Raytracing: krok po kroku

cz. 4 - światło

I. Co/Dlaczego?

Tym razem postaramy się dodać trochę światła do naszego świata. W tej części wprowadzimy trochę teorii oraz dodamy obsługę świateł punktowych i prostych, matowych materiałów.

II. Lepsze kolory

Zanim zaczniemy się zajmować trudniejszymi tematami, mamy do wykonania prostą robotę. Mianowicie, wbudowana w .NET framework klasa System.Drawing.Color okazuje się coraz mniej pasować do naszych potrzeb. Co konkretnie mamy jej do zarzucenia?

- Niewygodna - nie można w prosty sposób dodać lub pomnożyć dwóch kolorów.
- Mały zakres - nie można reprezentować kolorów 'poza skalą'. Obecnie nie ma to znaczenia bo kolor i tak będziemy obcinać, ale w przyszłości może się okazać problemem.
- Nadmiarowość - klasa Color przechowuje kanał alpha którego my nie potrzebujemy (w raytracingu kolor nie może być przeźroczysty, jest to własność materiału)
- Niedokładność - najważniejsze - mamy tylko 256 możliwych wartości dla każdego kanału! Nawet kilka prostych transformacji przy oświetleniu może spowodować że ilość kolorów w wyrenderowanym obrazie spadnie do kilkunastu.

Żeby rozwiązać ten problem napiszemy własny kolor. Kod nie potrzebuje chyba dodatkowego wyjaśnienia, prosta struktura z podstawowymi operacjami:

```
struct ColorRgb
{
    public double R { get; set; }
    public double G { get; set; }
    public double B { get; set; }

    public ColorRgb(double r, double g, double b)
        : this()
    {
        this.R = r;
        this.G = g;
        this.B = b;
    }

    public static implicit operator ColorRgb(Color color)
    { return new ColorRgb(color.R / 255.0, color.G / 255.0, color.B / 255.0); }

    public static ColorRgb operator +(ColorRgb col1, ColorRgb col2)
    { return new ColorRgb(col1.R + col2.R, col1.G + col2.G, col1.B + col2.B); }

    public static ColorRgb operator *(ColorRgb col1, double val)
    { return new ColorRgb(col1.R * val, col1.G * val, col1.B * val); }

    public static ColorRgb operator *(ColorRgb col1, ColorRgb col2)
    { return new ColorRgb(col1.R * col2.R, col1.G * col2.G, col1.B * col2.B); }

    public static ColorRgb operator /(ColorRgb col1, double val)
    { return col1 * (1 / val); }

    public static readonly ColorRgb White = new ColorRgb(1, 1, 1);
```

```
    public static readonly ColorRgb Black = new ColorRgb(0, 0, 0);
}
```

III. Światło punktowe

Nie stworzymy żadnej klasy abstrakcyjnej/interfejsu dla światła, ponieważ obecnie jedynym przez nas używanym rodzajem światła będzie światło punktowe.
W klasie reprezentującej światło nie umieścimy żadnych metod, zostawiając przetwarzanie informacji innym obiektom - wystarczą nam informację o kolorze światła i jego pozycji.

```
class PointLight
{
    public PointLight(Vector3 position, ColorRgb color)
    {
        this.Position = position;
        this.Color = color;
    }

    public Vector3 Position { get; private set; }
    public ColorRgb Color { get; private set; }
}
```

Nic więcej w tym miejscu, idziemy dalej.

