# Projekt 13

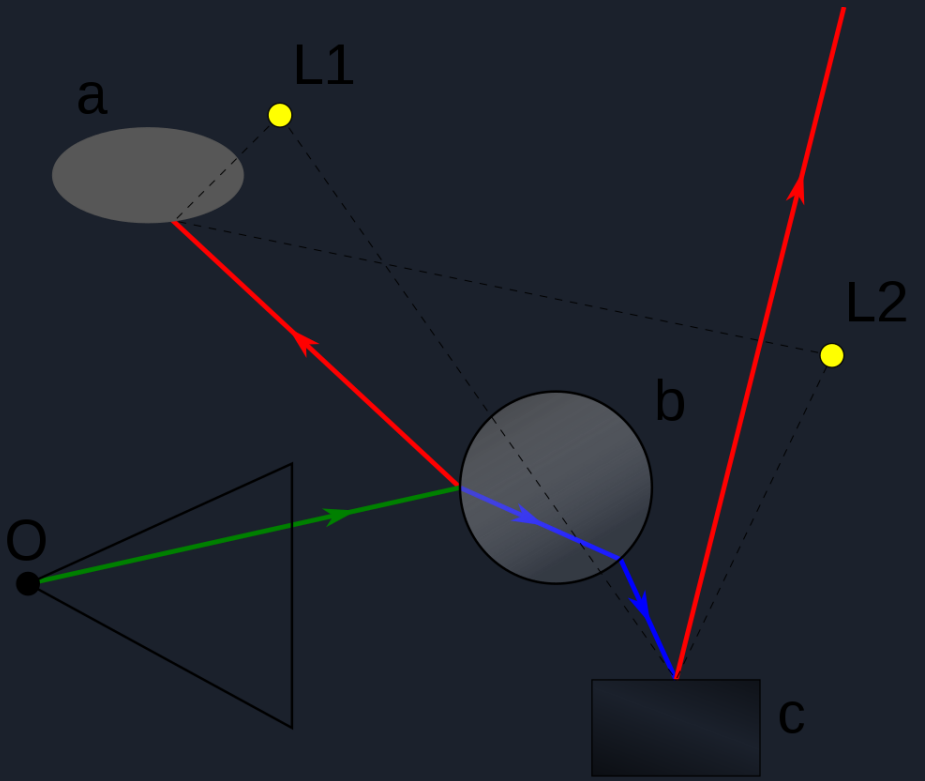
## Zastosowanie symulacji równoległych do tworzenia obrazu trójwymiarowego

