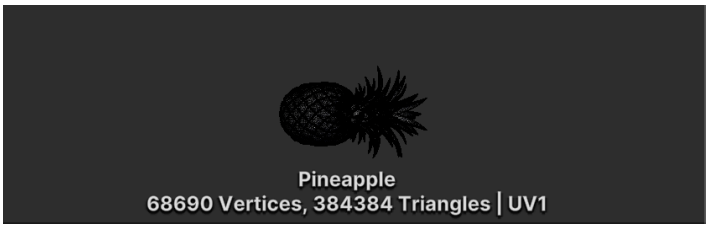
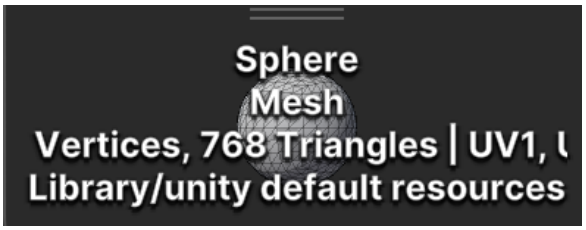
CSE462/562 – Augmented Reality (Fall 2024)

Homework #4
Physics Rendering via Ray Casting with Black Holes

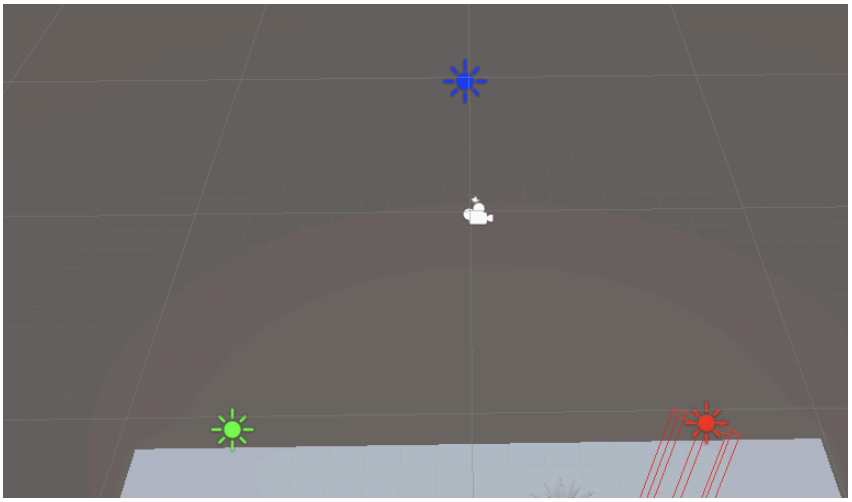
Berru Lafcı - 1901042681

- **Scene:** A 3D world with at least 4 objects and a total of 10,000 triangles. I made this with 2 different scenes. One of has 14 spheres (768*14 = 10752) and the other one has 4 pineapples.

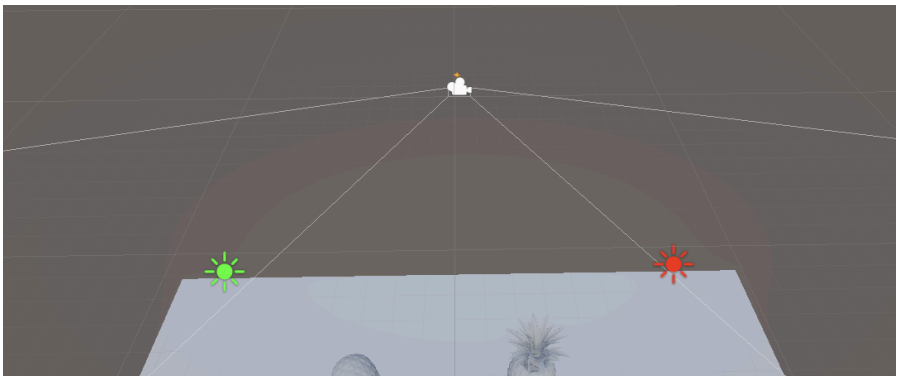




- **Lighting:** Three light sources.



- **Materials:** All objects use one type of material.
- **Camera:** A pinhole camera.



- **Ray Physics:** Rays travel straight unless influenced by a black hole, where they follow a quadratic curve.

The project includes two key scripts: **ObjectRay.cs** for handling object-based rays and visualizing them with Debug.Drawline and **RayCastRendered.cs** for rendering the scene and saving a png image for the scene.

Implementation Details

1. ObjectRay.cs

Ray Generation:

- The method GenerateRandomDirections initializes a set of random directions for the rays. This randomness is essential to see different kinds of rays in the scene.

Ray Casting Logic:

- The TraceRay method takes the origin and direction of a ray as inputs. Ray's origin is the starting point of the ray on the object's surface, converted to world space.
- First, it checks for intersections. If an intersection is detected within the specified range, a straight ray is drawn to the point of contact.

```
// If the ray hits something (like plane), draw a white line up to the hit point
if (Physics.Raycast(ray, out RaycastHit hit, straightRayLength))
{
    Debug.DrawLine(origin, hit.point, Color.white);
    return;
}
```

- If no intersection is detected, the method evaluates whether the ray is influenced by a black hole. This influence is determined by calculating the angle between the ray's direction and the vector pointing to the black hole. If the angle is smaller than the threshold, the ray is considered to curve toward the black hole. Angles below threshold are drawn as curved using a quadratic calculation.

toBlackHole: Calculates the direction from the ray's starting point to the black hole's position.

