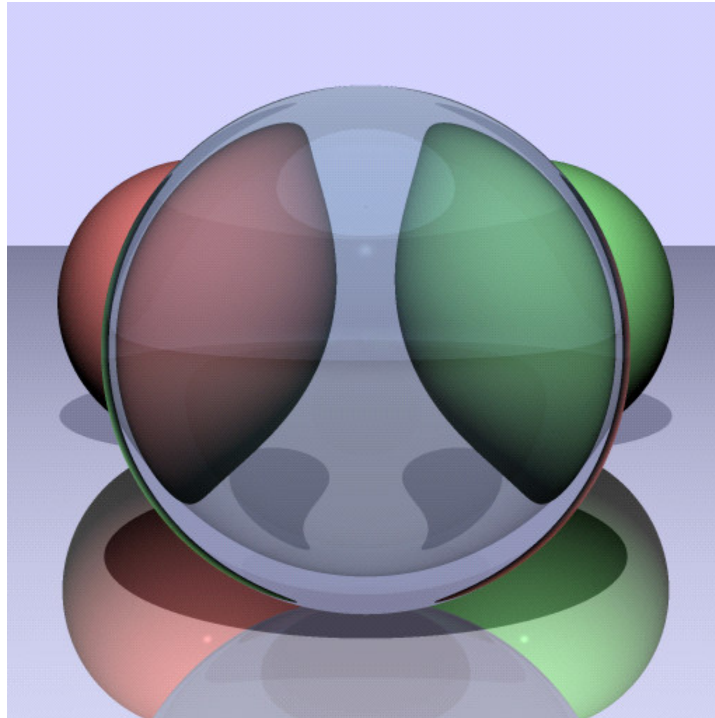
Raytracing: krok po kroku

cz. 11 - uproszczona przeźroczystość

I. Co/Dlaczego?

Przeźroczystość jest jednym z efektów przy którym raytracing naprawdę błyszczy pozwalając uzyskać efekty praktycznie niemożliwe przy tradycyjnych sposobach renderowania.



Przeźroczysta kula załamująca światło

Tak naprawdę model przeźroczystości zaprezentowany w tej części jest lekko uproszczony, ale już on potrafi dawać naprawdę dobre efekty - w przyszłości prawdopodobnie lekko go poprawimy zwiększając jego możliwości.

II. Podstawy fizyczne

Kiedy fala przechodzi przez przeźroczysty ośrodek, jak na przykład powietrze, szkło, woda - jest spowalniana przez jego cząsteczki przez co rozchodzi się z mniejszą prędkością.

Absolutny współczynnik załamania ośrodka *(ang. index of refraction, ior,* η*)* jest równy prędkości światła w próżni podzielonej przez przez prędkość światła w ośrodku

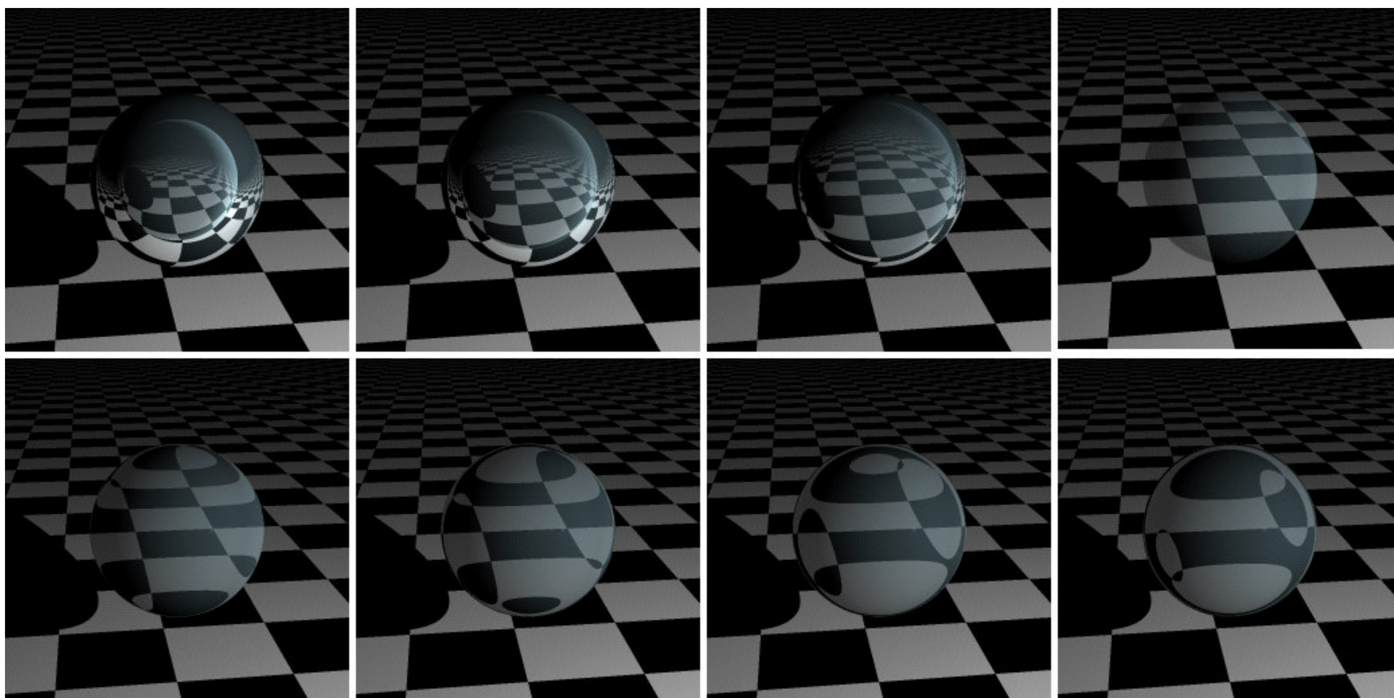
$$\eta_a = \frac{c}{v}$$

Relatywny współczynnik załamania na granicy dwóch materiałów jest równy prędkości fali wpadającej do ośrodka podzielonej przez prędkość fali w ośrodku:

$$\eta = \frac{v_i}{v_t} = \frac{\eta_t}{\eta_i}$$

Uwaga dotycząca oznaczeń - \`t\` w indeksie dolnym dotyczy zawsze promienia załamanego (*transmitted*), \`i\` oznacza zawsze promień padający (*incoming*).
Relatywny współczynnik załamania jest głównym parametrem interesującym nas przy przeźroczystości i to nim się w praktyce głównie zajmujemy, ponieważ to on wpływa bezpośrednio na to jak zachowuje się światło przy przechodzieniu przez ośrodek - absolutne wartości są użyteczne głównie do obliczania relatywnych.

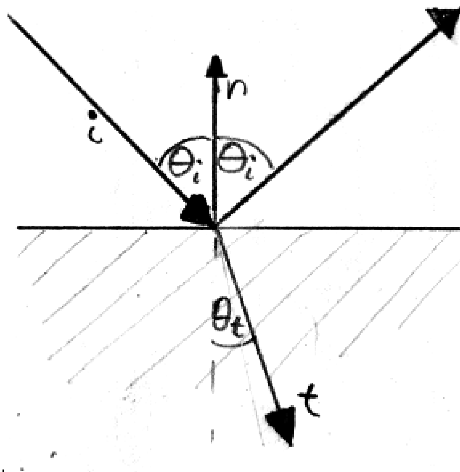
Współczynnik załamania próżni wynosi równe 1.
W przypadku powietrza, jest równy minimalnie więcej i wynosi około 1.0003, a dla wody jest równy 1.33.
Jednym z najbardziej charakterystycznych materiałów mocno załamujących światło jest diament mając η = 2.42.



Ta sama kula przy różnych niewielkich relatywnych współczynnikach załamania światła
Od lewej: 0.7, 0.8, 0.9, 1, 1.01, 1.02, 1.03, 1.04

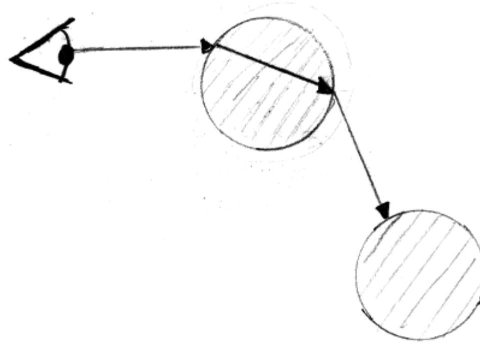
III. Załamanie światła

Kiedy promień trafia na przeźroczysty ośrodek powstaje, jak zawsze, odbity promień, ale tym razem *może* również powstać promień załamany.

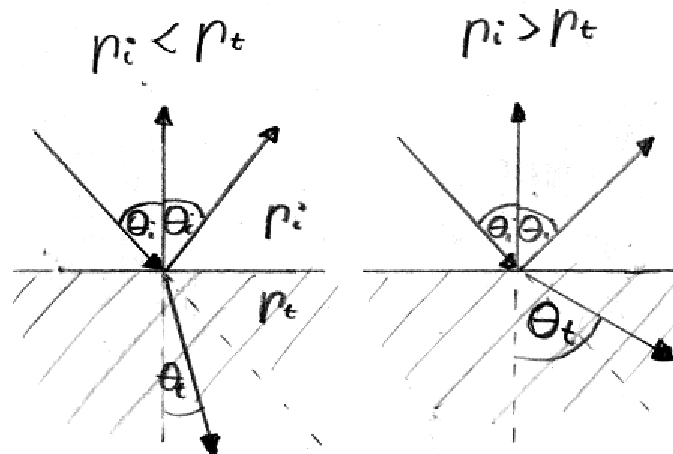


Najważniejsze co należy zauważyć na powyższej ilustracji jest to, że promień wychodzący (t) nie jest równoległy do promienia wpadającego (w).

Dzieje się tak z powodu zjawiska fizycznego zwanego refrakcją (załamaniem) - fala zmienia kierunek przechodząc między ośrodkami. Kąt pomiędzy załamanym promieniem a normalną powierzchni na którą pada jest zwany kątem refrakcji:



Promień może zostać załamany na dwa sposoby - w kierunku normalnej powierzchni i w kierunku przeciwnym do normalnej powierzchni.



Sytuacja pierwsza zachodzi kiedy promień wpada z ośrodka z mniejszym ior (współczynnikiem załamania) do ośrodka o większym ior, a druga kiedy promień wpada z ośrodka o większym ior do ośrodka o mniejszym ior.

Pomiędzy kątem refrakcji a kątem padania istnieje zależność znana jako prawo refrakcji (albo prawo załamania, albo prawo Snelliusa).

$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{\eta_i}{\eta_t} = \eta$$

Tzn. sinus kąta pod jakim pada promień (θi) podzielony przez sinus kąta załamanego promienia (θt) jest równy relatywnemu współczynnikowi załamania pomiędzy nimi.

Znając wzory na odbicie i refrakcję, można wyprowadzić (albo lepiej - uwierzyć mi na słowo) wzór na kierunek promienia wychodzącego.

$$t = \frac{1}{\eta}\omega_o - \left(\cos\theta_t - \frac{1}{\eta}\cos\theta_i\right)n$$

gdzie:

$$\cos\theta_i = \eta \cdot \omega_o$$

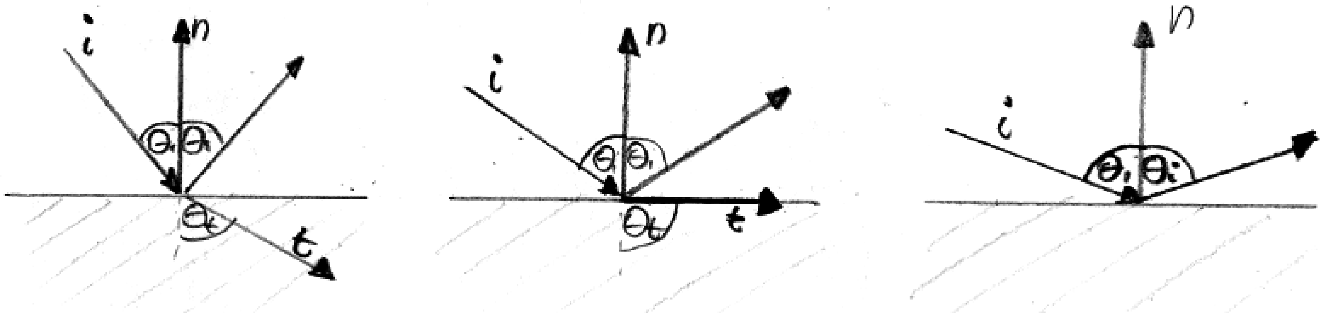
$$\cos\theta_t = \sqrt{1 - \frac{1}{\eta^2}\left(1 - \cos^2\theta_i\right)}$$

Wyglądają trochę strasznie... A to dopiero początek dziwnych wzorów. Czas na omówienie specjalnego przypadku załamania światła - całkowitego wewnętrznego odbicia.

III. Całkowite wewnętrzne odbicie

Całkowite wewnętrzne odbicie (ang. *total internal reflection*) to zjawisko występujące przy falach (w tym, oczywiście, przy świetle) na granicy różnych ośrodków.
Objawia się tym, że jeśli światło przechodzi z ośrodka o wyższym współczynniku załamania (ior) do ośrodka z niższym (czyli relatywny współczynnik załamania tych dwóch ośrodków < 1) oraz pada pod kątem większym niż pewien *kąt graniczny* (ang. *critical angle*), nie przechodzi do ośrodka ale ulega całkowitemu odbiciu.

Wynika to bezpośrednio z tego, że światło w takim wypadku jest załamywane *od* normalnej - patrz: rysunek.



Całkowite wewnętrzne odbicie następuje wtedy kiedy promień załamany musiałby wręcz wychodzić poza przeźroczysty ośrodek (jak na ostatniej sytuacji na ilustracji powyżej).

Jesli chcemy renderować poprawnie wyglądające sceny, musimy wziąść pod uwagę całkowite wewnętrzne odbicie. Pozostaje pytanie, w jaki sposób sprawdzić czy to zjawisko zachodzi?

Najoczywistszym sposobem na sprawdzenie tego jest test θi > θc (czyli kiedy kąt pod którym pada promień jest większy od kąta krytycznego) - do tego jednak trzeba by wyliczyć θc - nie jest to konieczne, można to samo zapisać w inny, równoważny sposób:

$$1-\frac{1}{\eta^2}\left(1-\cos^2\theta_i\right)<0$$

Dlaczego tak? Przypominając podany wcześniej wzór na cosθt:

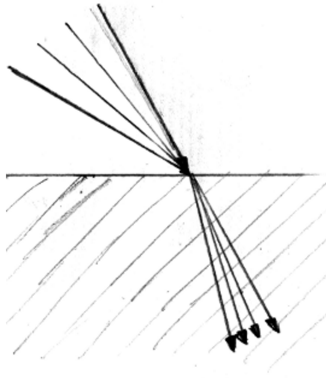
$$\cos\theta_t=\sqrt{1-\frac{1}{\eta^2}\left(1-\cos^2\theta_i\right)}$$

Jeśli wartość po lewej stronie nierówności jest mniejsza od zera, wtedy cosθt jest równa pierwsiastkowi z liczby ujemnej, czyli jest liczbą urojoną - co można traktować w tym przypadku jako inny sposób na powiedzenie że ten nie istnieje.

III. Kolor załamanego światła

Musimy jeszcze wiedzieć w jaki sposób obliczyć jak mocno załamane promienie wpływają na to co odbiera kamera.

Okazuje się że kiedy promienie przechodzą z ośrodka o mniejszym współczynniku załamania (np. powietrze) do ośrodka o większym współczynniku załamania (np. szkło), ich \`gęstość\` nie zostaje taka sama, ale załamanie powoduje ich *ścieśnienie*:



Zależność między natężeniem padającego i transmitowango światła można zapisać jako (można ja wyprowadzić za pomocą prawa refrakcji):

$$L_t = k_t\left(\frac{\eta_t^2}{\eta_i^2}\right)L_i = k_t\left(\frac{1}{\eta}\right)L_i$$

To tyle teorii potrzebnej do modelowania przeźroczystości - od tego momentu zajmiemy się trochę bardziej praktycznymi rzeczami, a zaczniemy od... upraszczania.

IV. Uproszczenia

Tak jak było napisane, w tej części przeźroczystość będzie trochę uproszczona - nie będziemy dla każdego załamywanego promienia wyliczać współczynnika załamania, ale będziemy przechowywać informacje o współczynniku załamania dla każdego obiektu i założymy że `*na zewnątrz*` każdego obiektu współczynnik załamania jest równy 1 (dla powietrza jest to bardzo dobre przybliżenie).

Upraszcza to trochę kod, ale niemożliwe jest przez to modelowanie zagnieżdżonych w sobie przeźroczystych obiektów - w tym celu musielibyśmy przechowywać dla każdego śledzonego promienia współczynnik załamania dla ośrodka w którym się aktualnie znajduje.

Z tego powodu, wystarczy że rozważymy dwa przypadki - promień wpadający do obiektu i promień wychodzący z obiektu.
Jeśli promień wpada do obiektu, korzystamy po prostu z zapisanego w polu klasy, stałego współczynnika załamania η. Jeśli promień wypada z obiektu, współczynnik załamania jest odwrotny czyli równy (1 / η) - oraz musimy odwrócić normalną używaną do obliczeń.

V. Szkielet Kodu

Mimo wszelkich starań, napisanie czytelnego kodu odpowiadającego za przeźroczystość niekoniecznie jest prostym zadaniem - to co zostanie zaprezentowane w tej części jest najładniejszym kodem jaki udało mi się napisać.

No więc może zanim przejdziemy do pisania faktycznego kodu, omówimy go trochę mniej formalnie.
Czynności jakie musimy wykonać po kolei:

1. Obliczyć oświetlenie bezpośrednie, za pomocą na przykład modelu Phonga
2. Sprawdzić czy następuje całkowite wewnętrzne odbicie
    3. Jeśli tak, dodać do wyniku kolor odbitego promienia
    4. Jeśli nie, dodać do wyniku kolor odbitego promienia i transmitowanego promienia
5. Zwrócić wynik

Brzmi prosto, ale łatwo nie będzie.

