -> Vector3 {  
        Vector3 {  
            vec: [  
                self.vec[0] * other.vec[0],  
                self.vec[1] * other.vec[1],  
                self.vec[2] * other.vec[2],  
            ],  
        }  
    }  
}
```

Fig 12. Unoptimized code snippet from vector3.rs (with mapping)

```
impl ops::Mul<f64> for Vector3 {  
    type Output = Vector3;  
  
    fn mul(self, other: f64) -> Vector3 {  
        Vector3 {  
            vec: [  
                self.vec[0] * other,  
                self.vec[1] * other,  
                self.vec[2] * other,  
            ],  
        }  
    }  
}
```

Fig 13. Optimized code snippet from vector3.rs (without mapping)

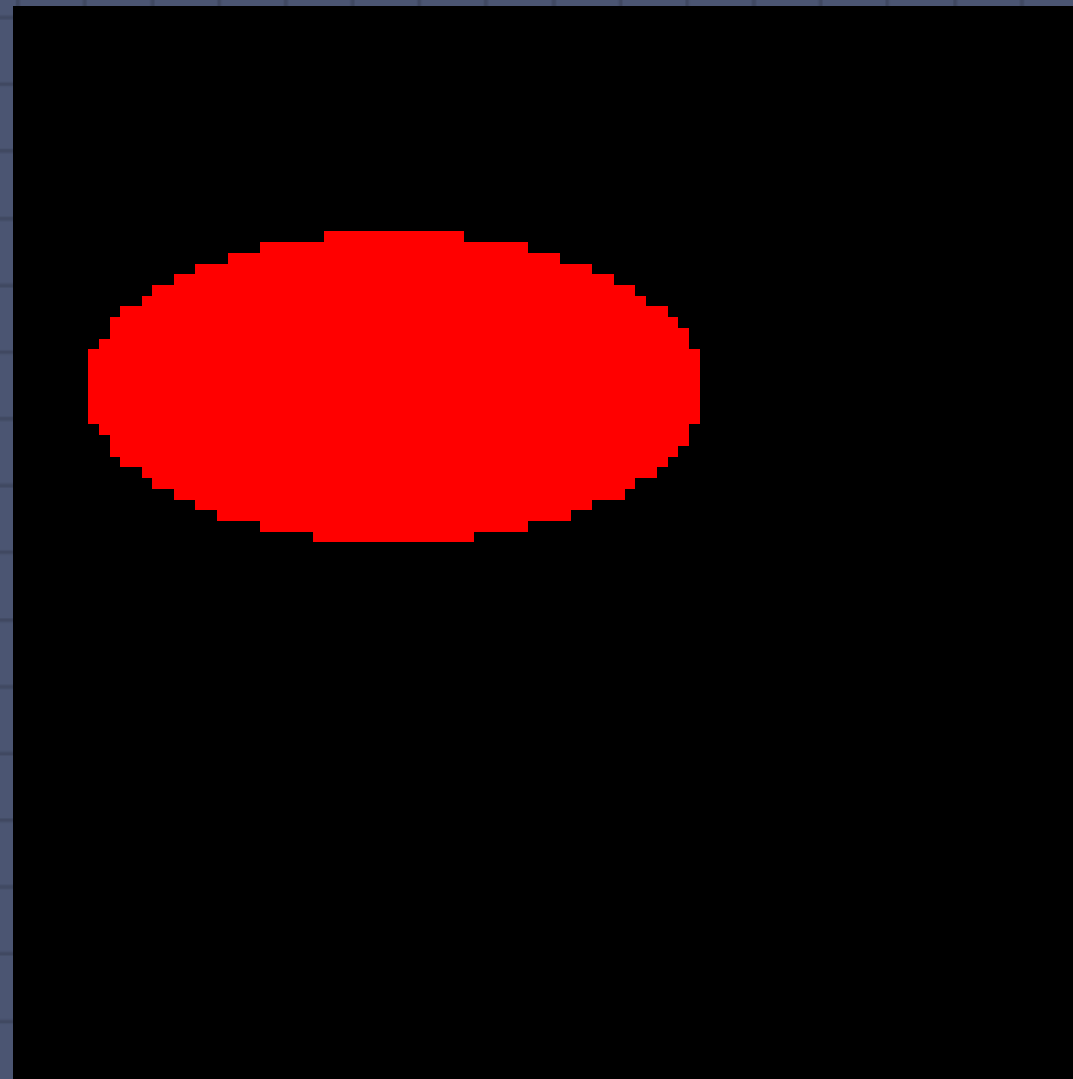


# RESULTS AND DISCUSSION



## RESULTS: RENDERED IMAGES

The following images show the results of the ray tracer program

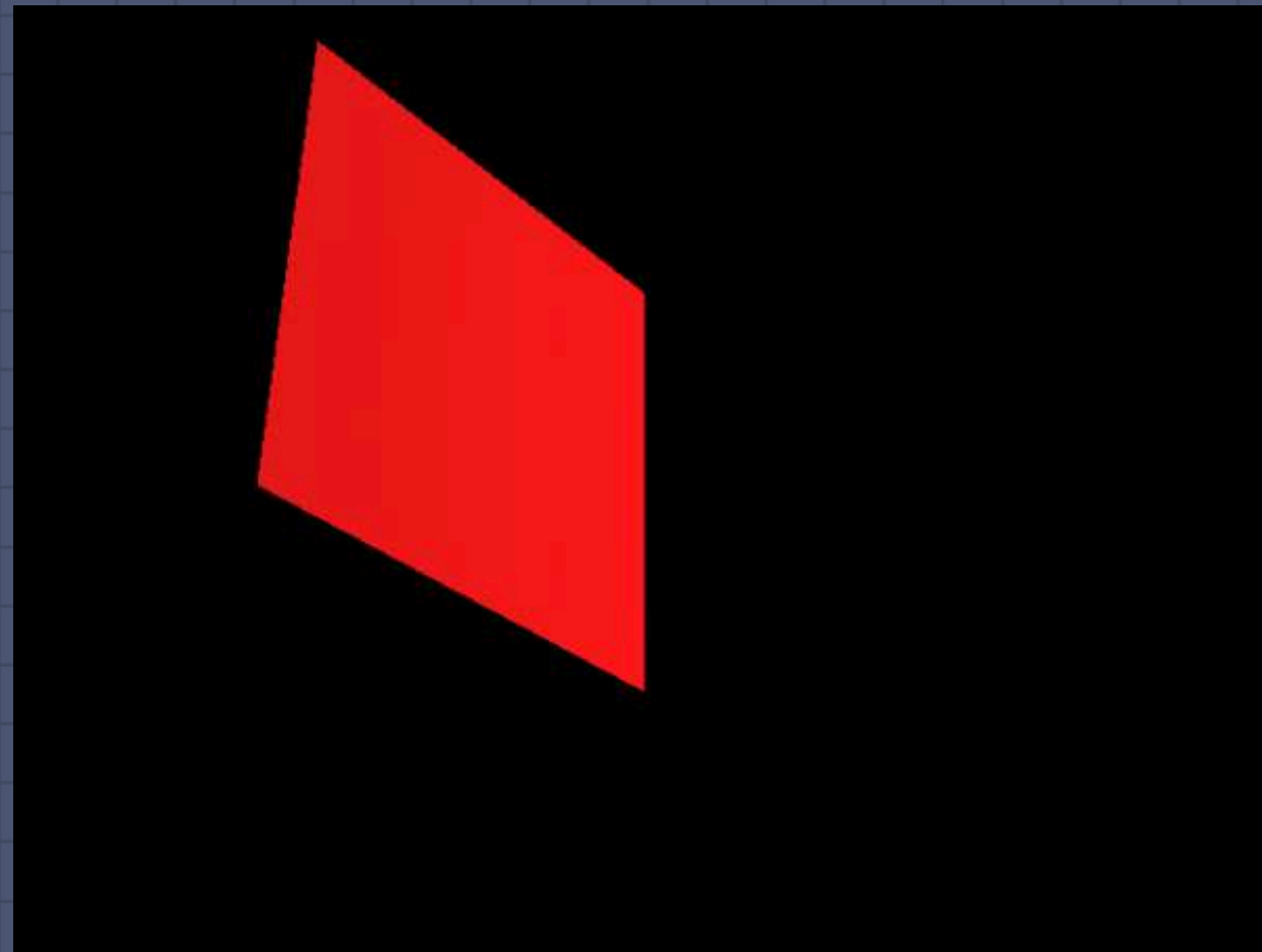


*Fig 14. Rendered image from Scene00*



## RESULTS: RENDERED IMAGES

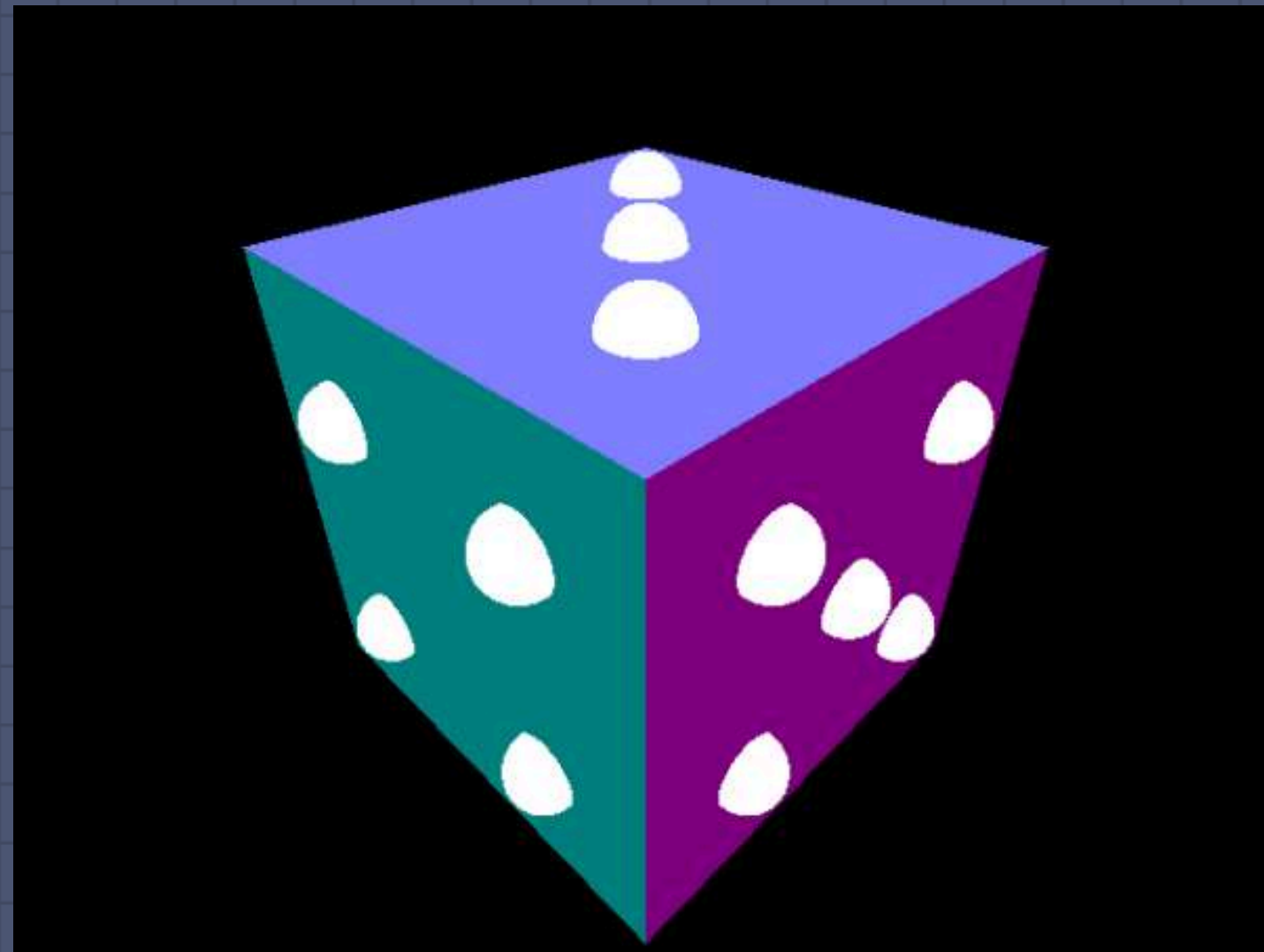
The following images show the results of the ray tracer program



*Fig 15. Rendered image from Scene01*

## RESULTS: RENDERED IMAGES

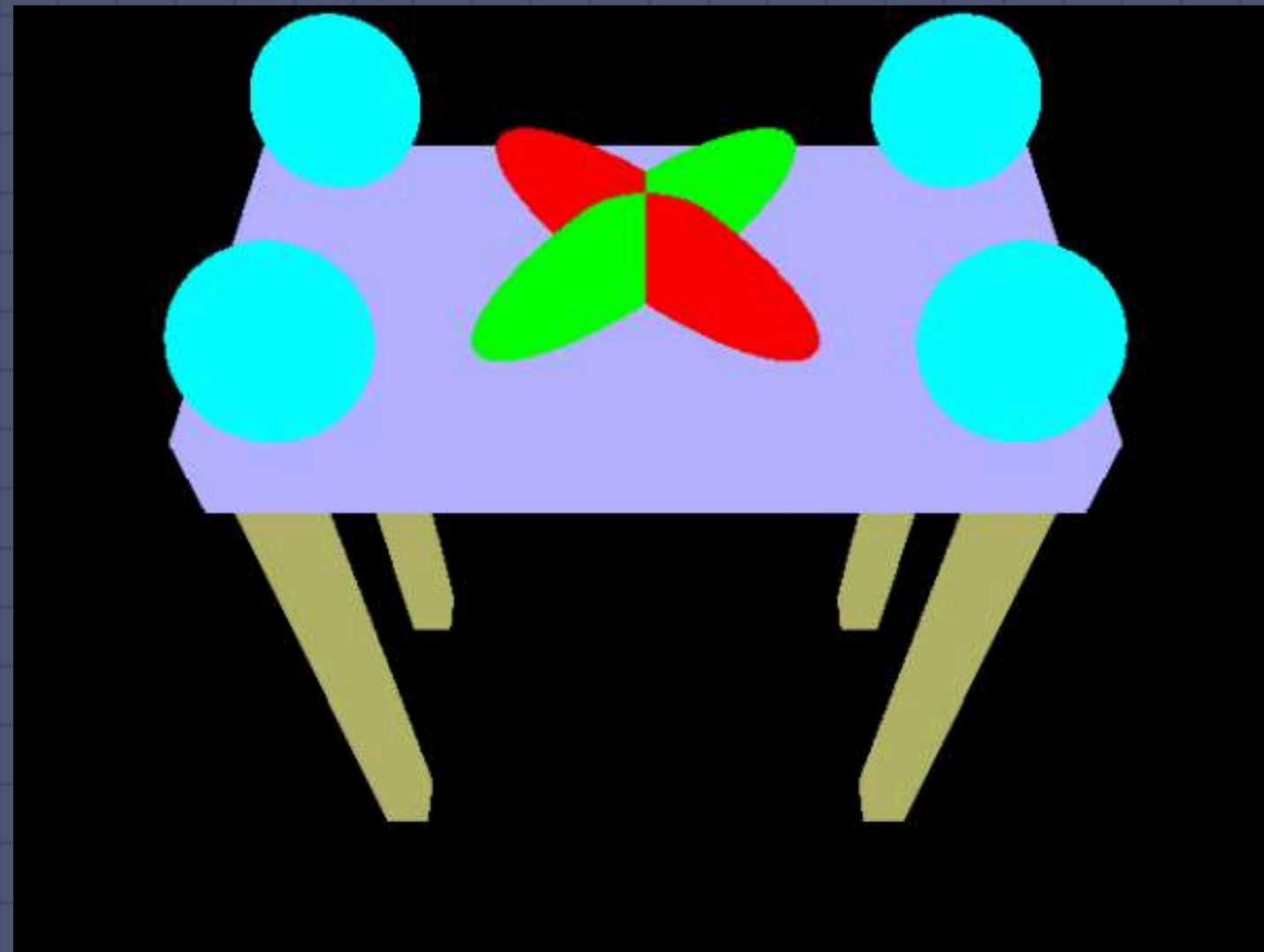
The following images show the results of the ray tracer program



*Fig 16. Rendered image from Scene02*

## RESULTS: RENDERED IMAGES

The following images show the results of the ray tracer program

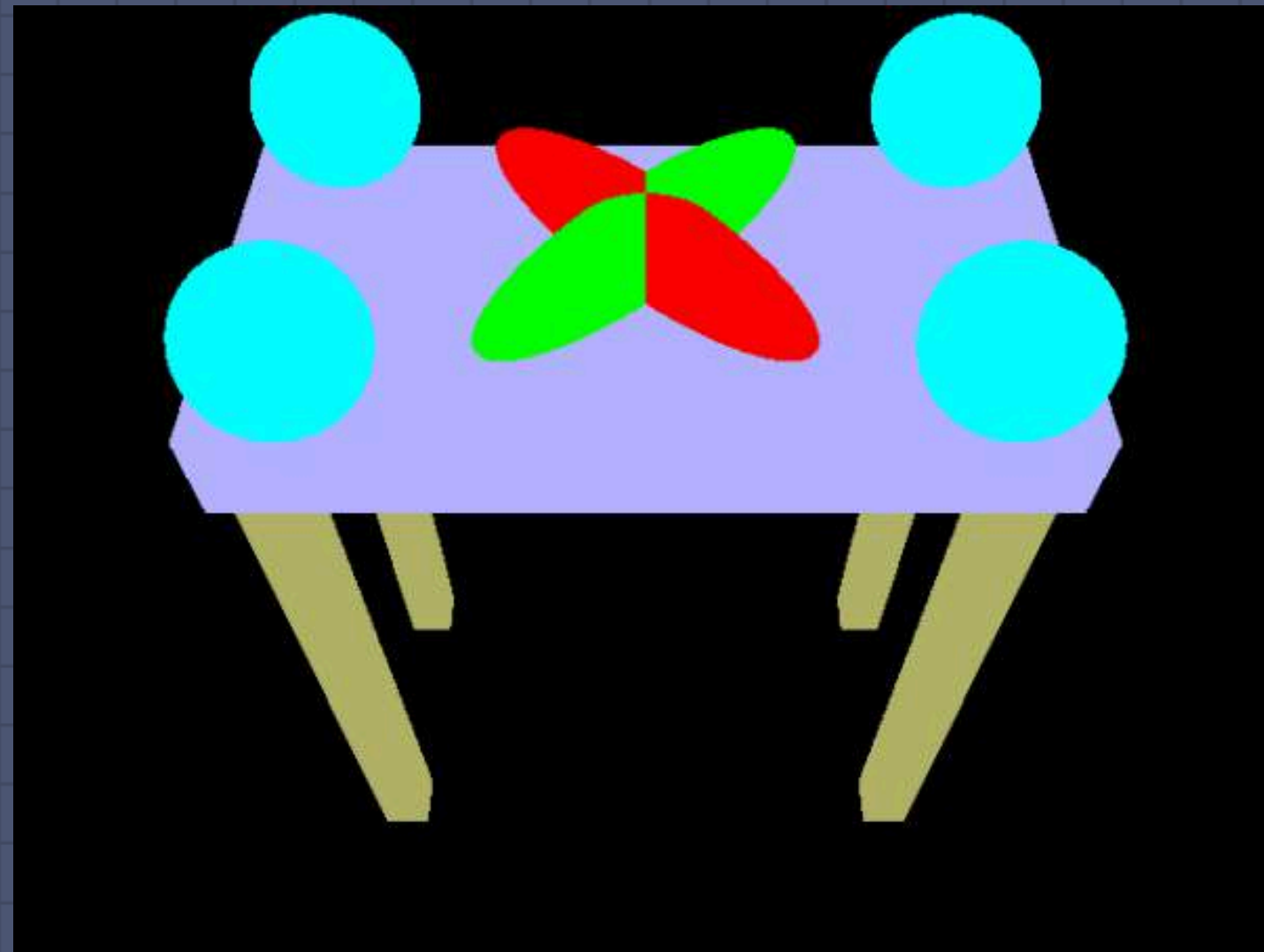


*Fig 17. Rendered image from Scene03*



## RESULTS: RENDERED IMAGES

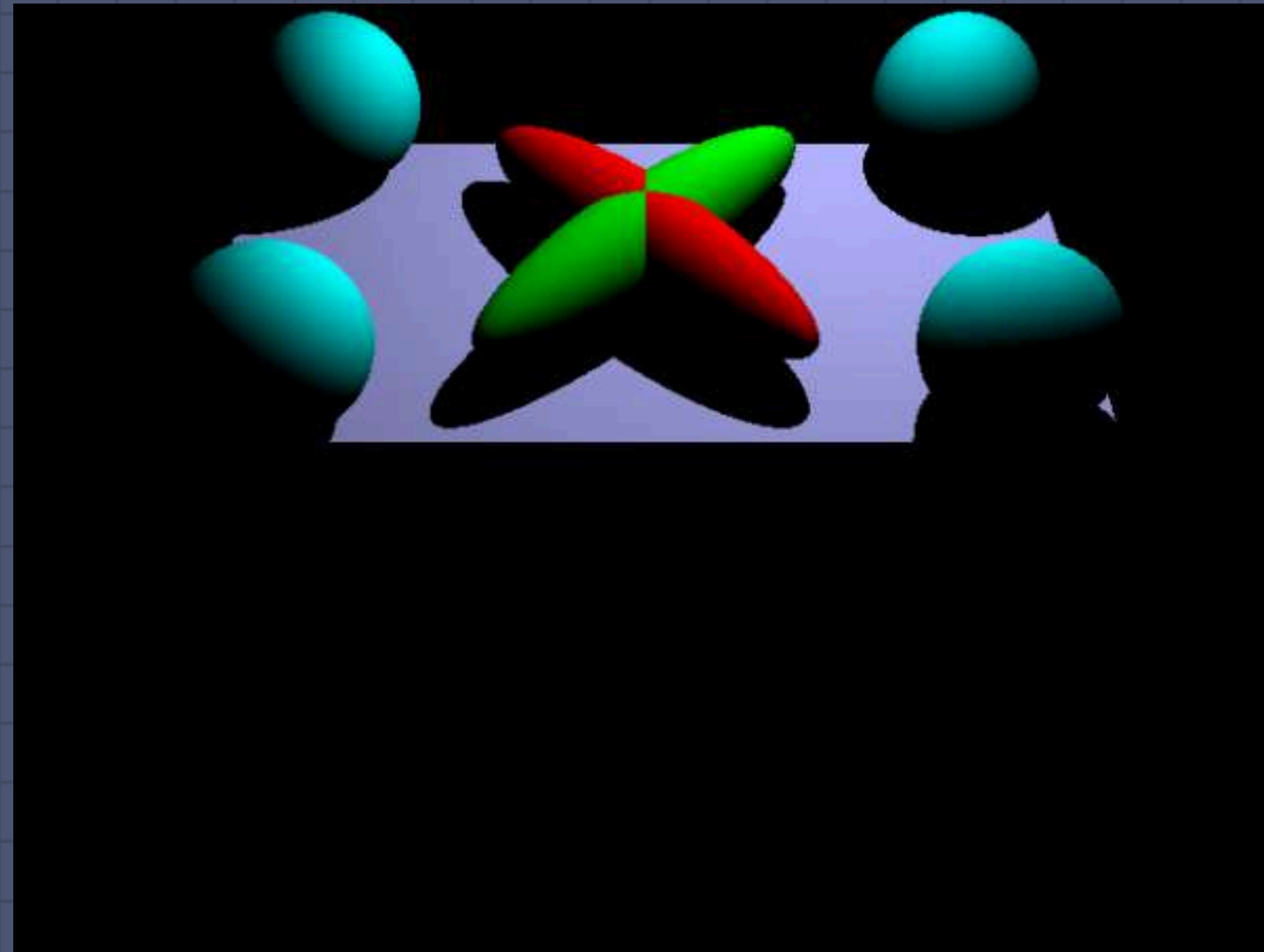
The following images show the results of the ray tracer program



*Fig 18. Rendered image from Scene04 – Ambient*

## RESULTS: RENDERED IMAGES

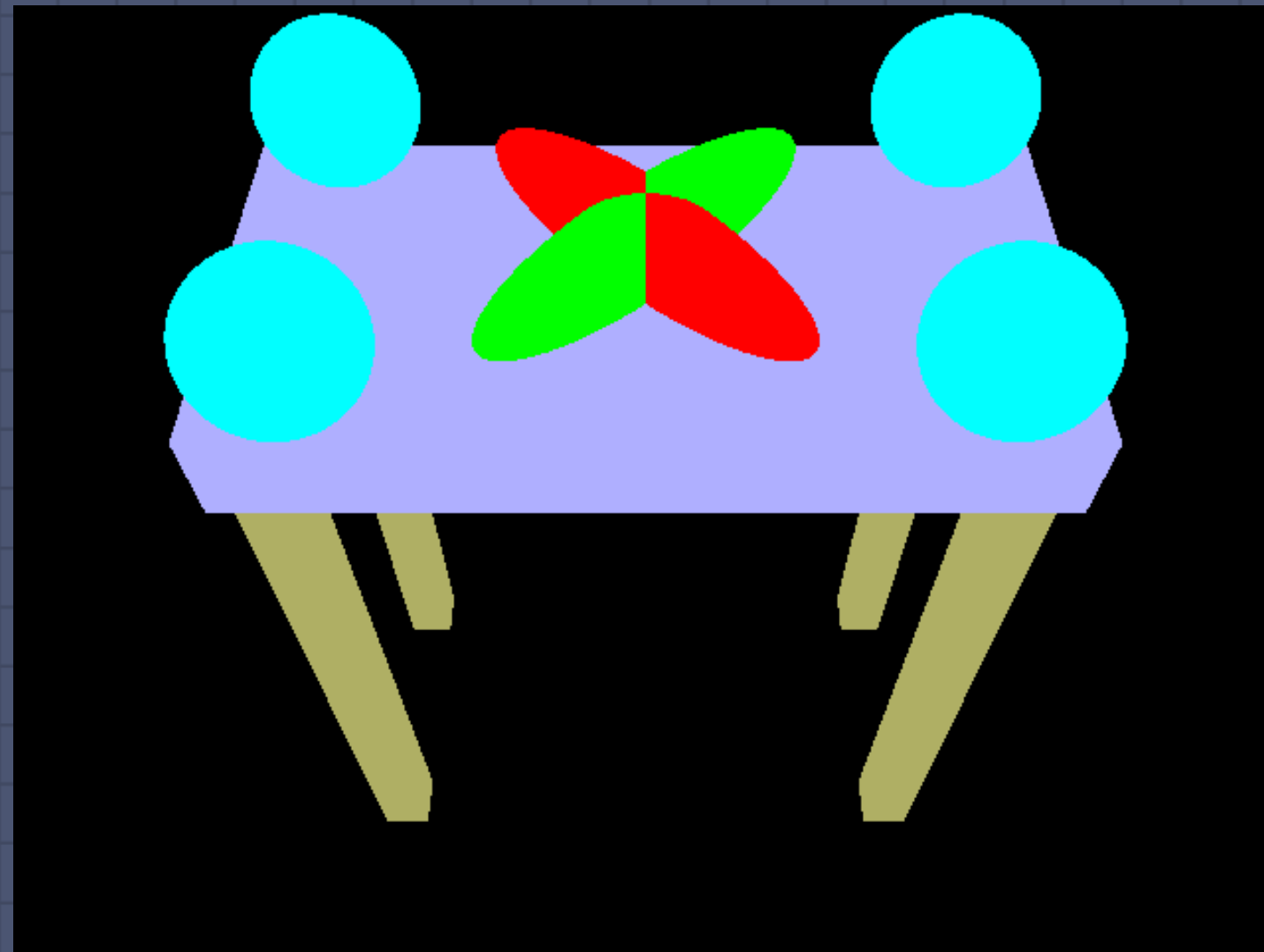
The following images show the results of the ray tracer program