Vector3.Angle: Finds the angle between the ray's direction and the direction to the black hole.

```
// If we do have a black hole, measure the angle
Vector3 toBlackHole = blackHole.position - origin;
float angle = Vector3.Angle(direction, toBlackHole);

// If the angle is below the threshold, draw a curved ray
if (angle < angleThreshold)
{
    Debug.Log("Angle to black hole is below threshold. Drawing a curved ray.");
    DrawCurvedRay(origin, blackHole.position, Color.red);
}
else
{
    Debug.Log("Angle to black hole is above threshold. Drawing a straight ray.");
    DrawStraightRay(origin, direction, straightRayLength, Color.white);
}
```

Drawing Curved Rays:

- The midpoint between the ray's origin and the black hole determines the curve of the ray. A quadratic Bézier curve is used for calculating the curve.

```
1 reference
private void DrawCurvedRay(Vector3 start, Vector3 end, Color color)
{
    Vector3 midpoint = (start + end) * 0.5f + Vector3.down; // Calculate the midpoint between start and end

    int steps = 30;
    float dt = 1f / steps;

    Vector3 prevPoint = start;
    for (int i = 1; i <= steps; i++)
    {
        float t = i * dt;

        // Compute the quadratic curve
        Vector3 currPoint = (1 - t)*(1 - t)*start + 2*(1 - t)*t*midpoint + t*t*end;

        Debug.DrawLine(prevPoint, currPoint, color);
        prevPoint = currPoint;
    }
}
```

---

2. RayCastRenderer.cs

Image Rendering:

- This method iterates over all pixels in the render resolution (640x480) and generates rays for each pixel using GenerateCameraRay. Each pixel's color is determined by tracing the corresponding ray through the scene. After processing all pixels, Texture2D.Apply is called to finalize the texture with updated pixel data.

```
1 reference
void BuildRayCastImage()
{
    for (int y = 0; y < imageHeight; y++)
    {
        for (int x = 0; x < imageWidth; x++)
        {
            // Generate a ray from camera through pixel
            Ray pixelRay = GenerateCameraRay(x, y);

            // Trace the ray and get the color
            Color color = TraceRay(pixelRay);

            // Write the color into the Texture2D
            renderTexture.SetPixel(x, y, color);
        }
    }

    // Apply all pixel changes
    renderTexture.Apply();
}
```

Generating Camera Rays:

- Normalize the pixel coordinates. These coordinates are transformed into a ray direction using the camera's field of view and aspect ratio.
- The ray is then converted from local camera space to world space.

Tracing Rays:

- If the ray encounters a black hole, the TraceQuadraticPath method calculates its curve. The direction of the curve is checked step-by-step to determine if the ray intersects with any object before reaching the black hole.
- If no black hole is nearby, the TraceStraightRay method is used, which checks if the ray hits an object directly.

```
Color TraceRay(Ray ray)
{
    // If a black hole is present, check if the ray is close to it
    if (blackHole != null)
    {
        Vector3 toBH = blackHole.position - ray.origin; // Subtract black hole position from ray origin
        float angle = Vector3.Angle(ray.direction, toBH);

        // If angle is below the threshold, do a curved ray
        if (angle < 10f)
        {
            return TraceQuadraticPath(ray.origin, blackHole.position);
        }
    }

    // Otherwise, do a normal straight ray
    return TraceStraightRay(ray);
}
```

Handling Shadows and Lambertian Shading:

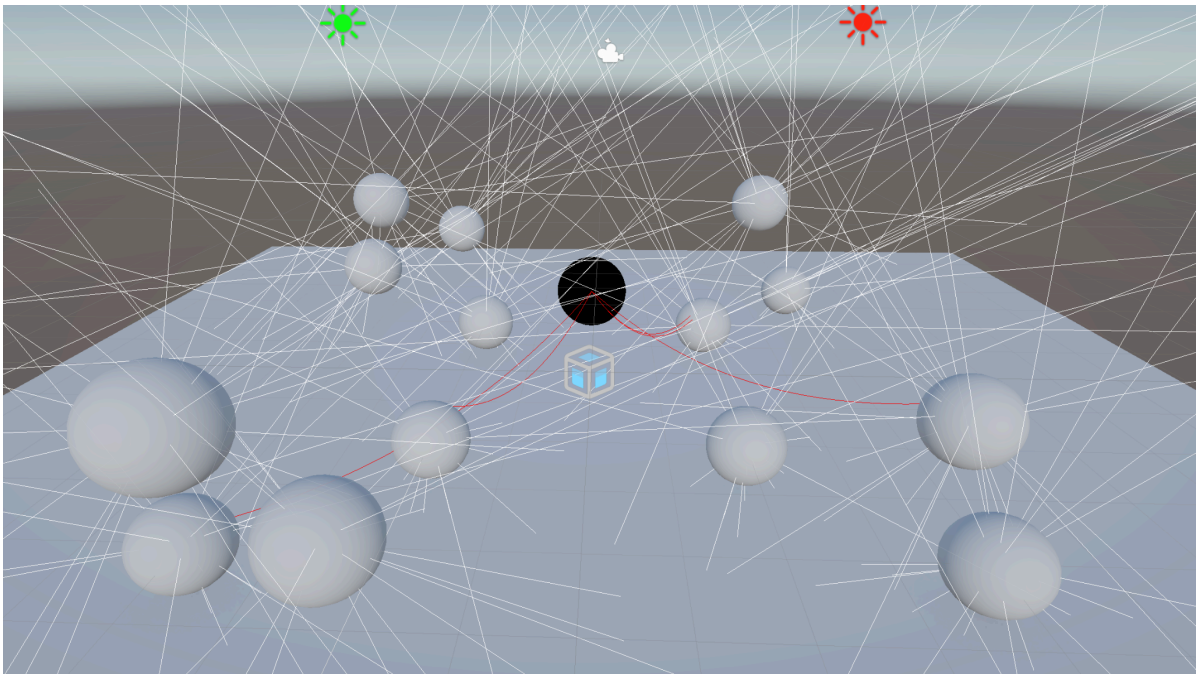
- For each light in the scene, the method calculates how much light reaches the hitPoint (the point where the ray intersects the object). For a given light source, it calculates the direction from the hitPoint to the light source and determines the distance to the light source.
- ```
  Vector3 toLight = (lt.transform.position - hitPoint).normalized; // Direction to light
  float distanceToLight = Vector3.Distance(hitPoint, lt.transform.position); // Distance to light
  ```
- If the ray intersects any object before reaching the light, the point is considered to be in shadow for this light.
- ```
  // Check if the hit point is in shadow of this light
  bool isInShadow = Physics.Raycast(hitPoint + normal * 0.001f, toLight, distanceToLight);
  if (isInShadow)
  {
      continue;
  }
  ```
- If the point is not in shadow, the diffuse lighting contribution is calculated. The contribution of the light to the finalColor is calculated as:
    - baseColor: The color of the object.

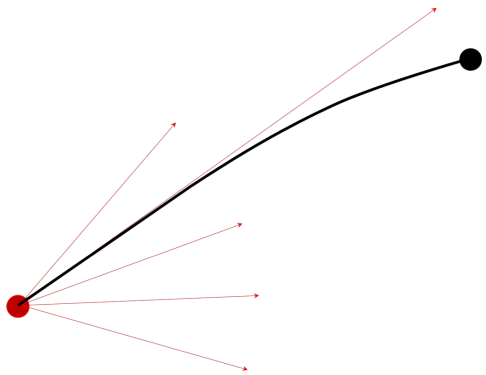
- lt.color: The color of the light source.
- lt.intensity: The intensity of the light source.
- ndotl: The diffuse factor determined by the Lambertian model.

```
// Compute the diffuse contribution
float ndotl = Mathf.Max(0f, Vector3.Dot(normal, toLight));
finalColor += baseColor * lt.color * lt.intensity * ndotl;
```

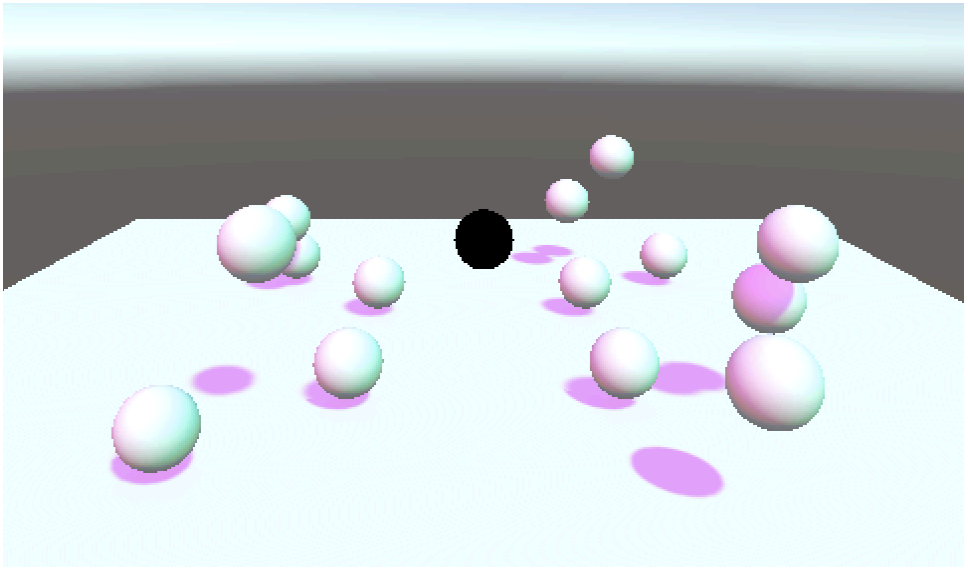
Results

With 14 spheres you can see straight white rays and curved red rays to the black hole. I applied similar logic with the pdf file.





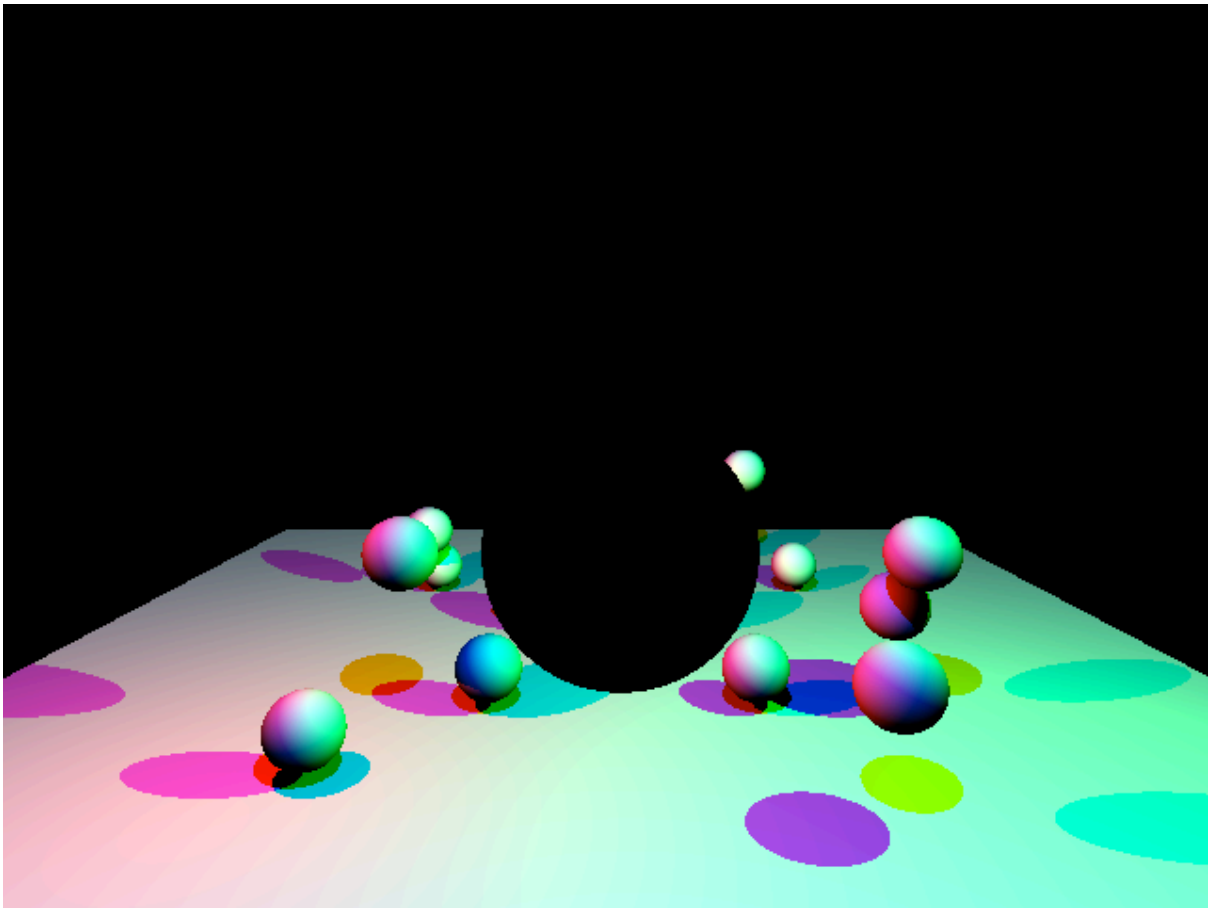
Game screen:



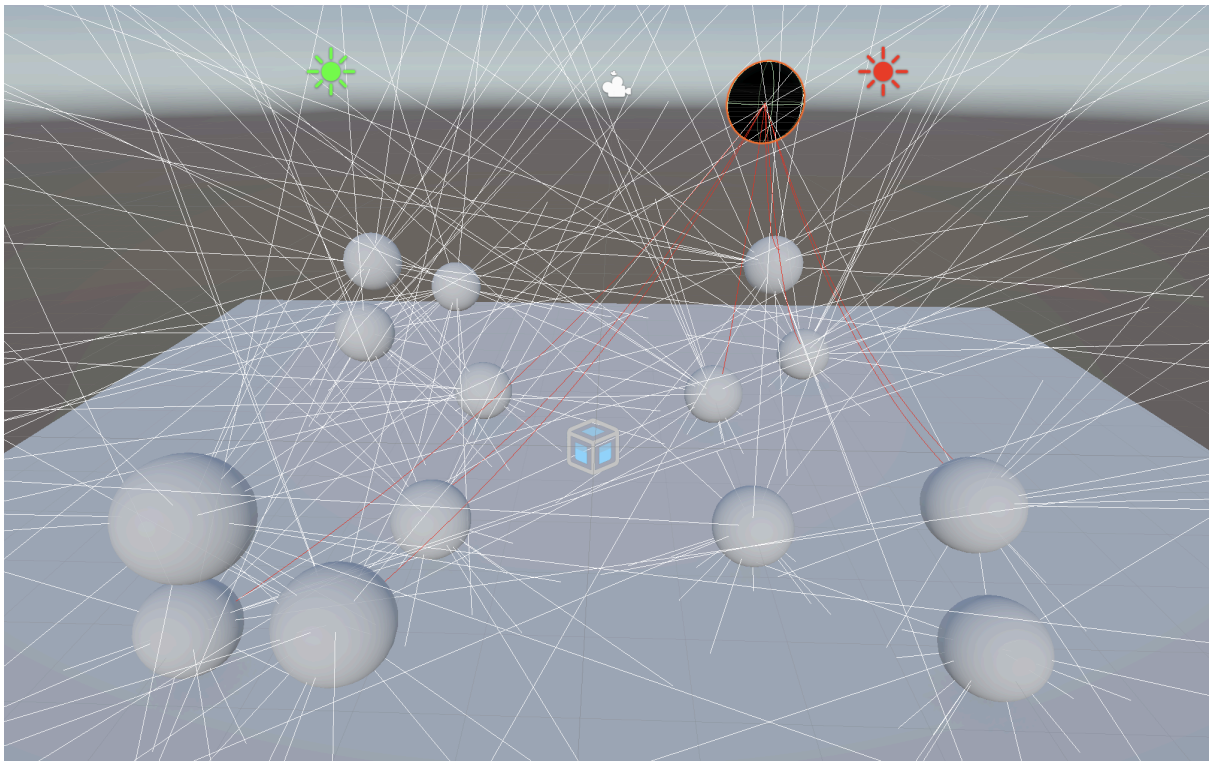
The saved png image. It shows different color of shadows because of the isInShadow variable. The middle black circle is black hole and it is returned in the TraceQuadraticPatch method.

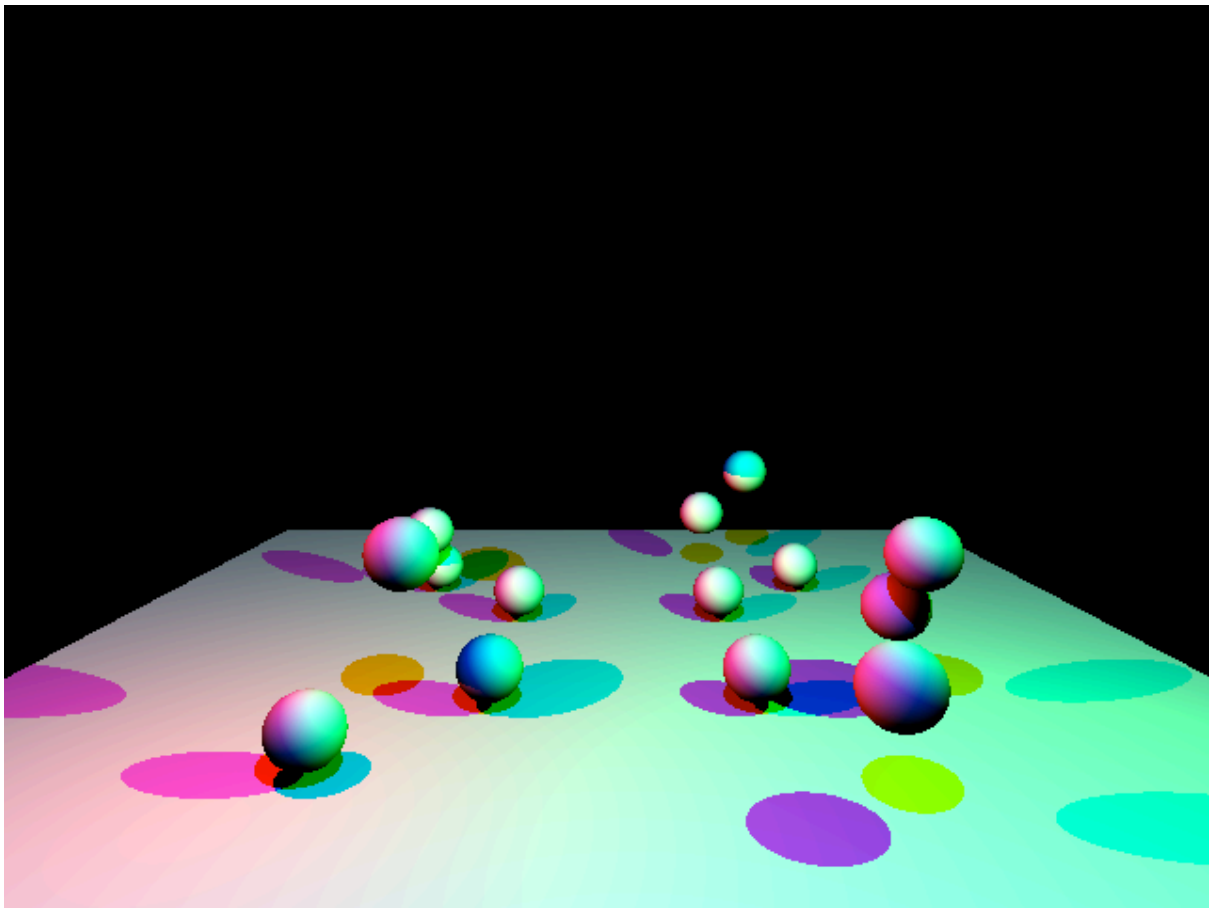
```
// If we made it to the black hole with no hits, return black
return Color.black;
```

.

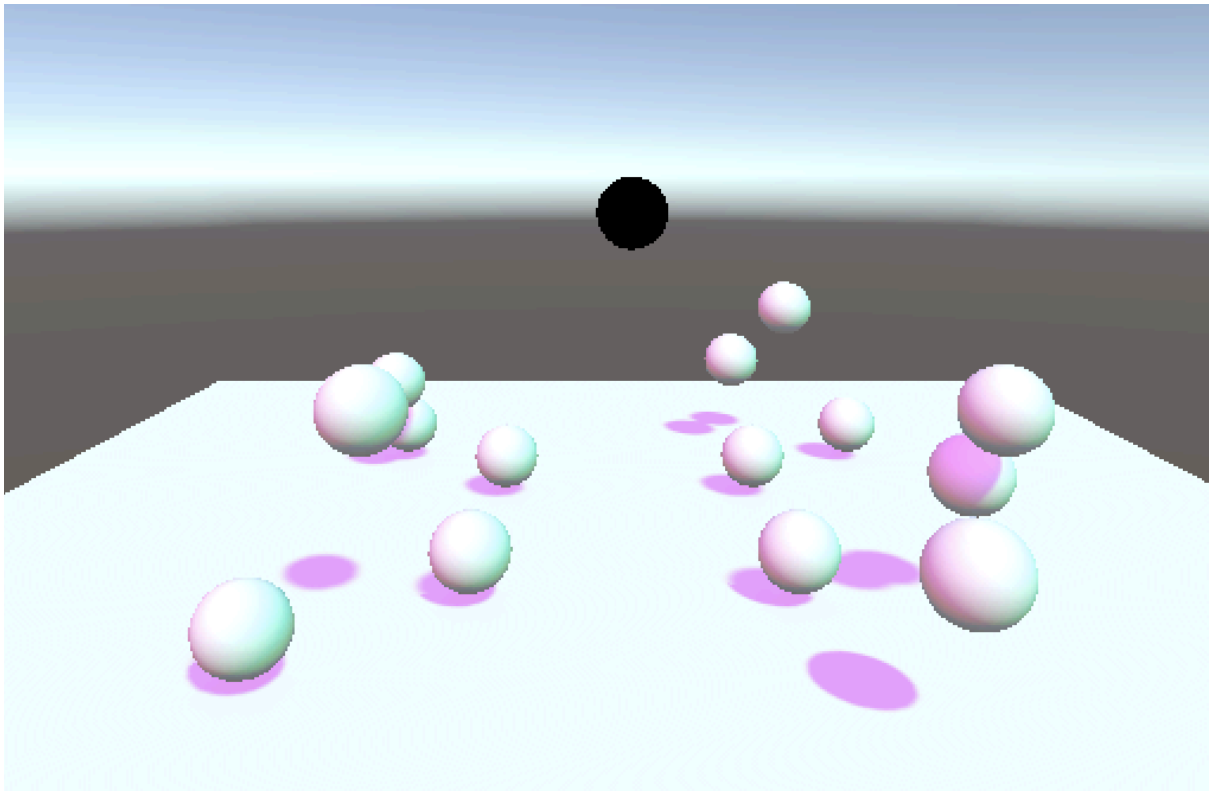


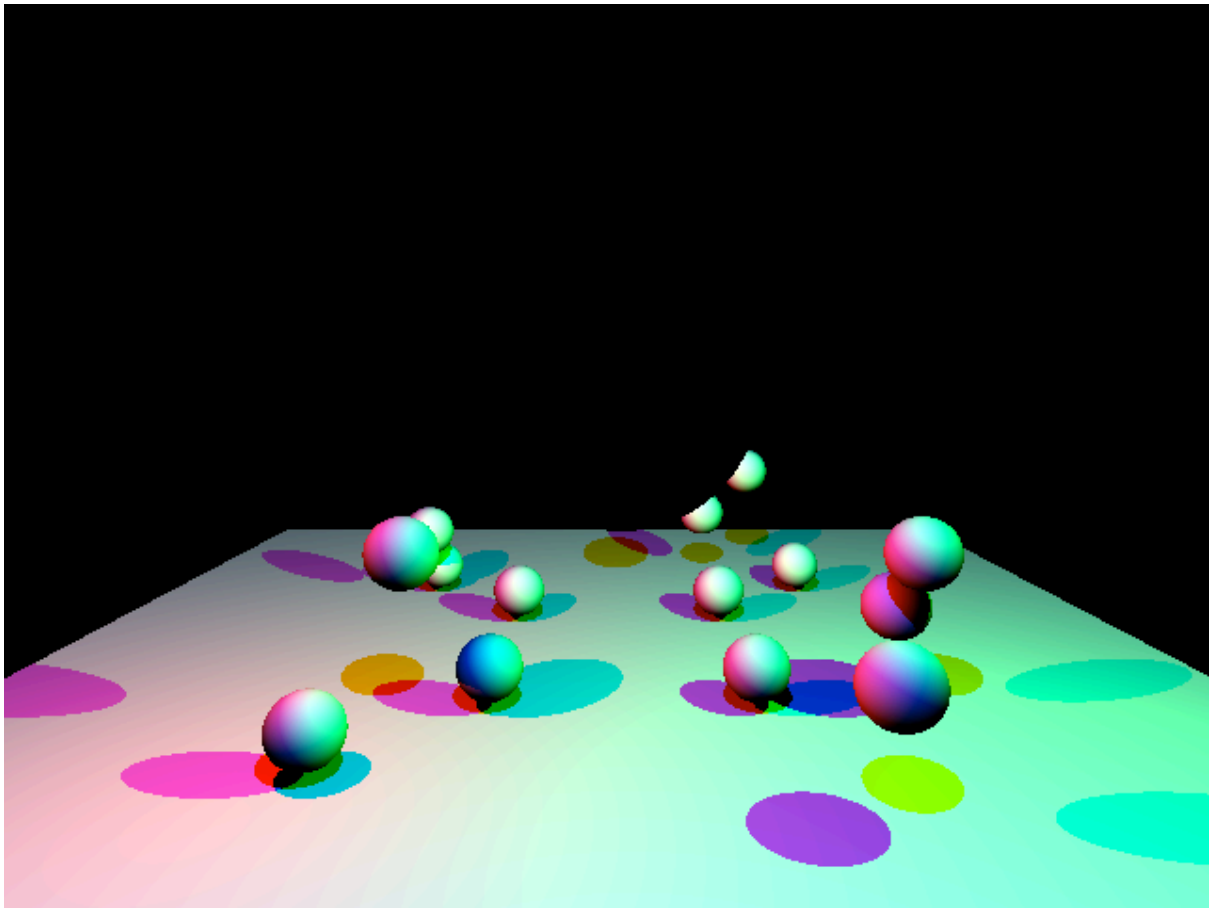
Different place of black hole:



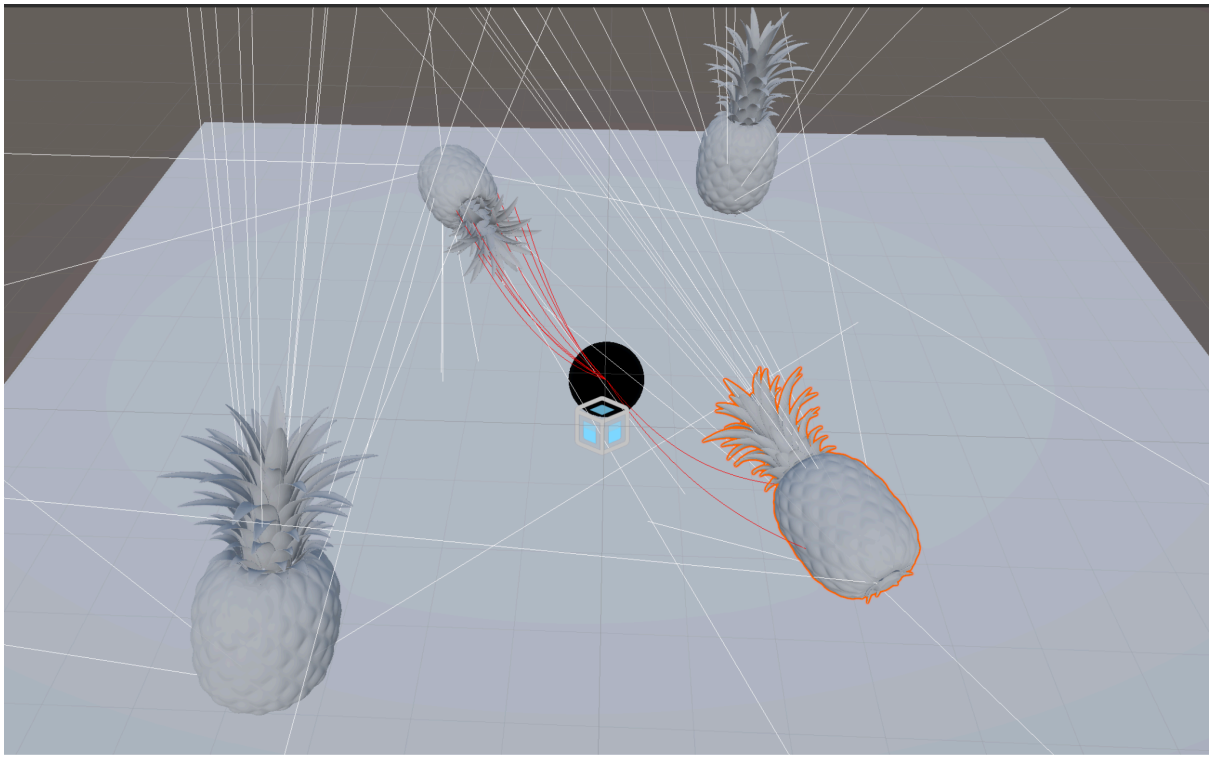


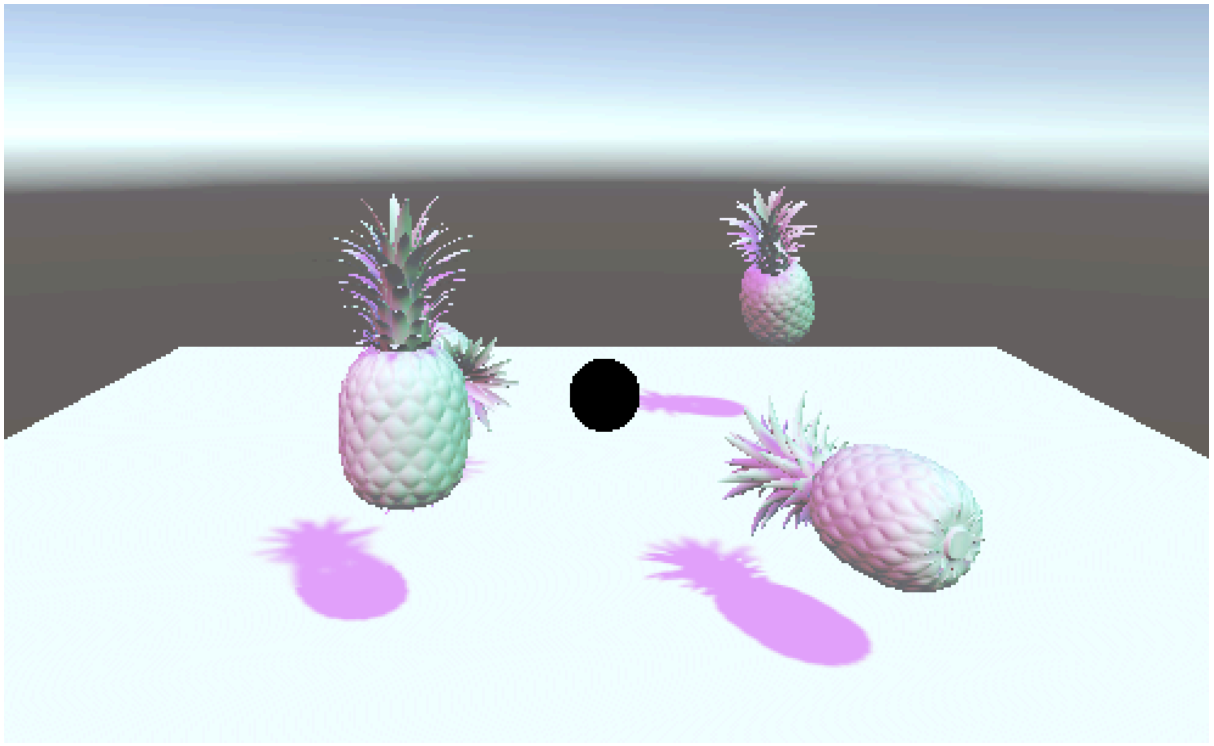
Changing the blackhole place and the effect of the saved image. You can see a little cropped part for the spheres up there.