Na początek stwórzmy na podstawie pseudokodu nic nie robiącą klasę którą będziemy później rozszerzać:

```
class Transparent : IMaterial
{
    Phong direct;
    double refraction;
    double reflection;
    double transmission;
    double specular;
    ColorRgb baseColor;

    public Transparent(ColorRgb color, double diffuse, double specular,
        double exponent, double reflection, double refraction, double transmission)
    {
        this.direct = new Phong(color, diffuse, specular, exponent);
        this.transmission = transmission;
        this.baseColor = color;
        this.reflection = reflection;
        this.specular = specular;
        this.refraction = refraction;
    }

    public ColorRgb Shade(Raytracer tracer, HitInfo hit)
    {
        ColorRgb final = direct.Shade(tracer, hit);

        if (IsTotalInternalReflection(...))
        {
        }
        else
        {
        }

        return final;
    }
}
```

Hmm, trochę dużo kodu jak na nic nie robiący szkielet...
Problemem też jest olbrzymia ilość parametrów w konstruktorze - to już zupełnie nie jest czytelne - ale niech zostanie na razie...

VI. Implementacja: Odbicie

No więc po kolei - na początku liczymy oświetlenie bezpośrednie:

```
ColorRgb final = direct.Shade(tracer, hit);
```

Jako że nic z tym później nie robimy, kod obecnie działa identycznie jak materiał Phonga.

Tak więc najpierw zajmiemy się implementacją odbicia światła - już poprzednio robiliśmy (część 7), teraz wystarczy to dostosować:

```
Vector3 toCameraDirection = -hit.Ray.Direction;
Ray reflectedRay = new Ray(hit.HitPoint, Vector3.Reflect(toCameraDirection, hit.Normal));
ColorRgb reflectionColor = baseColor * reflection;
```

Teraz wystarczy prześledzić promień `reflectedRay` i zmodyfikować go o współczynnik reflectionColor - ale to zrobimy za chwilę...

VII. Implementacja: Całkowite wewnętrzne odbicie

Więc może teraz czas na coś nowego - sprawdzanie czy światło uległo całkowitemu wewnętrznemu odbiciu.

Przypominając może warunek na całkowite wewnętrzne odbicie:

$$1 - \frac{\left(1 - \cos^2 \theta_i\right)}{\eta^2} < 0$$

Jako że lewa strona równania powtarza się w dwóch różnych wzorach, wyciągniemy jego obliczanie do oddzielnej funkcji:

```
double FindRefractionCoeff(double eta, double cosIncidentAngle)
{
    return 1 - (1 - cosIncidentAngle * cosIncidentAngle) / (eta * eta);
}
```

CosIncidentAngle, jak można się domyślić, jest cosinusem kąta pod którym światło pada na ośrodek (we wzorze zapisane jako cosθi).
Eta jest współczynnikiem załamania. Jak było opisane w podrozdziale `uproszczenia`, traktujemy relatywny współczynnik załamania jako stałą dla ośrodka i przechowujemy jako pole w klasie. Bierzemy jedynie poprawkę na to czy światło wpada czy wypada z ośrodka. Warunkiem pozwalającym sprawdzić z ktorej strony pada światło jest cosθi > 0 (kąt pod którym pada światło jest w zakresie -90 do 90 stopni, czyli cosinus kąta jest większy od zera). Zapisujemy to jako:

```
double cosIncidentAngle = hit.Normal.Dot(toCameraDirection);
double eta = cosIncidentAngle > 0 ? refraction : 1 / refraction;
```

A sprawdzenie czy następuje całkowite wewnętrzne odbicie redukuje się do (funkcja której jedynym zadaniem jest stworzenie czytelniejszego warunku w if-ie):

```
bool IsTotalInternalReflection(double refractionCoeff)
{
    return refractionCoeff < 0;
}
```

I co zrobić jeśli okazuje się że całkowite wewnętrzne odbicie następuje? W takim wypadku ignorujemy przeźroczystość i po prostu odbijamy światło w całości:

```
if (IsTotalInternalReflection(refractionCoeff))
{
    final += tracer.TraceRay(hit.World, reflectedRay, hit.Depth);
}
```

VIII. Implementacja: Przeźroczystość

Pozostaje pytanie - co zrobić jeśli światło *nie* odbija się w całości, ale jego część wpada do przeźroczystego ośrodka?

Na pewno musimy wyliczyć kierunek w którym światło się załamuje, po czym, podobnie jak przy odbiciu, prześledzić promień rzucony w tym kierunku.

Rozbijemy sobie tutaj dla czytelności robotę na dodatkowe funkcje, a główny kod sprowadzimy do:

```
Ray transmittedRay = ComputeTransmissionDirection(hit.HitPoint, toCameraDirection, hit.Normal,
    eta, Math.Sqrt(refractionCoeff), cosIncidentAngle);
ColorRgb transmissionColor = ComputeTransmissionColor(eta, hit.Normal,
transmittedRay.Direction);
```

Jako że dzień bez wzorów to dzień stracony, przypomnijmy poprzednio podany wzór na kierunek załamanego promienia:

$$t = \frac{1}{\eta}\omega_o - \left(\cos\theta_t - \frac{1}{\eta}\cos\theta_i\right)n$$

Zanim radośnie się rzucimy do jego implementowania, przypomnienie - jeśli światło pada na materiał od środka, musimy do obliczeń odwrócić kierunek normalnej którą bierzemy pod uwagę (co przy okazji powoduje zmianę branego pod uwagę kąta padania światła).

Po tym ostrzeżeniu, niezrażeni możemy napisać:

```
Ray ComputeTransmissionDirection(Vector3 hitPoint, Vector3 toCameraDirection, Vector3 normal,
    double eta, double cosTransmittedAngle, double cosIncidentAngle)
{
    if (cosIncidentAngle < 0)
    {
        normal = -normal;
        cosIncidentAngle = -cosIncidentAngle;
    }

    Vector3 direction = -toCameraDirection / eta
        - normal * (cosTransmittedAngle - cosIncidentAngle / eta);
    return new Ray(hitPoint, direction);
}
```

Tutaj naprawdę nie ma wiele do tłumaczenia, w końcu to tylko przepisanie wzoru do języka programowania.

Podobnie, przewrotnie korzystając z podanego wcześniej wzoru na kolor załamanego światła:

$$L_t = k_t\left(\frac{\eta_t^2}{\eta_i^2}\right)L_i = k_t\left(\frac{1}{\eta}\right)L_i$$

Możemy zapisać:

```
ColorRgb ComputeTransmissionColor(double eta)
{
    return ((ColorRgb.White * transmission) / (eta * eta));
}
```

I co my z tymi kątami i kolorami możemy zrobić? Jeśli nie zachodzi całkowite wewnętrzne odbicie, bierzemy pod uwagę zarówno odbite światło i transmitowane światło, otrzymując:

```
if (IsTotalInternalReflection(refractionCoeff)) { ... }
else
{
    Ray transmittedRay = ComputeTransmissionDirection(hit.HitPoint, toCameraDirection,
hit.Normal,
        eta, Math.Sqrt(refractionCoeff), cosIncidentAngle);
    ColorRgb transmissionColor = ComputeTransmissionColor(eta);

    final += reflectionColor * tracer.TraceRay(hit.World, reflectedRay, hit.Depth);
    final += transmissionColor * tracer.TraceRay(hit.World, transmittedRay, hit.Depth);
}
```

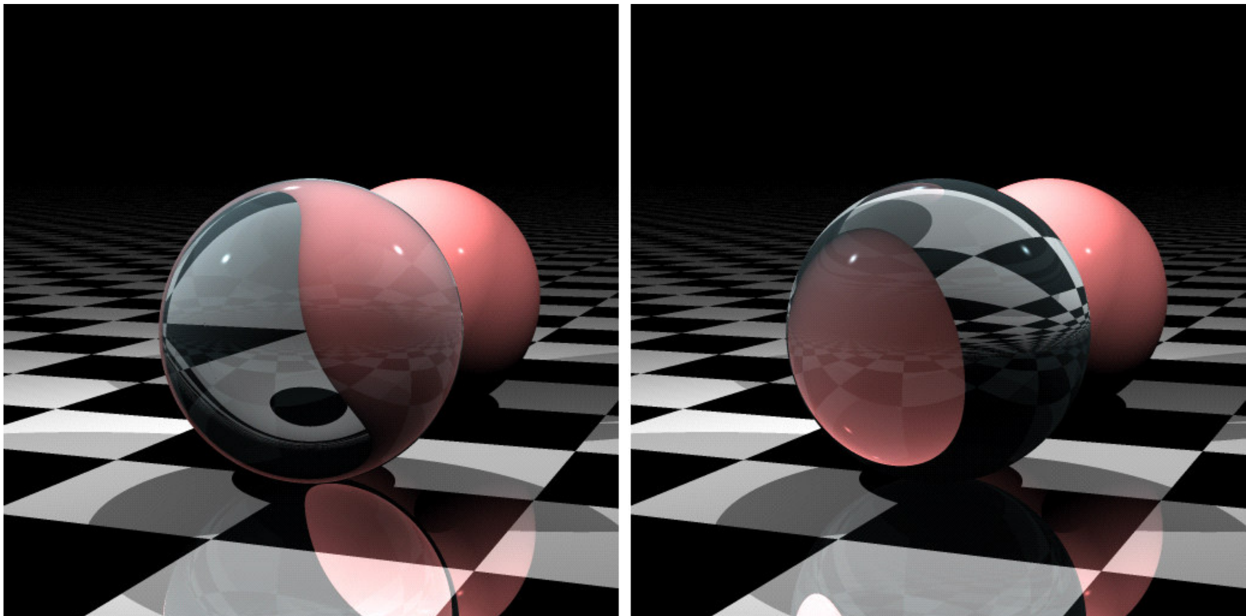
W końcu, po długim pisaniu, możemy połączyć funkcję Shade w całość:

```
public ColorRgb Shade(Raytracer tracer, HitInfo hit)
{
    ColorRgb final = direct.Shade(tracer, hit);

    Vector3 toCameraDirection = -hit.Ray.Direction;
    double cosIncidentAngle = hit.Normal.Dot(toCameraDirection);
    double eta = cosIncidentAngle > 0 ? refraction : 1 / refraction;
    double refractionCoeff = FindRefractionCoeff(eta, cosIncidentAngle);

    Ray reflectedRay = new Ray(hit.HitPoint, Vector3.Reflect(toCameraDirection, hit.Normal));
    ColorRgb reflectionColor = baseColor * reflection;

    if (IsTotalInternalReflection(refractionCoeff))
    {
        final += tracer.TraceRay(hit.World, reflectedRay, hit.Depth);
    }
    else
    {
        Ray transmittedRay = ComputeTransmissionDirection(hit.HitPoint, toCameraDirection,
            hit.Normal, eta, Math.Sqrt(refractionCoeff), cosIncidentAngle);
        ColorRgb transmissionColor = ComputeTransmissionColor(
            eta, hit.Normal, transmittedRay.Direction);

        final += reflectionColor * tracer.TraceRay(hit.World, reflectedRay, hit.Depth);
        final += transmissionColor * tracer.TraceRay(hit.World, transmittedRay, hit.Depth);
    }

    return final;
}
```

XI. Wyniki

Teraz pojawia się pytanie, co właściwie osiągnęliśmy przepisując te wszystkie tajemniczo wyglądające wzory fizyczne do kodu programu.

Tak więc, możemy całkiem zgodnie z prawami fizyki oddać wygląd przeźroczystych obiektów:

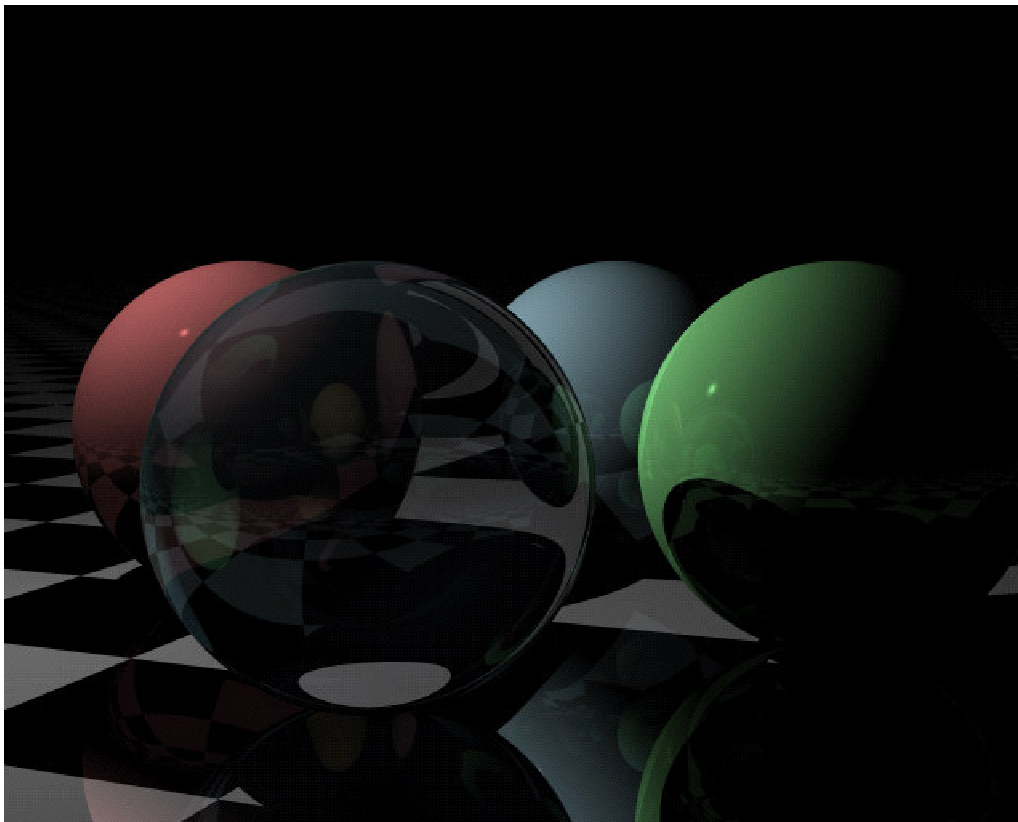


Dwie kule, pierwsza z współczynnikami załamania równymi odpowiednio 1.1 i 1.5

Żeby wykonać coś własnego, trzeba użyć stworzonego w tej części materiału - na przykład tworząc cztery kule i odpowiednio ustawiając podłoże:

```
world.Add(new Sphere(new Vector3(-3.5, 0, 0), 2,
    new Reflective(Color.LightCoral, 0.7, 0.5, 1000, 0.3)));
world.Add(new Sphere(new Vector3(3.5, 0, 0), 2,
    new Reflective(Color.LightGreen, 0.7, 0.5, 1000, 0.3)));
world.Add(new Sphere(new Vector3(0, 0, 3.5), 2,
    new Reflective(Color.LightBlue, 0.7, 0.5, 1000, 0.3)));
world.Add(new Sphere(new Vector3(0, 0, -3.5), 2,
    new Transparent(Color.LightBlue, 0.1, 0, 0, 0.3, 1.05, 0.9)));
world.Add(new Plane(new Vector3(0, -2, 0), new Vector3(0, 1, 0),
    new Reflective(Color.White, 0.4, 0, 1000, 0.6, true););

world.AddLight(new Light(ColorRgb.White, new Vector3(-5, 5, -3)));

ICamera camera = new Pinhole(new Vector3(6, 2, -15),
    new Vector3(0, 0.3, 0),
    new Vector3(0, -1, 0),
    new Vector2(0.7, 0.7),
    2);
```

Co pozwala na osiągnięcie takiego efektu:



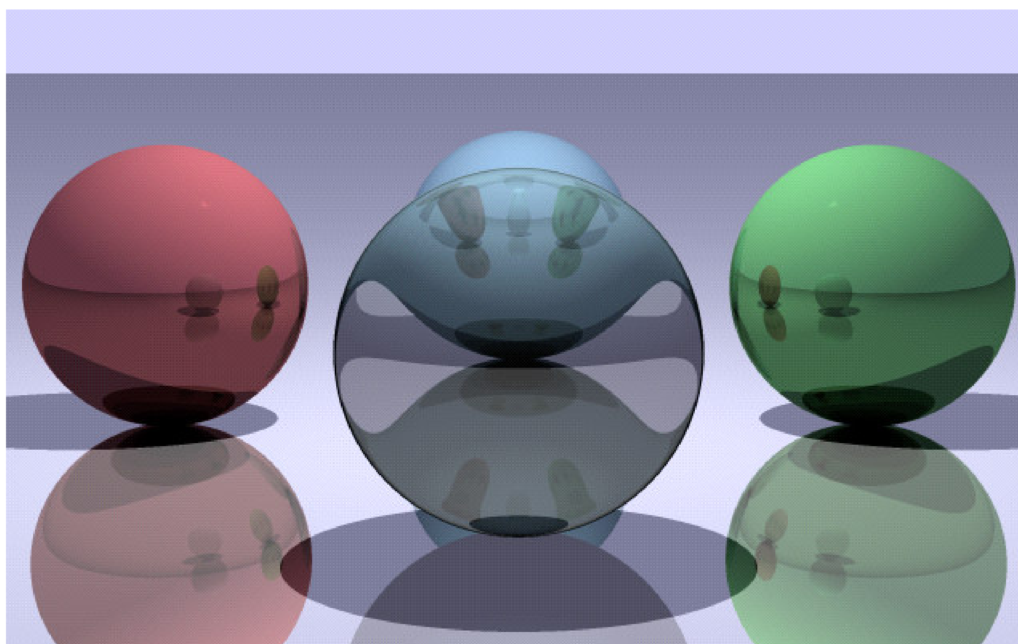
Efekt działania kodu powyżej, czyli "O nie, znowu to samo"

No, nie całkiem - w końcu perfidnie oszkukiwałem, nie pierwszy raz zresztą, używając jeszcze nie omówionego teksturowania. Będzie to jedna z rzeczy którą się zajmiemy w najbliższej przyszłości.

Ale nie od razu - ponieważ przeźroczyste obiekty potrafią generować wręcz wykładniczą ilość promieni (po każdym trafieniu na powierzchnię, mogą powstać dwa promienie), renderowanie ich zajmuje zazwyczaj bardzo dużo czasu. Na razie jedyne co możemy robić przez ten czas to patrzeć się w migający kursor na konsoli albo pójść na kawę, w następnej części zajmiemy się zmuszaniem raytracera do wypluiwania jakichś mądrze wyglądających informacji przez ten czas.

Dodatkowo, wszystkie sceny tutaj pokazane są siłą rzeczy nudne - w końcu ile można pokazać dysponując jedynie kulami i płaszczyznami? Zanim będziemy w stanie stworzyć coś ciekawego, musimy napisać kod który pozwoli nam renderować dowolne modele...

Na zakończenie, ta sama scena co zwykle z dodanym nowym, przeźroczystym obiektem:



Nowy kolega dołączył do naszych ulubionych trzech kul, prezentując efekty załamania światła