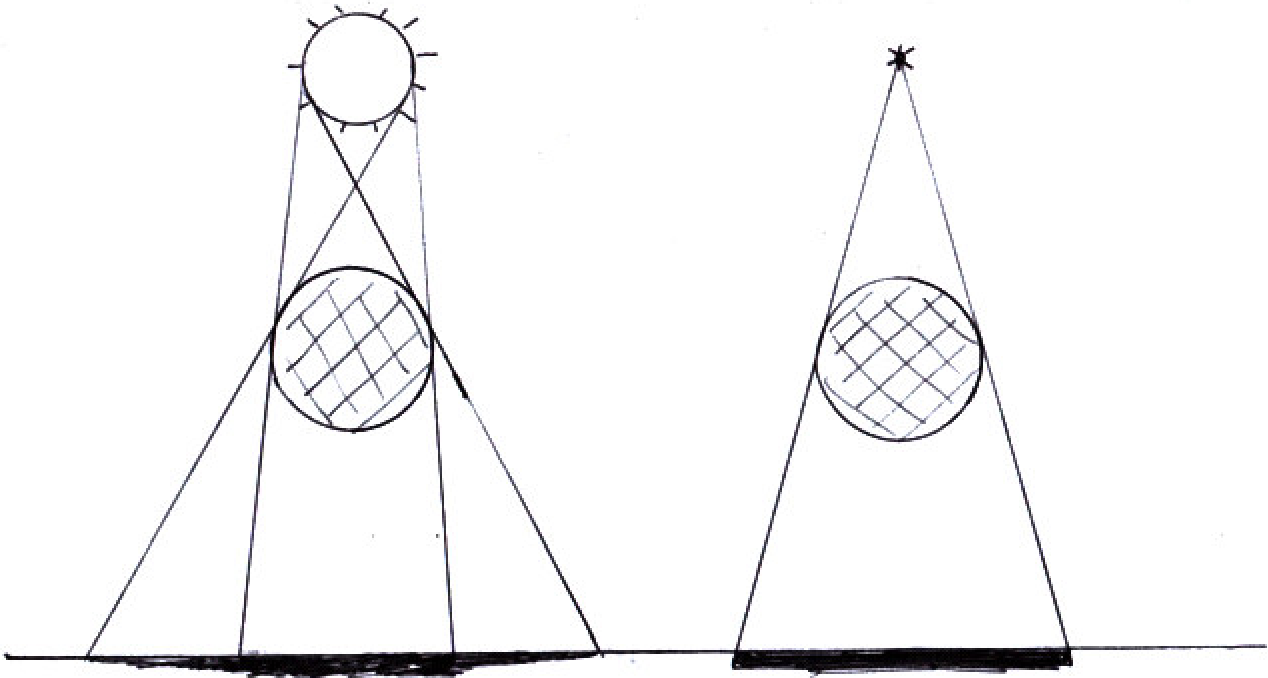
Raytracing: krok po kroku

cz. 10 - soft shading

I. Co/Dlaczego?

Kolejnym elementem który na naszych obrazach jest zawsze nierealistycznie ostry są cienie (ang. *hard shadows*).
Dzieje się tak dlatego że źródła światła w prawdziwym świecie mają pewną niezerową wielkość - a my używamy światła punktowego (*point light*).
W tym artykule zajmiemy się symulacją nieostrych, miękkich cieni (ang. *soft shadows*).



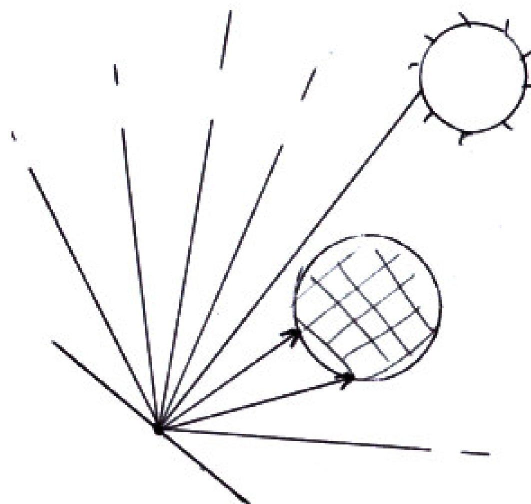
(po lewej - nasz cel, czyli soft shadow, po prawej - to co jest obecnie, czyli hard shadow)