*Fig 19. Rendered image from Scene04 – Diffuse*

## RESULTS: RENDERED IMAGES

The following images show the results of the ray tracer program

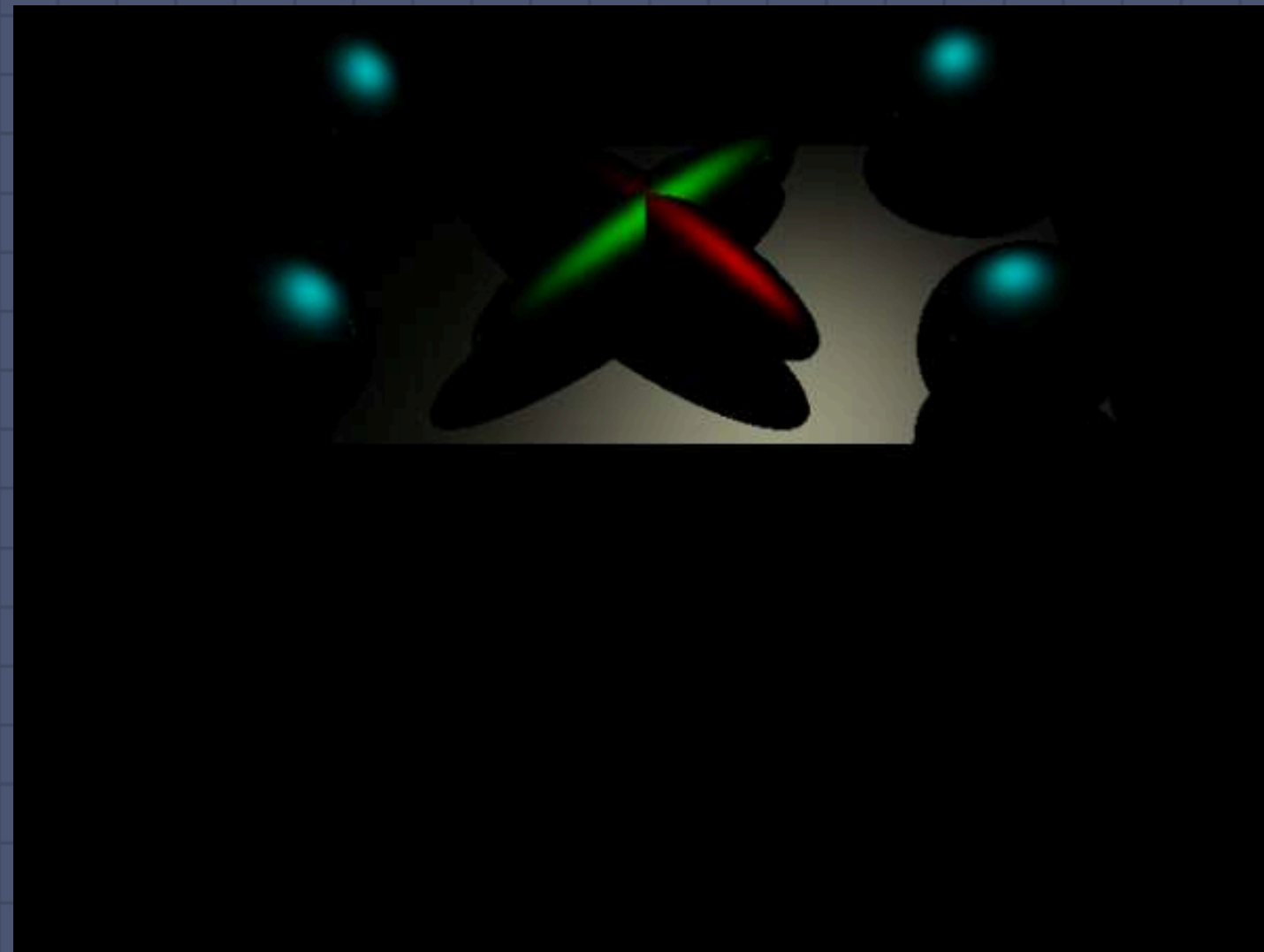


*Fig 20. Rendered image from Scene04 – Emission*



## RESULTS: RENDERED IMAGES

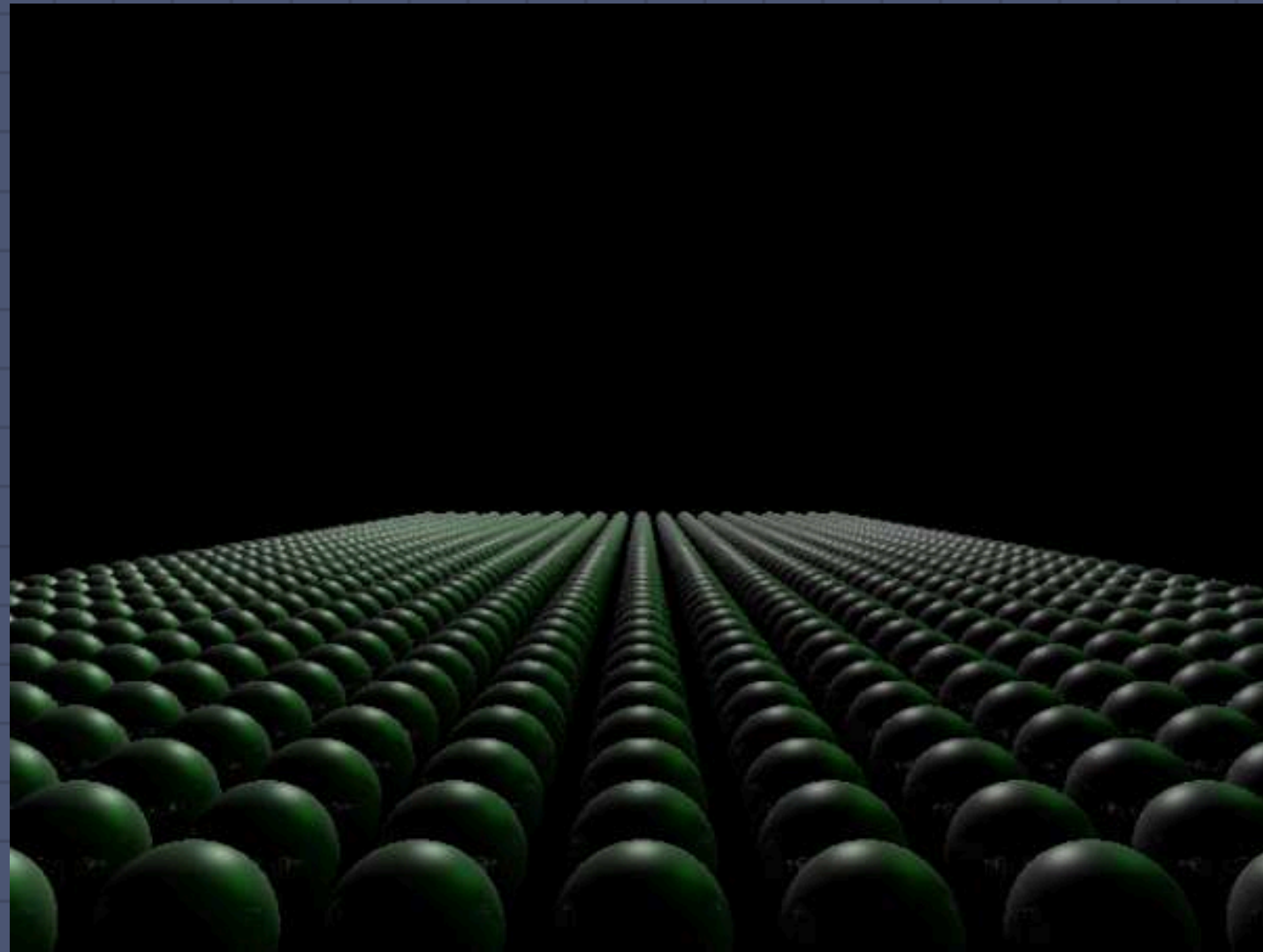
The following images show the results of the ray tracer program



*Fig 21. Rendered image from Scene04 – Specular*

## RESULTS: RENDERED IMAGES

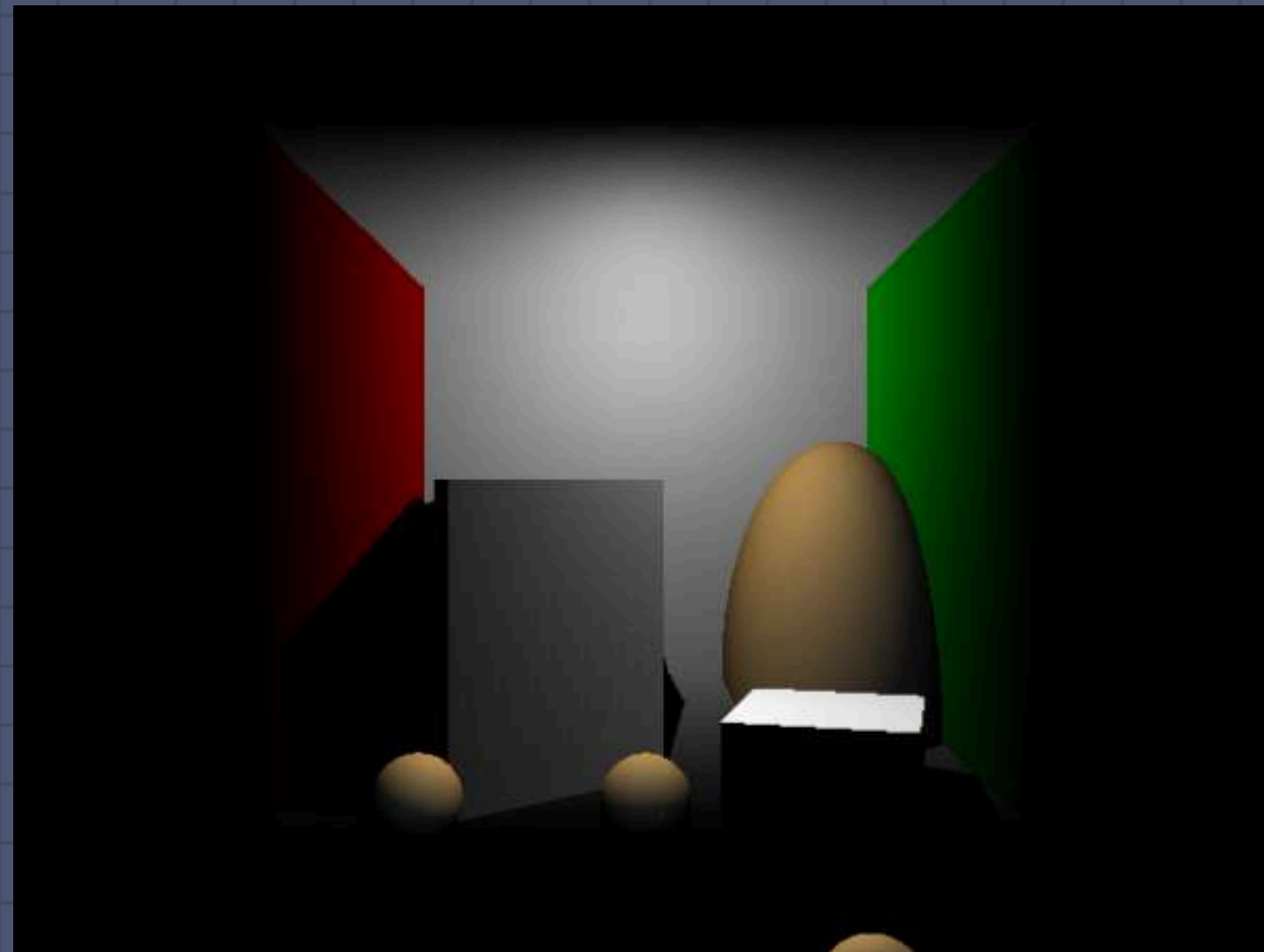
The following images show the results of the ray tracer program



*Fig 22. Rendered image from Scene05*

## RESULTS: RENDERED IMAGES

The following images show the results of the ray tracer program



*Fig 23. Rendered image from Scene06*



## RESULTS: RENDERED IMAGES

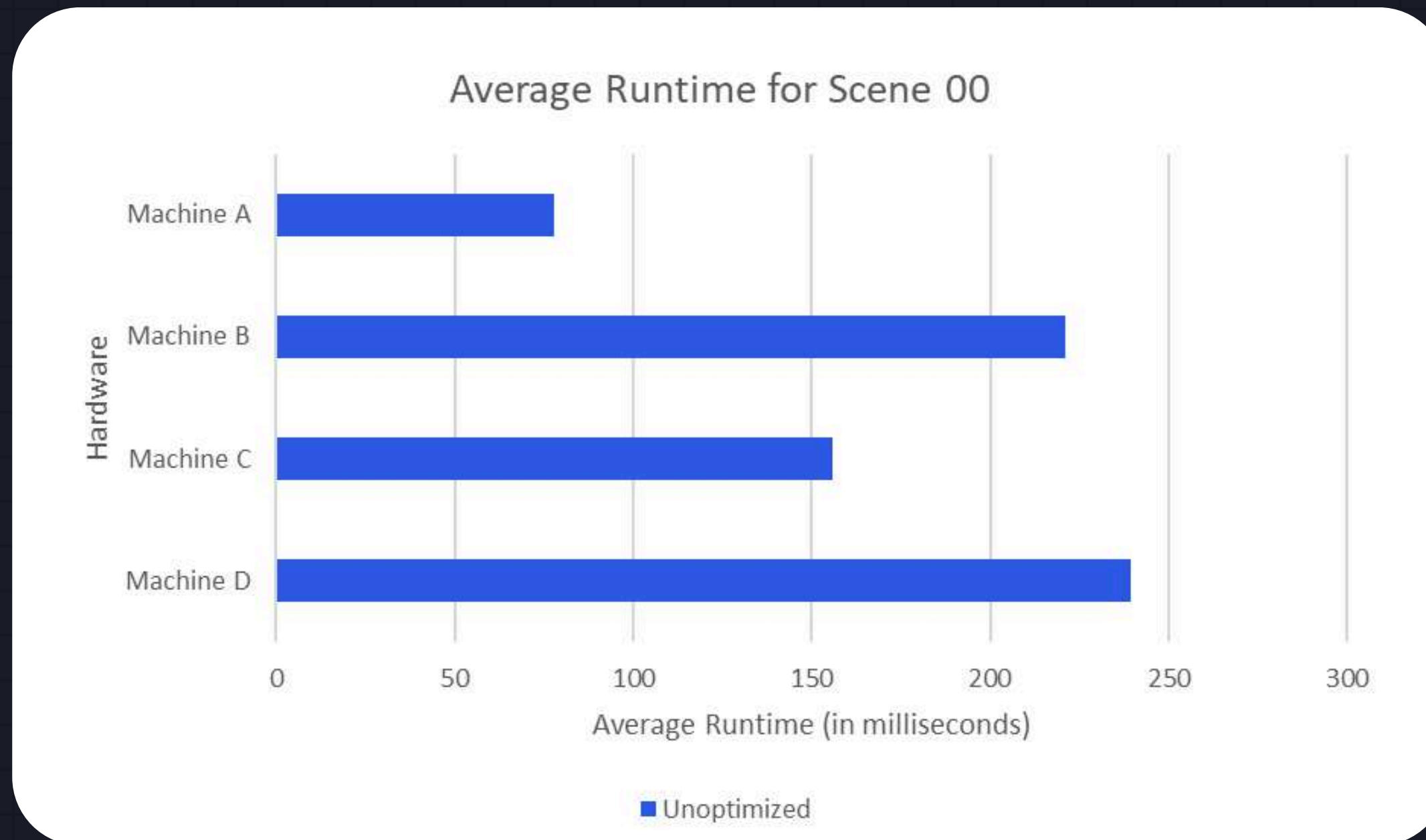
The following images show the results of the ray tracer program



*Fig 24. Rendered image from Scene07*

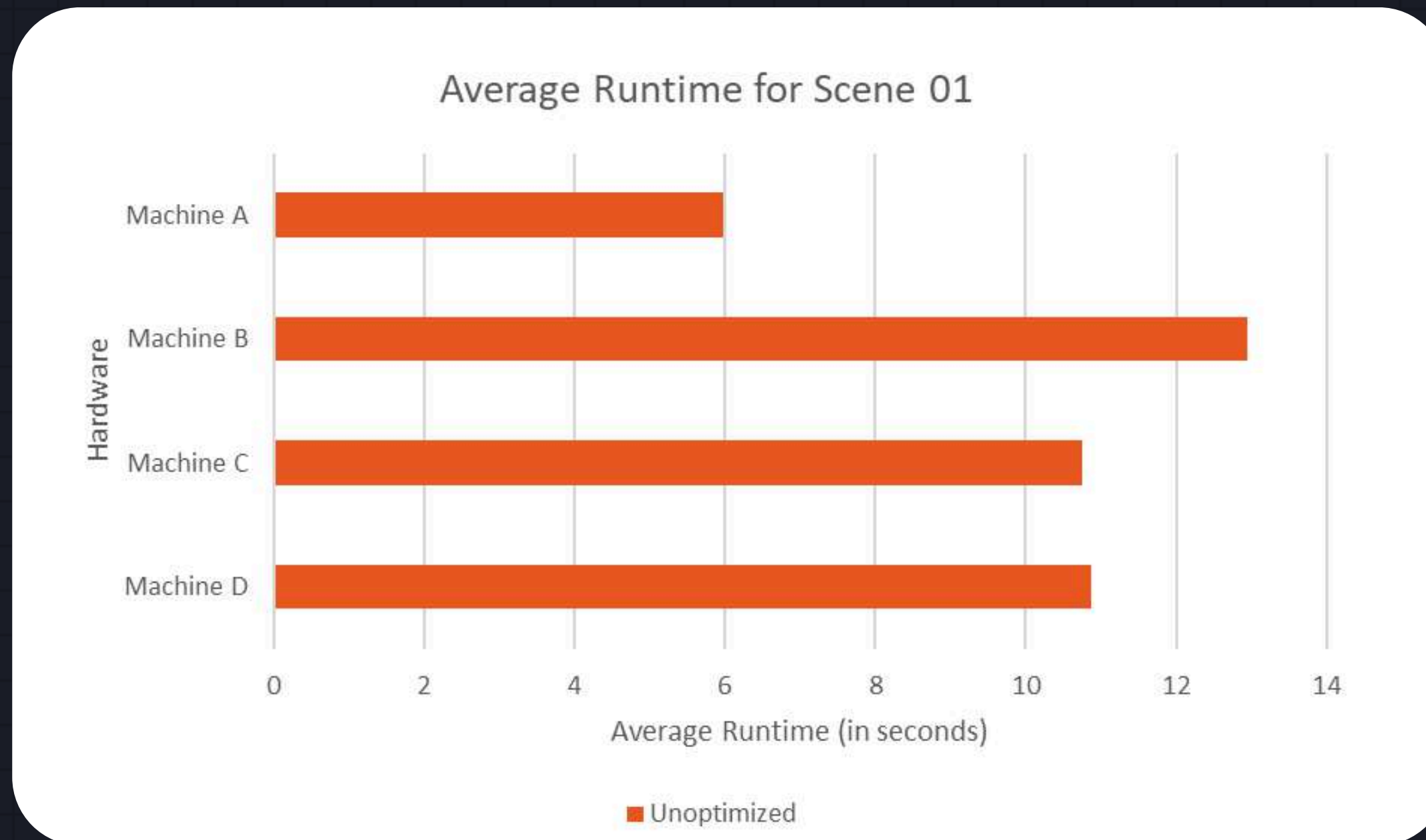
# RESULTS: UNOPTIMIZED IMPLEMENTATION

Summarized in the following graphs are the runtimes of the unoptimized ray tracer program across different hardware



# RESULTS: UNOPTIMIZED IMPLEMENTATION

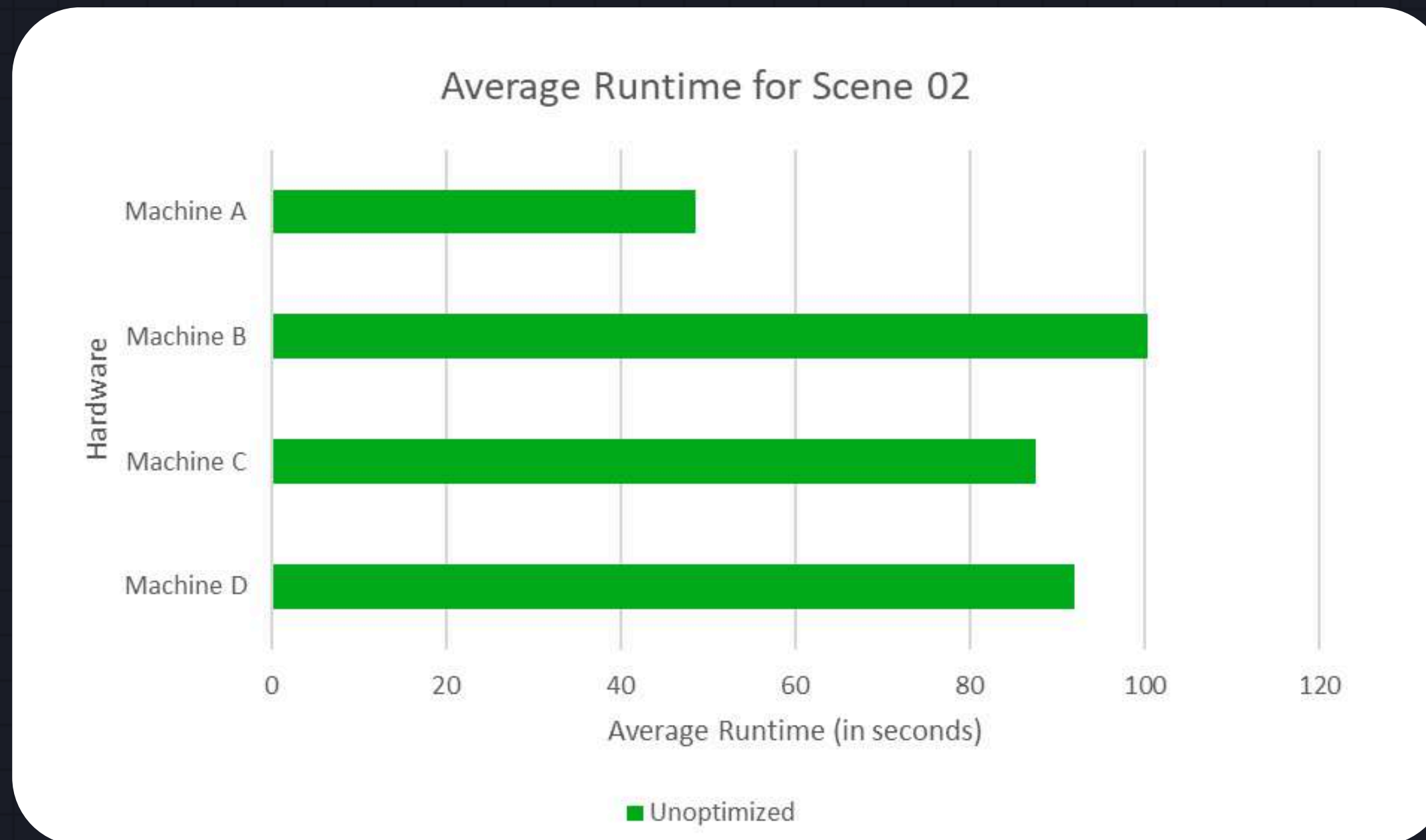
Summarized in the following graphs are the runtimes of the unoptimized ray tracer program across different hardware





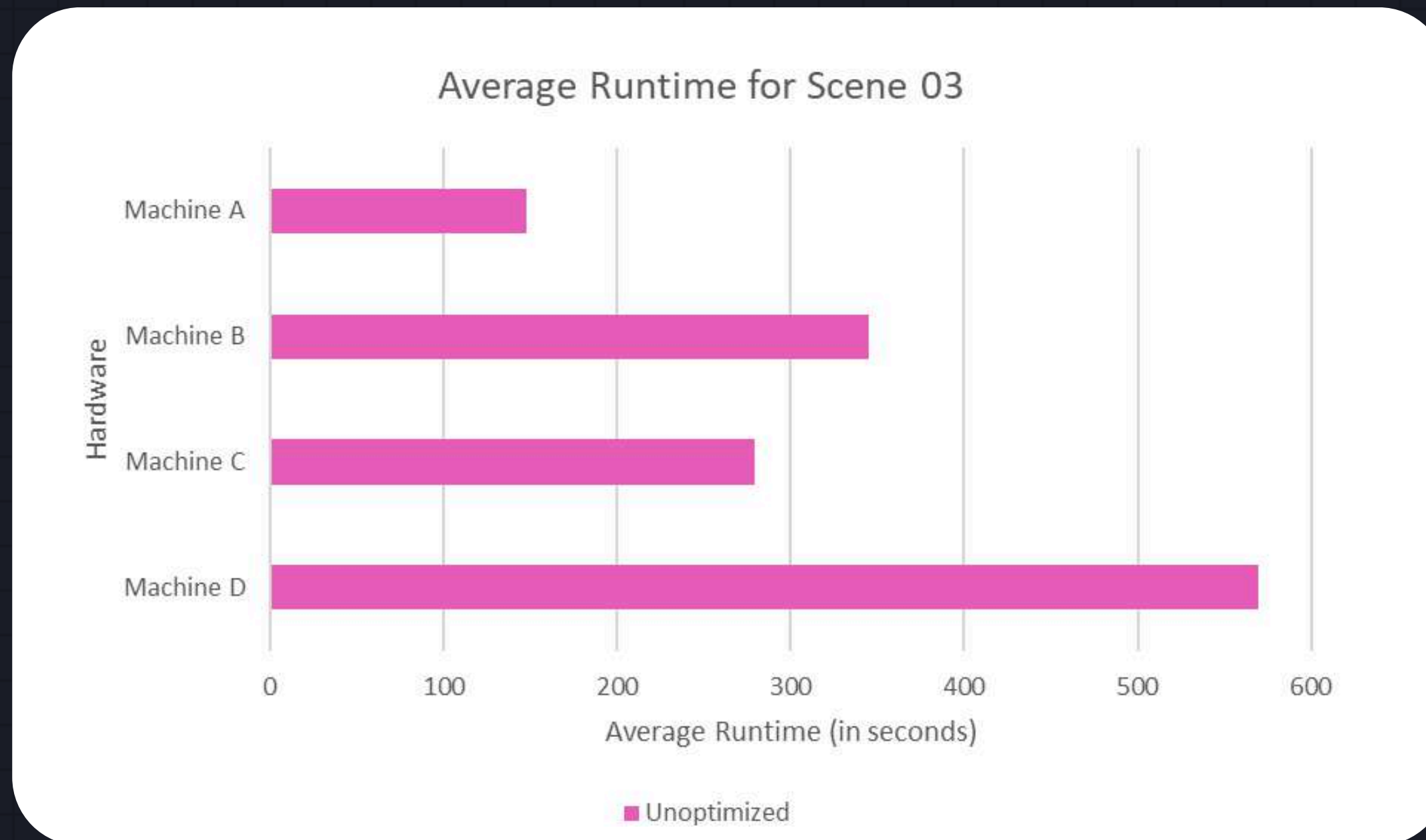
# RESULTS: UNOPTIMIZED IMPLEMENTATION

Summarized in the following graphs are the runtimes of the unoptimized ray tracer program across different hardware



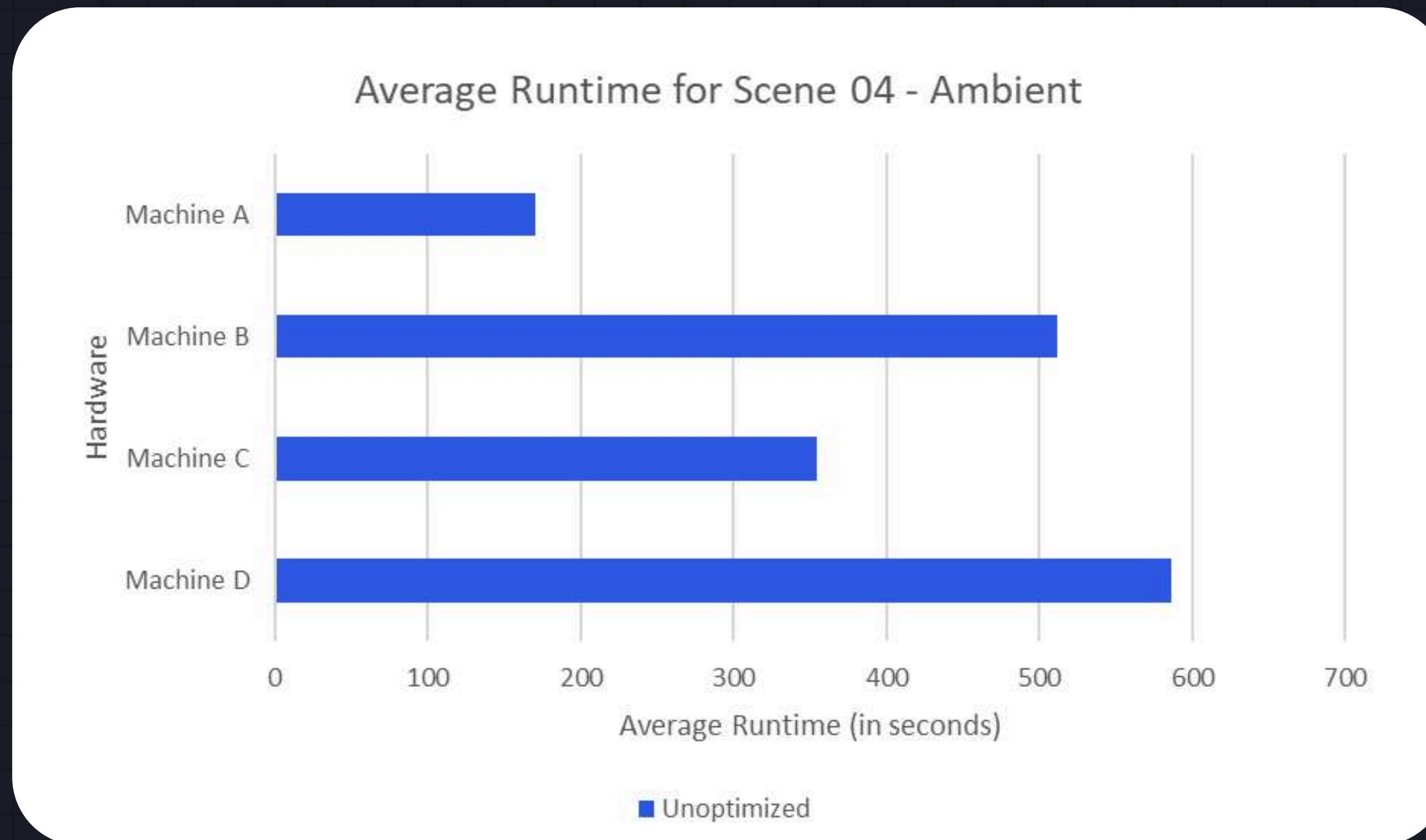
# RESULTS: UNOPTIMIZED IMPLEMENTATION

Summarized in the following graphs are the runtimes of the unoptimized ray tracer program across different hardware