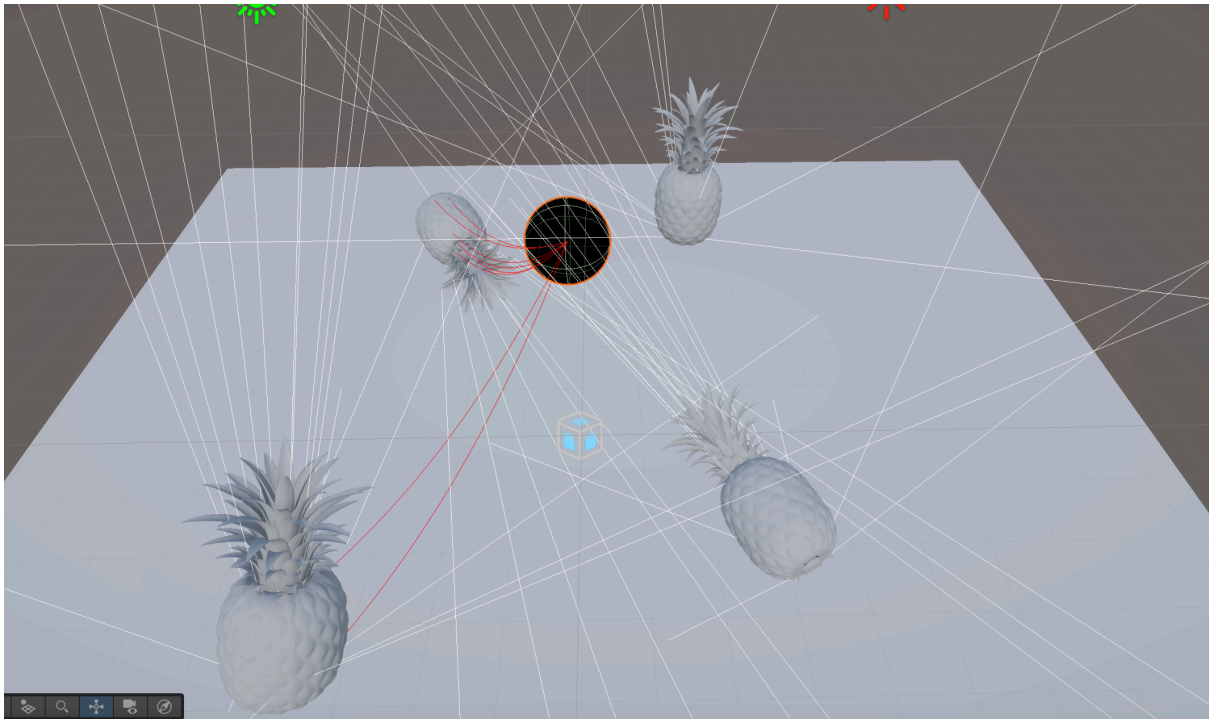


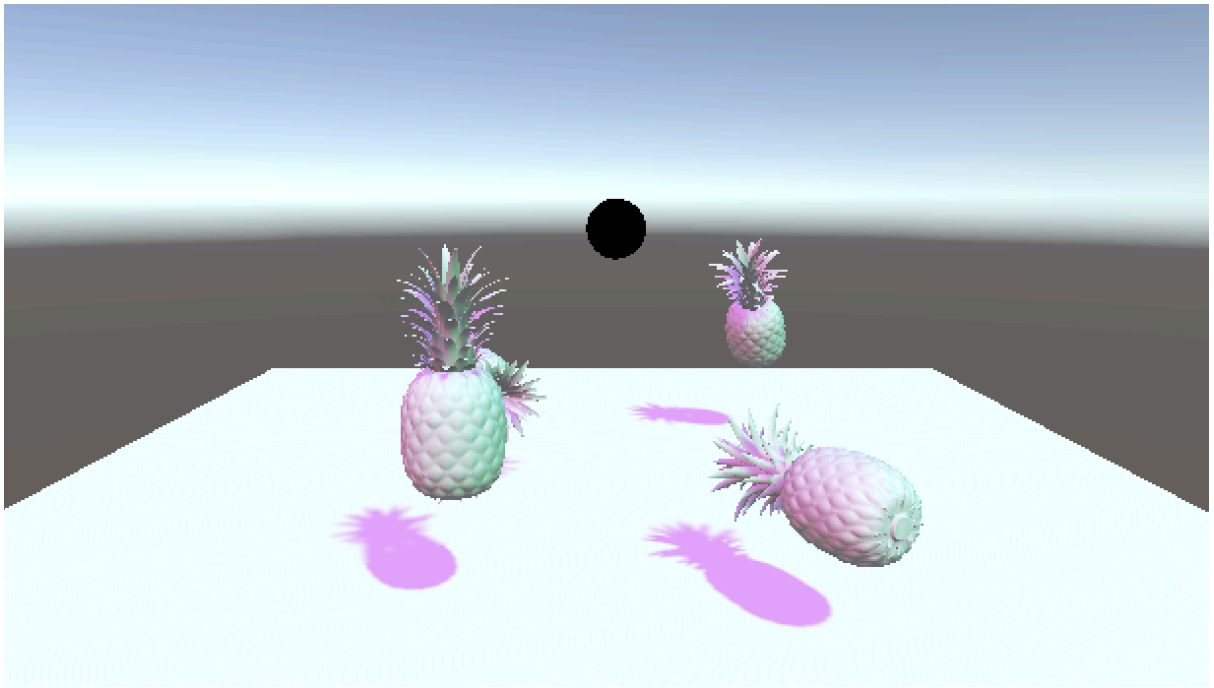
With different objects:





Different place of black hole:





Testing with more rays: