Raytracing: krok po kroku

cz. 8- sampling i antyaliasing

I. Co/Dlaczego?

W tej części zajmiemy się głównie omawianiem samplingu - czyli generowania losowych punktów, tzw. sampli (określanych również jako próbki) w wymaganym zakresie i w danym rozkładzie (na przykład na powierzchni dysku). Sampling często pojawia się przy raytracingu w miejscach gdzie dokładne wyliczenie jakiejś wartości jest niemożliwe/bardzo trudne - możemy wtedy przybliżyć wynik wykonując kilka prób za pomocą różnych sampli i mając nadzieję że oszacowanie jest dobre. Konkretne przykłady zastosowań to:

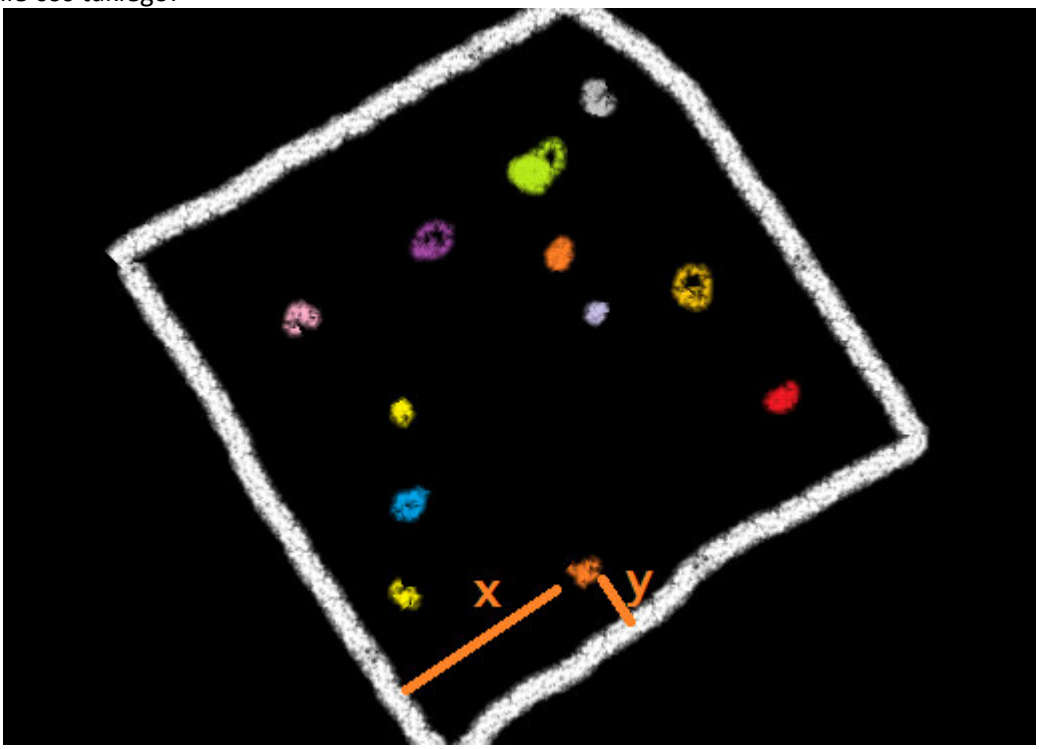
- Antyaliasing (ta część)
- Depth of field
- Area lights + Soft shadows
- Global illumination
- Glossy reflection

Inaczej mówiąc, jeśli chcemy uciec od klinicznie czystych i ostrych jak brzytwa obrazów - potrzebujemy samplingu.

II. Architektura

Czym właściwie jest sampling? Żeby to wytłumaczyć, wyobraźmy sobie na chwilę że jesteśmy samplerem (sampler, czyli coś tworzącego sample).

Bierzemy więc kredę i rysujemy na chodniku kwadrat o rozmiarach 1 x 1 metra. Następnie bierzemy drugą kredę, na przykład kolorową i rzucamy z całej siły w nasze dzieło. Powtarzamy kilka razy. Patrząc pod nogi, zobaczymy prawdopodobnie coś takiego:



Gratulacje! W pewnym sensie, ślady które zostawiła na ziemi kreda to nasze pierwsze wylosowane sample. Odległość próbek od krawędzi tego kwadratu to ich współrzędne. Tak właśnie należy rozumieć sampling - mamy kwadrat (tak naprawdę może to być dowolna inna powierzchnia) i wybieramy punkty leżące na nim.

Żeby osiągnąć taki (pozornie) prosty efekt, będziemy musieli napisać trochę kodu. Klasy które napiszemy będą się dzielić na dwie rodziny - generatory i dystrybutory.