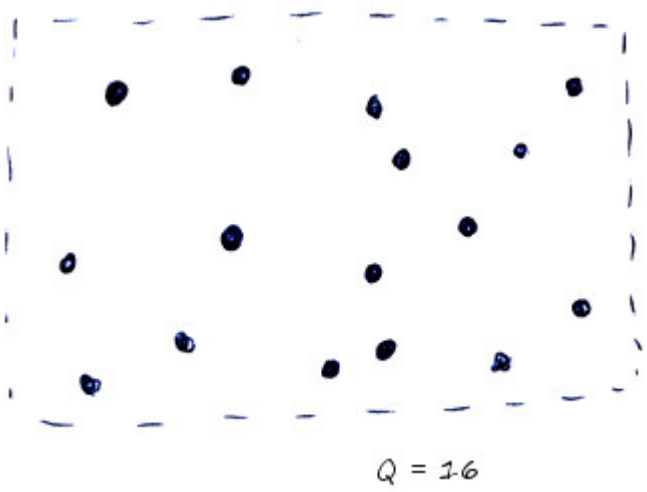
IV. Teoria

Ten podrozdział będzie się składał z definicji kilku pojęć których będziemy używali przy opisywaniu materiałów. Nie jest, zgodnie z założeniami, niezbędny do zrozumienia kodu i dalszych opisów, ale pojęcia radiancji i irradiancji będą sie często pojawiać przy omawianiu własności materiałów - oraz po prostu pomagają zrozumieć na głębszym poziomie proces raytracingu. Ogólnie jeśli ktoś jest bardzo oprzedzony do fizyki, można pominąć. Są obrazki.

Energia (Q)- podstawowa jedostka elektromagnetycznej energii. Jest mierzona w dżulach (J). Każdy foton przenosi pewną ilość energii równą

$$Q = \frac{hc}{\lambda}$$

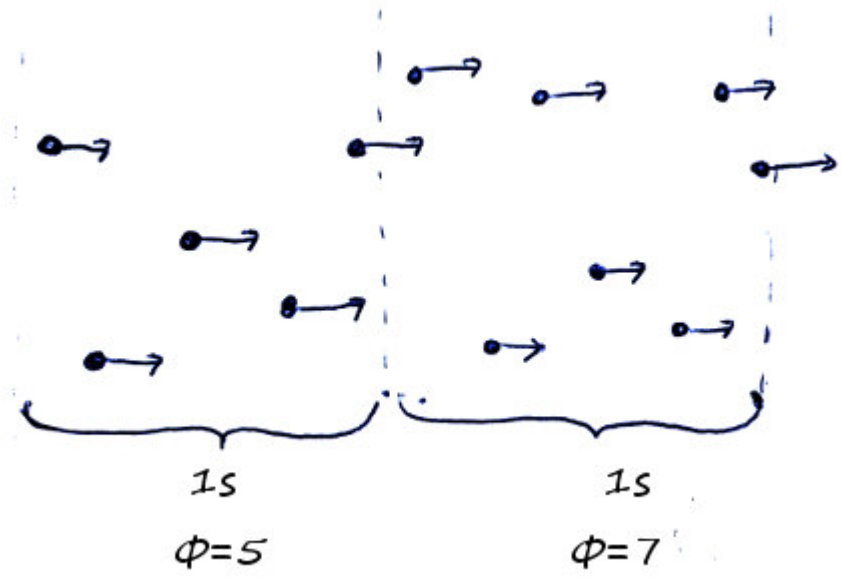
Gdzie h to stała Plancka równa 6.62*10e-34, c to prędkość światła a λ to długość fali.



Każdy foton niesie pewną ilość energii.

Strumień promieniowania (Φ) - najprościej ją zrozumieć jako *`energię na sekundę`*. Określa ilość energii która przechodzi przez region/powierzchnię w czasie sekundy. Jednostka to dżul na sekundę czyli wat. Jest określony wzorem

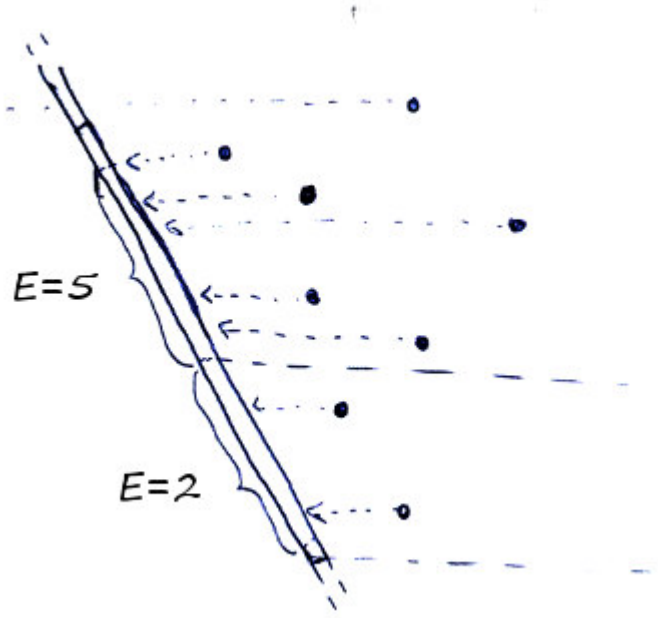
$$\Phi = \frac{Q}{t}$$



Strumień fotonów (nie-do-końca poprawnie w tym artykule nazywany światłem) `lecący` w jednym kierunku. Skala fotonów niezbyt dokładnie zachowana - odgrodzone pionowymi przerywane liniami obszary mają długość równą 299 792 458 = prędkość światła metrów.