# RESULTS: UNOPTIMIZED IMPLEMENTATION

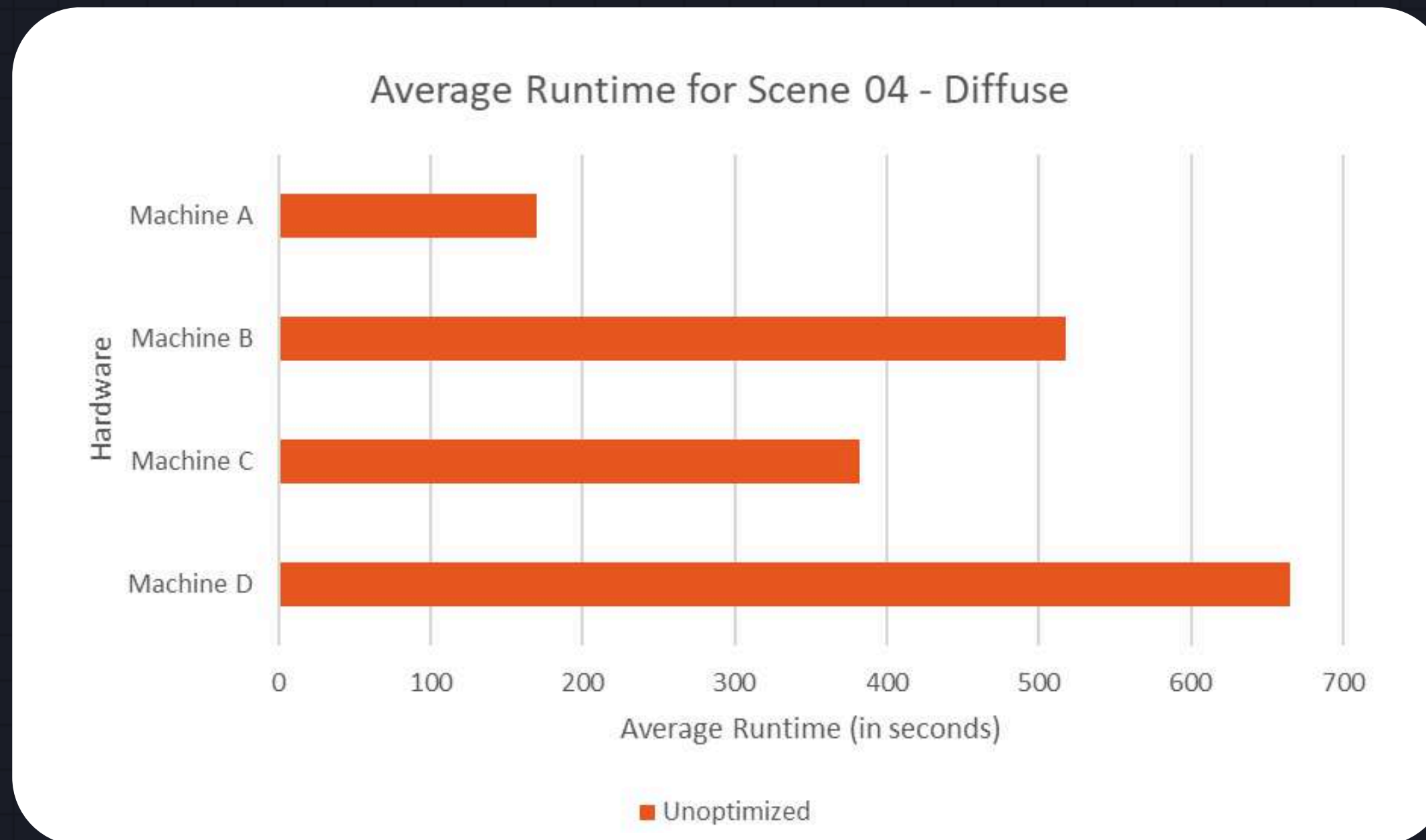
Summarized in the following graphs are the runtimes of the unoptimized ray tracer program across different hardware





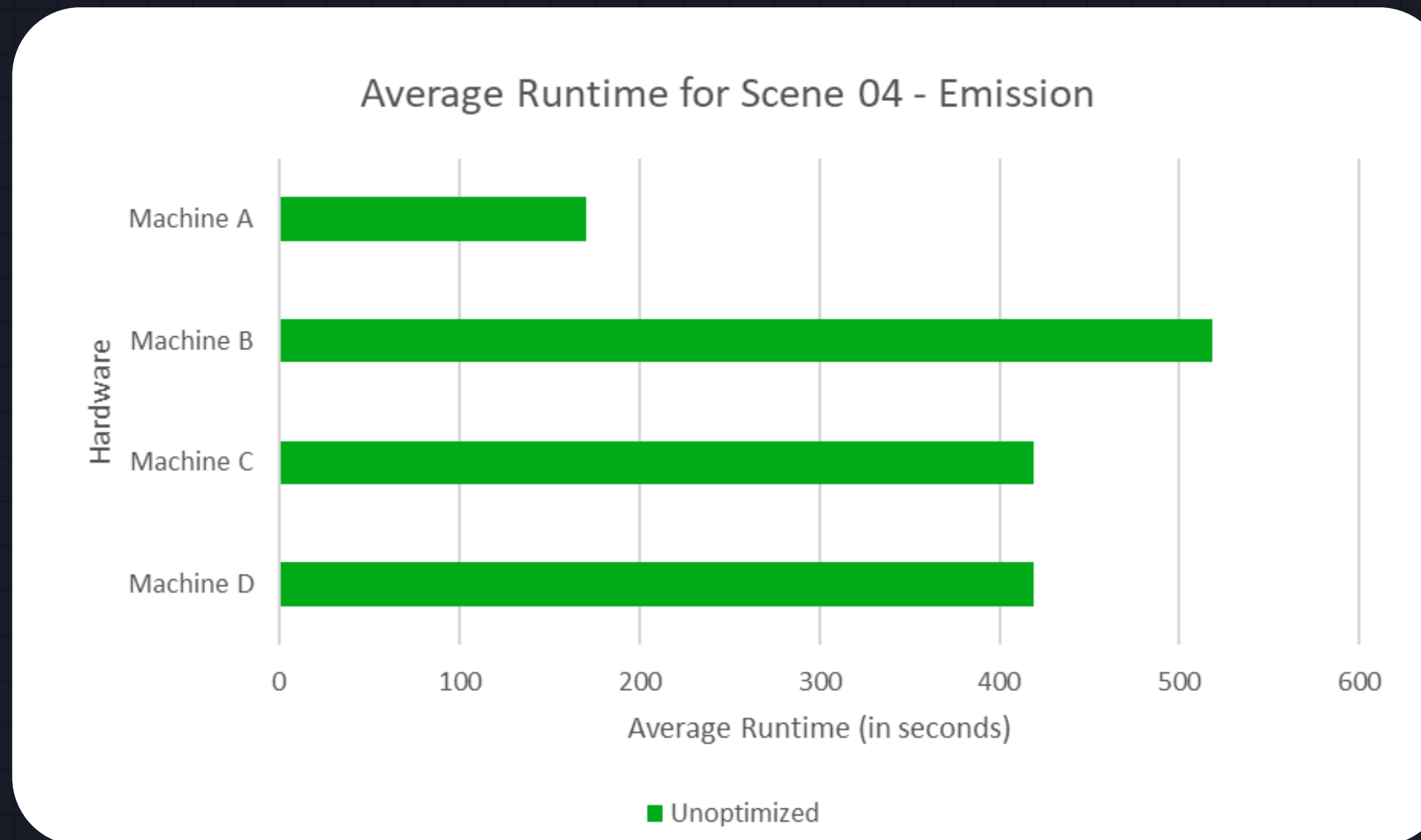
# RESULTS: UNOPTIMIZED IMPLEMENTATION

Summarized in the following graphs are the runtimes of the unoptimized ray tracer program across different hardware



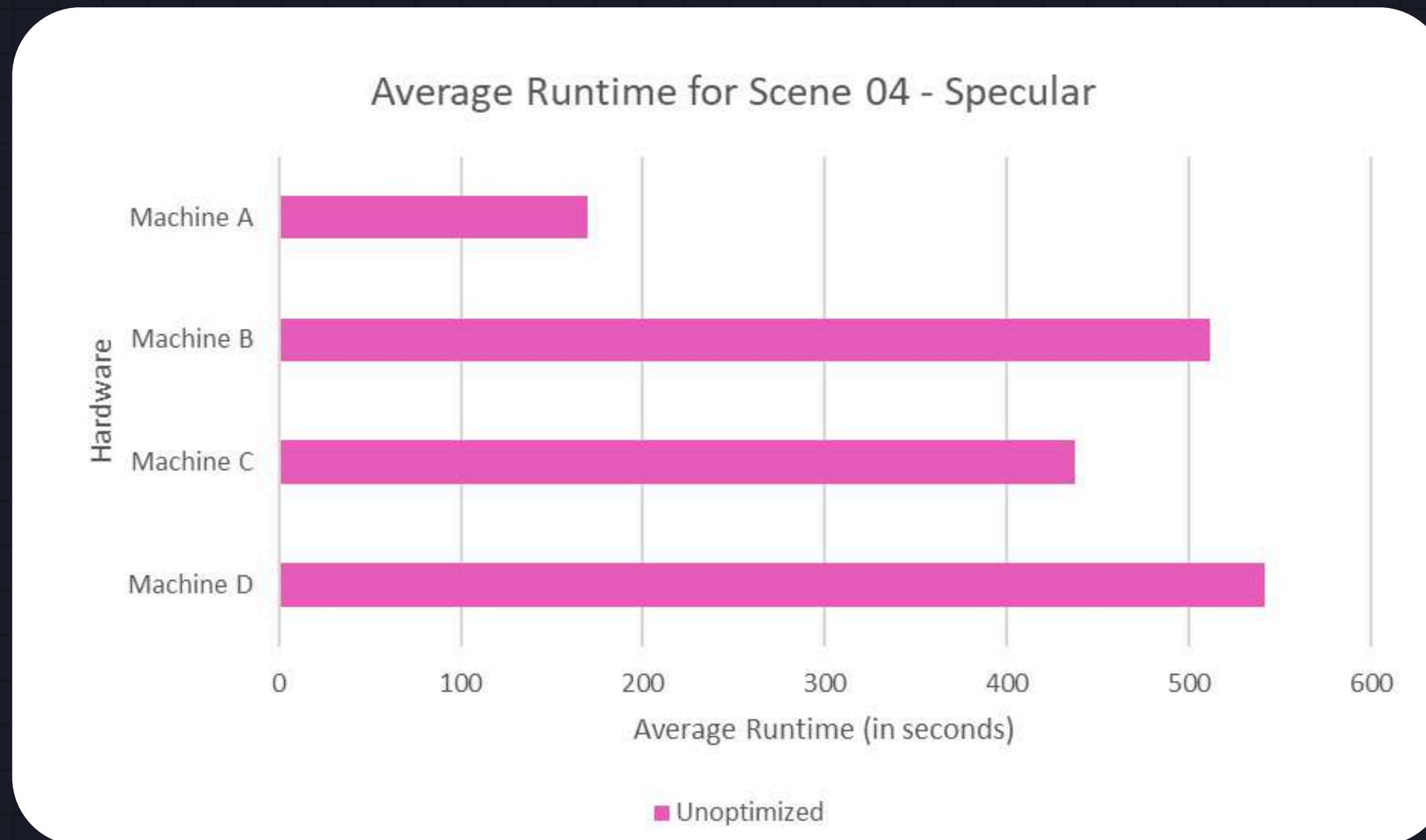
# RESULTS: UNOPTIMIZED IMPLEMENTATION

Summarized in the following graphs are the runtimes of the unoptimized ray tracer program across different hardware



# RESULTS: UNOPTIMIZED IMPLEMENTATION

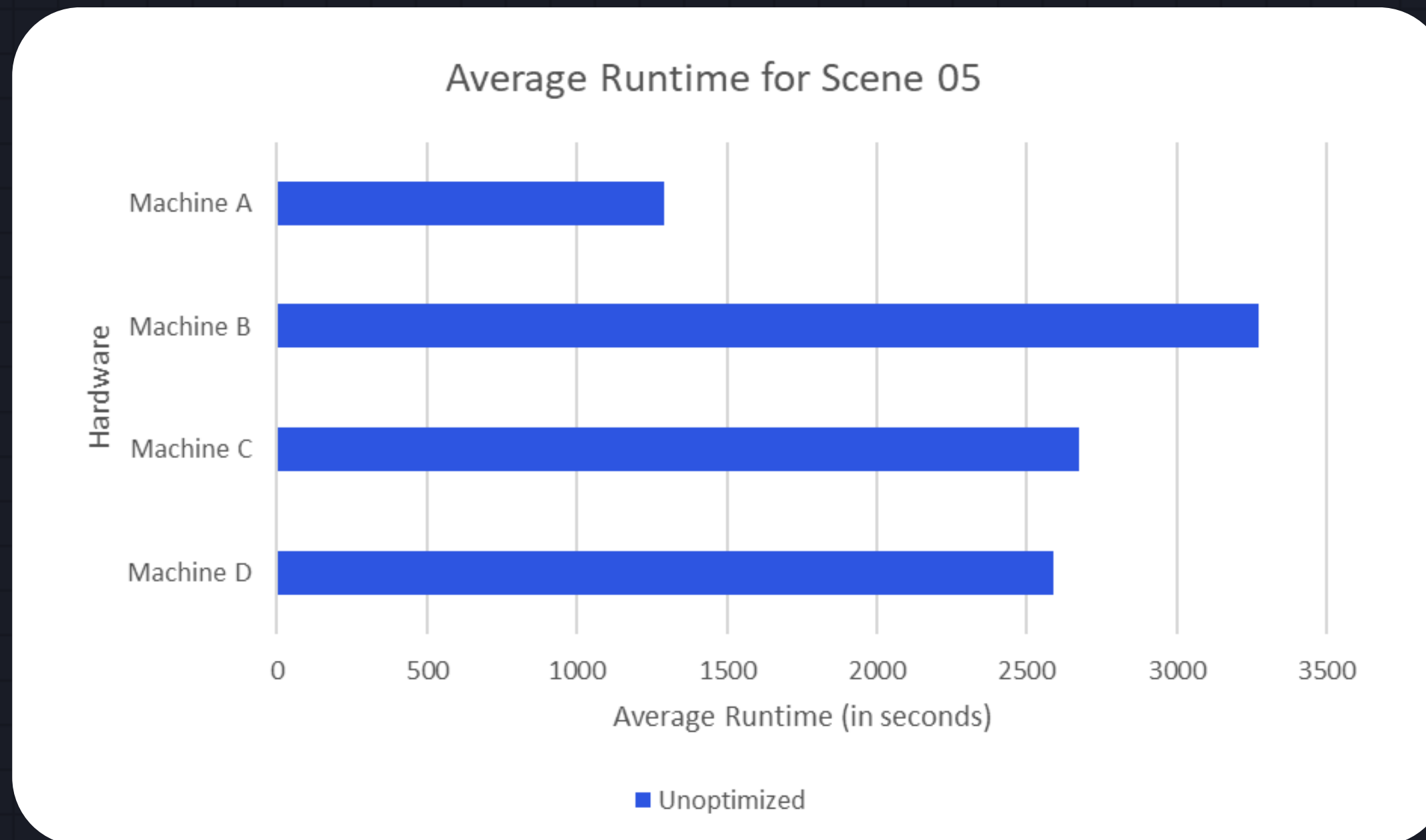
Summarized in the following graphs are the runtimes of the unoptimized ray tracer program across different hardware





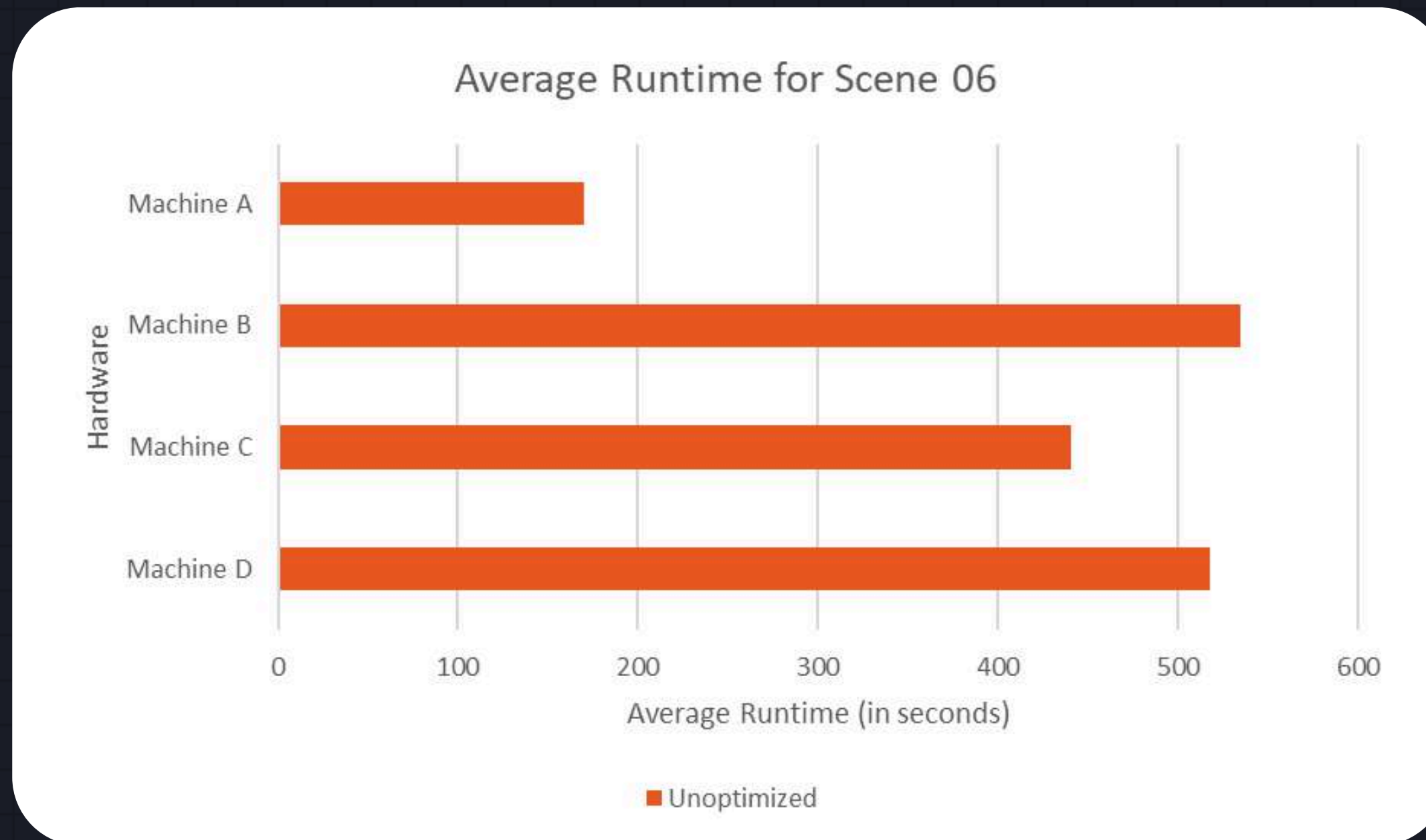
# RESULTS: UNOPTIMIZED IMPLEMENTATION

Summarized in the following graphs are the runtimes of the unoptimized ray tracer program across different hardware



# RESULTS: UNOPTIMIZED IMPLEMENTATION

Summarized in the following graphs are the runtimes of the unoptimized ray tracer program across different hardware



## RESULTS: UNOPTIMIZED IMPLEMENTATION

The average runtime for Scene 07 was not recorded, as the program did not finish running after 24 hours.

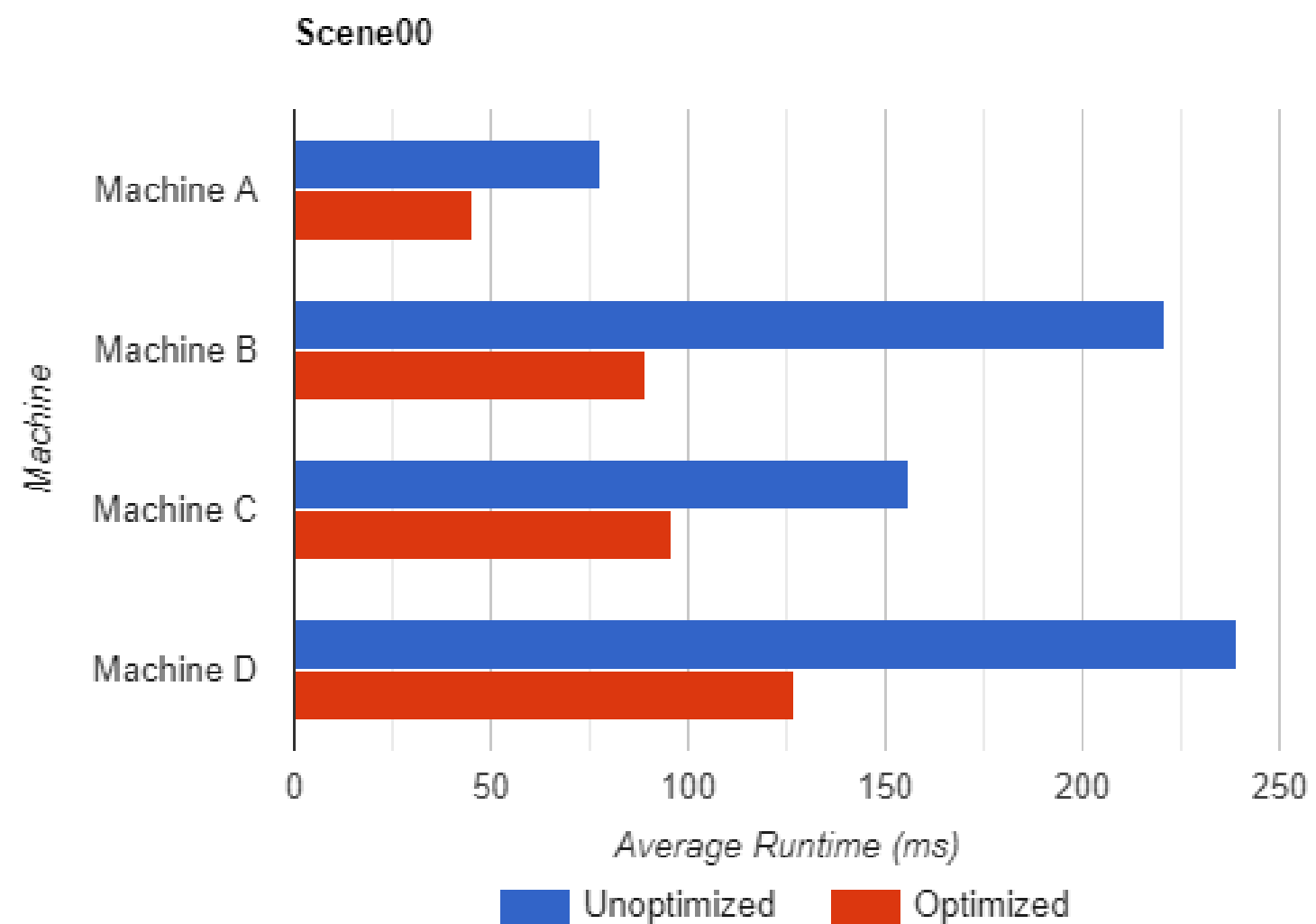


## DISCUSSION: UNOPTIMIZED IMPLEMENTATION

- As the size of the image and the number of vertices, shapes, and light increase, the runtime of the program significantly increases due to more elements being processed by the ray tracing algorithm.
- More memory is occupied relative to the increase in elements of the scene.
- Runtimes vary greatly across different machines. The device specifications especially the processor determine the speed of rendering.