I. Kilka pomysłów

Na świecie nie wszystko można podzielić jednoznacznie podzielić na światło i cień. W momencie kiedy część źródła światła dochodząca do danego punktu jest zasłonięta przez inny obiekt pojawia się półcień.
Niestety, dokładne obliczenie jak dużo światła dociera do obiektu byłoby zadaniem bardzo trudnym, powolnym, oraz niemożliwym do zrealizowania w sposób ogólny dla dowolnego obiektu.
Na szczęście, napisałem `dokładne obliczenie` - nie jesteśmy jeszcze na straconej pozycji. Na ratunek przybywa po raz kolejny sampling...

W jaki sposób wykorzystać sampling do symulowania miękkich cieni?
Możemy przetestować kilka różnych ścieżek którymi światło nadchodzi do obiektu i oszacować na tej podstawie ilość docierającego światła (patrz - ilustracja).

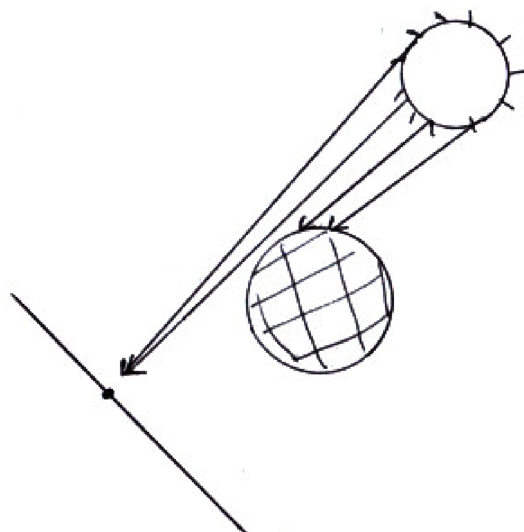
Pierwszym, naiwnym podejściem jest rzucenie promieni jednorodnie we wszystkich kierunkach dookoła i sprawdzanie czy trafiły na źródło światła:



Jeśli ktoś lubi doświadczenia i chciałby przetestować ten sposób w praktyce - należy wziąć kamień (reprezentujący promień) do rąk, zamknąć oczy, obrócić się kilka razy dookoła i rzucić kamieniem. Jeśli kamień trafi w lampę albo inne źródło światła - światło dociera do miejsca gdzie rzucający stoi. Jeśli kamień trafi w przeszkodę, jak ścianę, wiemy że z tego kierunki nie dociera światło. Należy eksperyment powtórzyć wielokrotnie - im więcej razy kamień trafi w światło na, powiedzmy, sto rzutów, w tym jaśniejszym stoimy miejscu (więcej lamp było dookoła).

Taki sposób jest bardzo niewydajny -w przypadku na przykład żarówki znajdującej się w pewnej odległości, szansa na to że losowo skierowany promień (rzucony kamień) z odległego punktu w nią trafi jest bardzo mała - większość sceny będzie kompletnie czarna, ponieważ najczęściej nie uda nam się trafić ani jednym promieniem w źródło światła.

Na szczęście jest lepszy sposób - możemy odwrócić sposób myślenia i śledzić promienie od światła w kierunku cieniowanego punktu:



Taki sposób jest dużo wydajniejszy.
Jeśli przyjąć znowu porównanie promienia do kamienia, to tak jakby lampa w pokoju nagle zyskała świadomość i zaczęła rzucać w nas kamieniami. Już po kilku rzuconych kamieniach poszkodowany możne dość trafnie określić jak bardzo jest odsłonięty przed rozwścieczoną lampą.

Ponieważ i tak wykonujemy antyaliasing, problem jeszcze się upraszcza - dla każdego piksela tak czy inaczej testujemy wiele promieni, więc obliczanie stosunku `*trafiających*` i `*zablokowanych*` promieni dostajemy za darmo (jeśli na przykład do piksela, przy użyciu soft shadingu, dojdzie 12 promieni a 4 zostaną zablokowane przez inny obiekt, suma wchodzącego światła wyniesie 3/4 pełnej jasności - ponieważ 12 promieni liczymy normalnie, a 4 promienie pomijamy).

Czyli tak - nasz dotychczasowy (przedstawiony w poprzednich artykułach) sposób postępowania można podsumować tak:

1. Sprawdź czy pomiędzy **światłem** a cieniowanym punktem jest przeszkoda
2. Jeśli nie, wkład światła wynosi zero. Jeśli tak, licz dalej normalnie.

A dodając miękkie cienie, będziemy działać tak:

1. **Wybierz punkt na powierzchni światła**
2. Sprawdź czy pomiędzy **punktem na powierzchni światła** a cieniowanym punktem jest przeszkoda
3. Jeśli nie, wkład światła wynosi zero. Jeśli tak, licz dalej normalnie.

Wygląda wystarczająco prosto...

II. Implementacja

Zaczniemy od stworzenia od nowa klasy światła - powinna ona, podobnie ja poprzednia, przechowywać kolor, oraz dodatkowo udostępniać metodę Sample() zwracającą punkt leżący na powierzchni źródła światła.
Uprościmy sobię trochę pisanie, i zaimplementujemy jedynie kuliste źródła światła, ponieważ oszczędzi nam to tworzenia dodatkowego interfejsu i klas, a strata w większości przypadków będzie zerowa.

Za pomocą tej jednej klasy będziemy symulować jednocześnie światło obszarowe (ang. *area light*) i światło punktowe (jako światło obszarowe z zerowym promieniem) - nazwiemy ją więc po prostu *Light*.

Tak więc musimy znaleźć sposób generowania punktów na powierzchni kuli - wykorzystamy do tego metodę zaprezentowaną na przykład na tej stronie:

http://www.altdevblogaday.com/2012/05/03/generating-uniformly-distributed-points-on-sphere/

Jeśli kogoś interesuje w jaki sposób został wyprowadzony ten wzór (żeby nie dublować materiału, obliczenia nie są tutaj kopiowane), może przeanalizować to:

http://repository.upenn.edu/cgi/viewcontent.cgi?article=1188&context=cis_reports

Kroki wykonywane w tym algorytmie można, jak zaprezentowano w pierwszym liknu, podsumować następująco:

1. Wybrać współrzędną z z zakresu $[-1;1]$
2. Wybrać współczynnik t z zakresu $[0;2\pi]$
3. $r = \sqrt{1 - z^2}$
4. $x = r * \cos(t)$
5. $y = r * \sin(t)$

Mapowanie sampli na powierzchnię kuli, korzystając z przedstawionego algorytmu, realizowane jest taką oto prostą funkcją:

```
Vector3 RemapSampleToUnitSphere(Vector2 sample)
{
    double z = 2 * sample.X - 1;
    double t = 2 * Math.PI * sample.Y;
    double r = Math.Sqrt(1 - z * z);
    return new Vector3(
        r * Math.Cos(t),
        r * Math.Sin(t),
        z);
```

```
}
```

I... to właściwie tyle jesli chodzi o pisanie klasy światła. Możemy spokojnie dopisać resztę, czyli konstruktory i obsługę przypadku światła punktowego:

```
class Light
{
    Vector3 center;
    double radius;
    Sampler sampler;
    ColorRgb color;

    // Konstruktor udający światło punktowe
    public Light(Vector3 position, ColorRgb color)
        : this(position, color, null, 0) { }

    // pełny konstruktor
    public Light(Vector3 center, ColorRgb color, Sampler sampler, double radius)
    {
        this.center = center;
        this.radius = radius;
        this.sampler = sampler;
        this.color = color;
    }

    public Vector3 Sample()
    {
        if (radius == 0) { return center; } // symulowanie światła punktowego

        var sample = sampler.Single();
        return center + RemapSampleToUnitSphere(sample) * radius;
    }

    Vector3 RemapSampleToUnitSphere(Vector2 sample)
    {
        double z = 2 * sample.X - 1;
        double t = 2 * Math.PI * sample.Y;
        double r = Math.Sqrt(1 - z * z);
        return new Vector3(
            r * Math.Cos(t),
            r * Math.Sin(t),
            z);
    }

    public ColorRgb Color { get { return color; } }
}
```

III. Łączenie w całość

Teraz pozostaje dostosować resztę systemu tak żeby działała z nowym światłem. W tym celu musimy zmienić wszystkie wystąpienia poprzedniej, przestarzałej już klasy PointLight na nową klasę Light.

Nie jest ich aż tak dużo jak mogłoby się wydawać. Zaczynając od klasy World, pojawia się ona w liście świateł na świecie:

```
List<PointLight> lights;
```

Co zamieniamy, razem z operującymi na niej metodami, na

```
List<Light> lights;
```

Dalej, musimy zmienić fragment w klasie Phong (i analogicznie w PerfectDiffuse) w związku z tym że znikła nam właściwość Position. Tak więc zmieniamy ten kod:

```
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; }
```

Musimy pobrać pozycję światła za pomocą metody .Sample() i użyć jej zamiast light.Position:

```
Vector3 lightPos = light.Sample();
Vector3 inDirection = (lightPos - hit.HitPoint).Normalized;
double diffuseFactor = inDirection.Dot(hit.Normal);

if (diffuseFactor < 0) { continue; }

if (hit.World.AnyObstacleBetween(hit.HitPoint, lightPos))
{ continue; }
```

IV. Wyniki

Jak zwykle, pod koniec wypada zaprezentować działanie kodu przedstawionego w artykule.

Żeby użyć nowego, obszarowego światła w renderowanej scenie należy wykonać dwa kroki:

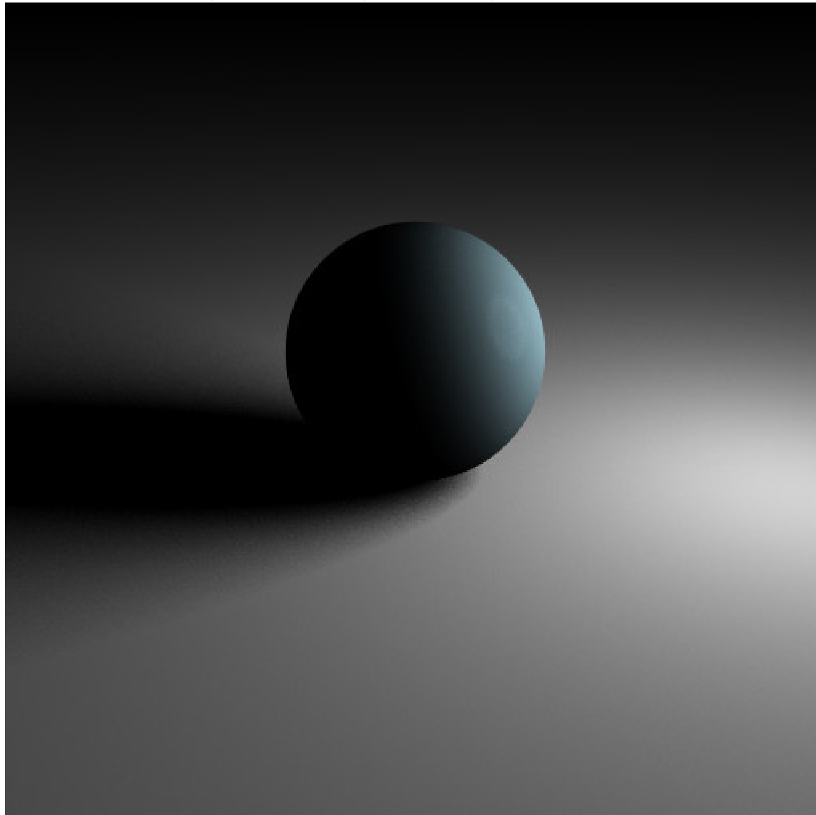
1) Stworzyć sampler dla światła:

```
Sampler areaLightSampler = new Sampler(
    new Jittered(0),          // generator
    new SquareDistributor(),// dystrybutor
    SampleCt,                 // ilość sampli
    97);                      // ilość zestawów sampli
```

2) Stworzyć samo światło:

```
world.AddLight(new Light(
    new Vector3(6, 2, 0),     // pozycja światła
    Color.White,              // kolor światła
    areaLightSampler,         // sampler
    2));                      // promień
```

Im większy promień światła tym bardziej rozmyte krawędzie cienia - za to przy wielkości światła równej zero, otrzymujemy zupełnie ostre cienie, identyczne do dotychczasowych:



Pozostaje dodać wariację na temat jedynej słusznej sceny z trzema kulami, tym razem w trochę 'mroczniejszym' nastroju: