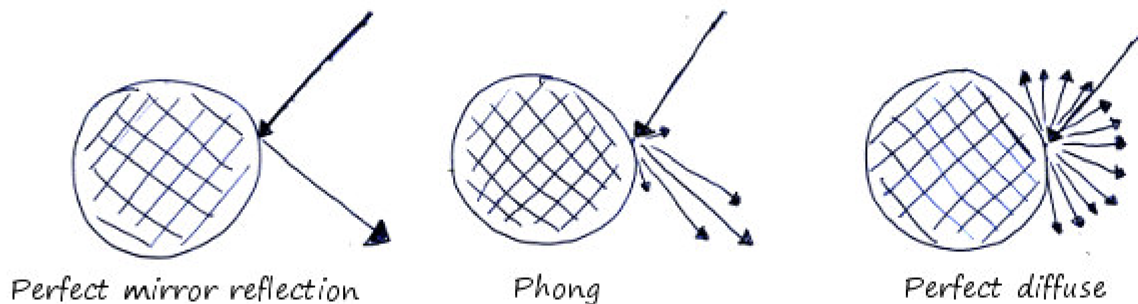
Raytracing: krok po kroku

cz. 7 - doskonałe odbicie lustrzane

I. Co/Dlaczego?

Jako że dodawanie kolejnych materiałów jest jednocześnie dość proste i powoduje szybkie efekty, spróbujemy w tej części pójść dalej w tym kierunku - ale prędzej czy później będzie trzeba wrócić do trudniejszych i bardziej nużących tematów...

Więc co tym razem? Stworzymy materiał potrafiący odbijać światło. Koniec z czasami kiedy wygląd obiektu był niezależny od reszty świata - pojawią się teraz odbicia lustrzane pozwalające na nowe efekty.
Materiał którym się zajmiemy będzie doskonałym materiałem lustrzanym odbijającym promienie bez najmniejszego rozproszenia. Nie jest to oczywiście rzecz możliwa w prawdziwym świecie, ale ostatecznie to samo dotyczy materiałów perfekcyjnie rozpraszających i modelu Phonga.

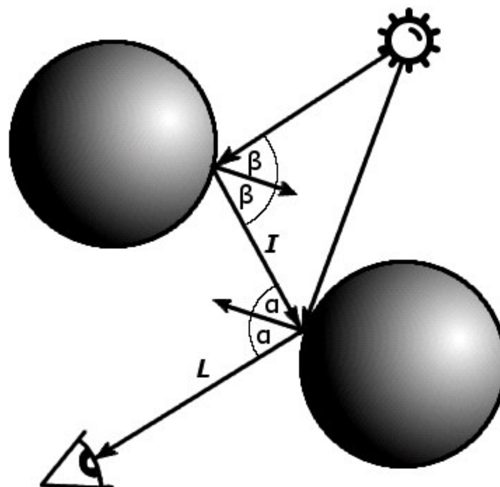


II. Teoria

Do tego momentu kolor cieniowanego punktu był zależny tylko od światła padającego na niego w prostej linii ze źródła - jest to tzw. *direct illumination* - bezpośrednie oświetlenie, jako że światło dociera do punktu *bezpośrednio*. W tym momencie spotykamy się z *indirect illumination* - światło odbite od innych obiektów. Ostateczny kolor jest sumą *direct* i *indirect illumination.*

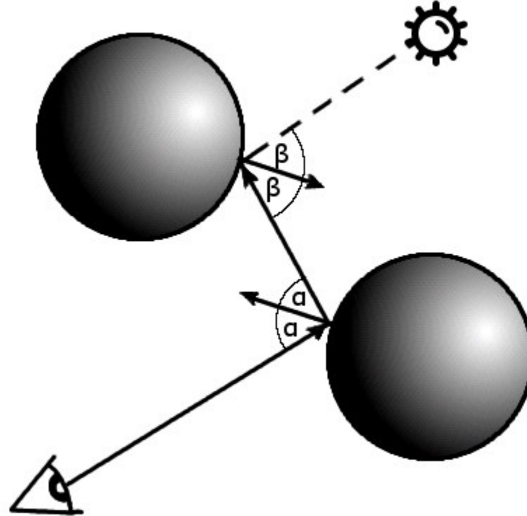
$$L(p,\omega) = L_{direct}(p,\omega) + L_{indirect}(p,\omega)$$

Żeby wyliczyć kolor pokrytego lustrzanym materiałem punktu, musimy zsumować te dwa czynniki. Wyliczenie oświetlenia bezpośredniego przyjdzie nam łatwo - skorzystamy z kodu napisanego w poprzedniej części. Pytanie na dzisiaj brzmi - jak wyliczyć dla naszego materiału oświetlenie pośrednie?



Można zauważyć że wektor odbitego światła L jest wektorem wchodzącego światła pośredniego I odbitym od normalnej powierzchni w tym punkcie - jako że znamy wektor odbitego światła (jest to wektor przeciwny do kierunku naszego promienia) i znamy normalną powierzchni, daje nam to prosty sposób na znalezienie wektora wchodzącego światła - postępujemy w odwrotny sposób i odbijamy wektor odbitego światła od normalnej.

Do czego nam wektor wchodzącego światła? Możemy kontynuować robienie wszystkiego na opak, i śledzić wchodzący promień żeby zobaczyć z jakiego kierunku promień do nas przyszedł.



A kiedy już wiemy skąd promień nadszedł... Możemy obliczyć jego jasność względem cieniowanego (lustrzanego) punktu w taki sam sposób jak obliczamy jego jasność względem kamery (obliczyć ile światła odbija w kierunku cieniowanego, pierwotnie punktu). Dzięki temu otrzymamy współczynnik pośredniego oświetlenia i skończymy z materiałem odbijającym światło.

III. Zmiany

Zanim zaczniemy kombinować z odbiciem, musimy zrobić coś jeszcze. Pamiętacie część mówiącą o cieniowaniu? Dodaliśmy tam między innymi taką pętlę (w klasie Raytracer):

```
foreach (var light in world.Lights)
{
    if (world.AnyObstacleBetween(info.HitPoint, light.Position)) { continue; }

    finalColor += material.Radiance(light, info);
}
```

Jest to ładny, czytelny kod i dobrze oddaje nasze intencje.
Niestety - czas się za wstydem przyznać - to było podwójne oszustwo:

- Założyliśmy że jeśli do jakiegoś punktu nie dochodzi światło bezpośrednio, ten punkt z pewnością jest zupełnie nieoświetlony (czarny) - w przypadku materiału lustrzanego to już nieprawda, ponieważ materiał może być oświetlony wyłącznie za pomocą światła odbitego.
- Założyliśmy że możemy obliczyć jasność punktu sumując wkład poszczególnych świateł - co jest nieprawdą, ponieważ czynniki takie jak odbicie liczymy tylko raz.

Co więc z tym zrobimy? Niestety, wymienimy siekierkę na kijek, piękny kod na copy&paste.

Każdy materiał będzie musiał osobiście zajmować się sprawdzaniem wszystkich świateł po kolei i sumowaniem ich wkładu.

Interfejs IMaterial będzie się musiał mocno zmienić:

```
interface IMaterial
{
    ColorRgb Shade(Raytracer tracer, HitInfo hit);
}
```

Jak widać, nie ma już parametru `light` (światło którego wkład liczymy), za to pojawił się `tracer` (o nim napiszemy później - na razie niech tam siedzi).

Zmienią się w związku z tym klasy PerfectDiffuse i Phong - w obydwóch trzeba dodać pętlę foreach iterującą po światłach w świecie (przekazywanym wśród parametrów).

Czyli poprzedni kod który można streścić mniej-więcej tak:

```
ColorRgb result;
/* obliczenia */

return result;
```

Zmieni się na taki:

```
ColorRgb totalColor = ColorRgb.Black;
foreach (var light in hit.World.Lights)
{
    ColorRgb result;
    /* obliczenia */

    totalColor += result;
}
return totalColor;
```

Dla kompletności, poniżej podany jest kompletny, zmieniony kod:

klasa PerfectDiffuse:

```
public ColorRgb Shade(Raytracer tracer, HitInfo hit)
{
    ColorRgb totalColor = ColorRgb.Black;

    foreach (var light in hit.World.Lights)
    {
        Vector3 inDirection = (light.Position - hit.HitPoint).Normalized;
        double diffuseFactor = inDirection.Dot(hit.Normal);

        if (diffuseFactor < 0) { continue; }

        if (hit.World.AnyObstacleBetween(hit.HitPoint, light.Position))
        { continue; }

        totalColor += light.Color * materialColor * diffuseFactor;
    }

    return totalColor;
}
```

klasa Phong:

```
public ColorRgb Shade(Raytracer tracer, HitInfo hit)
{
    ColorRgb totalColor = ColorRgb.Black;

    foreach (var light in hit.World.Lights)
    {
        Vector3 inDirection = (light.Position - hit.HitPoint).Normalized;
        double diffuseFactor = inDirection.Dot(hit.Normal);
```

```
        if (diffuseFactor < 0) { continue; }

        if (hit.World.AnyObstacleBetween(hit.HitPoint, light.Position))
        { continue; }

        ColorRgb result = light.Color * materialColor * diffuseFactor * diffuseCoeff;
        double phongFactor = PhongFactor(inDirection, hit.Normal, -hit.Ray.Direction);

        if (phongFactor != 0)
        { result += materialColor * specular * phongFactor; }

        totalColor += result;
    }

    return totalColor;
}
```

Skromnym zdaniem autora, nowa wersja jest dużo brzydsza od poprzedniej, ale będziemy się z tym musieli nauczyć żyć (reszta pisanego dzisiaj kodu będzie już schludniejsza). Jedyną zaletą dokonanych zmian jest to, że wspomniana pętla:

```
foreach (var light in world.Lights)
{
    if (world.AnyObstacleBetween(info.HitPoint, light.Position)) { continue; }

    finalColor += material.Radiance(light, info);
}
return finalColor;
```

Zamienia się w proste

```
return material.Shade(this, info);
```

III. Implementacja

Pisanie materiału modelującego odbicie (nazwiemy go Reflective) zaczniemy jak zwykle od napisania szkieletu klasy:

```
class Reflective : IMaterial
{
    Phong direct; // do bezpośredniego oświetlenia
    double reflectivity;
    ColorRgb reflectionColor;

    public Reflective(ColorRgb materialColor,
        double diffuse,
        double specular,
        double exponent,
        double reflectivity)
    {
        this.direct = new Phong(materialColor, diffuse, specular, exponent);
        this.reflectivity = reflectivity;
        this.reflectionColor = materialColor;
    }

    public ColorRgb Shade(Raytracer tracer, HitInfo hit)
    {
        throw new System.NotImplementedException();
    }
}
```

Jak zwykle - kolejny materiał, więcej parametrów w konstruktorze.
MaterialColor to podstawowy kolor materiału i nie wymaga raczej żadnych wyjaśnień.
Diffuse, specular i exponent są już znane - oznaczają odpowiednio jaka część światła jest przez obiekt rozpraszana, jak silne są specular highlights, oraz jaki wykładnik jest użyty do ich modelowania.
Reflectivity oznacza jak mocno materiał odbija światło - przy wartości 0 nie ma żadnych odbić i materiał Reflective

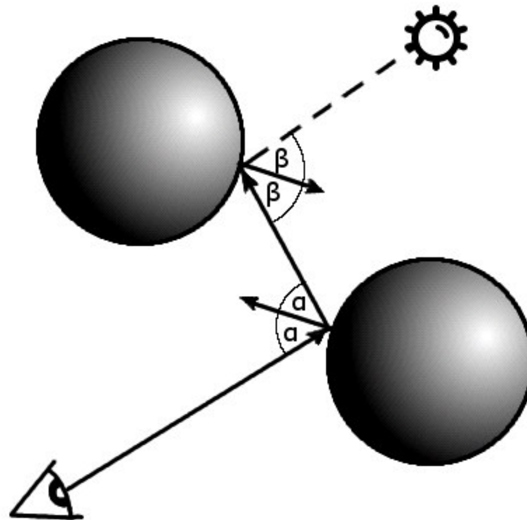
jest identyczny do Phonga. Przy wartości 1 (i diffuse równym 0) materiał jest idealnym lustrem i odbija całe padające na niego światło.

Współczynniki diffuse i reflectivity powinny zawsze się sumować maksymalnie do 1 - ponieważ część światła jest przez obiekt rozpraszana, część doskonale odbijana, ale nigdy więcej światła nie opuści obiekt niż na niego pada. Nie jest to wymuszone w kodzie, ale lepiej o tym pamiętać - w przeciwnym wypadku wyrenderowane sceny mogą wyjść nienaturalnie jasne.

Dzięki polu `direct` całe obliczenia dla światła bezpośredniego wykonamy jedną linijką:

```
ColorRgb radiance = direct.Radiance(light, hit);
```

Oświetlenie bezpośrednie załatwione.
I co dalej? Do wyliczenia odbicia, potrzebujemy informacji skąd wpada odbijane światło. Możemy się tego dowiedzieć, jak było wspominane, odbijając wektor światła odbitego (liczymy światło odbijające się *w kierunku kamery*, więc bierzemy odwrotność kierunku promienia który idzie *od kamery*) od normalnej:




```
Vector3 toCameraDirection = -hit.Ray.Direction;
Vector3 reflectionDirection = Vector3.Reflect(toCameraDirection, hit.Normal);
Ray reflectedRay = new Ray(hit.HitPoint, reflectionDirection);
```

Znając kierunek (reflectionDirection) z którego przychodzi odbijane światło, możemy stworzyć promień (reflectedRay) w tamtym kierunku i `śledzić` go.

```
tracer.ShadeRay(hit.World, reflectedRay);
```

Ale musimy wziąć pod uwagę to, że promień może teoretycznie odbijać się w nieskończoność od powierzchni (na przykład wewnątrz lustrzanej sfery) - dlatego musimy ograniczyć ilość odbić do jakiejś, ustalonej z góry wartości.

Trzeba w jakiś sposób śledzić informację o poziomie rekurencji w jakim się znajdujemy podczas cieniowania punktu. Do przechowywania takich informacji przystosowana jest klasa HitInfo, dlatego dorzucimy jej jeszcze jedno pole:

```
/// <summary>Zwiększana przy śledzeniu odbitego lub załamanego promienia</summary>
public int Depth { get; set; }
```

Zmodyfikujemy lekko funkcję Raytracer.ShadeRay żeby brała pod uwagę możliwość zapędzenia się w rekurencji:

```
public ColorRgb ShadeRay(World world, Ray ray, int currentDepth)
{
    if (currentDepth > maxDepth) { return ColorRgb.Black; }
```

```
    HitInfo info = world.TraceRay(ray);
    info.Depth = currentDepth + 1;

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

    IMaterial material = info.HitObject.Material;

    return material.Shade(this, info);
}
```

Zmienna `maxDepth` nie weźmie się znikąd niestety, więc musimy ją zadeklarować i przypisać wartość w konstruktorze:

```
class Raytracer
{
    int maxDepth;

    public Raytracer(int maxDepth)
    {
        this.maxDepth = maxDepth;
    }

    ...
}
```

W metodzie Main, zmieniamy tworzenie tracera na:

```
Raytracer tracer = new Raytracer(5);
```

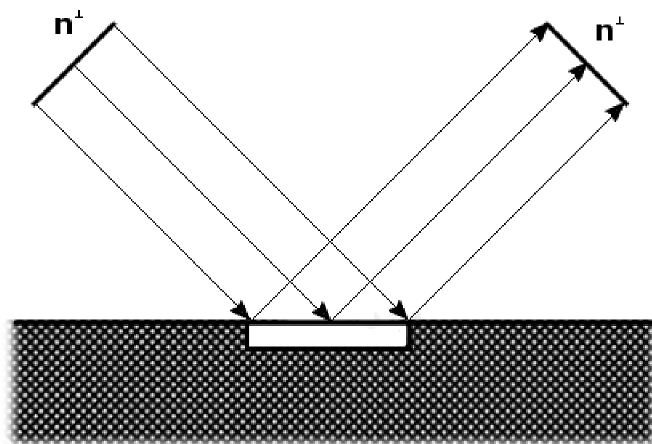
W samym tracerze, usunęliśmy wersję metody ShadeRay nie pobierającą parametru currentDepth (zapomnienie o dodaniu tego parametru mogłoby mieć fatalne skutki), więc musimy zmienić linijkę kodu w funkcji Raytrace:

```
bmp.SetPixel(x, y, StripColor(ShadeRay(world, ray, 0)));
```

Teraz możemy w końcu prześledzić odbity promień pisząc:

```
tracer.ShadeRay(hit.World, reflectedRay, hit.Depth);
```

Co jeszcze może wpływać na jasność odbitego światła? Może czynnik lamberta? Otóż okazuje się że nie, nie w tym przypadku:



Przekrój przez strumień promieni wchodzących jest równy przekrojowi promieni odbitych - oznacza to że gęstość strumienia promieni odbitych nie zależy od kąta padania.

Teraz możemy w końcu wyliczyć odbite światło:

```
ColorRgb reflected = tracer.ShadeRay(hit.World, reflectedRay, hit.Depth) *
    reflectionColor *
    reflectivity;
```

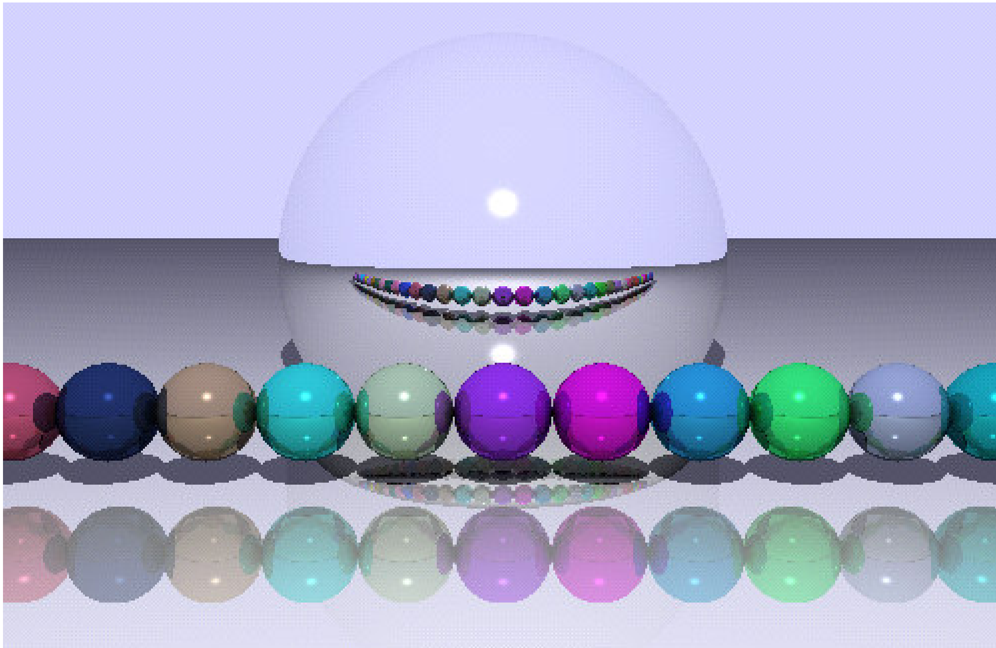
I połączyć wszystko w całość:

```
public ColorRgb Shade(Raytracer tracer, HitInfo hit)
{
    Vector3 toCameraDirection = -hit.Ray.Direction;

    ColorRgb radiance = direct.Shade(tracer, hit);
    Vector3 reflectionDirection = Vector3.Reflect(toCameraDirection, hit.Normal);
    Ray reflectedRay = new Ray(hit.HitPoint, reflectionDirection);

    radiance += tracer.ShadeRay(hit.World, reflectedRay, hit.Depth) *
        reflectionColor *
        reflectivity;

    return radiance;
}
```

Pomijając zmiany wprowadzone w innych klasach, kodu wykonującego odbicie jak widać nie jest dużo.

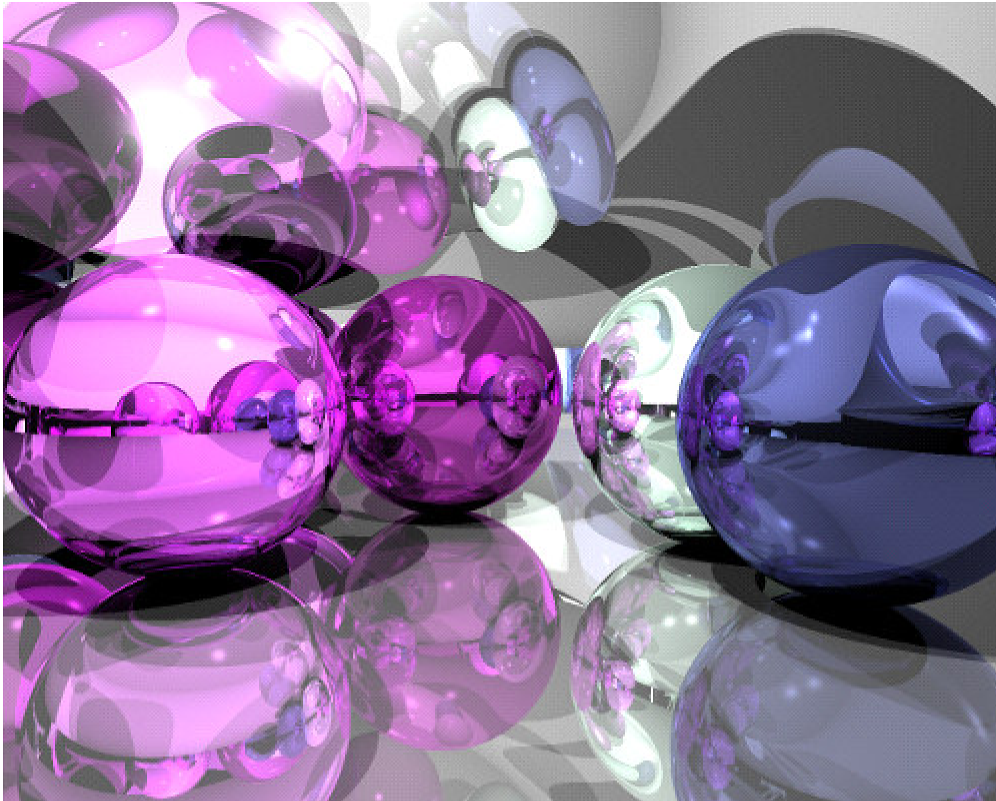
IV. Wyniki

Wszystkie przykładowe sceny wyrenderowane oczywiście za pomocą tylko i wyłącznie dotychczas napisanego kodu. Nareszcie mamy coś czym można się ewentualnie komuś pochwalić:



Przy odrobinie wyobraźni można bez problemu stworzyć robiące wrażenie, niemożliwe w prawdziwym świecie sceny (czy kiedykolwiek zastanawiałeś się co się stanie jeśli ustawisz dwa lustra naprzeciwko siebie?).

A jeśli ustawimy kilka kul, oświetlimy je i przykryjemy odbijającą światło sferą?



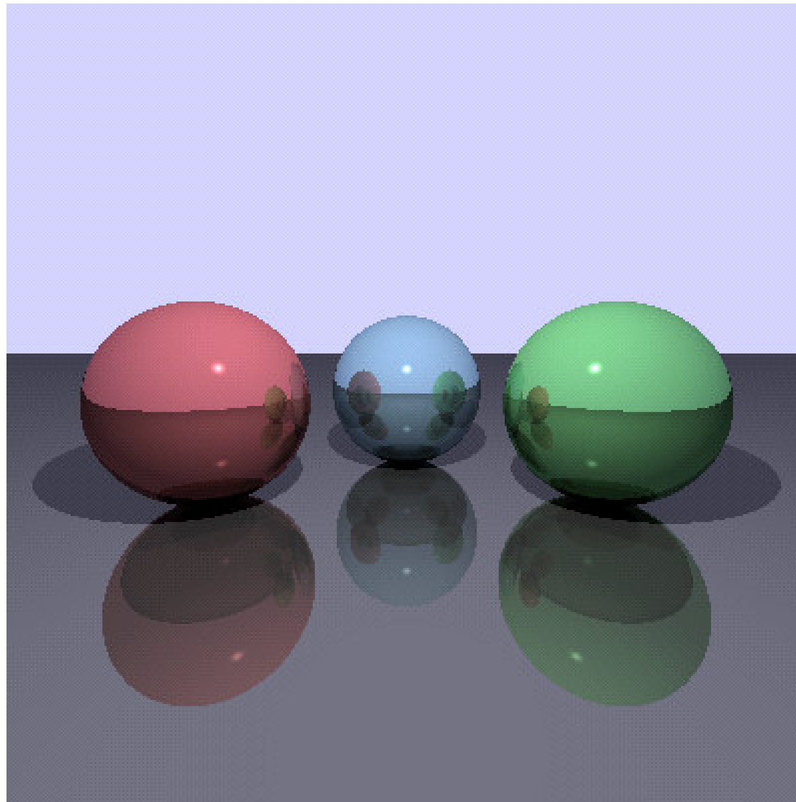
Ciekawostka - jeśli połączymy cztery kule krawędziami i dodamy odpowiednie światło, otrzymamy pewien znany fraktal:

I na sam koniec - jak prezentuje się nasza ulubiona scena?

Zmodyfikujmy materiały obiektów na scenie:

```
IMaterial redMat = new Reflective(Color.LightCoral, 0.4, 1, 300, 0.6);
IMaterial greenMat = new Reflective(Color.LightGreen, 0.4, 1, 300, 0.6);
IMaterial blueMat = new Reflective(Color.LightBlue, 0.4, 1, 300, 0.6);
IMaterial grayMat = new Reflective(Color.Gray, 0.4, 1, 300, 0.6);
```

A oto efekt:



Całkiem nieźle w porównaniu z poprzednią wersją...

Niestety - tak jak groziłem - następna część będzie nudniejsza i nie przyniesie tak rzucających się w oczy rezultatów - ale bez niej wiele byśmy stracili...