Irradiancja (E) - jest określana jako strumień promieniowania *padający* na jednostkę powierzchni. Wzór irradiancji to

$$E = \frac{d\Phi}{dA}$$



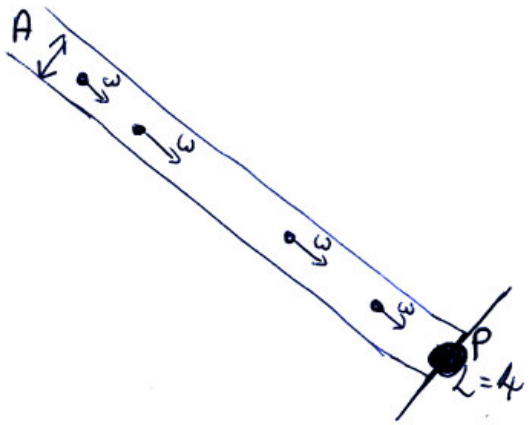
Irradiancja, czyli ilość fotonów padających na jednostkę powierzchni w czasie sekundy.

Radiancja (L) - zdefiniowany jako strumień promieniowania na jednostkę kąta bryłowego. Opisuje ona strumień promieniowania *w dowolnym punkcie w przestrzeni* p , nadchodzący z konkretnego kierunku ω i mierzony na prostopadłej do kierunku powierzchni $A^{\perp}$.

Radiancja może być określona dla dowolnego punktu w przestrzeni, niekoniecznie na jakiejś powierzchni. Na przykład dla centrum kamery.
Radiancja nie zależy od tego czy strumień promieniowania jest odbity, wyemitowany, rozproszony...

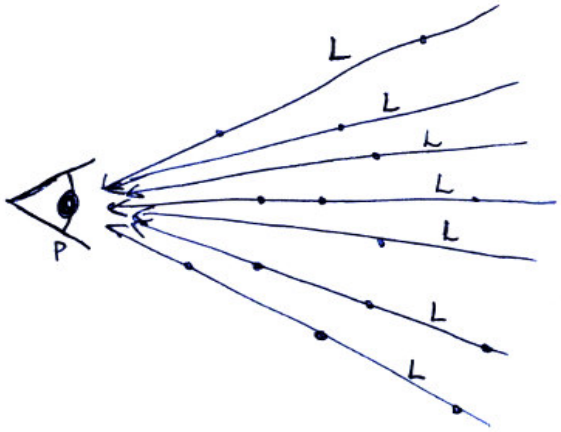
Jest zapisywana za pomocą wzoru:

$$L = \frac{d^2\Phi}{dA^{\perp} * d\omega}$$



Radiancja, czyli ilość fotonów nadchodząca z konkretnego kierunku do konkretnego punktu w przestrzeni.

Radiancja jest własnością która nas najbardziej interesuje - proces renderowania sceny można zdefiniować jako mierzenie radiancji w centrum kamery i kierunku wyznaczanym przez piksel:



Ciekawostka - wszystkie wartości radiometryczne zależą od energii, a energia zależy od długości fali - gdzie jest długość fali w naszej aplikacji? Okazuje się że kryje się w strukturze ColorRgb - Red, Green i Blue to po prostu trzy różne długości fali - mimo że ich długość w nanometrach nie jest nigdzie podana.