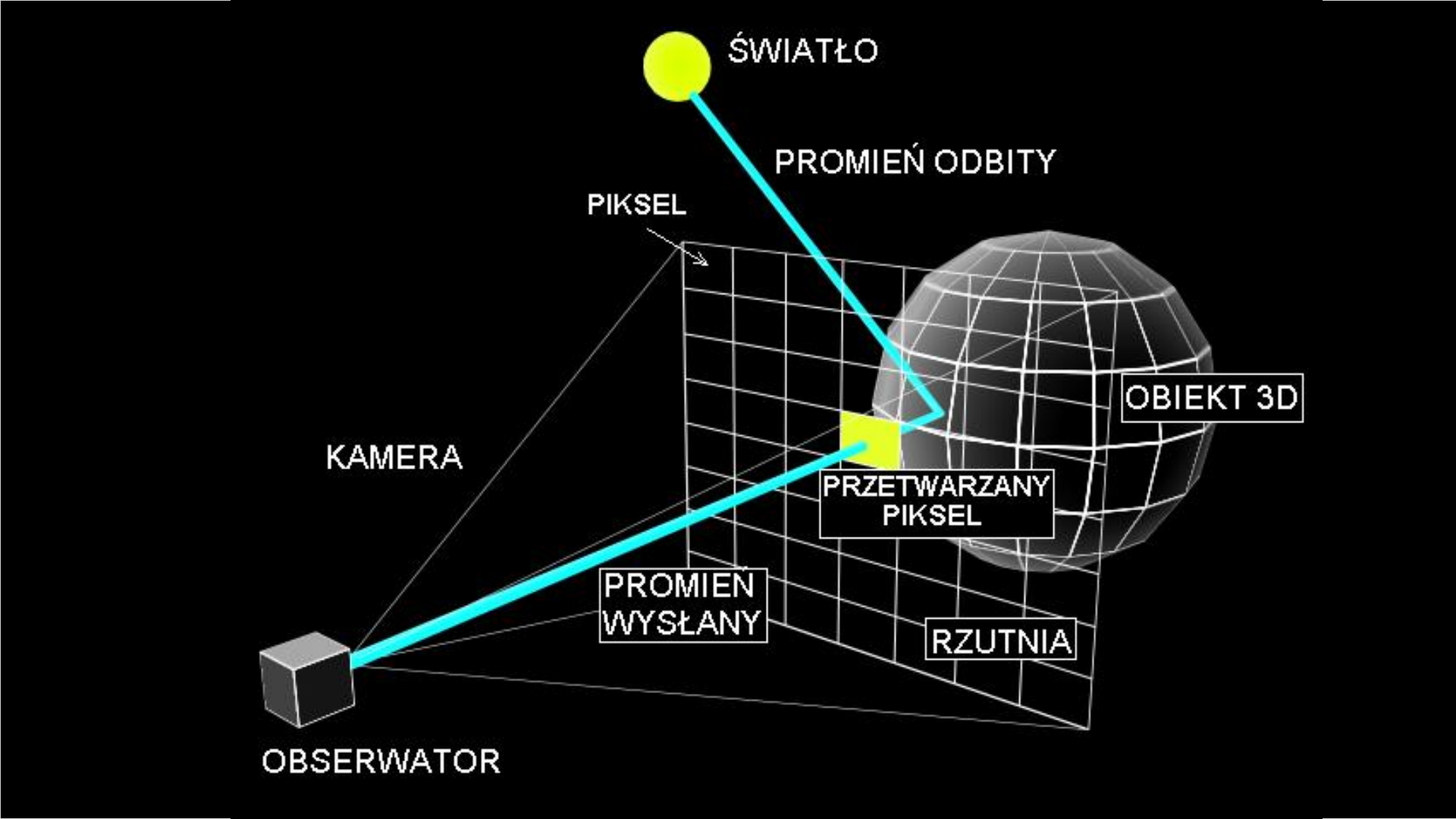
### Ray tracing

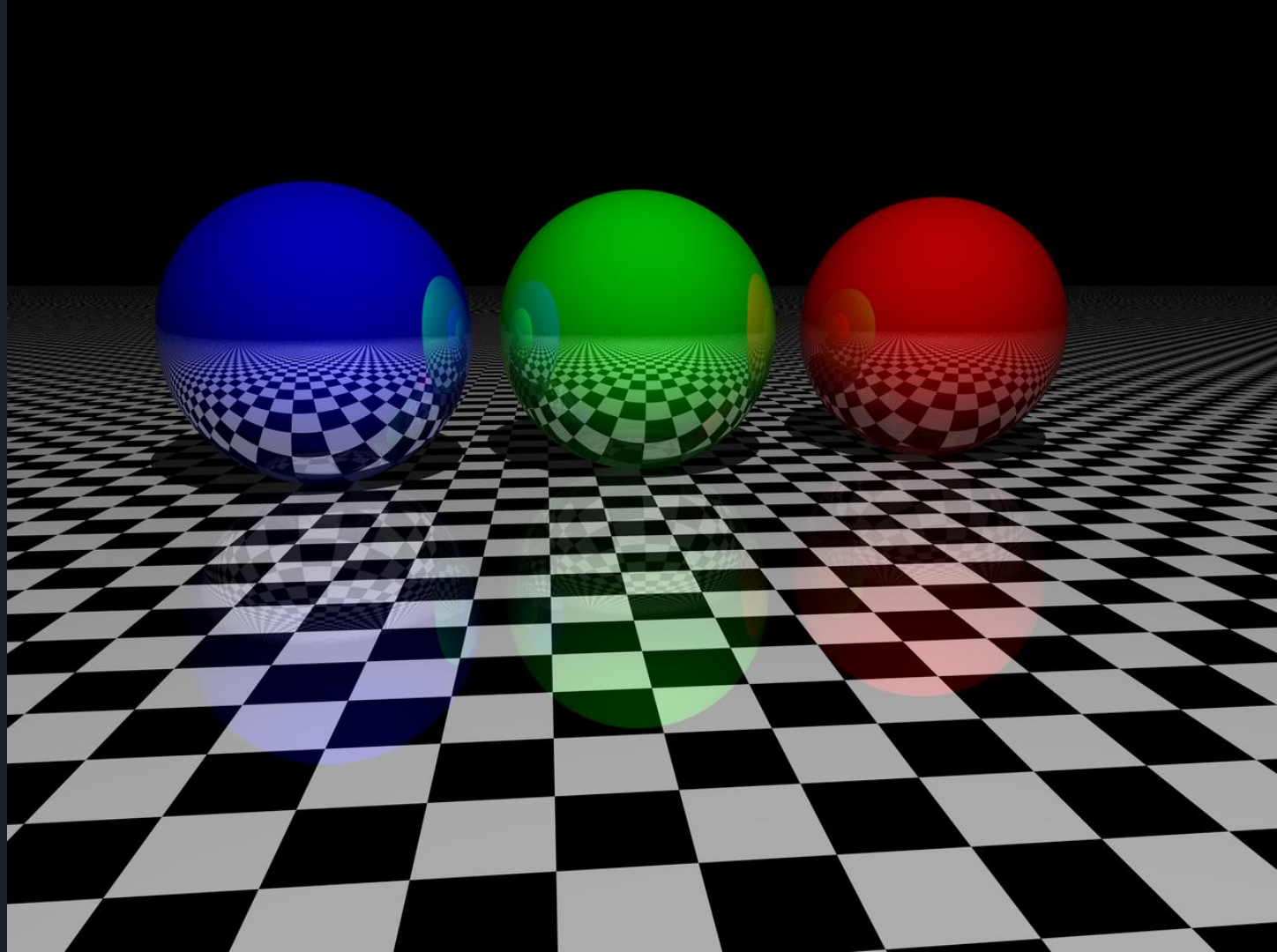
Kamil Mikołajczuk

# Ray tracing

**Śledzenie promieni (ang. ray tracing)** – technika generowania fotorealistycznych obrazów scen trójwymiarowych opierająca się na analizowaniu tylko tych promieni światła, które trafiają bezpośrednio do obserwatora.