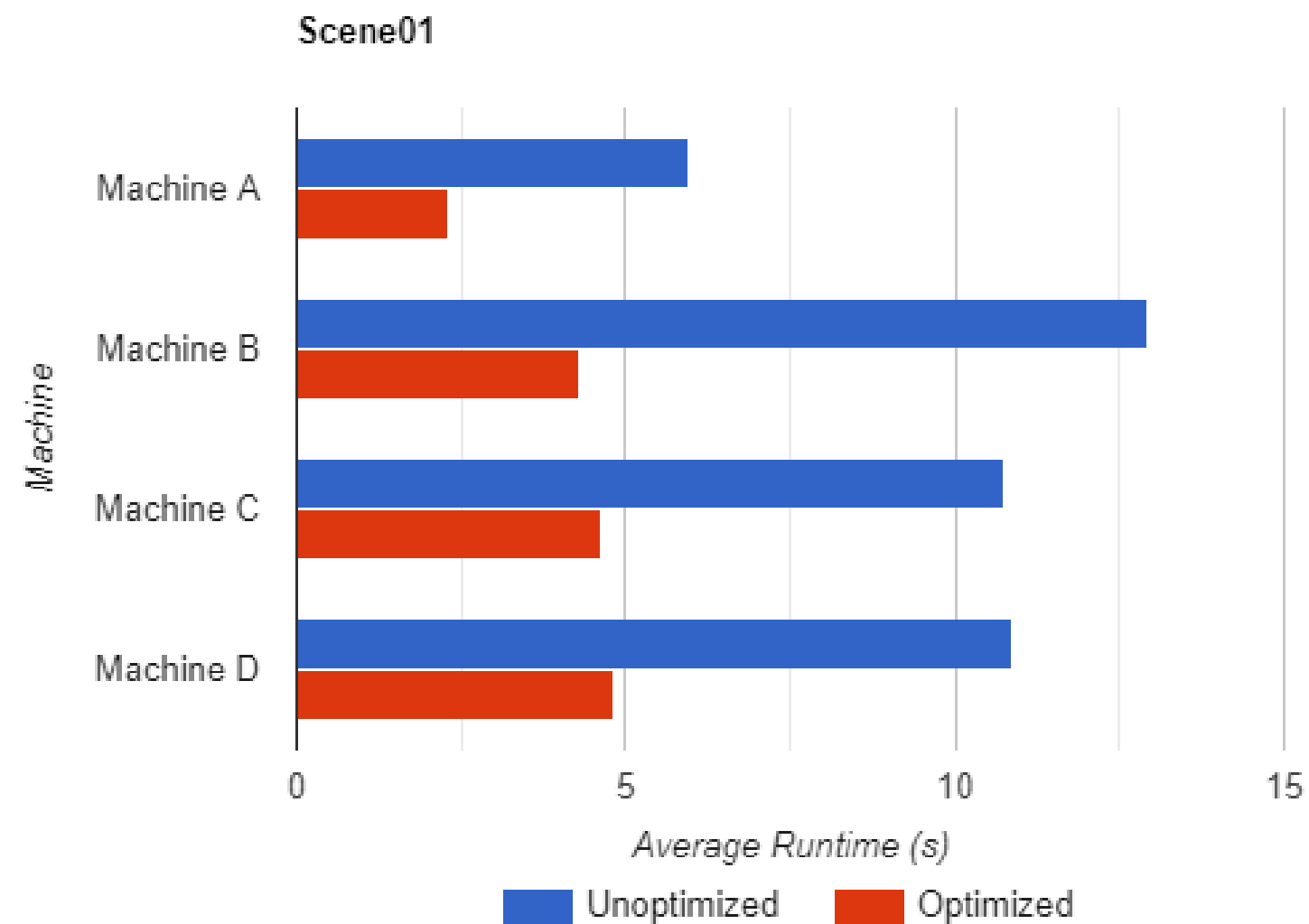
# RESULTS: CODEBASE OPTIMIZATION I (RENDER + BUILD\_IMAGE FUNCTION)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



# RESULTS: CODEBASE OPTIMIZATION I (RENDER + BUILD\_IMAGE FUNCTION)

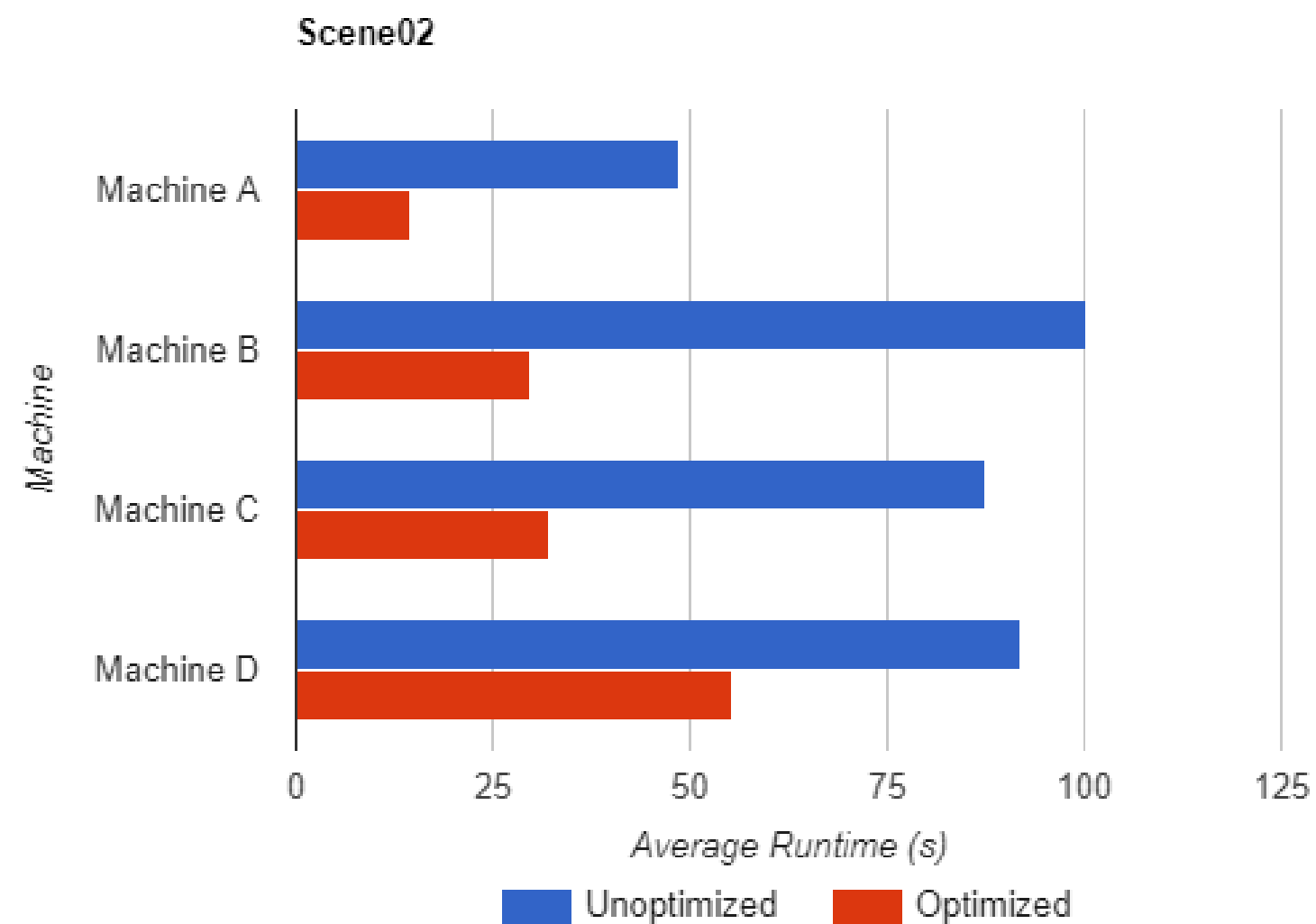
Summarized in the following graphs are the runtimes of the ray tracer program across different hardware





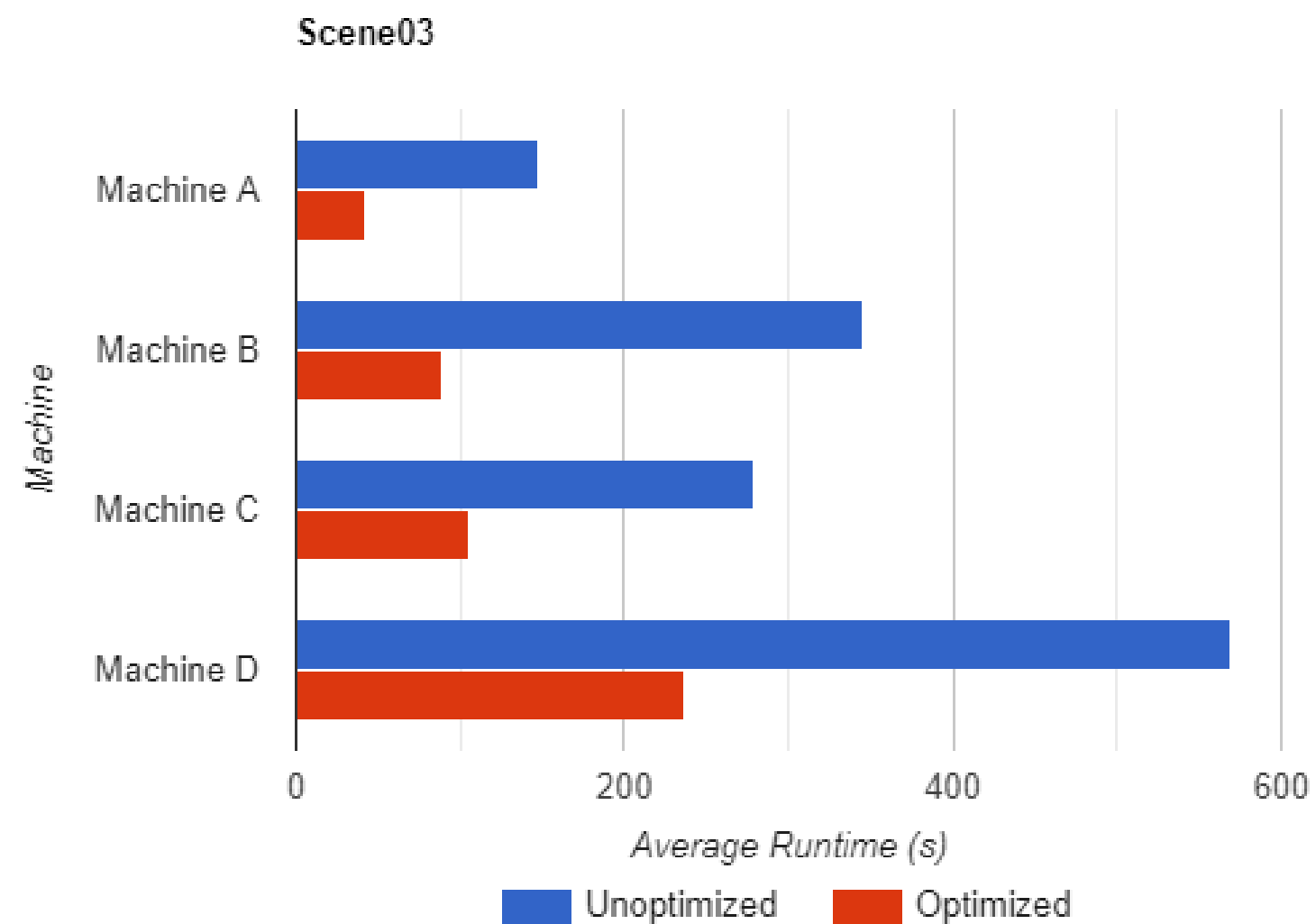
# RESULTS: CODEBASE OPTIMIZATION I (RENDER + BUILD\_IMAGE FUNCTION)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



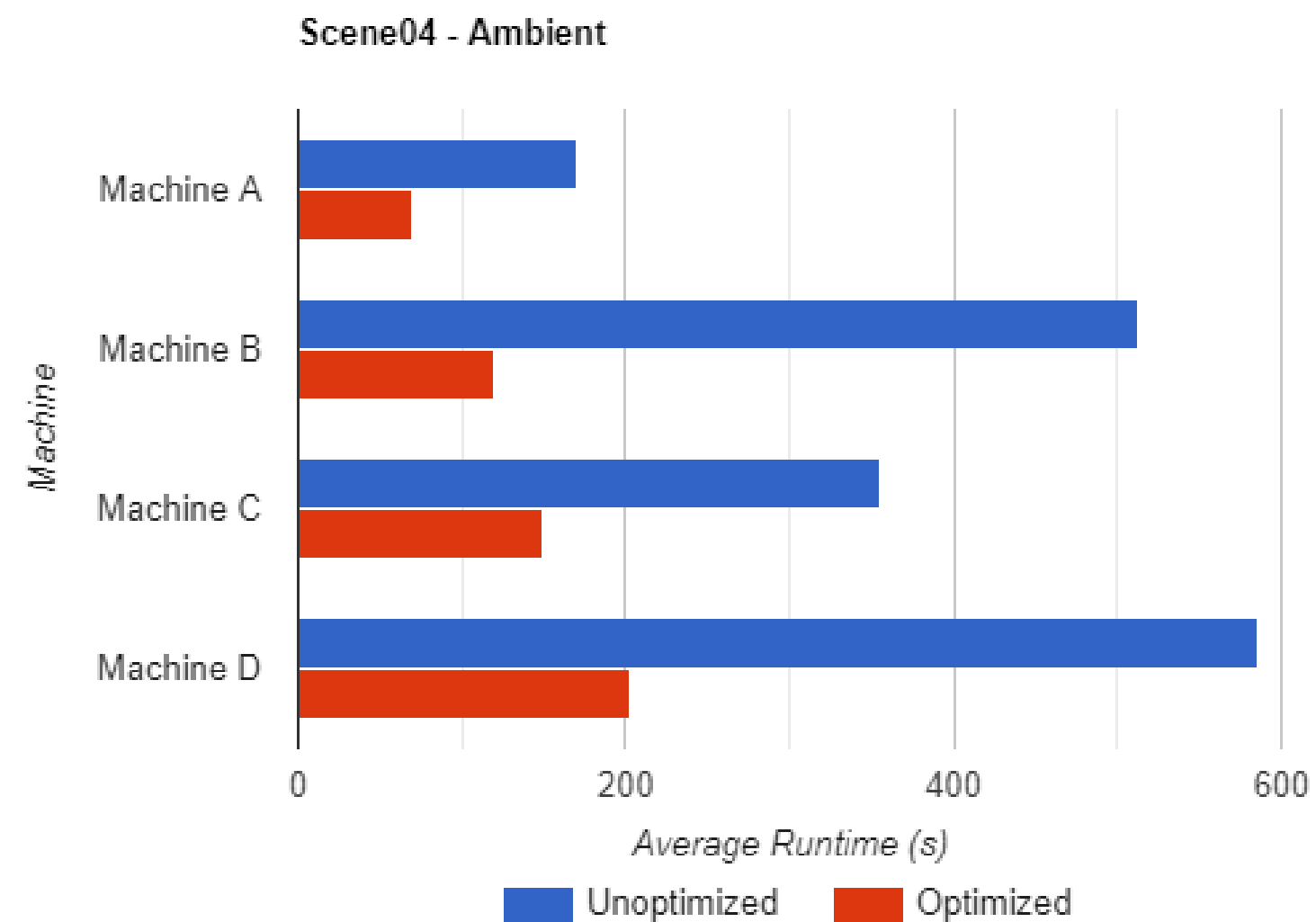
# RESULTS: CODEBASE OPTIMIZATION I (RENDER + BUILD\_IMAGE FUNCTION)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



# RESULTS: CODEBASE OPTIMIZATION I (RENDER + BUILD\_IMAGE FUNCTION)

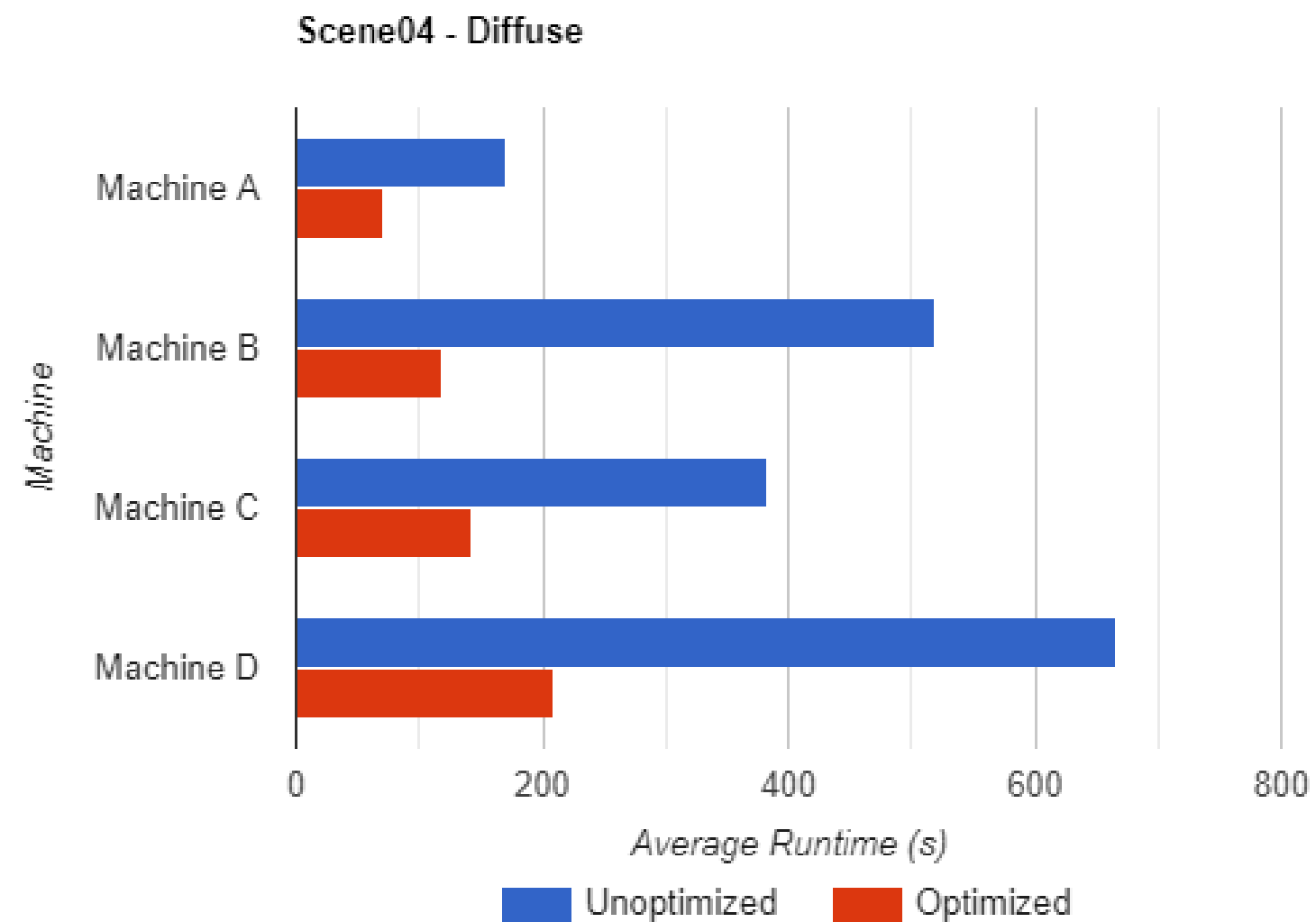
Summarized in the following graphs are the runtimes of the ray tracer program across different hardware





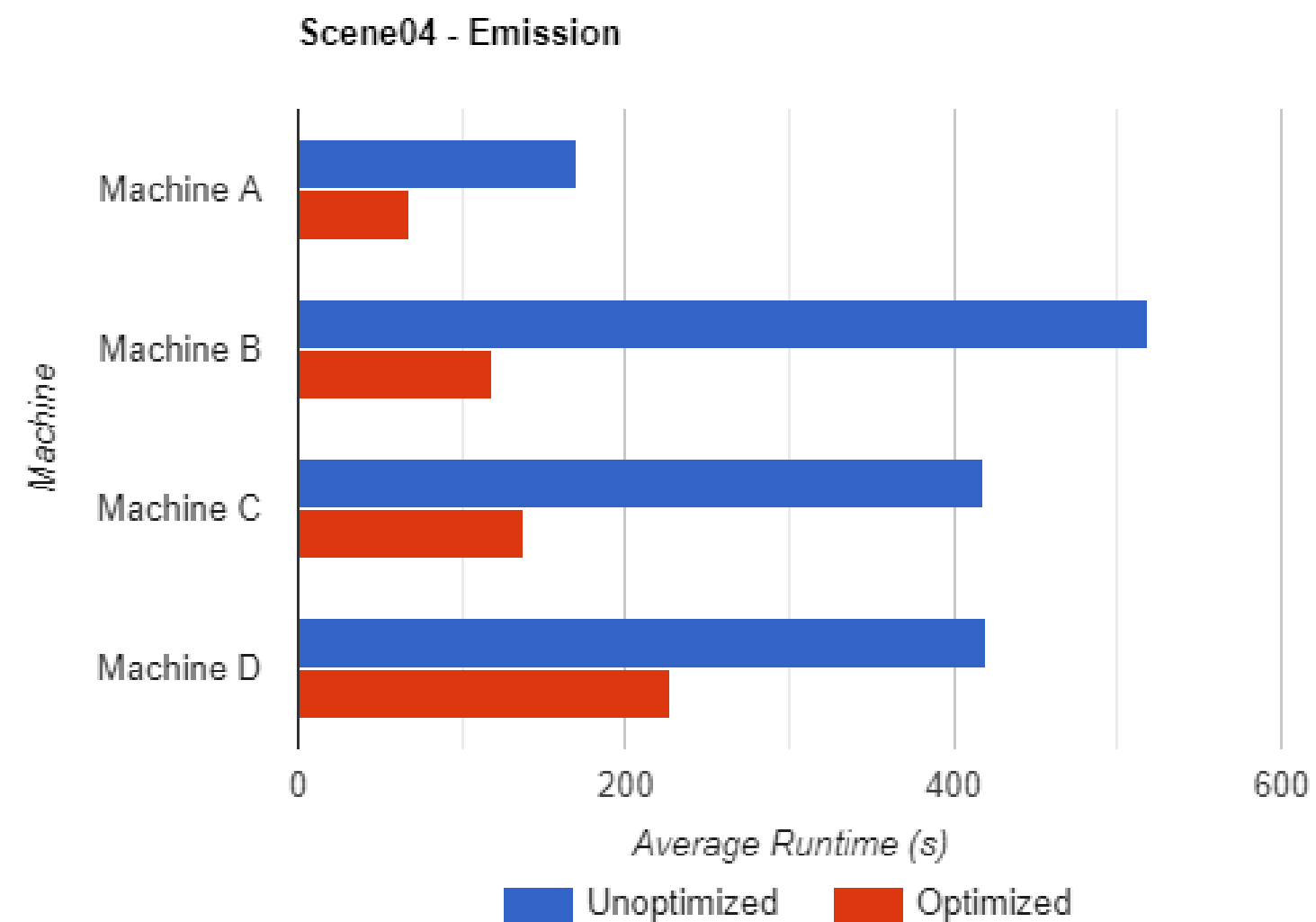
# RESULTS: CODEBASE OPTIMIZATION I (RENDER + BUILD\_IMAGE FUNCTION)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



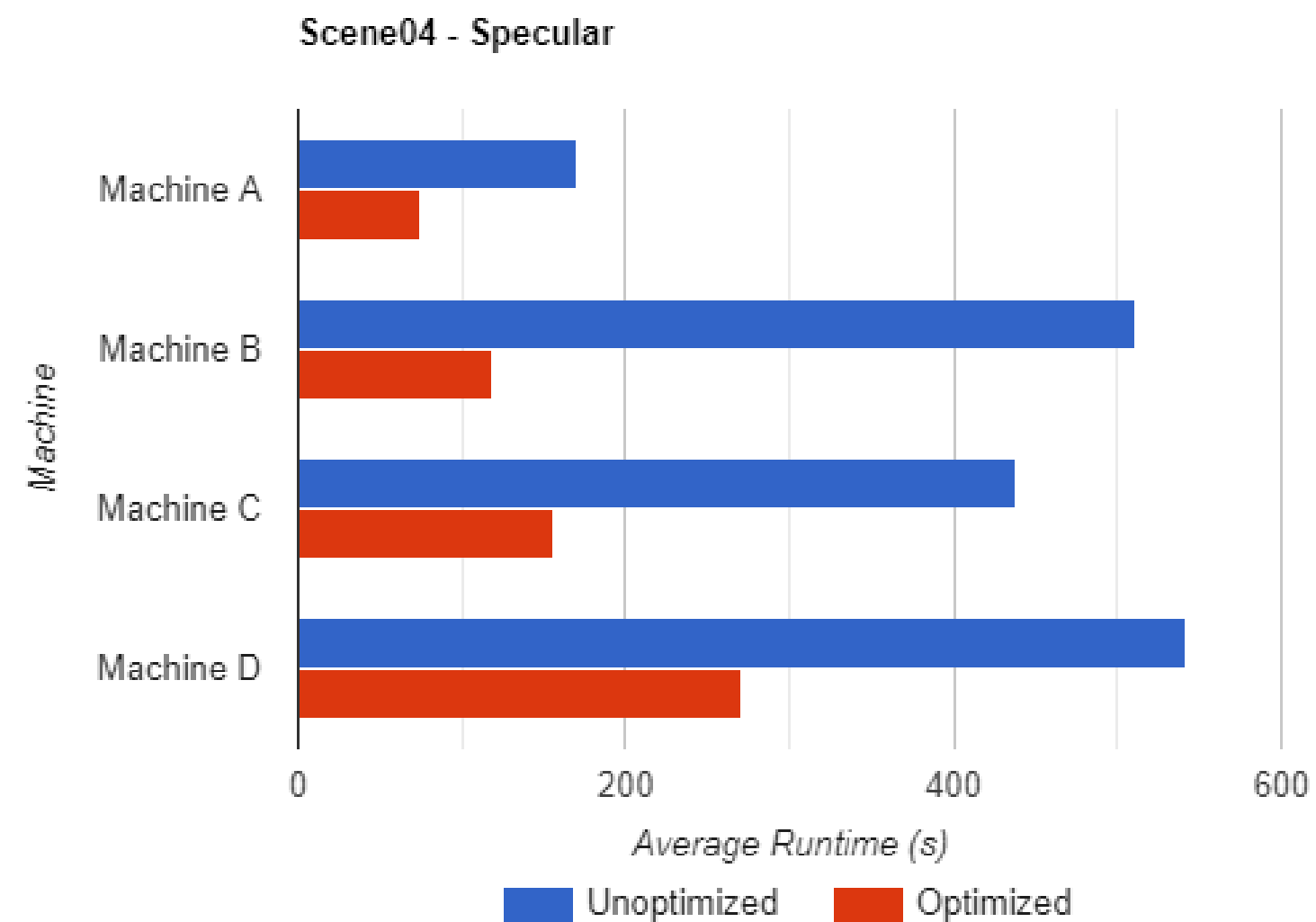
# RESULTS: CODEBASE OPTIMIZATION I (RENDER + BUILD\_IMAGE FUNCTION)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



# RESULTS: CODEBASE OPTIMIZATION I (RENDER + BUILD\_IMAGE FUNCTION)

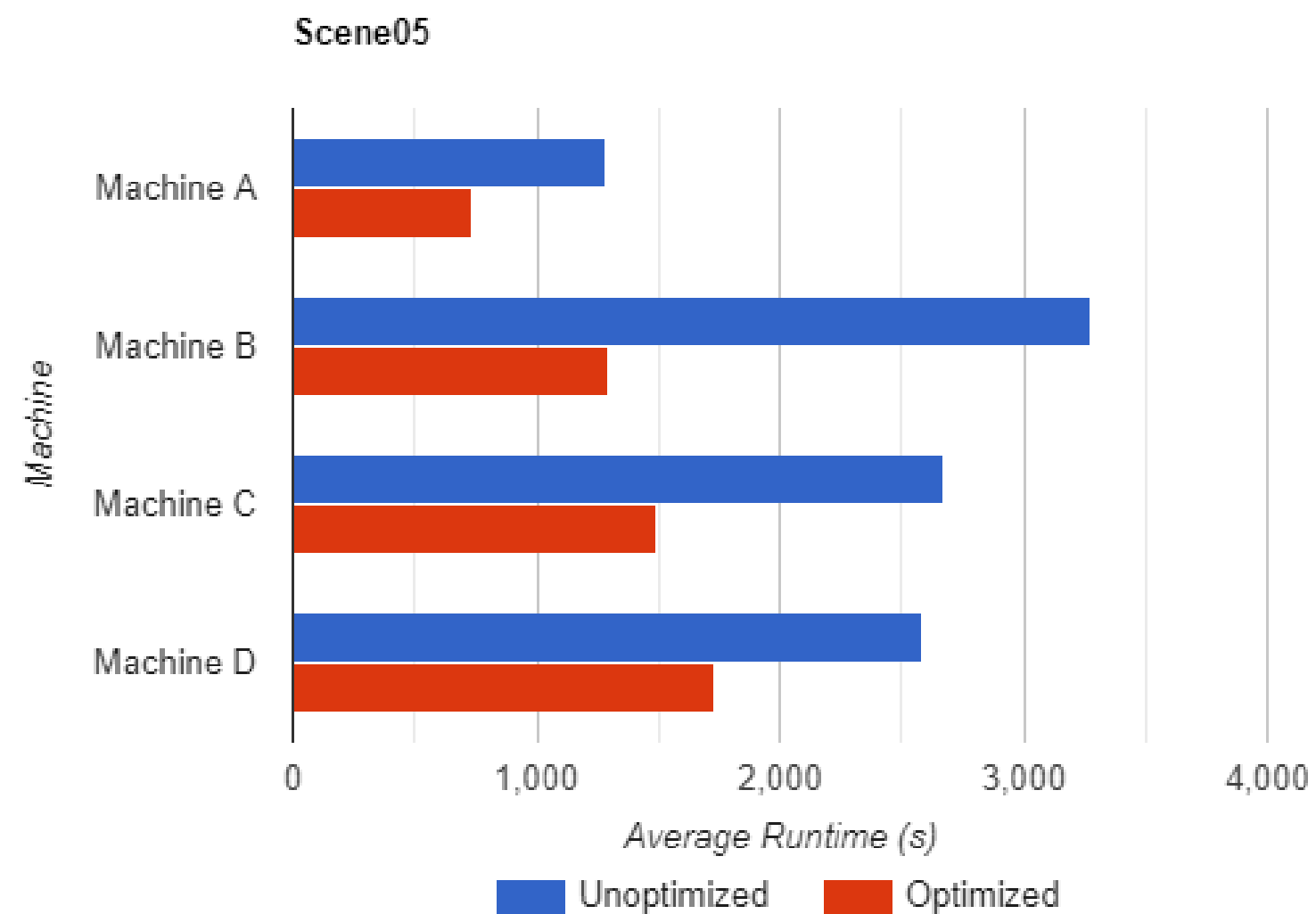
Summarized in the following graphs are the runtimes of the ray tracer program across different hardware





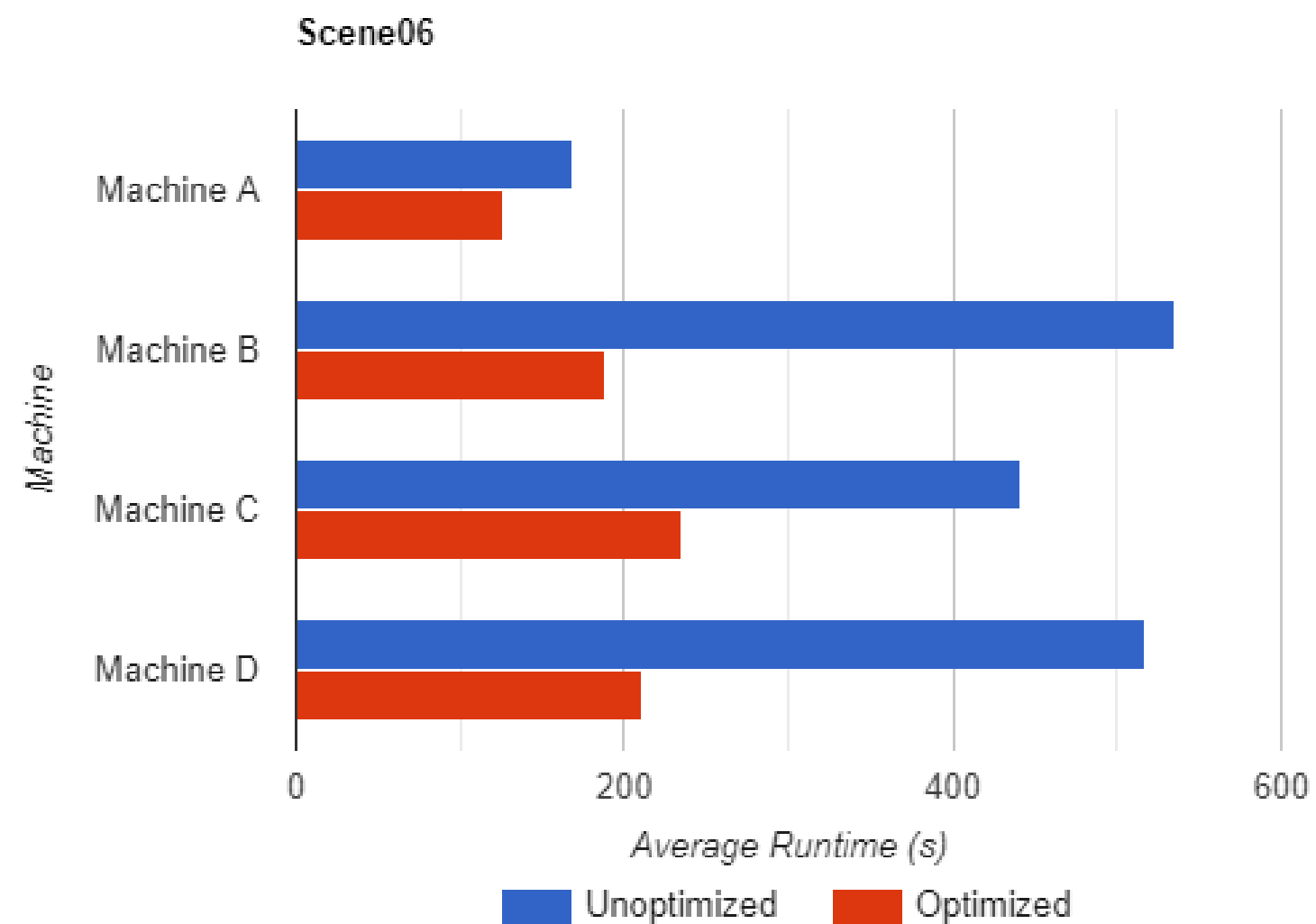
# RESULTS: CODEBASE OPTIMIZATION I (RENDER + BUILD\_IMAGE FUNCTION)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



# RESULTS: CODEBASE OPTIMIZATION I (RENDER + BUILD\_IMAGE FUNCTION)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



RESULTS: CODEBASE OPTIMIZATION I  
(RENDER + BUILD\_IMAGE FUNCTION)

The average runtime for Scene 07 was not recorded, as the program did not finish running after 24 hours.



# DISCUSSION: CODEBASE OPTIMIZATION I

## (RENDER + BUILD\_IMAGE FUNCTION)

Presented below is the average percent decrease in runtime for each scene across all four (4) machines utilized in the project after codebase optimization I.

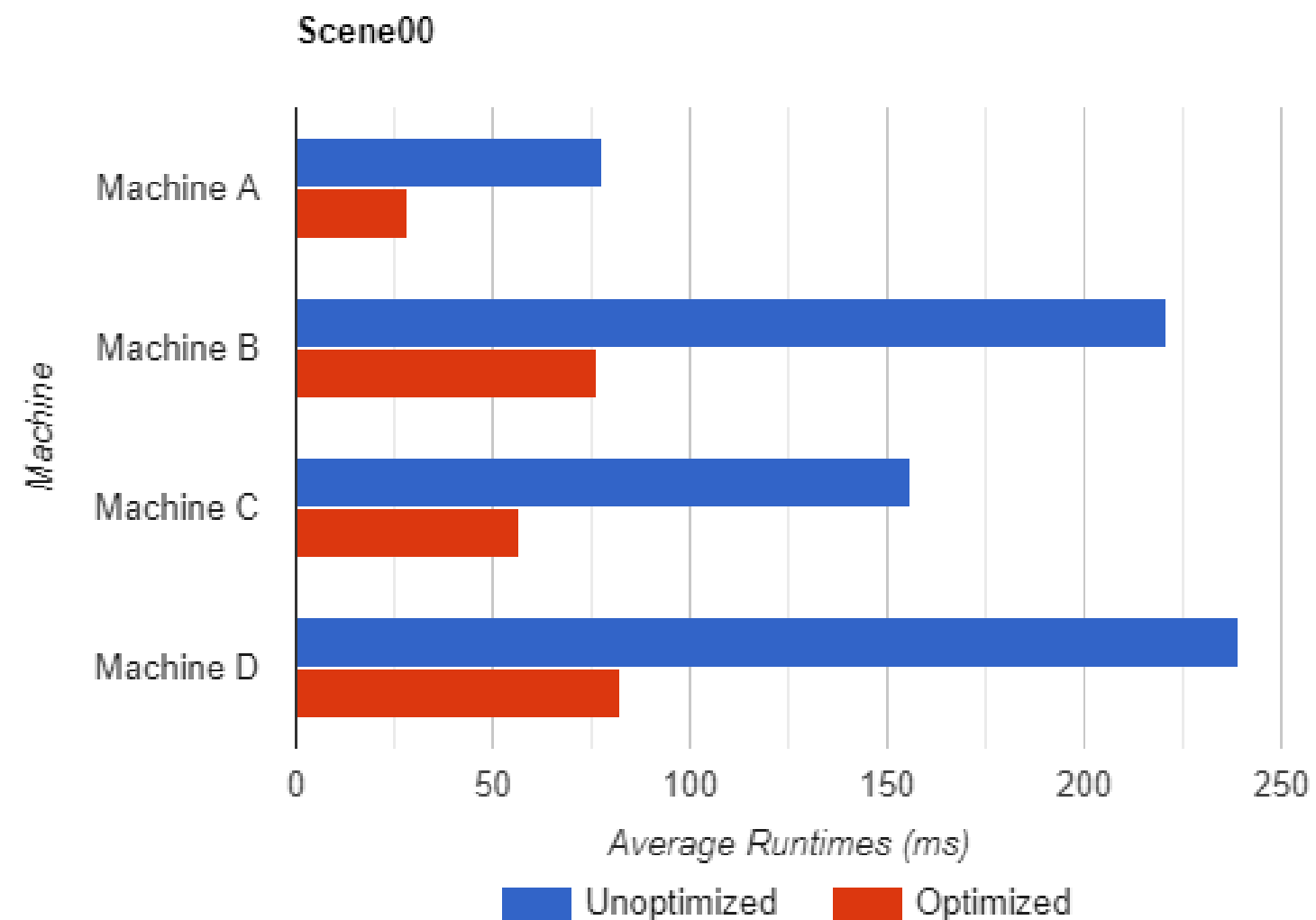
Scene	Percent Decrease
Scene 00	46.64
Scene 01	60.24
Scene 02	60.99
Scene 03	66.55
Scene 04 - Ambient	64.66
Scene 04 - Diffuse	66.7
Scene 04 - Emission	62.41
Scene 04 - Specular	61.89
Scene 05	45.32
Scene 06	49.09

## DISCUSSION: CODEBASE OPTIMIZATION I (RENDER + BUILD\_IMAGE FUNCTION)

After implementing the render and build\_image optimization, it was observed that *Scene 04 - Diffuse* benefited the most with a 66.7% average decrease in runtime while *Scene 05* had the least benefit with an average of 45.32% decrease in runtime.

# RESULTS: CODEBASE OPTIMIZATION I + II (INCLUDED MAKE\_RAY + RENDER REFACTORING)

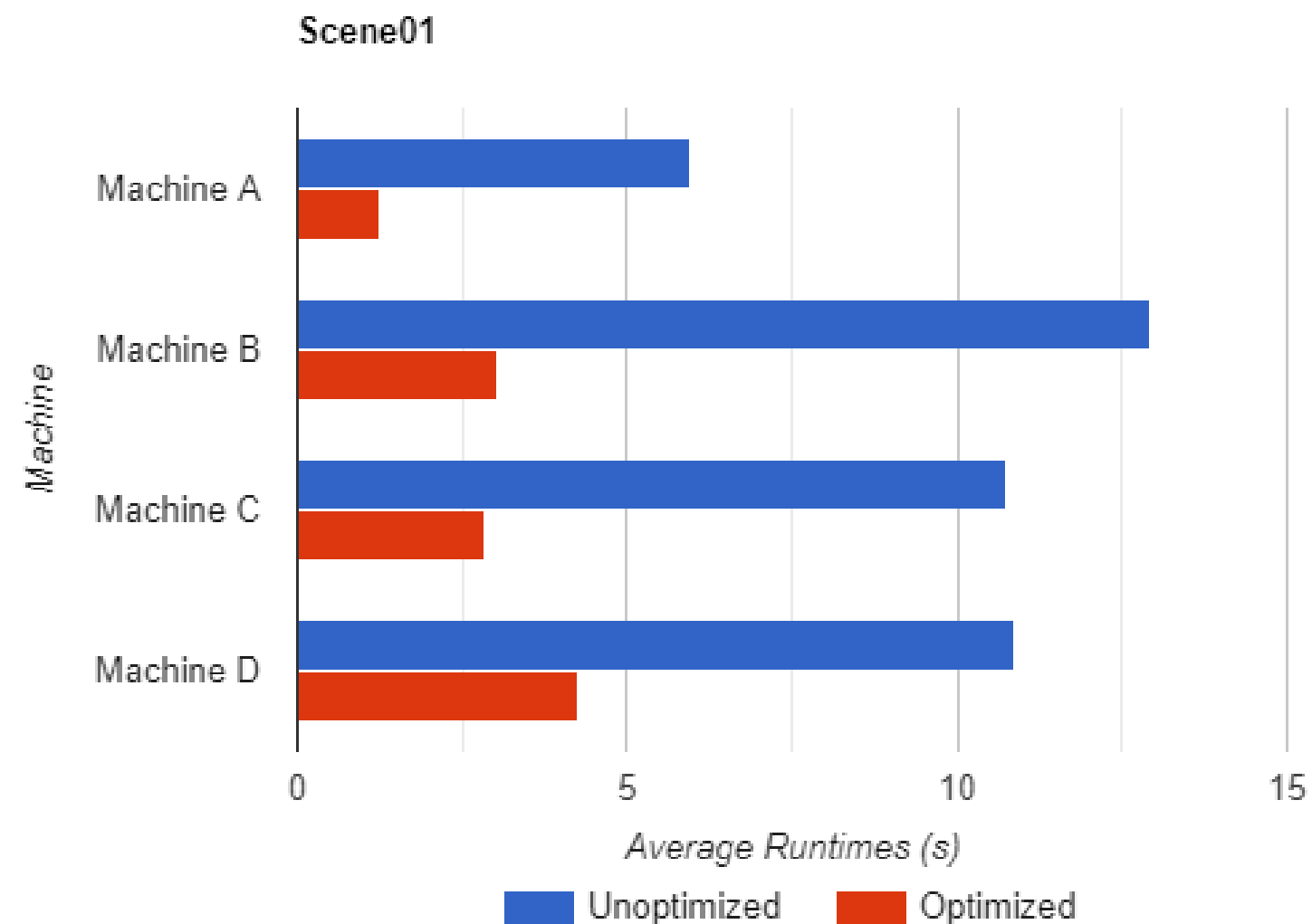
Summarized in the following graphs are the runtimes of the ray tracer program across different hardware





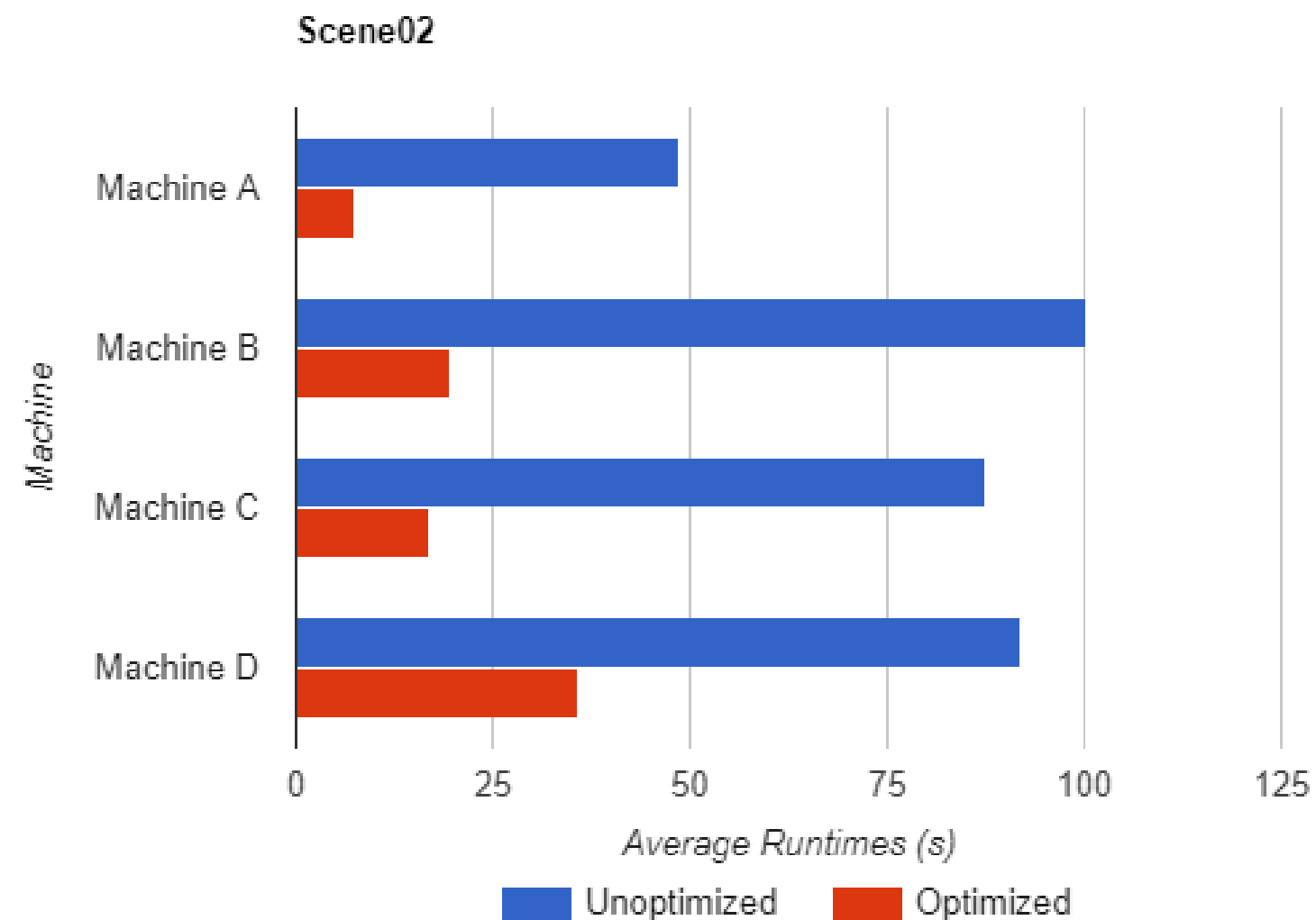
# RESULTS: CODEBASE OPTIMIZATION I + II (INCLUDED MAKE\_RAY + RENDER REFACTORING)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



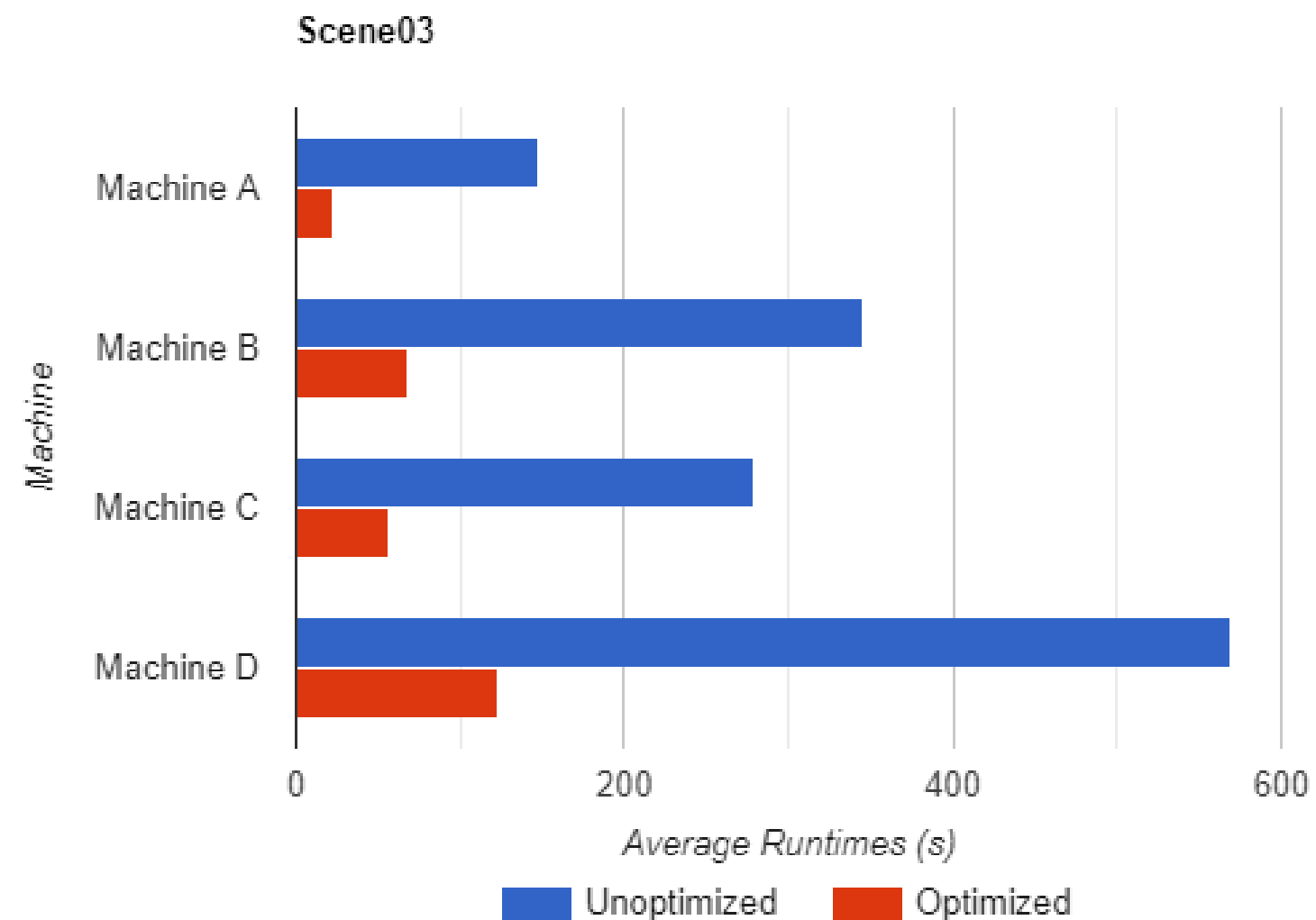
# RESULTS: CODEBASE OPTIMIZATION I + II (INCLUDED MAKE\_RAY + RENDER REFACTORING)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



# RESULTS: CODEBASE OPTIMIZATION I + II (INCLUDED MAKE\_RAY + RENDER REFACTORING)

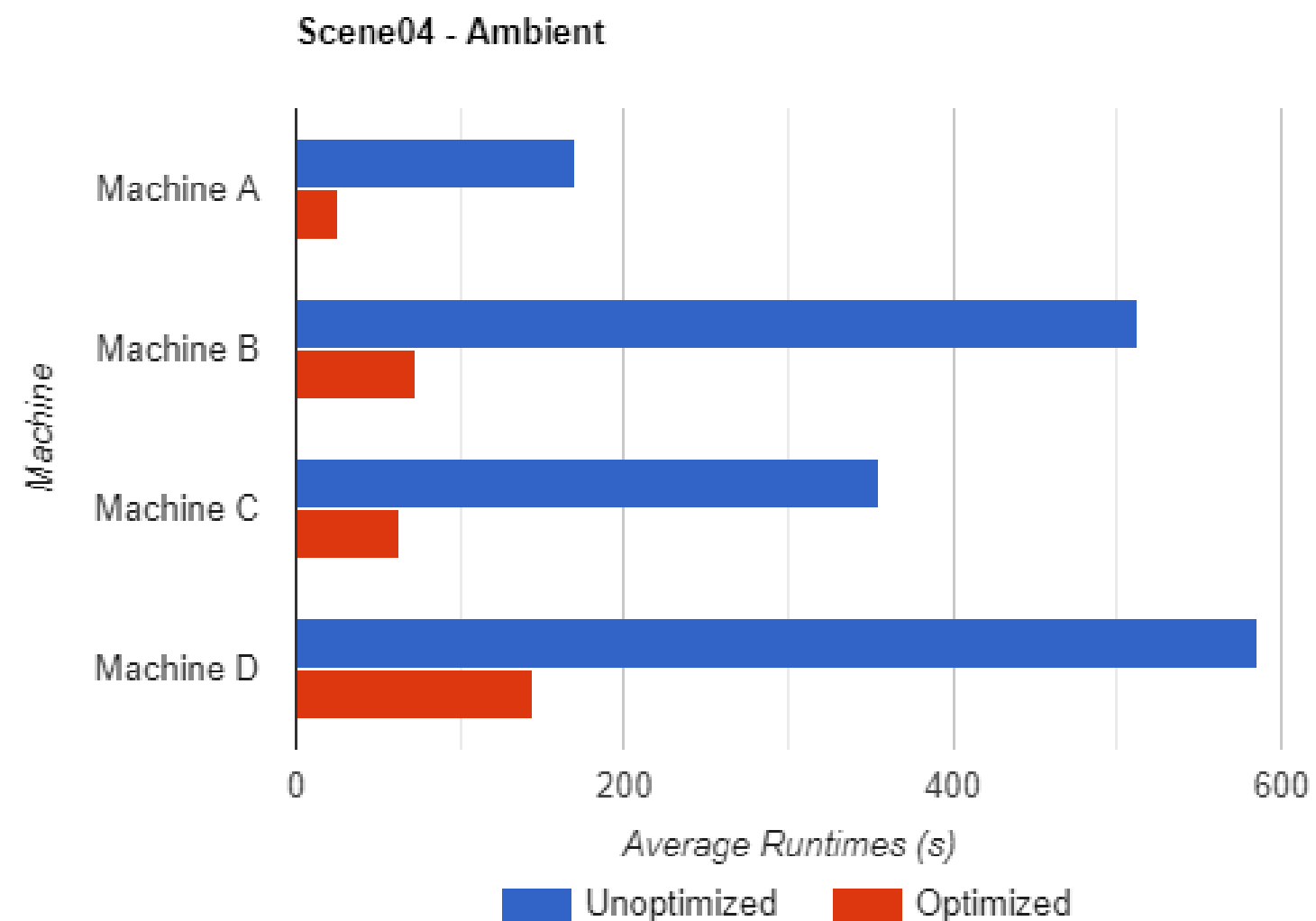
Summarized in the following graphs are the runtimes of the ray tracer program across different hardware





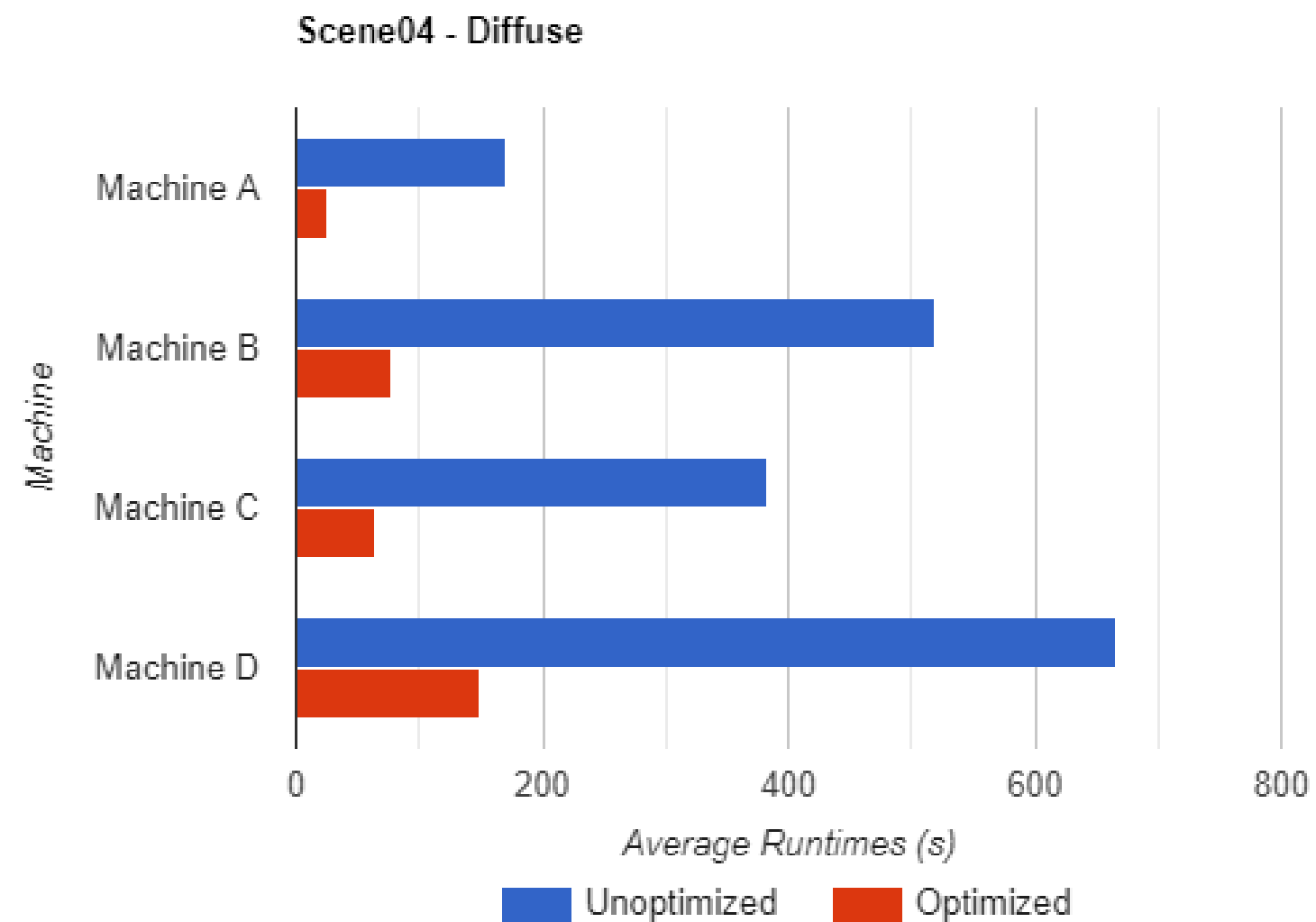
# RESULTS: CODEBASE OPTIMIZATION I + II (INCLUDED MAKE\_RAY + RENDER REFACTORING)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



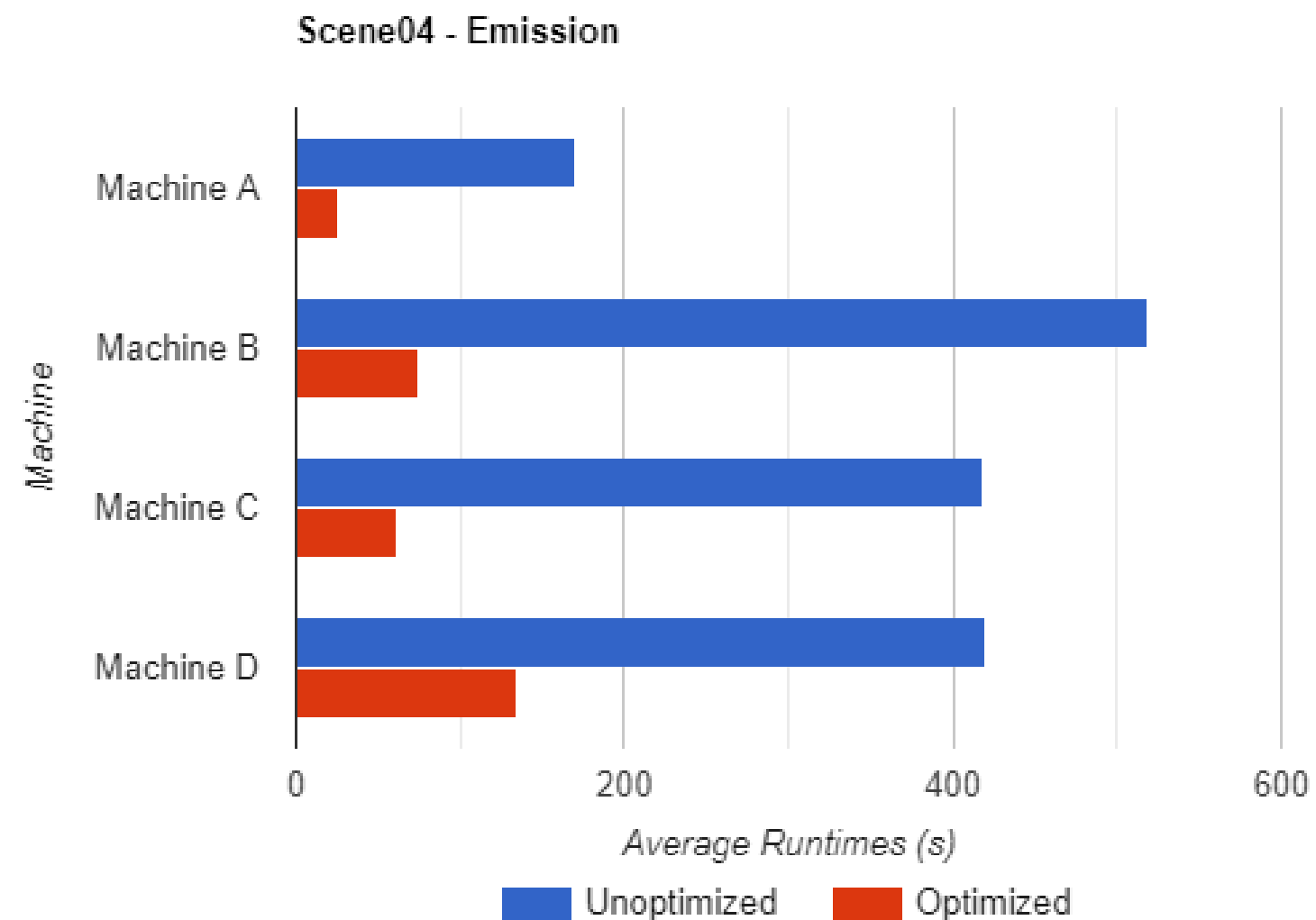
# RESULTS: CODEBASE OPTIMIZATION I + II (INCLUDED MAKE\_RAY + RENDER REFACTORING)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



# RESULTS: CODEBASE OPTIMIZATION I + II (INCLUDED MAKE\_RAY + RENDER REFACTORING)

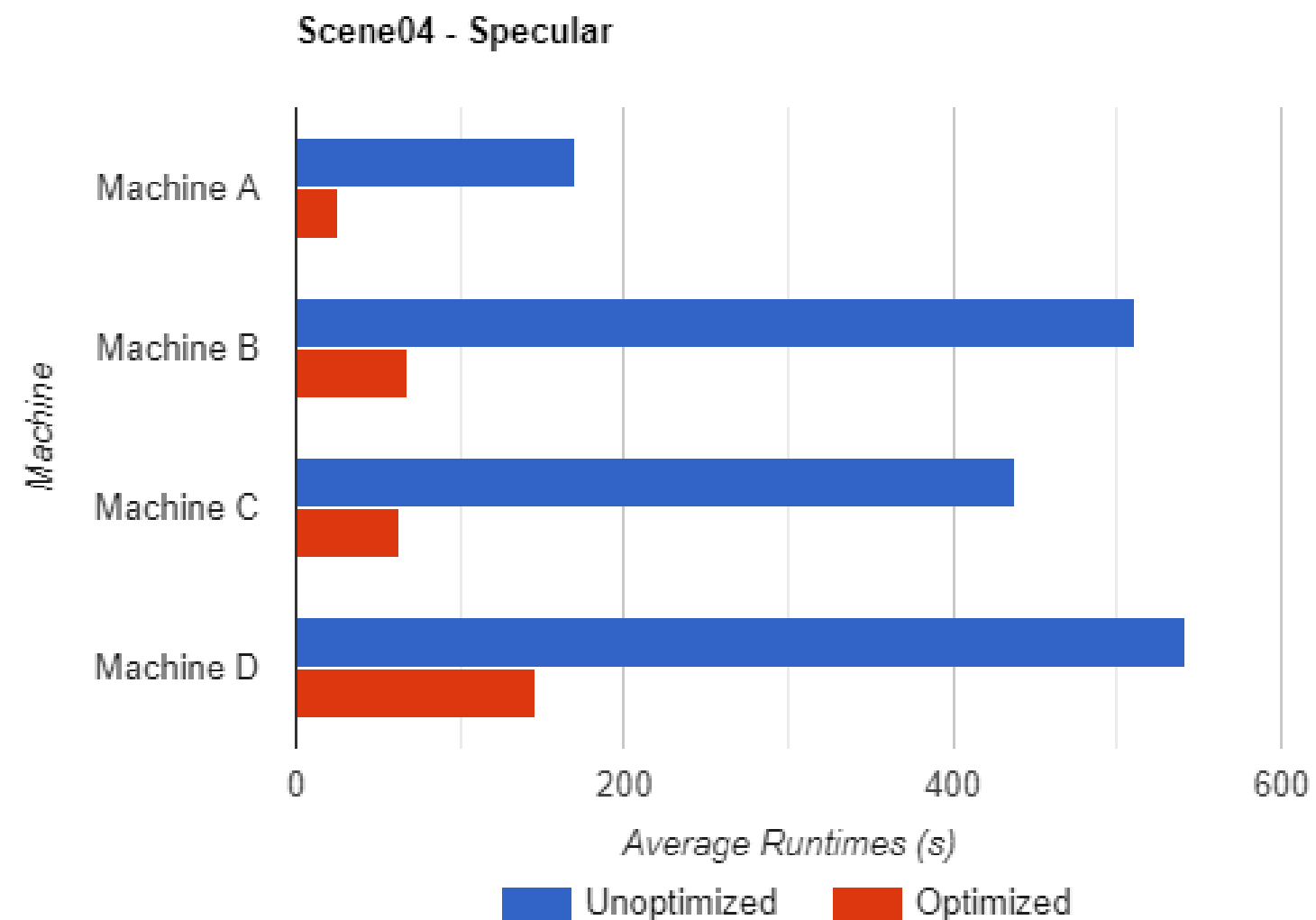
Summarized in the following graphs are the runtimes of the ray tracer program across different hardware





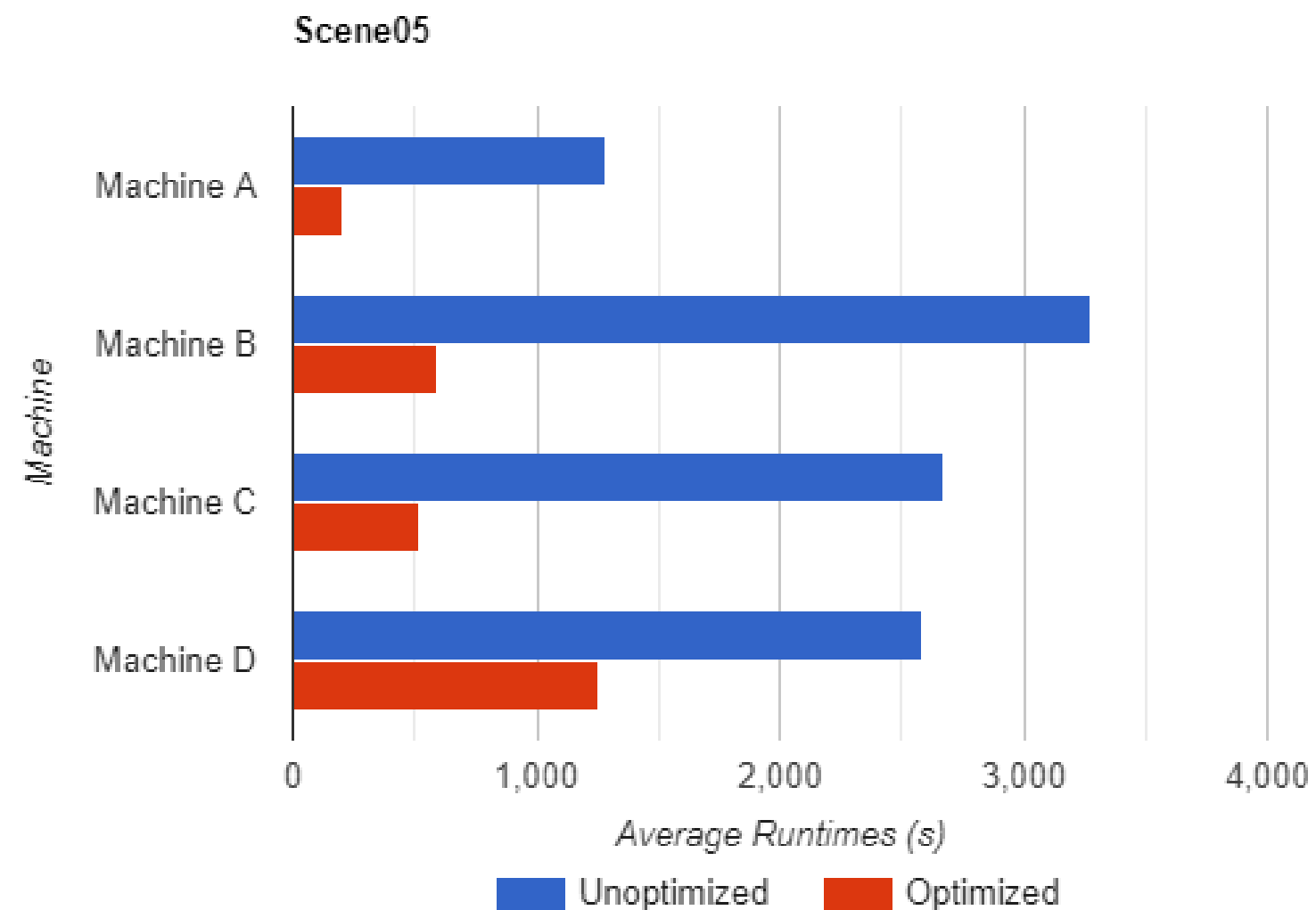
# RESULTS: CODEBASE OPTIMIZATION I + II (INCLUDED MAKE\_RAY + RENDER REFACTORING)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



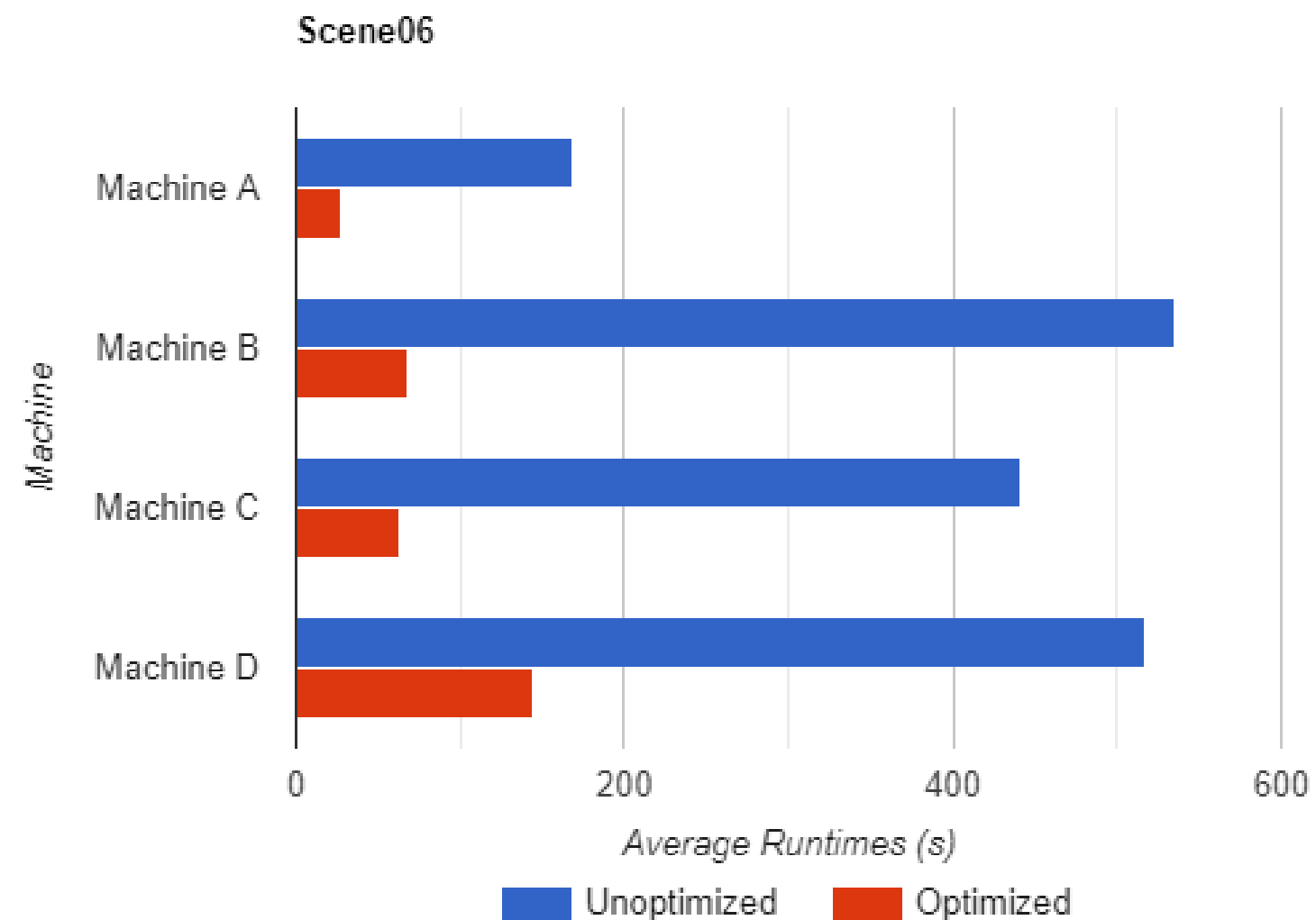
# RESULTS: CODEBASE OPTIMIZATION I + II (INCLUDED MAKE\_RAY + RENDER REFACTORING)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



# RESULTS: CODEBASE OPTIMIZATION I + II (INCLUDED MAKE\_RAY + RENDER REFACTORING)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware





RESULTS: CODEBASE OPTIMIZATION I + II  
(INCLUDED MAKE\_RAY + RENDER REFACTORING)

Scene 07 is only ran on Machine A (the fastest device available) with a time of 9122 seconds or approximately 2 hours and 30 minutes.

# DISCUSSION: CODEBASE OPTIMIZATION I + II

## (INCLUDED MAKE\_RAY + RENDER REFACTORING)

Presented below is the average percent decrease in runtime for each scene across all four (4) machines utilized in the project after codebase optimization I + II.

Scene	Percent Decrease
Scene 00	64.47
Scene 01	72.49
Scene 02	76.6
Scene 03	80.8
Scene 04 – Ambient	82.01
Scene 04 – Diffuse	82.56
Scene 04 – Emission	80.99
Scene 04 – Specular	82.56
Scene 05	74.57
Scene 06	82.19

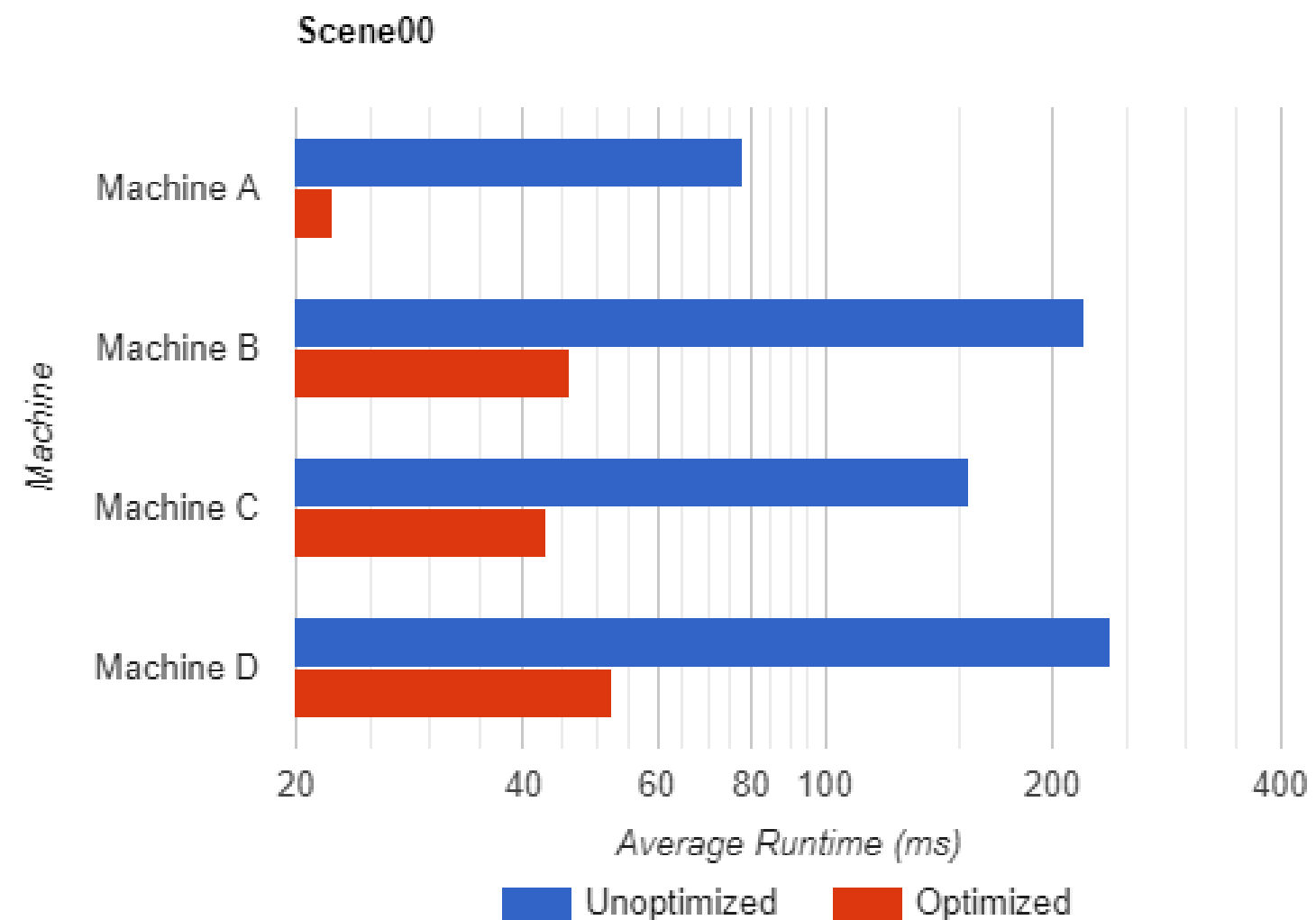
# DISCUSSION: CODEBASE OPTIMIZATION I + II (INCLUDED MAKE\_RAY + RENDER REFACTORING)

After implementing optimization II on top of optimization I, *Scene 00* showed the least average percent decrease in runtime with 64.47%. *Scene 04 - Diffuse* and *Scene 04 - Specular* on the other hand showed the highest average percent decrease in runtime with 82.56%.



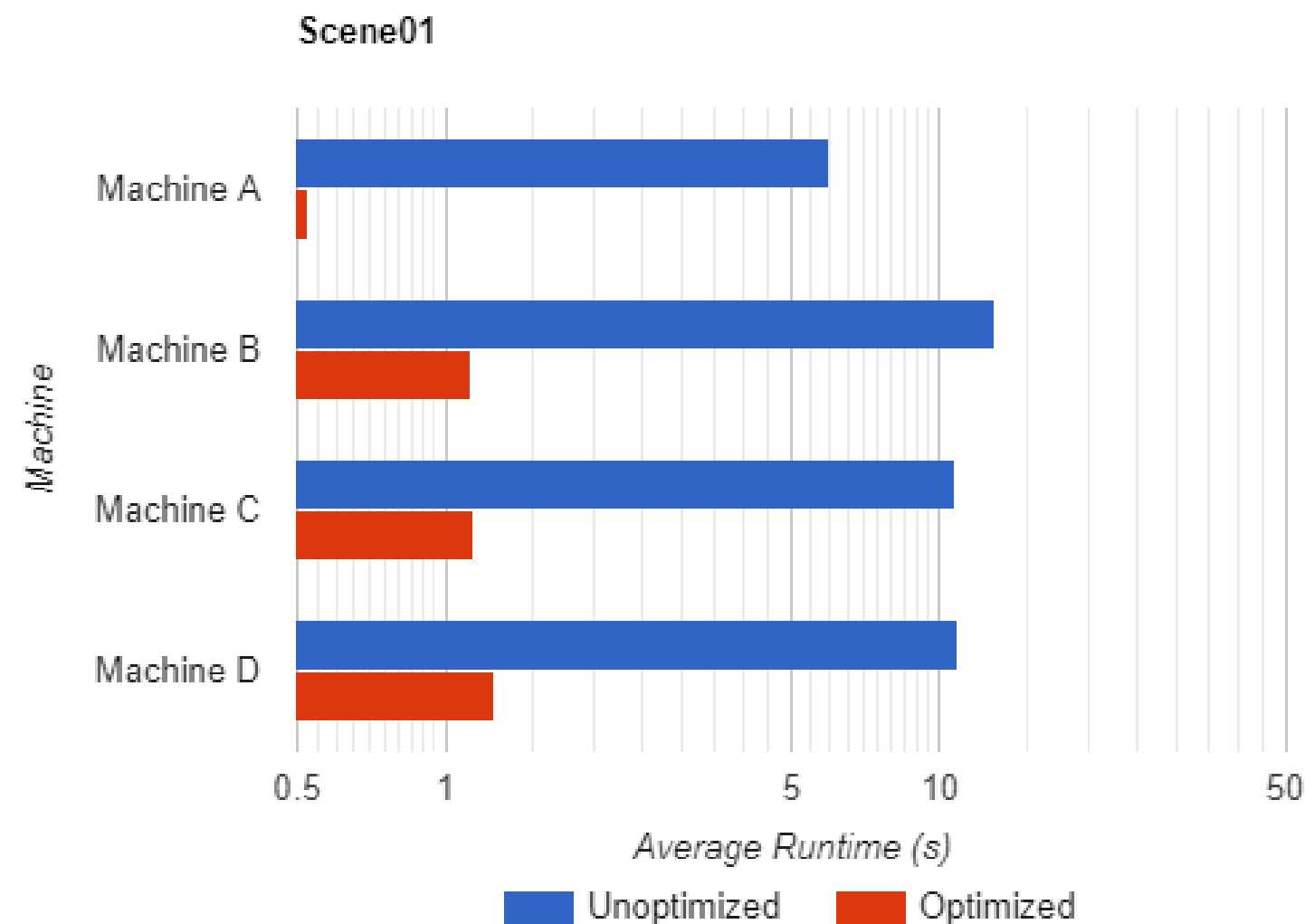
# RESULTS: CODEBASE OPTIMIZATION I + II + III (INCLUDED REFACTORING FUNCTIONS INVOLVING ITERATING OVER MAPS)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



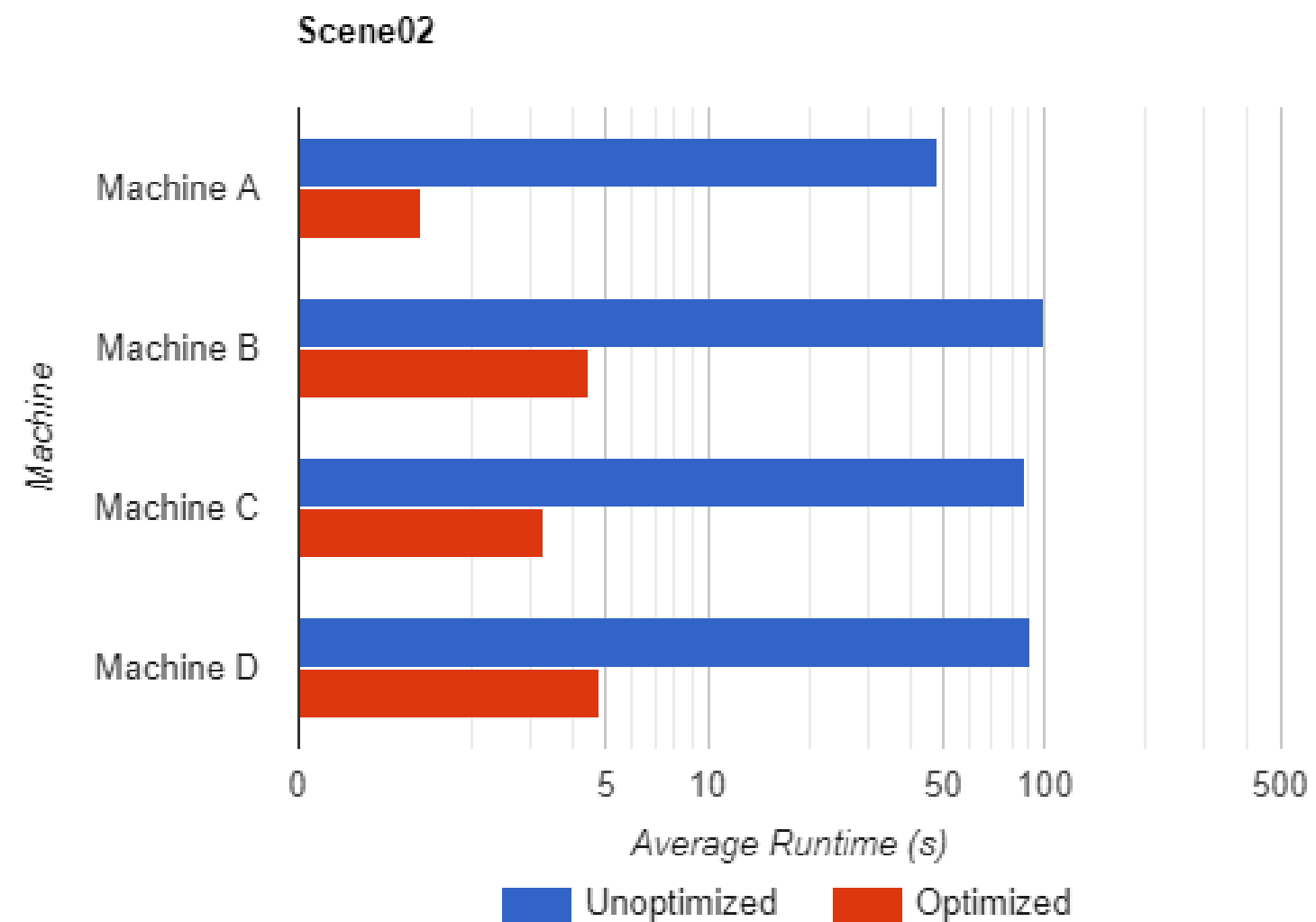
# RESULTS: CODEBASE OPTIMIZATION I + II + III (INCLUDED REFACTORING FUNCTIONS INVOLVING ITERATING OVER MAPS)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



# RESULTS: CODEBASE OPTIMIZATION I + II + III (INCLUDED REFACTORING FUNCTIONS INVOLVING ITERATING OVER MAPS)

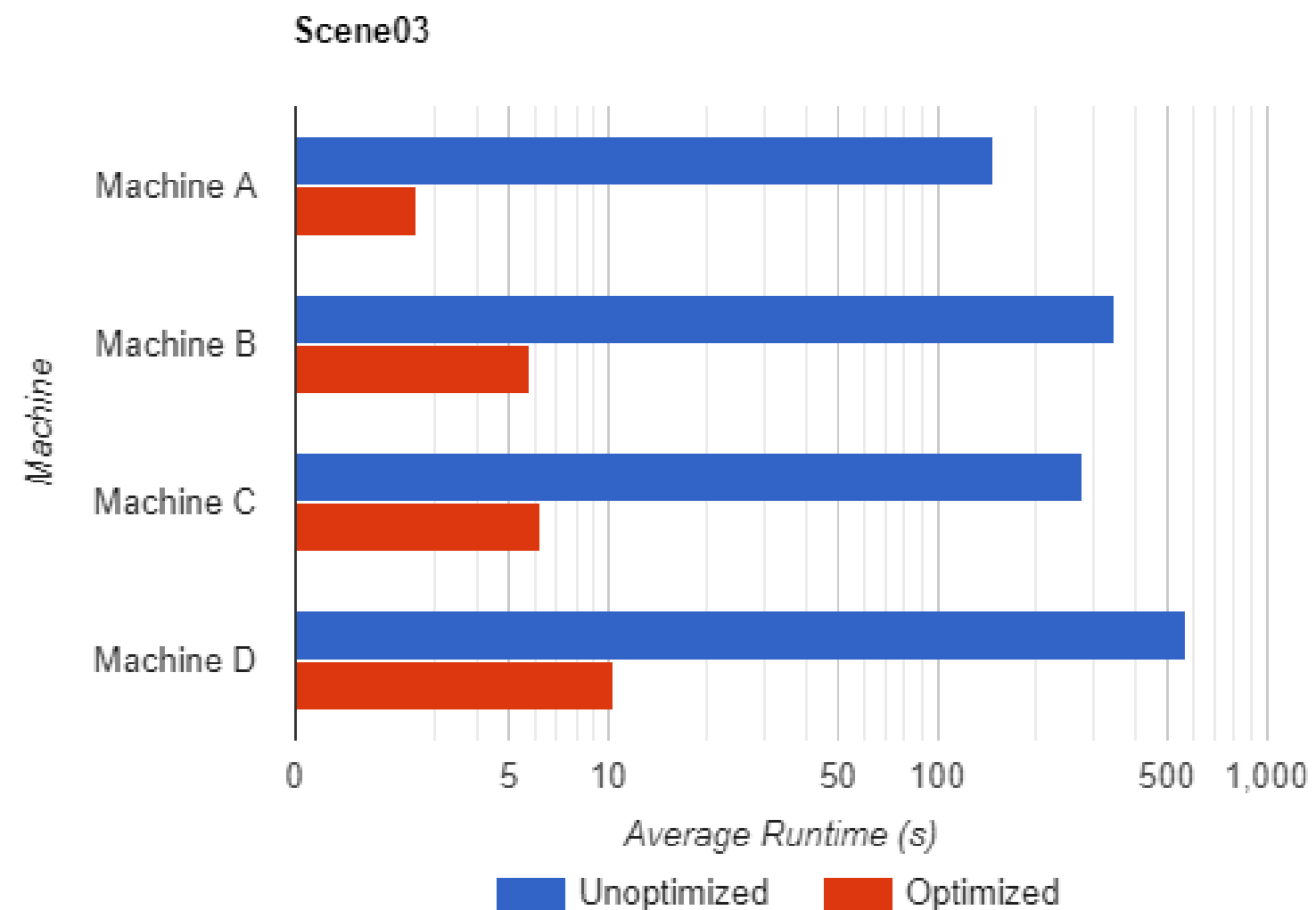
Summarized in the following graphs are the runtimes of the ray tracer program across different hardware





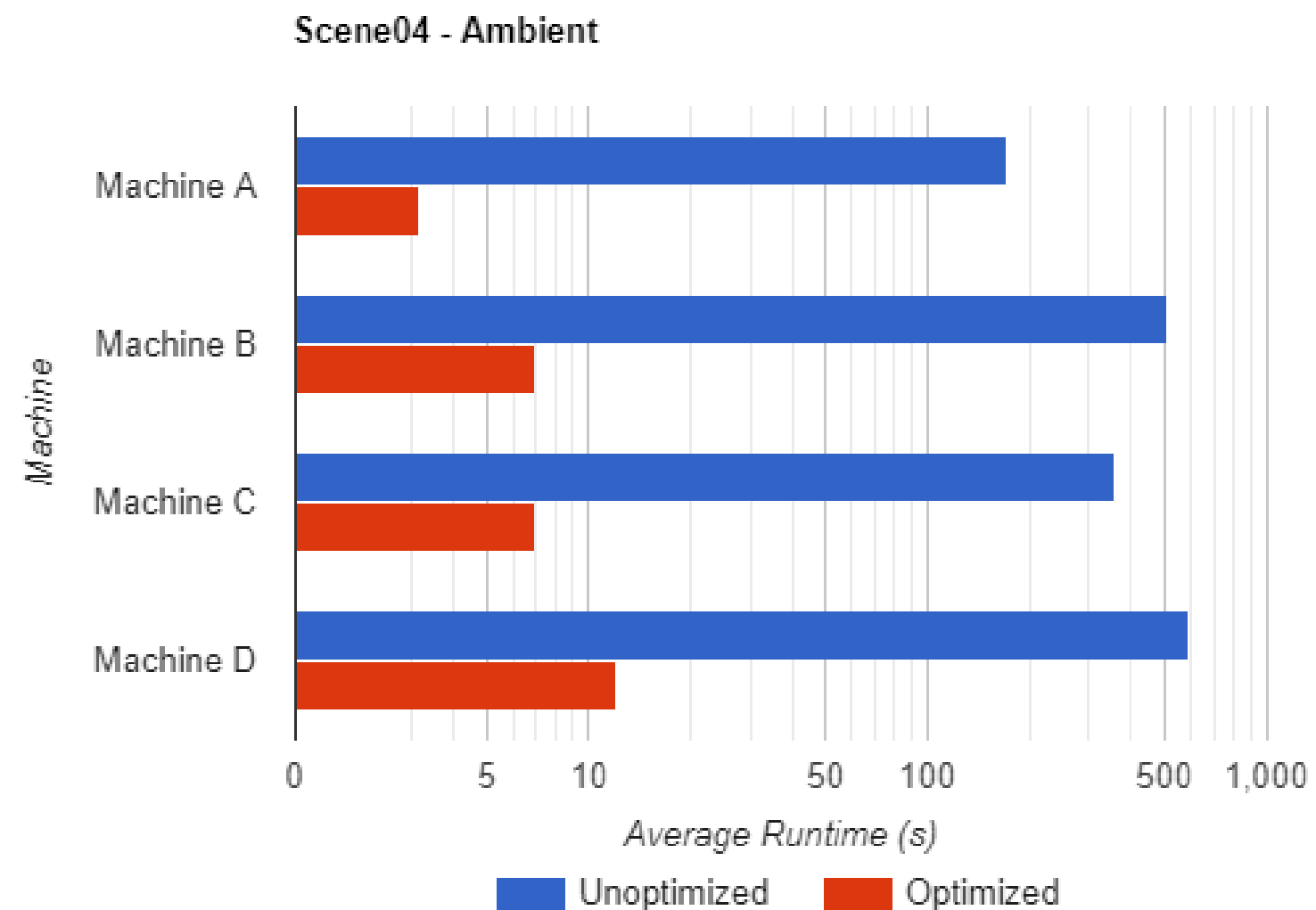
# RESULTS: CODEBASE OPTIMIZATION I + II + III (INCLUDED REFACTORING FUNCTIONS INVOLVING ITERATING OVER MAPS)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



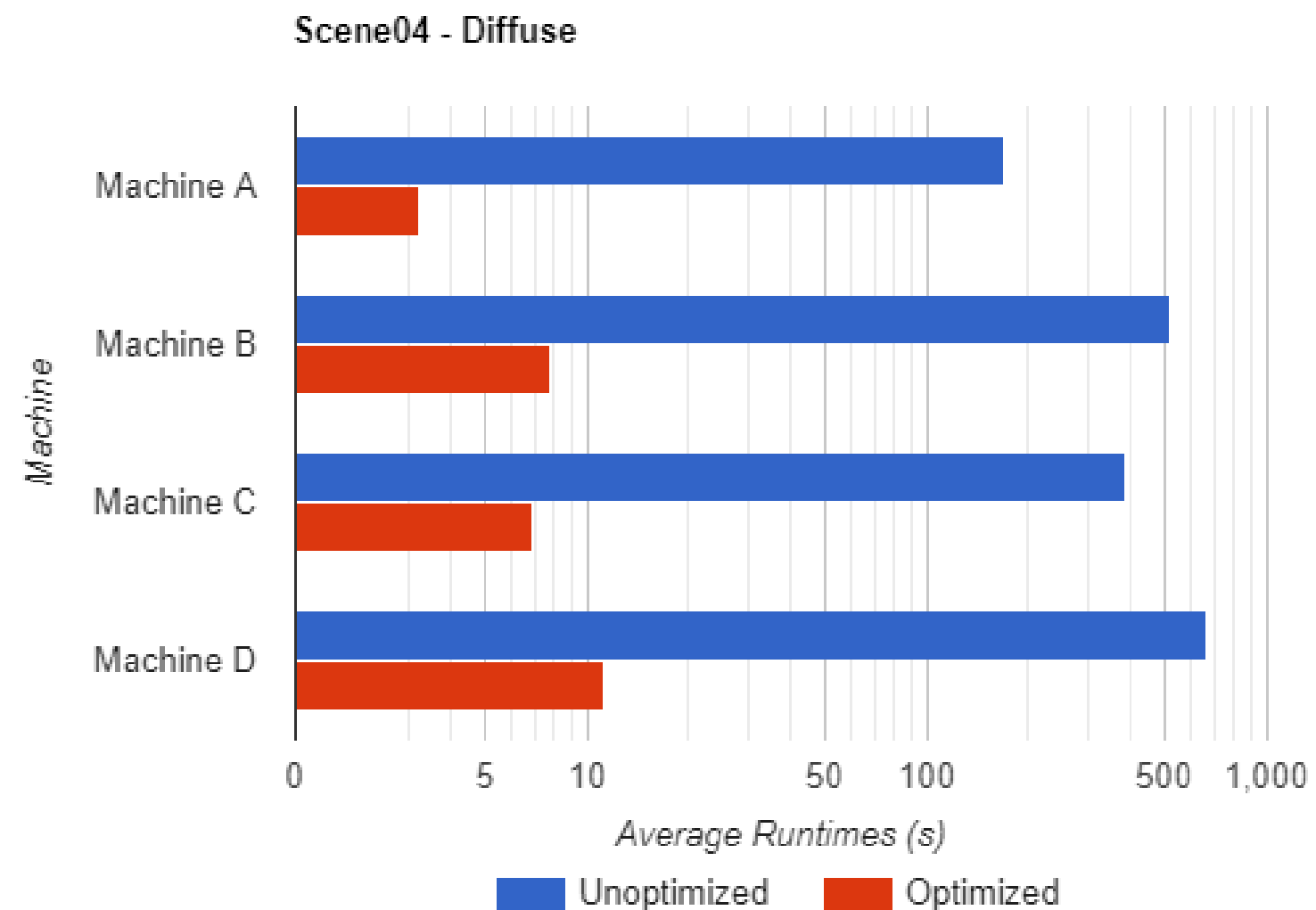
# RESULTS: CODEBASE OPTIMIZATION I + II + III (INCLUDED REFACTORING FUNCTIONS INVOLVING ITERATING OVER MAPS)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



# RESULTS: CODEBASE OPTIMIZATION I + II + III (INCLUDED REFACTORING FUNCTIONS INVOLVING ITERATING OVER MAPS)

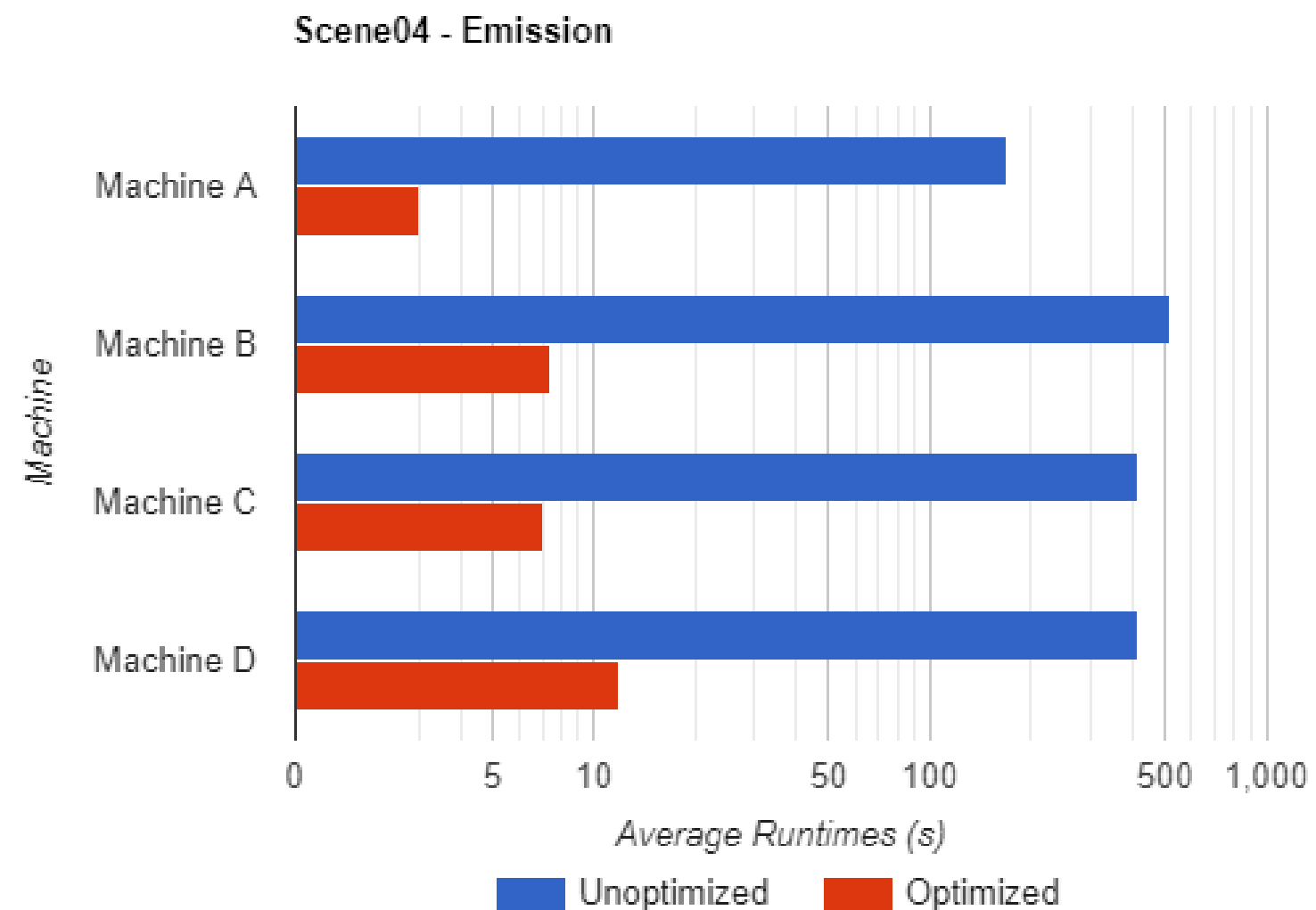
Summarized in the following graphs are the runtimes of the ray tracer program across different hardware





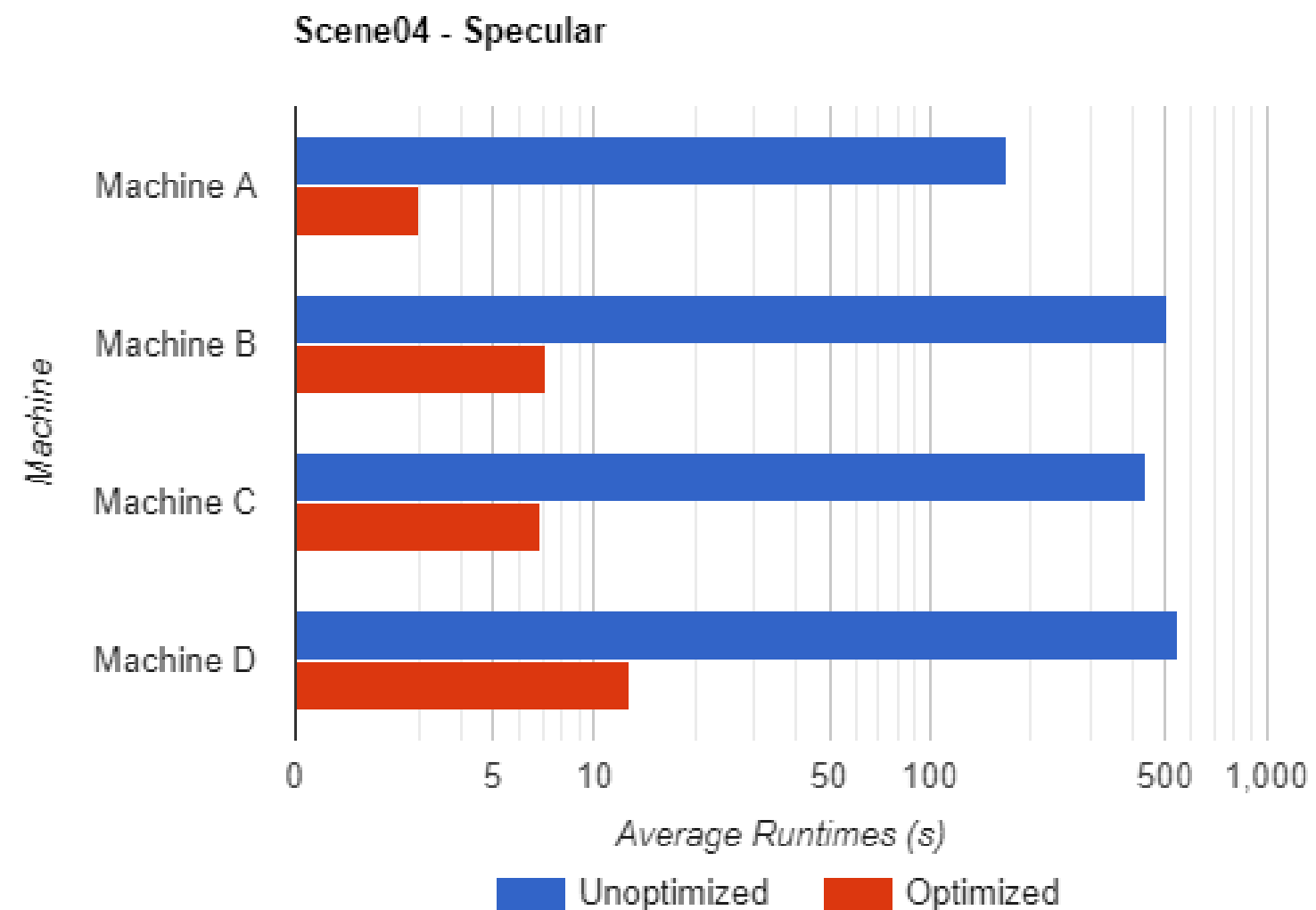
# RESULTS: CODEBASE OPTIMIZATION I + II + III (INCLUDED REFACTORING FUNCTIONS INVOLVING ITERATING OVER MAPS)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



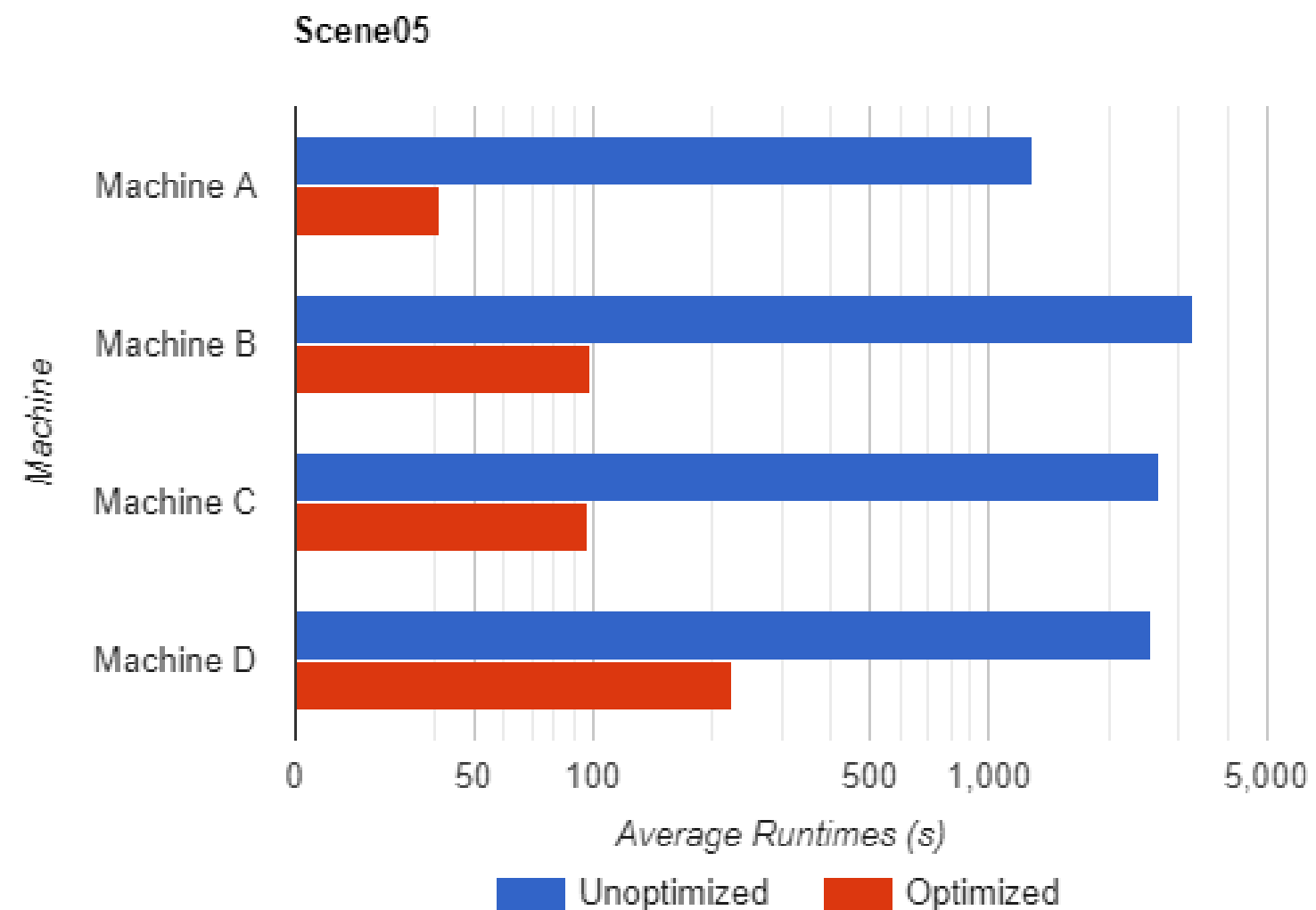
# RESULTS: CODEBASE OPTIMIZATION I + II + III (INCLUDED REFACTORING FUNCTIONS INVOLVING ITERATING OVER MAPS)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



# RESULTS: CODEBASE OPTIMIZATION I + II + III (INCLUDED REFACTORING FUNCTIONS INVOLVING ITERATING OVER MAPS)

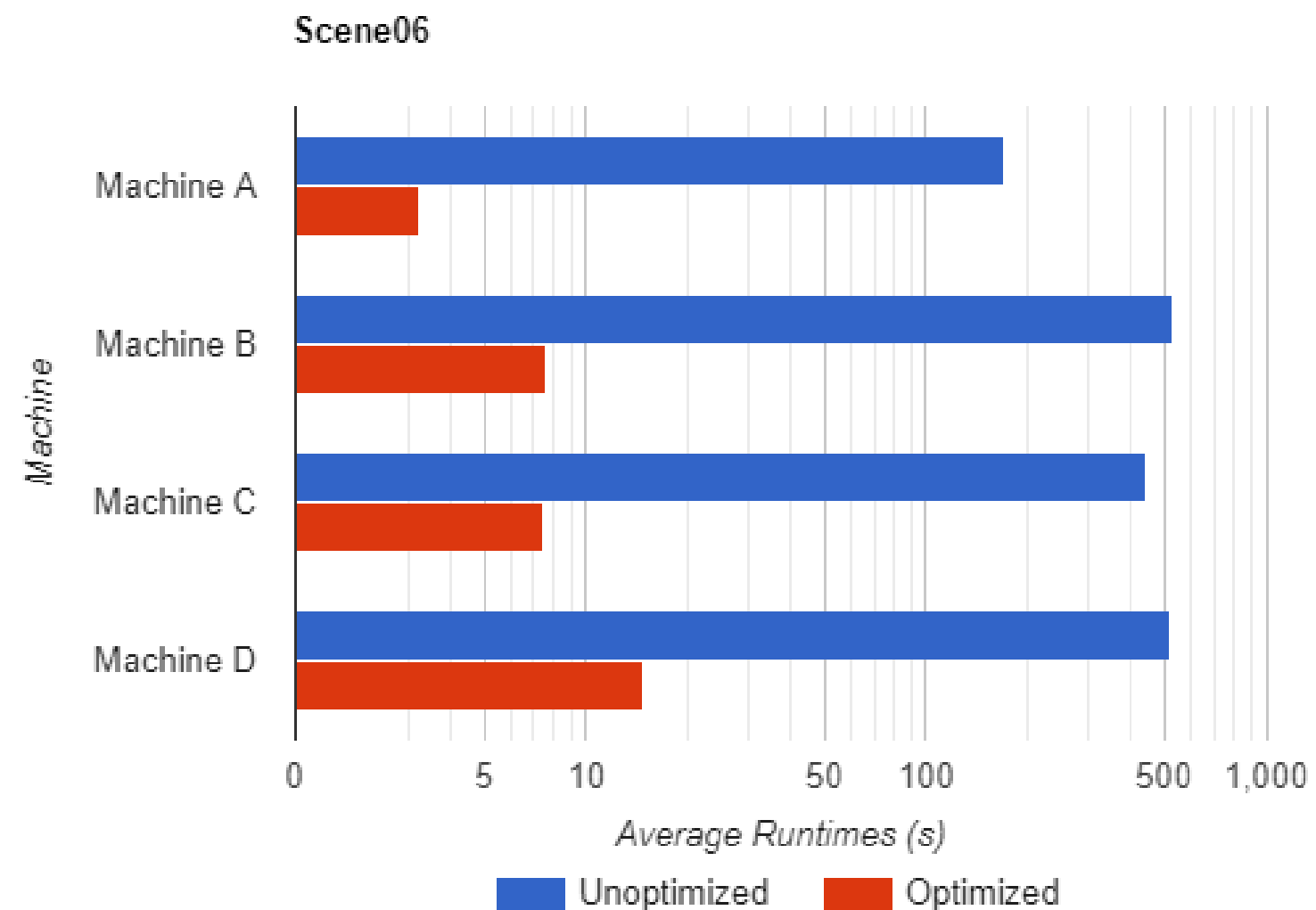
Summarized in the following graphs are the runtimes of the ray tracer program across different hardware





# RESULTS: CODEBASE OPTIMIZATION I + II + III (INCLUDED REFACTORING FUNCTIONS INVOLVING ITERATING OVER MAPS)

Summarized in the following graphs are the runtimes of the ray tracer program across different hardware



RESULTS: CODEBASE OPTIMIZATION I + II + III  
(INCLUDED REFACTORING FUNCTIONS INVOLVING ITERATING OVER MAPS)

Scene 07 is only ran on Machine A (the fastest device available) with a time of 1939 seconds or approximately 32 minutes.

# DISCUSSION: CODEBASE OPTIMIZATION I + II + III

(INCLUDED REFACTORING FUNCTIONS INVOLVING ITERATING OVER MAPS)

Presented below is the average percent decrease in runtime for each scene across all four (4) machines utilized in the project after codebase optimization I + II + III.

Scene	Percent Decrease
Scene 00	75.26
Scene 01	90.09
Scene 02	95.94
Scene 03	98.13
Scene 04 – Ambient	98.19
Scene 04 – Diffuse	98.28
Scene 04 – Emission	98.06
Scene 04 – Specular	98.22
Scene 05	95.38
Scene 06	88.81

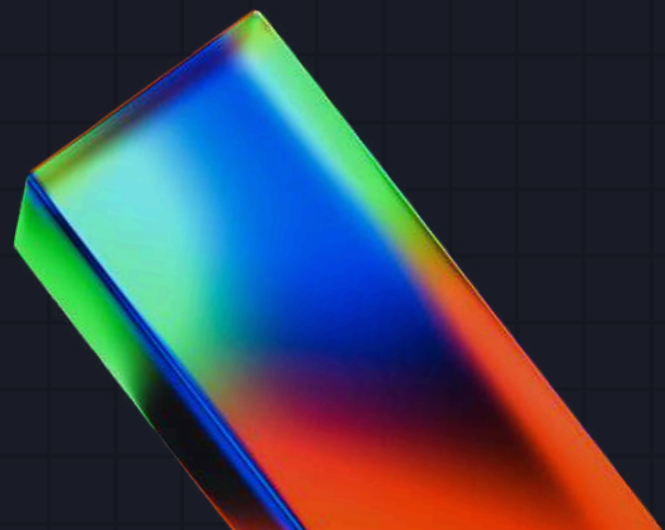
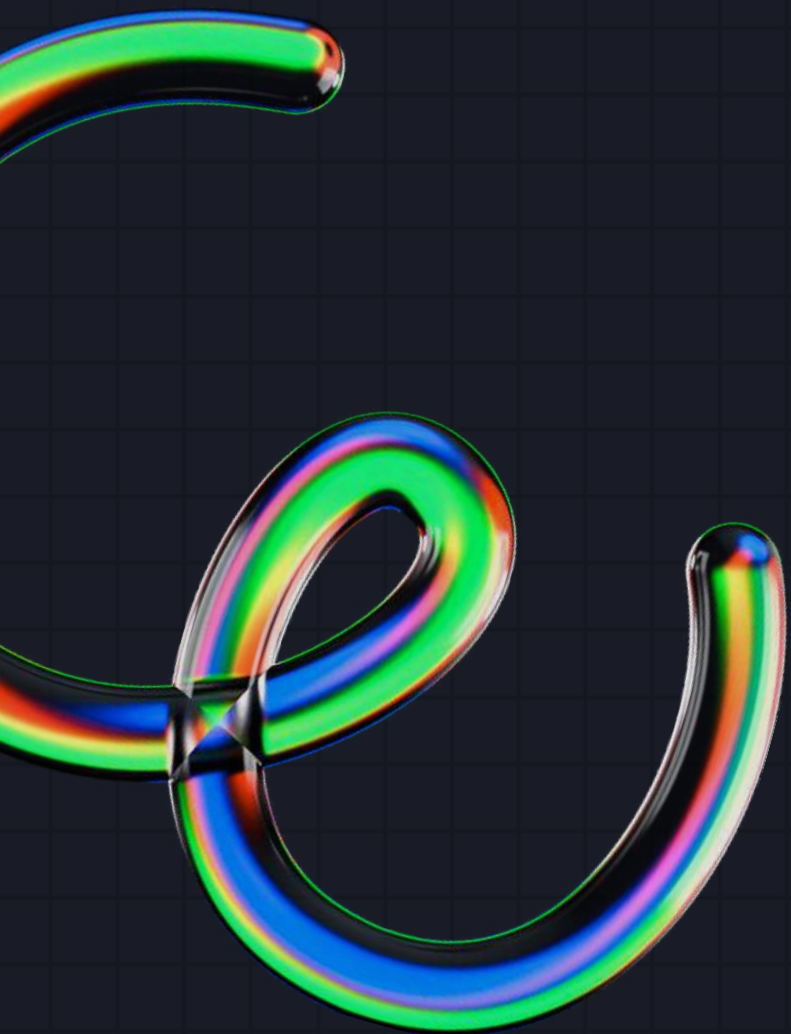
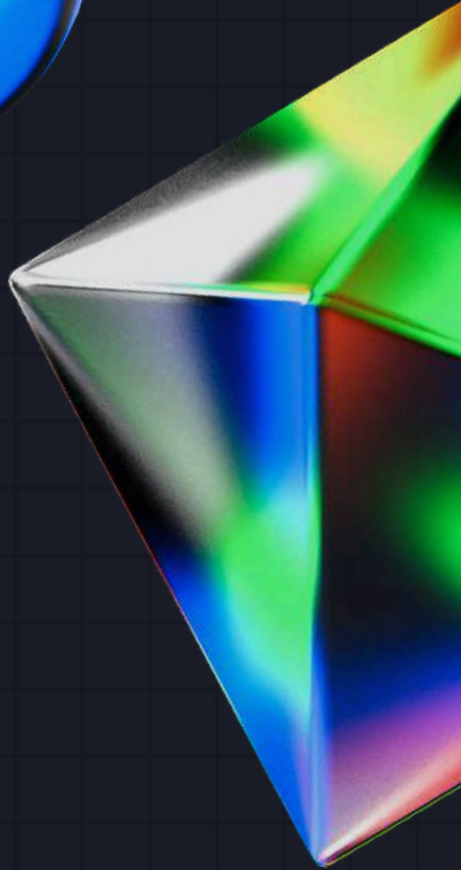


## DISCUSSION: CODEBASE OPTIMIZATION I + II + III

(INCLUDED REFACTORING FUNCTIONS INVOLVING ITERATING OVER MAPS)

After combining all the codebase optimizations done by the group, *Scene 04 - Diffuse* experienced the highest average decrease in runtime with a result of 98.28%. *Scene 00* showed the least improvement with only 75.26% average decrease in runtime.

other code-based  
optimization  
proposals



## DISCUSSION: OTHER CODE-BASED OPTIMIZATION PROPOSALS

- **Linear Algebra Operation Improvements via Basic Linear Algebra Subprograms (BLAS) Library** – since the raytracing algorithm involves lots of vector and matrix manipulations, it is good to utilize these provided optimized linear algebra operation implementations
- **Concurrency** – distributes workload across multiple threads to mitigate downtimes by efficiently utilizing available resources, this is useful especially in ray traversals, ray-surface intersection tests, and shading calculations



## DISCUSSION: OTHER CODE-BASED OPTIMIZATION PROPOSALS

- **Single Instruction, Multiple Data (SIMD) Instructions** – simultaneously perform computations on multiple data elements, also useful in the previously enumerated computations
- **Improve Data Structures (Acceleration Structures)**
  - Spatial Trees like Bounding Volume Hierarchies (BVH) – hierarchizes scene objects into "trees" to detect early terminations and use fewer intersections, but implementation is quite complex
  - Others – oct-tress, kd trees, BSP trees

## DISCUSSION: OTHER CODE-BASED OPTIMIZATION PROPOSALS

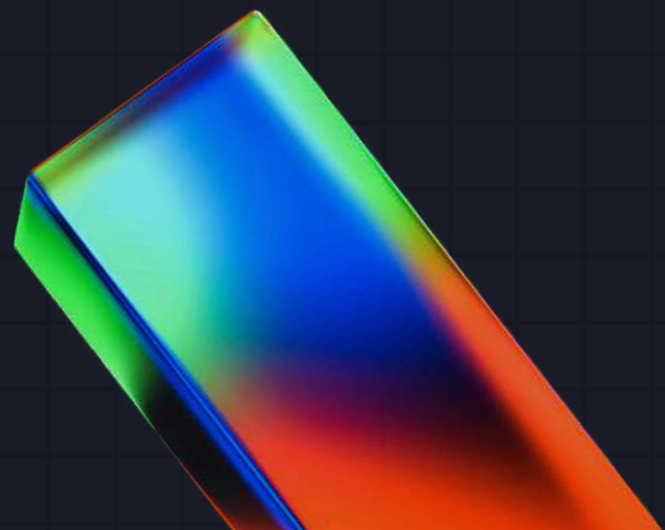
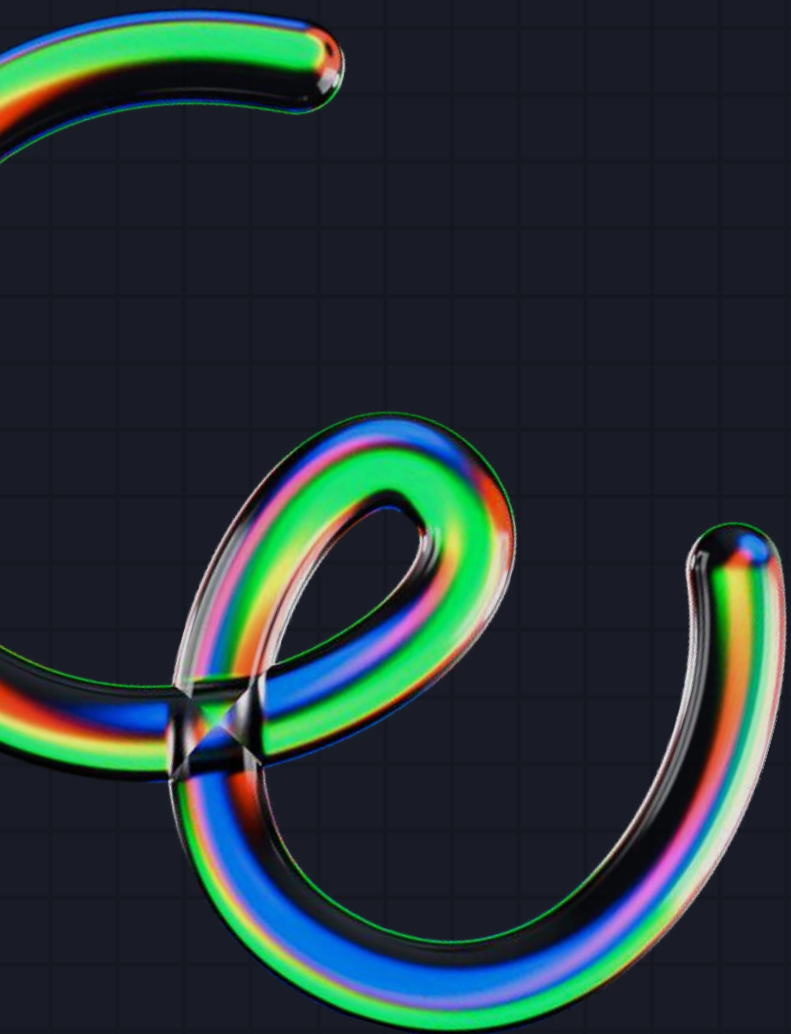
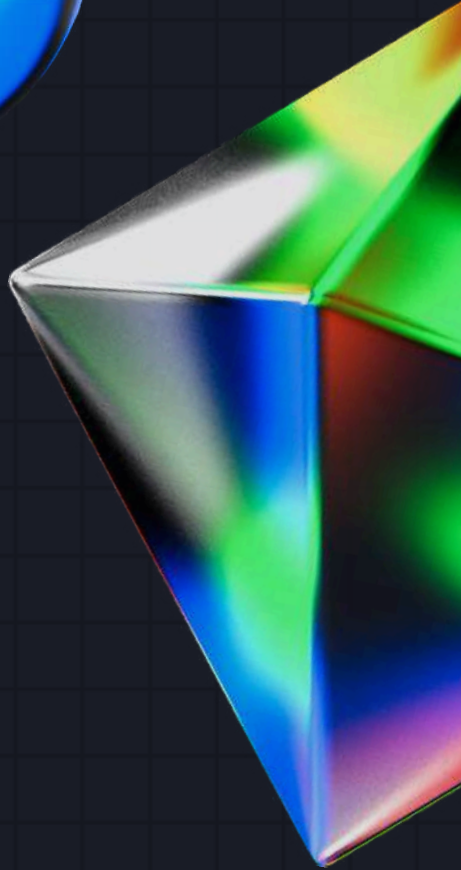
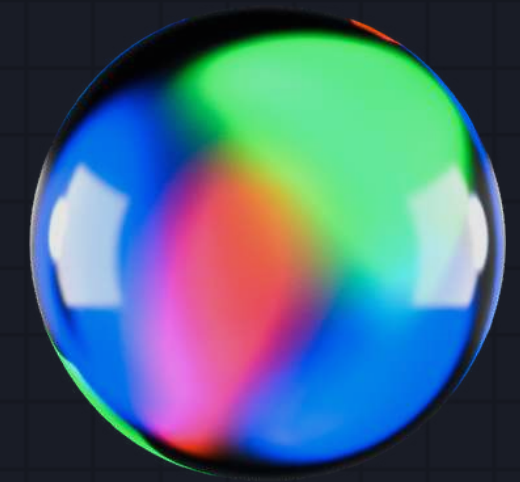
- **Adaptive Sampling** – reallocates rays on areas requiring more complex shading, allowing fewer rays to be used
- Implementing other tracings involving generalized rays
  - **Beam Tracing** – uses thicker rays (beams) to represent groups of rays
  - **Cone Tracing** – uses cone-shaped rays spanning larger scene area

## DISCUSSION: OTHER CODE-BASED OPTIMIZATION PROPOSALS

- **Improve Primitives** – primitives are the vectors, matrices, transformations, geometrical elements, material properties, camera and lights, and improving these would ease the computations required
- **Implement Code on a Lower-Level Programming Language** such as C/C++, assembly, as well as lower-level libraries, that provides more direct accesses and control over the hardware such as memory and CPU



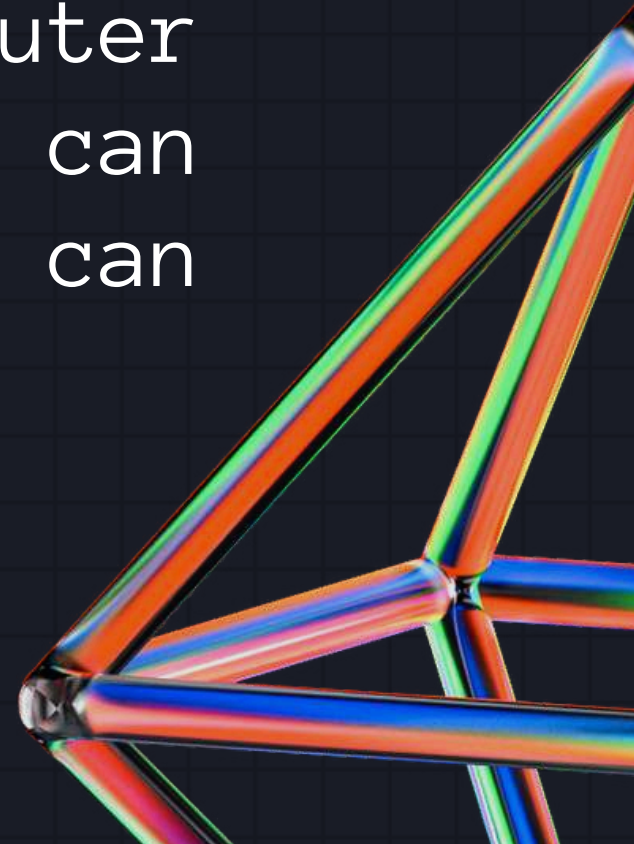
platform-specific  
optimization  
proposals





## DISCUSSION: PLATFORM-SPECIFIC OPTIMIZATION PROPOSALS

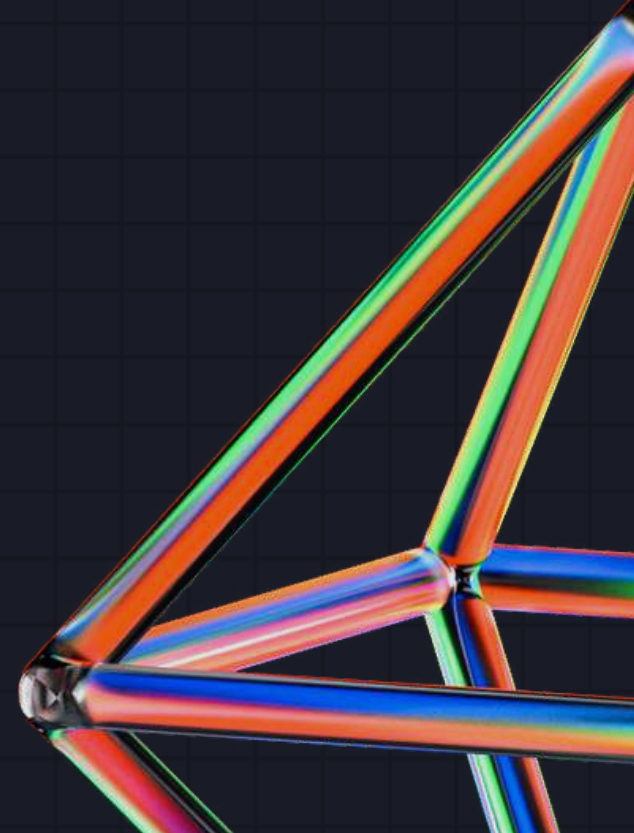
### component overclocking

- **Overclocking** is the action of increasing a component's clock rate, **running it at a higher speed** than it was designed to run. This technique is usually applied to CPUs and GPUs.