V. Irradiancja i kąty

Irradiancja zależy od kąta padania światła. Stosunek powierzchni projekcji do powierzchni płaszczyzny w danym punkcie wynosi

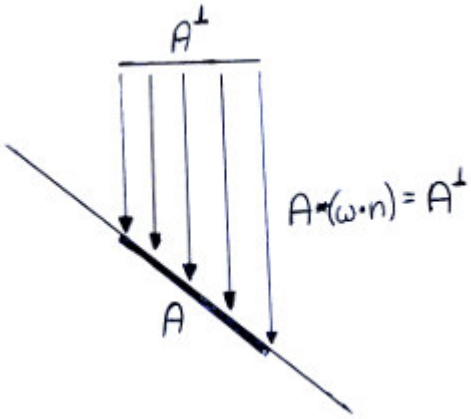
$$dA^{\perp} = \cos\theta * dA$$

Podstawiając to do wzoru na irradiancję, otrzymujemy tak zwane Prawo Lamberta - irradiancja jest proporcjonalna do cosinusa kąta padającego światła.

Możemy to prawo zapisać również za pomocą dot productu:

$$\cos\theta = \omega \cdot n$$

Gdzie w i n są znormalizowanymi wektorami oznaczającymi odpowiednio kąt wpadającego światła i normalną powierzchni.



VI. Materiały

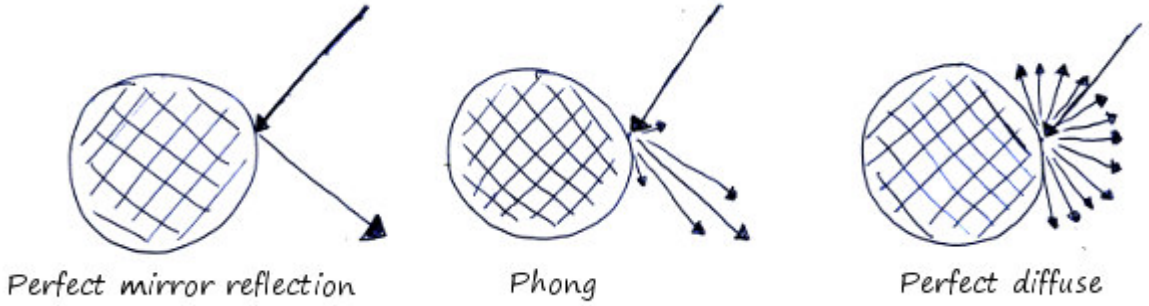
Materiał opisuje to w jaki sposób światło jest odbijane i pochłaniane przy trafieniu na obiekt (nie będziemy tutaj zajmować się pojęciem BRDF - jest to precyzyjny sposób przedstawiania w jaki sposób światło odbija się od powierzchni. My połączymy ten termin z materiałem).

Zadaniem materiału jest, znając źródło światła i punkt w przestrzeni, obliczyć radiancję - ilość światła odbijanego przez ten materiał w danym punkcie w kierunku oka (ściśle - w kierunku z którego nadszedł promień który śledzimy) - resztę światła ignorujemy ponieważ nie ma ona bezpośrednio wpływu na to co widzi kamera.

Zapiszemy to za pomocą takiego interfejsu:

```
interface IMaterial
{
    ColorRgb Radiance(PointLight light, HitInfo hit);
}
```

Rysunek przedstawiający kilka typowych rodzajów materiałów (jeszcze go kilka razy zobaczymy):



W tej części naszym celem jest ostatni z nich - Perfect Diffuse.

VII. Rozpraszanie

Pierwszym i jedynym na razie rodzajem materiału którym się zajmiemy będzie materiał doskonale rozpraszający. Jest to materiał który w każdym punkcie odbija światło równomiernie we wszystkich kierunkach, być może pochłaniając pewne barwy.

Łatwo się domyślić że jest to najprostszy w implementacji rodzaj światła. Stwórzmy więc szkielet klasy.

```
class PerfectDiffuse : IMaterial
{
    ColorRgb materialColor;

    public PerfectDiffuse(ColorRgb materialColor)
    {
        this.materialColor = materialColor;
    }

    public ColorRgb Radiance(PointLight light, HitInfo hit)
    {
        throw new System.NotImplementedException();
    }
}
```

Mimo że żadne prawdziwe materiały nie działają się dokładnie w ten sposób, perfect diffuse jest dobrym przybliżeniem zachowania nieobrobionego drewna, papieru, matowych farb itd.
Ilosć światła odbitego jest opisany przez wzór

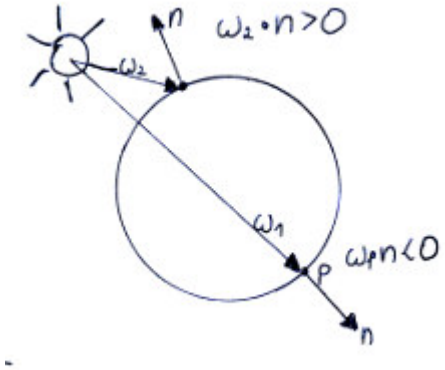
$$L = L_i * k_d * c_d$$

Gdzie kd to współczynnik rozpraszania (diffuse coefficient), cd to kolor rozpraszania (diffuse color) a Li to irradiancja (dla nieczytających - ilosć światła padającego na materiał w danym punkcie).
Cd jest stały dla materiału (to po prostu kolor materiału) a współczynnik kd <0, 1> opisuje jak wiele światła zostaje pochłonięte.

Co jednak z irradiancją? Patrz punkt VI - znając kąt padania światła i jego siłę możemy ją łatwo wyliczyć (gdzie n to normalna powierzchni, w to kąt padania a ks to jasność światła)

$$L_i = (n \cdot \omega) * k_s$$

Musimy wziąść pod uwagę jeszcze jedną rzecz - światło może `padać` na punkt *`od spodu`* (na przykład jeśli źródło światła jest za kulą to do frontalnej powierzchni światło będzie docierać *`z tyłu`*) Łatwo to rozpoznamy po tym że dot product wektorów n i w jest mniejszy od zera - w takim wypadku do powierzchni nie dociera żadne faktyczne światło i zwracamy kolor czarny (czyli brak światła).



Możemy teraz zapisać to w postaci kodu:

```
public ColorRgb Radiance(PointLight light, HitInfo hit)
{
    Vector3 inDirection = (light.Position - hit.HitPoint).Normalized;
    double diffuseFactor = inDirection.Dot(hit.Normal);

    if (diffuseFactor < 0) { return ColorRgb.Black; }

    return light.Color * materialColor * diffuseFactor;
}
```

I mamy już gotowy, rozpraszający materiał.
Niestety, nasze HitInfo nie wie na razie nic o HitPoint i Normal - musimy go trochę zmodyfikować...

VIII. Łączenie w całość

Mamy już gotowe wszystkie cegiełki potrzebne do zbudowania oświatlenia, czas zacząć modyfikować resztę kodu.

Przede wszystkim informacje zawarte w HitInfo bardzo różnią się od tego czego potrzebujemy do cieniowania.
Naprawmy to:

```
class HitInfo
{
    /// <summary>Trafiony obiekt lub null jeśli promień w nic nie trafił</summary>
    public GeometricObject HitObject { get; set; }

    /// <summary>Referencja do świata który renderujemy</summary>
    public World World { get; set; }

    /// <summary>Normalna do punktu trafienia</summary>
    public Vector3 Normal { get; set; }

    /// <summary>Punkt trafienia (w koordynatach świata)</summary>
    public Vector3 HitPoint { get; set; }

    /// <summary>Promień który trafił obiekt</summary>
    public Ray Ray { get; set; }
}
```

Oczywiście powoduje to błędy kompilacji w kilkunastu innych miejscach. Naprawianie zaczniemy od klasy World.

Jako że teraz obsługujemy światło a World jest najlepszym miejscem na przechowywanie listy świateł, dodajemy kilka nowych pól. Musimy też zmodyfikować metodę TraceRay żeby uwzględnić wyliczanie normalnych

```
class World
{
    List<GeometricObject> objects;
    List<PointLight> lights;

    public World(Color background)
    {
        this.BackgroundColor = background;
        this.objects = new List<GeometricObject>();
        this.lights = new List<PointLight>();
    }

    public void Add(GeometricObject obj)
    {
        objects.Add(obj);
    }

    public void AddLight(PointLight light)
    {
        lights.Add(light);
    }

    public HitInfo TraceRay(Ray ray)
    {
        HitInfo result = new HitInfo();
        Vector3 normal = default(Vector3);
        double minimalDistance = Ray.Huge; // najbliższe trafienie
        double lastDistance = 0; // zmienna pomocnicza, ostatnia odległość

        foreach (var obj in objects)
        {
            if (obj.HitTest(ray, ref lastDistance, ref normal) &&
                lastDistance < minimalDistance) // jeśli najbliższe trafienie
            {
                minimalDistance = lastDistance; // nowa najmniejsza odległość
                result.HitObject = obj; // nowy trafiony obiekt
                result.Normal = normal; // normalna trafienia
            }
```

```
        }

        if (result.HitObject != null) // jeśli trafiliśmy cokolwiek
        {
            result.HitPoint = ray.Origin + ray.Direction * minimalDistance;
            result.Ray = ray;
            result.World = this;
        }

        return result;
    }

    public ColorRgb BackgroundColor { get; private set; }
    public List<GeometricObject> Objects { get { return objects; } }
    public List<PointLight> Lights { get { return lights; } }
}
```