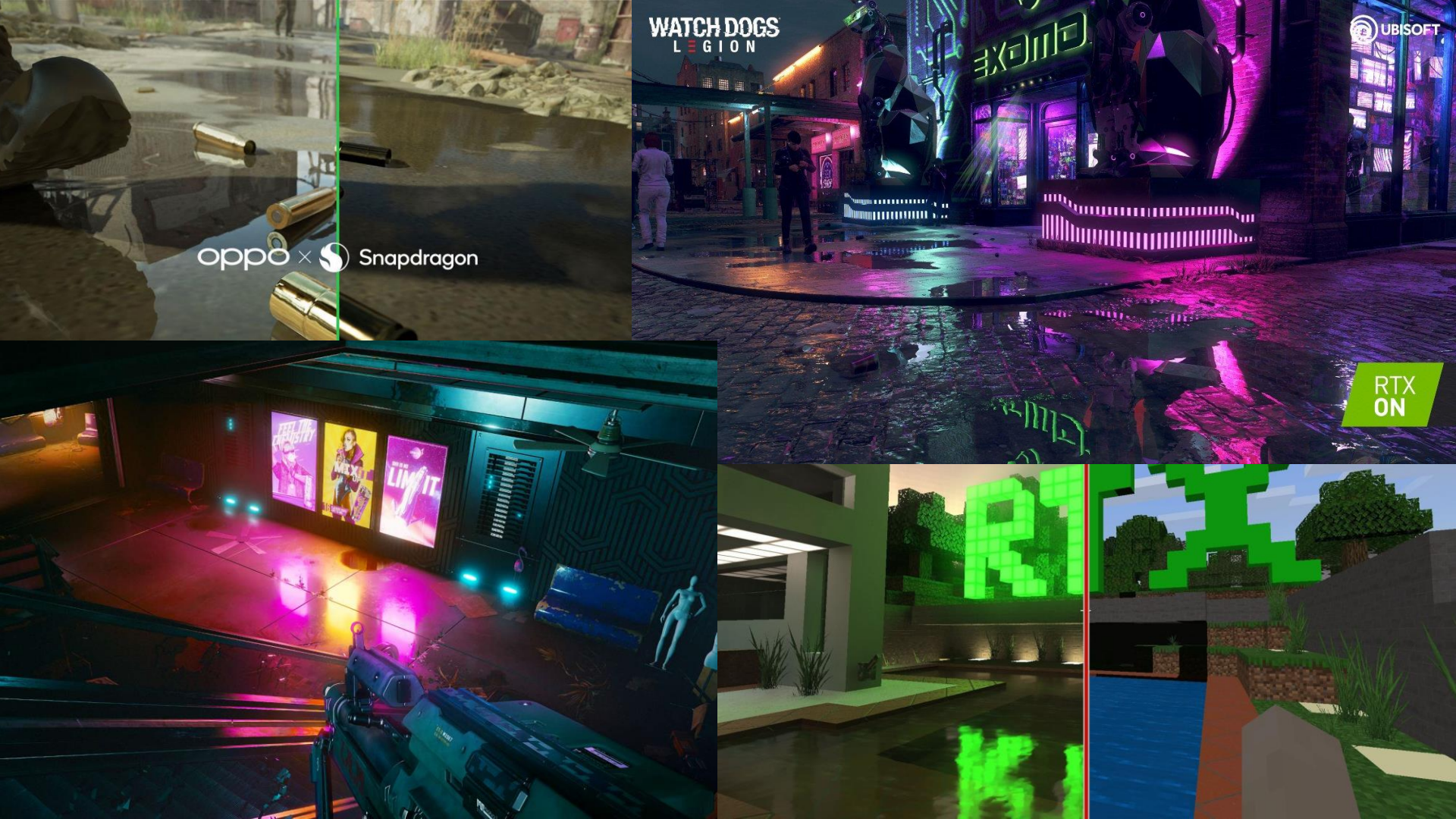



ŚWIATŁO
PROMIEŃ ODBITY
PIKSEL
OBIEKT 3D
KAMERA
PRZETWARZANY PIKSEL
PROMIEŃ WYSŁANY
RZUTNIA
OBSERWATOR



Śledzenie promieni – Wikipedia

Ray tracing (graphics) - Wikipedia


oppo × Snapdragon
WATCH DOGS LEGION
UBISOFT
RTX ON

# Przecięcie promienia z płaszczyzną

P - Punkt należący

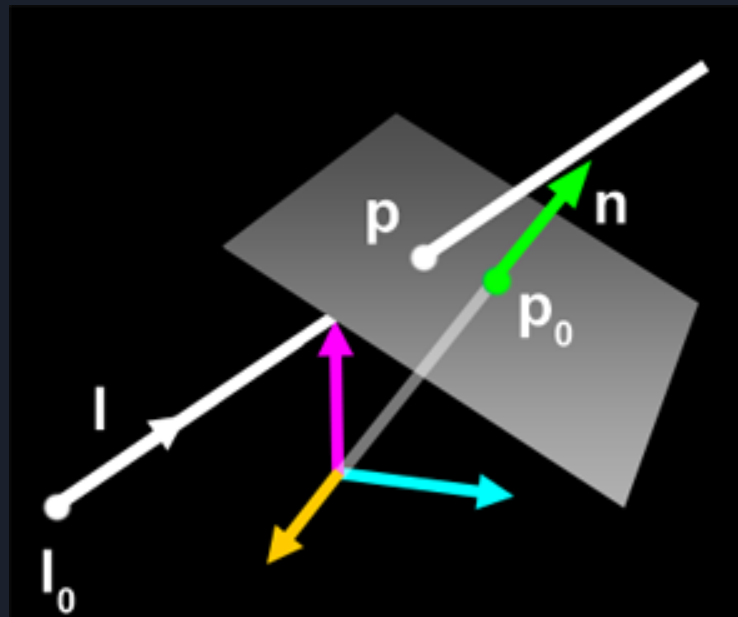
S - Punkt startowy
K - Kierunek
**Promień:** ***P = S + tK***

D - Przesunięcie
**Płaszczyzna:** ***P * N + D = 0***

Po podstawieniu promienia w równanie płaszczyzny mamy:

***(S + tK) * N + D = 0***

***t = -(S * N + D) / (K * N)***



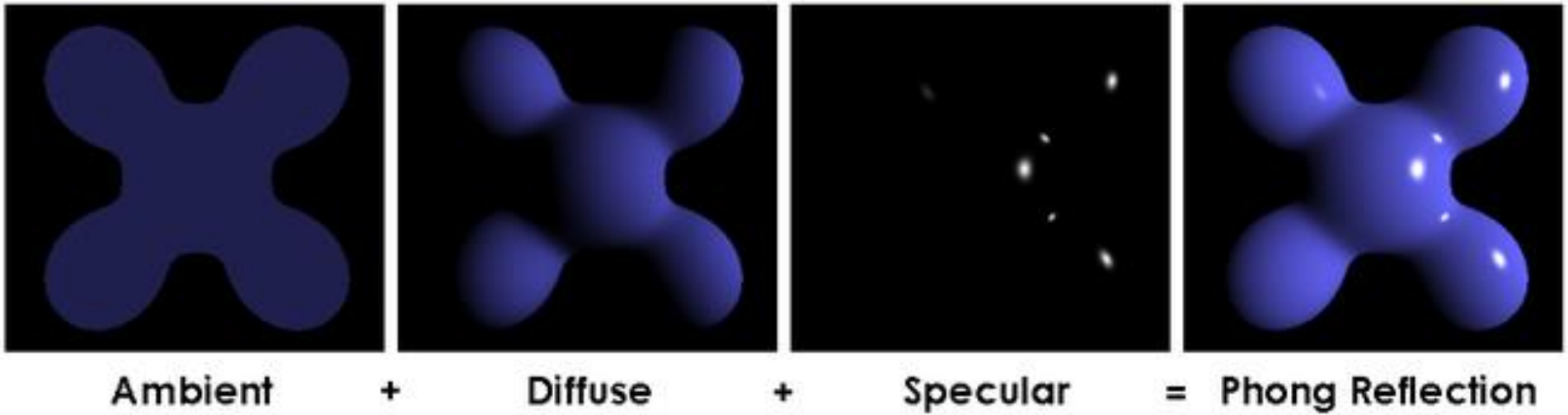
```
public Plane(Vector norm, double offset, Surface surface)
{
    this.norm = norm;
    this.offset = offset;
    this.surface = surface;
}

Odwołania: 2
public Intersection intersect(Ray ray)
{
    var normRayDirectionDot = norm.dot(ray.direction);

    if (normRayDirectionDot > double.Epsilon) return null;

    var dist = -(norm.dot(ray.origin) + offset) / normRayDirectionDot;

    return new Intersection(this, ray, norm, dist);
}
```

# Model oświetlenia

$$I_i(p) = \varepsilon \cdot \frac{I_i^{Max}}{1 + \alpha \left\| p - p_i \right\|^2} \cdot \cos \sphericalangle \left( \vec{n}_p, \vec{l}_p \right)$$