  - Increasing a component's clock rate enables a computer component to **run more operations per second** which can help the codebase to run faster since the CPU can process more operations per second ([howtogeek.com](http://howtogeek.com)).
- 



## DISCUSSION: PLATFORM-SPECIFIC OPTIMIZATION PROPOSALS

### component overclocking

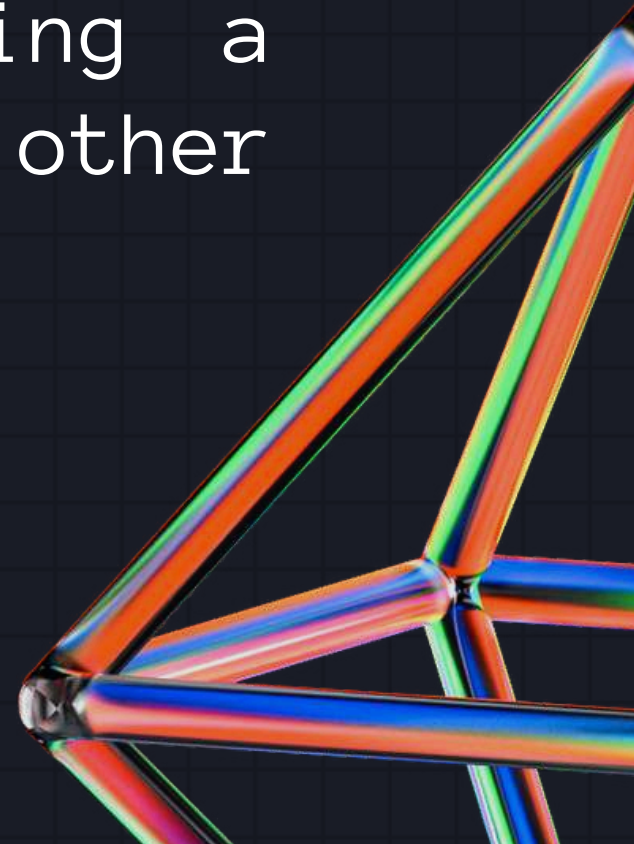
- This technique can squeeze more performance out of your components provided that a sufficient cooling method is in place ([howtogeek.com](http://howtogeek.com)) [8].
- 



## RAM configuration


- The Random Access Memory or RAM is the temporary storage in a computer that gives applications a place to store and access data on a short-term basis. Increasing a machine's RAM means that **more data can be accessed and read almost instantly** as opposed to being written on the machine's hard drive.
- There are **different configurations** that provide more efficient use of RAM such as the following:
  1. Single Channel Configuration
  2. Flex Mode Configuration
  3. Dual Channel Configuration

## RAM configuration

- **Single Channel Configuration**
    - In a **single-channel RAM configuration**, the RAM operates on a 64-bit data memory channel. As only one of the memory channels on your motherboard is functional, the memory throughput is limited to the rated speed of the RAM stick in the machine, consequently rendering a notable difference in speed compared to other configurations
- 

## RAM configuration

- **Flex Mode Configuration**

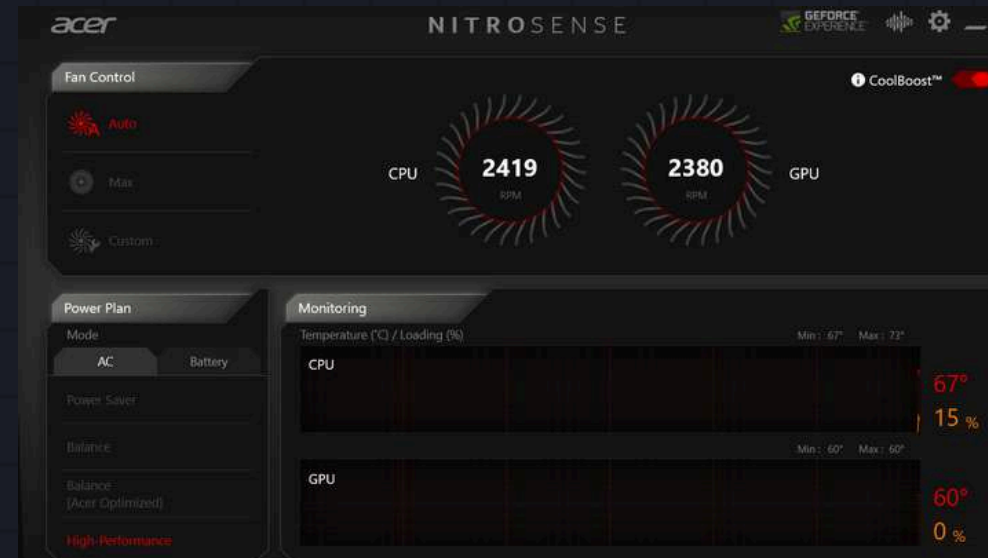
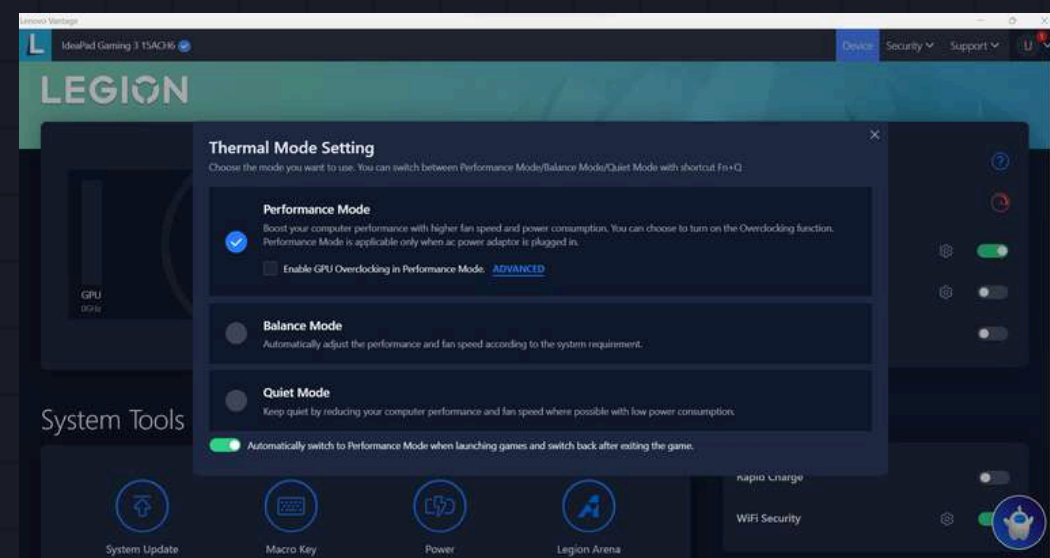
- **Flex Mode** aims to **optimize an otherwise less optimal situation**. When a computer has two unequal RAM modules, the system switches to Flex Mode instead of operating solely in single-channel mode.
  - As an example, machine B operated on 4GB and 8GB RAM modules. 4GB from each module (totaling 8GB) will work in dual-channel mode, while the remaining 4GB from the 8GB module will function in single-channel mode [9].
- 




# DISCUSSION: PLATFORM-SPECIFIC OPTIMIZATION PROPOSALS

## machine performance mode

- Machines A, B, and C are equipped with different operating modes that are designed to improve the machine's performance. Enabling the highest operating modes can provide substantial increase in the code's runtime [10].



## reduce background processes

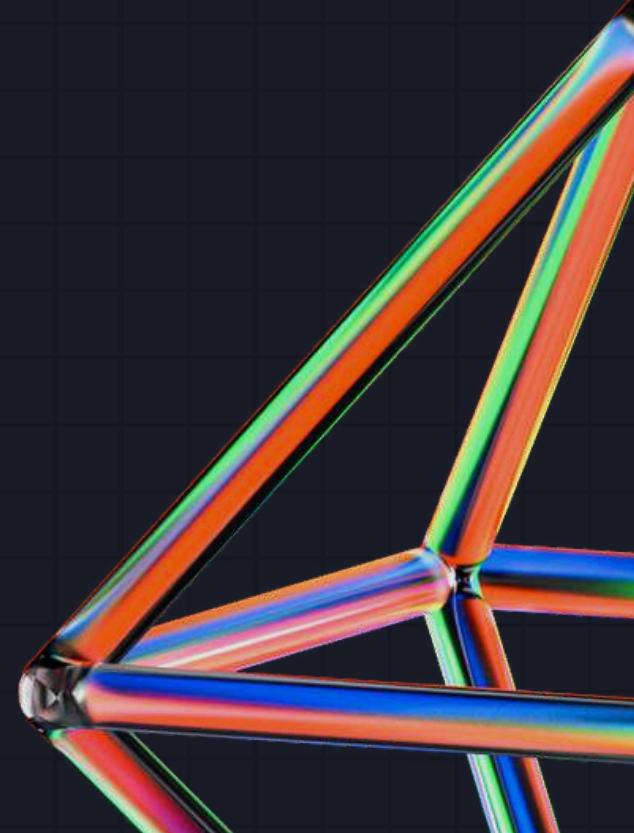
- Reducing background processes can help improve the runtime of code by minimizing the competition for system resources such as CPU, memory, and disk I/O. When a computer is running multiple processes simultaneously, each process requires a share of these resources to execute its tasks.
  - Some of the main background applications or processes that can greatly affect the code's runtime are Google Chrome and Screen sharing.
- 





## DISCUSSION: PLATFORM-SPECIFIC OPTIMIZATION PROPOSALS

### reduce background processes

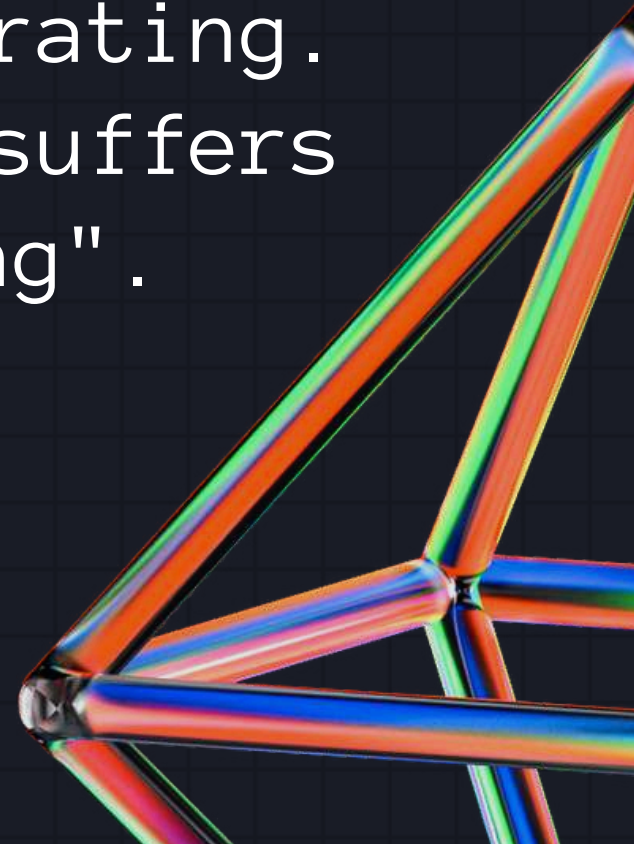
- The performance will also experience significant improvements if the program is executed one at a time rather than running it three times in parallel across different terminals.
- 





## DISCUSSION: PLATFORM-SPECIFIC OPTIMIZATION PROPOSALS

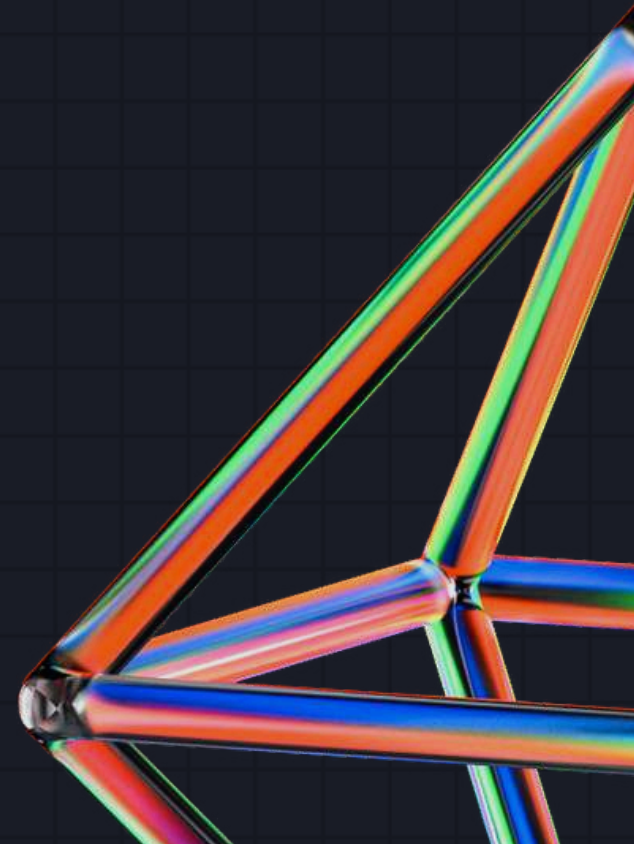
### prevent thermal throttling

- Thermal throttling is the process that computer components such as the CPU and GPU perform when they start operating under extreme temperatures.
  - The process generally adjusts the clock speed of the component based on the amount of heat it is generating. During the duration of this process, the component suffers significant performance dips, Hence the term "throttling".
- 

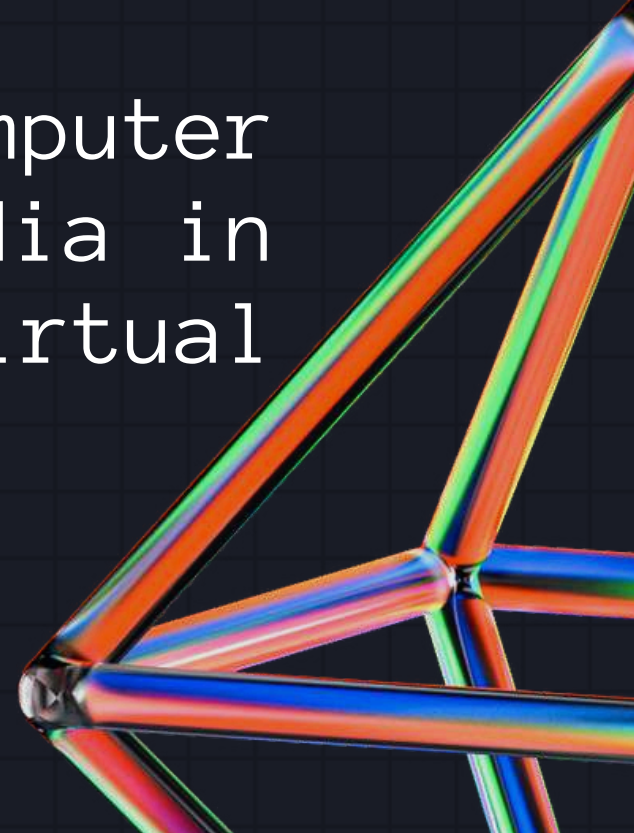


## DISCUSSION: PLATFORM-SPECIFIC OPTIMIZATION PROPOSALS

### prevent thermal throttling

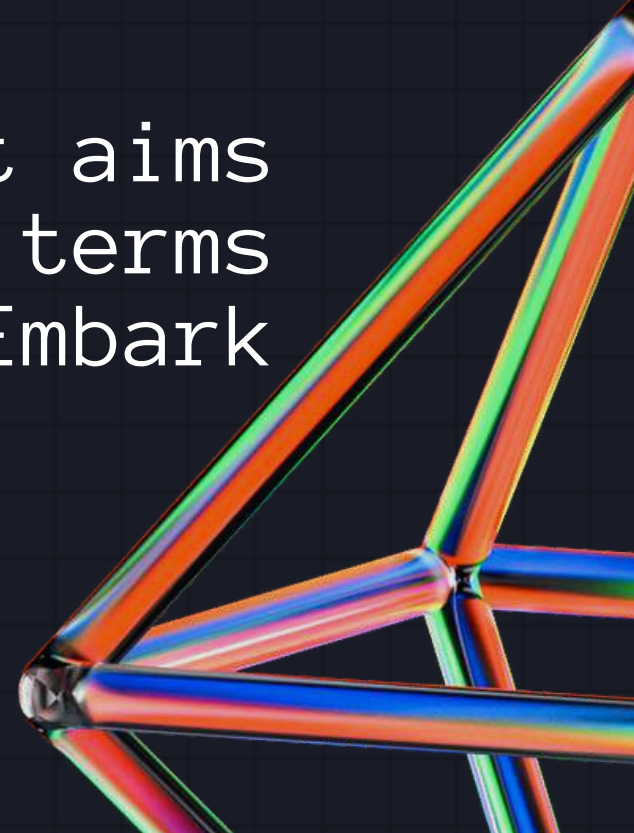
- To minimize the occurrence of such process during runtime, the machines can be placed on top of a cooling pad or be operated in a cool environment [11][12].
- 

## GPU programming

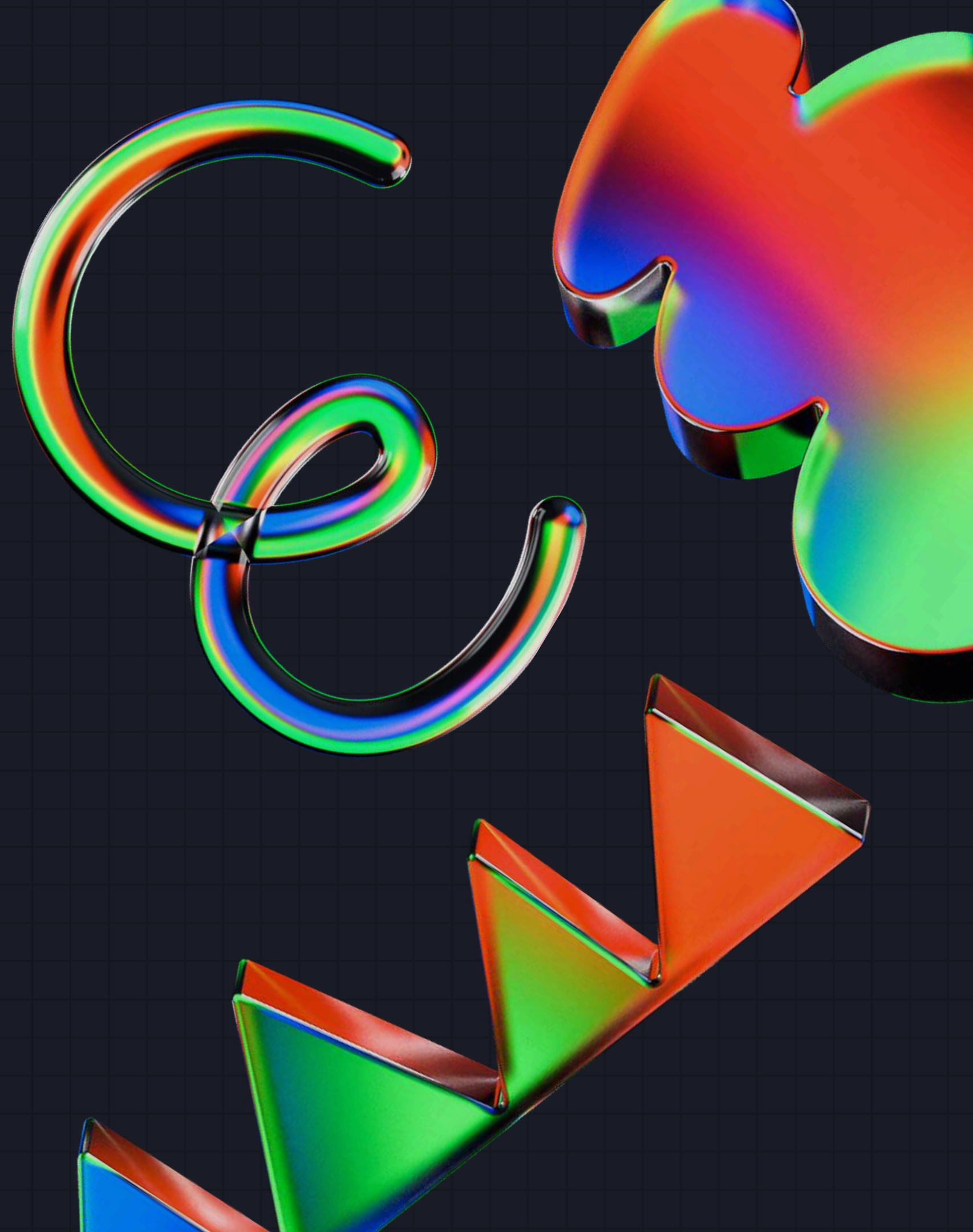
- Graphical Processing Units or GPUs are originally designed exclusively for computer graphics. However, GPUs today are extensively used for general purpose computing as well. Additionally, GPU-driven parallel computing is used for scientific modelling, machine learning, and other parallelization tasks.
  - There are GPU Programming APIs such as the Computer Unified Device Architecture (CUDA) developed by Nvidia in 2006, that gives direct access to the GPU's virtual instruction set for the execution of compute kernels.
- 



## GPU programming

- CUDA programming model allows developers to use CUDA-enabled GPUs for general purpose processing in C/C++ and FORTRAN, with third party wrappers in Python, R, and other programming languages [13].
  - Unfortunately, Rust does not have a wrapper for CUDA programming at the moment.
  - However, there are numerous open-source projects that aims to fill this gap between Rust and other languages in terms of GPU Programming such as the rust-gpu project by Embark Studios [14].
- 

# CONCLUSION





# CONCLUSION

- Raytracing involves linear algebra and lots of simple computations done repeatedly, thus optimizing linear algebra operations and implementing parallelism yields significant decrease in runtimes.
  - Reordering MMM implemented as nested for loops have little effects to runtimes due to the matrix size used being small.
- Transitioning to better data structures accordingly eases computational requirements and reduces time complexity.

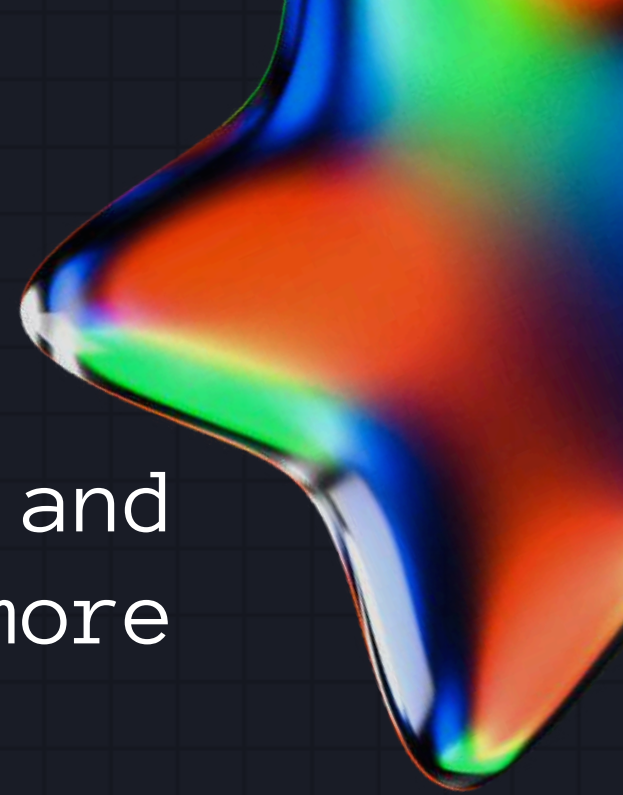


# CONCLUSION

- Mitigating redundant operations, usually by changing their order in the code, reduces the total processes and makes the code easier to read.
- Optimizing the device used for simulation by changing its settings accordingly as well as closing unnecessary process also factors.
  - The brand, model and specifications of the hardware used matters.

# CONCLUSION

- Having more elements in the scene, such as shapes and lights, causes massive runtime increases since more intersections are possible.
- The optimizations implemented by the group significantly reduced the runtimes across different platforms, by approximately 98%.
- Overall, raytracing is a computationally expensive rendering method, but there are lots of possible optimizations that greatly improves performance, with the expense of complex implementations.





# BIBLIOGRAPHY

[1] F.P. Vidal, F. Bello, K.W. Brodlie, N.W. John, D. Gould, R. Phillips, and N.J. Avis, "Principles and applications of computer graphics in medicine," *Computer Graphics Forum*, vol. 25, no. 1, pp. 113–137, 2006.

[2] P. H. Christensen, J. Fong, D. M. Laur and D. Batali, "Ray Tracing for the Movie `Cars'," 2006 IEEE Symposium on Interactive Ray Tracing, Salt Lake City, UT, USA, 2006, pp. 1–6, doi: 10.1109/RT.2006.280208.

[3] A. Tewari, J. Thies, B. Mildenhall, P. Srinivasan, E. Tretschk, W. Yifan, C. Lassner et al. "Advances in neural rendering." *Computer Graphics Forum*, vol. 41, no. 2, pp. 703–735. 2022.

[4] Ray Tracing and Other Methods (August 1, 2018). Accessed June 30, 2023. [Online Video]. Available: <https://www.youtube.com/watch?v=LAsnQoBUG4Q>

[5] A. S. Glassner, *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989.





# BIBLIOGRAPHY

[6] Fourier Transform in 5 minutes: The Case of the Splotched Van Gogh, Part 3 (June 6, 2022). Accessed June 30, 2023. [Online Video]. Available: <https://www.youtube.com/watch?v=JciZYrh36LY>

[7] F. Quatresooz, S. Demey and C. Oestges, "Tracking of Interaction Points for Improved Dynamic Ray Tracing," in IEEE Transactions on Vehicular Technology, vol. 70, no. 7, pp. 6291-6301, July 2021, doi: 10.1109/TVT.2021.3081766.

[8] C. Hoffman. "What Is Overclocking? The Beginner's Guide to Understanding How Geeks Speed Up Their PCs". How-To Geek. <https://www.howtogeek.com/165064/what-is-overclocking-the-absolute-beginners-guide-to-understanding-how-geeks-speed-up-their-pcs/> (accessed June 30, 2023).

[9] "What is RAM and what does RAM do?". Crucial.com. <https://www.crucial.com/articles/about-memory/support-what-does-computer-memory-do> (accessed June 30, 2023).



# BIBLIOGRAPHY

[10] R. Rowley. "How to Select Your Perfect RAM Configuration". Overclockers. <https://www.overclockers.co.uk/blog/how-to-select-your-perfect-ram-configuration/> (accessed June 30, 2023).

[11] "Definition of thermal throttling". PCMag. <https://www.pcmag.com/encyclopedia/term/thermal-throttling> (accessed June 30, 2023).

[12] Y. Abidi. "What Is CPU Thermal Throttling and How Does It Affect Performance?". MakeUseOf. <https://www.makeuseof.com/what-is-cpu-thermal-throttling/> (accessed June 30, 2023).

[13] M. Levinas. "A Complete Introduction to GPU Programming With Practical Examples in CUDA and Python". Cherry savers. <https://www.cherryservers.com/blog/introduction-to-gpu-programming-with-cuda-and-python> (accessed June 30, 2023).





# BIBLIOGRAPHY

[14] EmbarkStudios. rust-gpu (Version 0.8). GitHub. <https://github.com/EmbarkStudios/rust-gpu> (accessed June 30, 2023)

[15] Ray Tracing in 5 minutes: Part 2 -- implementing a basic ray tracer (October 16, 2022). Accessed June 30, 2023. [Online Video]. Available: <https://www.youtube.com/watch?v=mT011vinv-U>

[16] Ray Tracing in 5 minutes: Part 3 -- recursive ray tracing (October 16, 2022). Accessed June 30, 2023. [Online Video]. Available: <https://www.youtube.com/watch?v=tUh6gCx08LI>

[17] Rasterizer Algorithm Explanation (October 31, 2019). Accessed June 30, 2023. [Online Video]. Available: <https://www.youtube.com/watch?v=t7Ztio8cwqM>





# BIBLIOGRAPHY

[18] What is Ray Tracing? (September 8, 2018). Accessed June 30, 2023. [Online Video]. Available: <https://www.youtube.com/watch?v=0FM1PUEAZfs>

[19] Speed up your Rust code with Rayon (February 20, 2023). Accessed June 30, 2023. [Online Video]. Available: <https://www.youtube.com/watch?v=YxG7PhZ3fb4>

[20] University of California San Diego. (2023). Computer Graphics. [Online]. Available: <https://cseweb.ucsd.edu/~viscomp/classes/cse167/wi23/schedule.html>

[21] Ray Tracing in 5 minutes (October 16, 2022). Accessed June 30, 2023. [Online Video]. Available: <https://www.youtube.com/watch?v=H5TB217zq6s>