Co zmieniło się w metodzie TraceRay? Po pętli foreach dodatkowo wypełniamy kilka pól klasy HitInfo - punkt trafienia, promień i referencję do świata.

Dodatkowo pojawił się nowy argument metody `HitTest`. Zła wiadomość jest taka że dodanie go jest niestety konieczne i musimy modyfikować wszystkie figury. Dobra wiadomość jest taka, że zarówno w przypadku kuli jak i płaszczyzny napisanie jej jest banalne.

Tak wygląda teraz definicja GeometricObject:

```
abstract class GeometricObject
{
    public IMaterial Material { get; set; }

    public abstract bool HitTest(Ray ray, ref double distance, ref Vector3 normal);
}
```

Płaszczyzna zna już swoją normalną więc po prostu ją zwracamy - nowy HitTest płaszczyzny:

```
public override bool HitTest(Ray ray, ref double distance, ref Vector3 outNormal)
{
    double t = (point - ray.Origin).Dot(normal) / ray.Direction.Dot(normal);

    if (t > Ray.Epsilon)
    {
        distance = t;
        outNormal = normal;
        return true;
    }

    return false;
}
```

W przypadku kuli normalna to (znormalizowana) odległość punktu trafienia od środka:

```
public override bool HitTest(Ray ray, ref double minDistance, ref Vector3 outNormal)
{
    // bez zmian
    // ...

    Vector3 hitPoint = (ray.Origin + ray.Direction * t);
    outNormal = (hitPoint - center).Normalized;
    minDistance = t;
    return true;
```

```
}
```

Tyle, to było proste. Jeśli już jesteśmy przy obiektach geometrycznych - niepotrzebnie przyjmują kolor w konstruktorze, od teraz używamy materiałów.

```
public Sphere(Vector3 center, float radius, IMaterial material)
{
    this.center = center;
    this.radius = radius;
    base.Material = material;
}

public Plane(Vector3 point, Vector3 normal, IMaterial material)
{
    this.point = point;
    this.normal = normal.Normalized;
    base.Material = material;
}
```

Już prawie skończyliśmy, ale najgorsze przed nami. Trzeba jeszcze zmodyfikować klasę Raytracer która zajmuje się faktycznym renderowaniem obrazu.

```
public Bitmap Raytrace(World world, ICamera camera, Size imageSize)
{
    Bitmap bmp = new Bitmap(imageSize.Width, imageSize.Height);

    for (int y = 0; y < imageSize.Height; y++)
        for (int x = 0; x < imageSize.Width; x++)
        {
            // przeskalowanie x i y do zakresu [-1; 1]
            Vector2 pictureCoordinates = new Vector2(
                ((x + 0.5) / (double)imageSize.Width) * 2 - 1,
                ((y + 0.5) / (double)imageSize.Height) * 2 - 1);

            // wysłanie promienia i sprawdzenie w co właściwie trafił
            Ray ray = camera.GetRayTo(pictureCoordinates);

            bmp.SetPixel(x, y, StripColor(ShadeRay(world, ray)));
        }

    return bmp;
}
```

Co powinno się znaleźć w miejscu oznaczonym za pomocą `???`? Powinniśmy tam obliczać kolor na podstawie informacji o trafionym obiekcie. W tym celu po prostu sumujemy wkład każdego światła na scenie w ilość światła odbitego przez materiał.

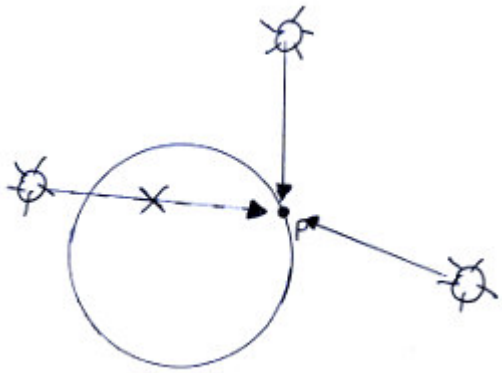
```
public ColorRgb ShadeRay(World world, Ray ray)
{
    HitInfo info = world.TraceRay(ray);

    if (info.HitObject == null) { return world.BackgroundColor; }

    ColorRgb finalColor = ColorRgb.Black;
    IMaterial material = info.HitObject.Material;

    foreach (var light in world.Lights)
    {
        finalColor += material.Radiance(light, info);
    }
```

```
    return finalColor;
}
```



W tej chwili znamy już kolor promienia którego szukamy, ale niestety - jest to obiekt naszej własnej klasy ColorRgb a żeby zapisać go do bitmapy potrzebujemy mieć System.Drawing.Color. Na razie załatwimy to w prosty sposób - obetniemy kolor jeśli jest zbyt jasny/ciemny i pomnożymy razy 255.

```
Color StripColor(ColorRgb colorInfo)
{
    colorInfo.R = colorInfo.R < 0 ? 0 : colorInfo.R > 1 ? 1 : colorInfo.R;
    colorInfo.G = colorInfo.G < 0 ? 0 : colorInfo.G > 1 ? 1 : colorInfo.G;
    colorInfo.B = colorInfo.B < 0 ? 0 : colorInfo.B > 1 ? 1 : colorInfo.B;

    return Color.FromArgb((int)(colorInfo.R * 255),
        (int)(colorInfo.G * 255),
        (int)(colorInfo.B * 255));
}
```

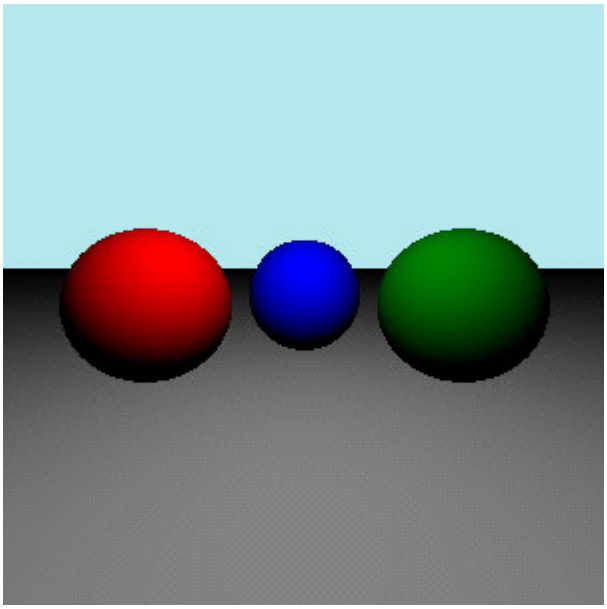
Gotowe. Teraz możemy ustawić scenę od nowa i cieszyć się efektem:

```
// Materiały
IMaterial redMat = new PerfectDiffuse(Color.Red);
IMaterial greenMat = new PerfectDiffuse(Color.Green);
IMaterial blueMat = new PerfectDiffuse(Color.Blue);
IMaterial grayMat = new PerfectDiffuse(Color.Gray);

// Trzy różnokolorowe kule
world.Add(new Sphere(new Vector3(-4, 0, 0), 2, redMat));
world.Add(new Sphere(new Vector3(4, 0, 0), 2, greenMat));
world.Add(new Sphere(new Vector3(0, 0, 3), 2, blueMat));
world.Add(new Plane(new Vector3(0, -2, 0), new Vector3(0, 1, 0), grayMat));

world.AddLight(new PointLight(new Vector3(0, 5, -5), Color.White));
```

Efekt ciągle nie powoduje opadu szczęki (trzeba będzie poczekać do odbić i załamań), ale jest znacznie lepiej. Widać w końcu że działamy w 3D, oraz nie trzeba się już wysilać żeby zobaczyć tam trzy kule i płaszczyznę.



Cóż, w tej części tyle. Od tego momentu efekty naszej ciężkiej pracy powinny pojawiać się szybciej i robić większe wrażenie.