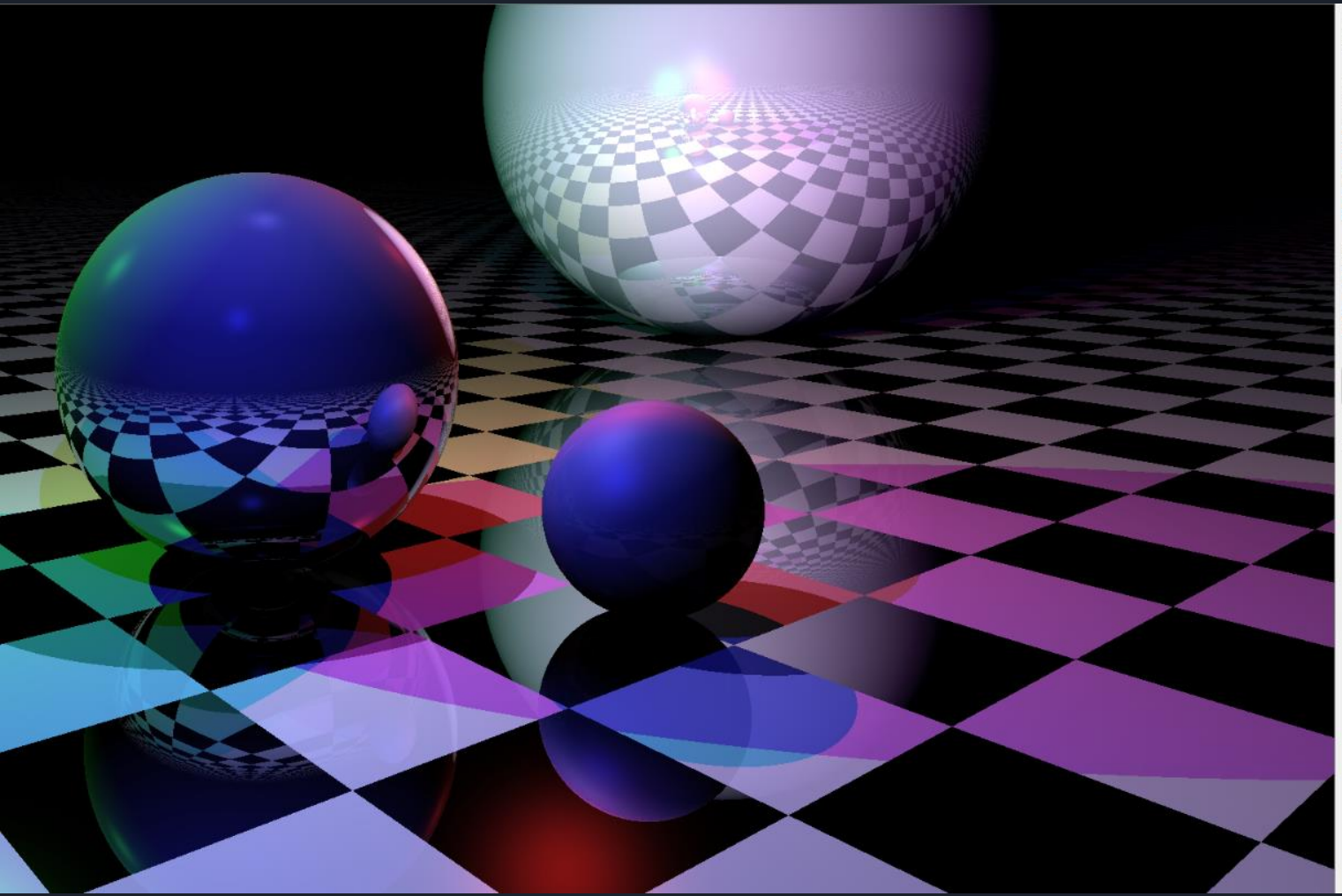
$$I_{tot}(p) = I_{amb} + \sum_i I_{i(L)}(p) + \sum_i I_{i(Ph)}(p)$$

$$I_{i(Ph)}(p) = \varepsilon \cdot \frac{I_i^{Max}}{1 + \alpha \left\| p - p_i \right\|^2} \cdot f\left( \sphericalangle \left( \vec{n}_p, \vec{l}_p \right) \right) \cdot \cos^m \sphericalangle \left( \vec{r}_p, \vec{v}_p \right)$$



# Szachownica

```
public class Checkerboard : Surface
{
    Odwołania: 2
    public double roughness => 150.0;

    Odwołania: 2
    public RColor diffuse(Vector pos)
    {
        if (isBlackField(pos)) return RColor.white;

        return RColor.black;
    }

    Odwołania: 2
    public double reflect(Vector pos)
    {
        if (isBlackField(pos)) return 0.1;

        return 0.7;
    }

    Odwołania: 2
    public RColor specular(Vector pos)...

    Odwołania: 2
    private bool isBlackField(Vector pos)
    {
        return (int) (Math.Floor(pos.z) + Math.Floor(pos.x)) % 2 != 0;
    }
```



Ilość wątków

4

Zoom

1,5

**Zawartość**

- [x] Wielka kula
- [x] Mała kula
- [x] Duża kula
- [x] Podłoga szachowa

**Światła**

- [x] Czerwone
- [x] Niebieskie
- [x] Szare
- [x] Zielone

Start

**Czas trwania animacji (s):** 1,973

# Fragmenty kodu

```
        var scene = getScene();
        var threads = getThreads();
        var width = renderedImage.Width;
        var height = renderedImage.Height;

        Task.Run(() =>
        {
            var rayTracer = new RayTracer(scene);
            var image = new Bitmap(width, height);

            renderedImage.Image = image;

            Task.WhenAll(
                range(threads).Select((_, fragment) =>
                Task.Run(() =>
                {
                    var renderedFragment = rayTracer.fragmentRender(width, height, fragment, threads);

                    Invoke(new Action(delegate ()
                    {
                        showFragment(image, renderedFragment, fragment, threads);
                        renderedImage.Refresh();
                    }));
                })
            )).Wait();

            stopwatch.Stop();

            Invoke(new Action(delegate ()
            {
                timeLabel.Text = (stopwatch.ElapsedMilliseconds / 1000.0).ToString();
                startButton.Enabled = true;
            }));
        });
    }
```

```
1 odwołanie
private Scene getScene()
{
    Thing[] things = new List<Thing> {
        groundControl.Checked    ? new Plane (new Vector( 0.0, 1.0,  0.00), 0.0, new Checkerboard()) : null,
        bigBallControl.Checked   ? new Sphere(new Vector( 0.0, 1.0, -0.25), 1.0, new Shiny())        : null,
        smallBallControl.Checked ? new Sphere(new Vector(-1.0, 0.5,  1.50), 0.5, new Matt())         : null,
        hugeBallControl.Checked  ? new Sphere(new Vector(-9.0, 3.0,  -4.5), 3.0, new Shiny())        : null,
    }.Where(x => x != null).ToArray();

    var red = RColor.from(125, 18, 18);
    var green = RColor.from(18, 125, 18);
    var blue = RColor.from(18, 18, 125);
    var gray = RColor.from(54, 54, 89);

    Light[] lights = new List<Light> {
        redLightControl.Checked   ? new Light(new Vector(-2.0, 2.5,  0.0), red)   : null,
        blueLightControl.Checked  ? new Light(new Vector( 1.5, 2.5,  1.5), blue)  : null,
        greenLightControl.Checked ? new Light(new Vector( 1.5, 2.5, -1.5), green) : null,
        grayLightControl.Checked  ? new Light(new Vector( 0.0, 3.5,  0.0), gray)  : null
    }.Where(x => x != null).ToArray();

    var zoomSelected = (string)zoomControll.SelectedItem;
    var zoom = zoomSelected != null && zoomSelected.Length > 0 ? double.Parse(zoomSelected) : 1.0;
    var camera = new Camera(new Vector(3.0, 2.0, 5.0), new Vector(-1.0, 0.5, 0.0), zoom);

    return new Scene(things, lights, camera);
}
```

```
1 odwołanie
public Bitmap fragmentRender(int screenWidth, int screenHeight, int fragmentIndex, int fragments)
{
    var image = new Bitmap(screenWidth, fragmentIndex == fragments - 1 ? screenHeight - fragmentIndex * (screenHeight / fragments) : screenHeigh

    var camera = scene.camera;

    var up = camera.up.times(camera.zoom);
    var right = camera.right.times(camera.zoom);

    var minAxis = Math.Min(screenWidth, screenHeight);

    var start = fragmentIndex * (screenHeight / fragments);

    for (var y = 0; y < image.Height; y++)
    {
        double recenterY = ((start + y) - (minAxis / 2.0)) / 2.0 / minAxis;
        var pointY = camera.forward.plus(up.times(-recenterY));

        for (var x = 0; x < screenWidth; x++)
        {
            double recenterX = (x - (minAxis / 2.0)) / 2.0 / minAxis;

            var point = pointY.plus(right.times(recenterX)).norm();
            var ray = new Ray(camera.position, point);
            var color = traceRay(ray, scene, 0);

            image.SetPixel(x, y, color.toDrawingColor());
        }
    }

    return image;
```

```
private RColor traceRay(Ray ray, Scene scene, int depth)
{
    var isect = closestIntersection(ray, scene);

    if (isect == null) return background;

    var direction = isect.ray.direction;
    var position = direction.times(isect.dist).plus(isect.ray.origin);
    var reflectDir = direction.minus(
        isect.norm.times(2 * isect.norm.dot(direction))
     );

    var naturalColor = background.plus(
        getNaturalColor(isect.thing, position, isect.norm, reflectDir, scene)
     );

    var reflectedColor = (depth >= maxDepth) ?
        RColor.grey :
        getReflectionColor(isect.thing, position, reflectDir, scene, depth);

    return naturalColor.plus(reflectedColor);
}
```

```
private RColor getNaturalColor(Thing thing, Vector position, Vector norm, Vector reflectDir, Scene scene)
{
    var color = defaultColor;
    var thingDiffuse = thing.surface.diffuse(position);
    var thingSpecular = thing.surface.specular(position);

    foreach (var light in scene.lights)
    {
        var toLightDirection = light.position.minus(position);
        var toLightDirectionNorm = toLightDirection.norm();
        var closest = closestIntersection(new Ray(position, toLightDirectionNorm), scene);

        var isInShadow = closest == null ? false : (closest.dist <= toLightDirection.mag());

        if (!isInShadow)
        {
            var illum = toLightDirectionNorm.dot(norm);
            var lColor = illum > 0 ? light.color.scale(illum) : defaultColor;

            var specular = toLightDirectionNorm.dot(reflectDir.norm());
            var sColor = specular > 0 ? light.color.scale(Math.Pow(specular, thing.surface.roughness)) : defaultColor;

            color = color.plus(lColor.times(thingDiffuse).plus(sColor.times(thingSpecular)));
        }
    }

    return color;
}
```

```
Odwołania: 2
private Intersection closestIntersection(Ray ray, Scene scene)
{
    var closest = double.PositiveInfinity;
    Intersection closestInter = null;

    foreach (var thing in scene.things)
    {
        var inter = thing.intersect(ray);

        if (inter != null && inter.dist < closest)
        {
            closestInter = inter;
            closest = inter.dist;
        }
    }

    return closestInter;
}

Odwołania: 2
private RColor traceRay(Ray ray, Scene scene, int depth)...

1 odwołanie
private RColor getReflectionColor(Thing thing, Vector position, Vector reflectDir, Scene scene, int depth)
{
    return traceRay(new Ray(position, reflectDir), scene, depth + 1).scale(thing.surface.reflect(position));
}
```