Generatory - Od generatorów sampli będziemy wymagali umiejętności wygenerowania `count` punktów na powierzchni jednostkowego kwadratu.

Interfejs dla generatora zapiszemy w taki sposób:

```
interface ISampleGenerator
{
    Vector2[] Sample(int count);
}
```

Zaimplementujemy kilka generatorów sampli (w kolejności zgodnej z poziomem skomplikowania) prezentujących różne podejścia do losowania i porównamy ich właściwości. W przyszłości nie będziemy potrzebować wcale wszystkich jednocześnie - wystarczy nawet jeden co jest pewnym wyjściem dla chcących uniknąć pisania.

Ostatnia uwaga - dotychczas renderowane dla tej samej sceny obrazy były sobie równe co do piksela (jeśli kilka razy uruchomimy raytracer, obraz wynikowy będzie dokładnie taki sam). Jest to bardzo przyjemna własność z którą szkoda się rozstawać (może być pomocna chociażby w przypadku szukania błędu w kodzie) - dlatego będziemy przyjmować seed w konstruktorach generatorów *(notatka: generatory liczb pseudolosowych, zgodnie z nazwą, generują liczby w sposób zupełnie przewidywalny, zależny od ziarna (seed) - inaczej mówiąc, jeśli stworzymy generator z seedem, powiedzmy, zero to jesteśmy w stanie z góry powiedzieć ze 100% pewnością jakie wartości pojawią się w następnych `losowaniach`)*. Nie jest to oczywiście konieczne, ale to dość sensowne podejście.

Dystrybutory - Zadaniem dystrybutorów jest odpowiednie rozprowadzenie sampli - samplery generują punkty na powierzchni kwadratu, ale czasami potrzebne są punkty na powierzchni na przykład koła (albo półkuli która ostatecznie nie załapała się do tego artykułu).
Dystrybutor zmienia położenie sampli tak, żeby mieściły się w wymaganym zakresie:

```
interface ISampleDistributor
{
    Vector2 MapSample(Vector2 sample);
}
```

III. Rozkład

Jakie własności powinien mieć dobry generator sampli?

I. Sample powinny być mniej-więcej jednorodnie rozłożone w całym zakresie.
II. Jeśli rzutujemy sample na oś X lub Y, wyniki również powinny być dobrze rozłożone.
III. Między samplami powinna być pewna minimalna odległość

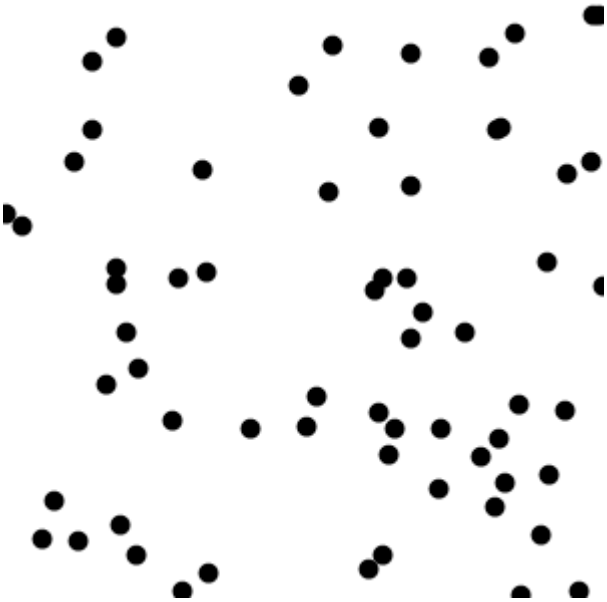
Dlaczego? Ponieważ takie zasady zapewniają dobry rozkład próbek. W uproszczeniu - dobry rozkład zapewnia estetycznie wyglądające obrazy, zły rozkład powoduje widoczny szum. Tak naprawdę każdy rozkład dąży do 'idealnego' obrazu, ale inna sprawa jak szybko dąży - 1000 całkowicie losowych próbek często daje efekty porównywalne lub gorsze od 100 lepiej rozprowadzonych - a czas renderowania jest w pierwszym przypadku 10 razy większy...

Wszystko stanie się być może bardziej jasne za chwilę, kiedy porównamy wyniki działania różnych generatorów i wytkniemy im nieprzestrzeganie tych zasad.

PureRandom: Pierwszym generatorem który napiszemy będzie PureRandom - jak sama nazwa wskazuje, wszystkie sample będą w 100% losowe. Implementacja: wystarczy użyć Random.Next() wymaganą ilość razy.

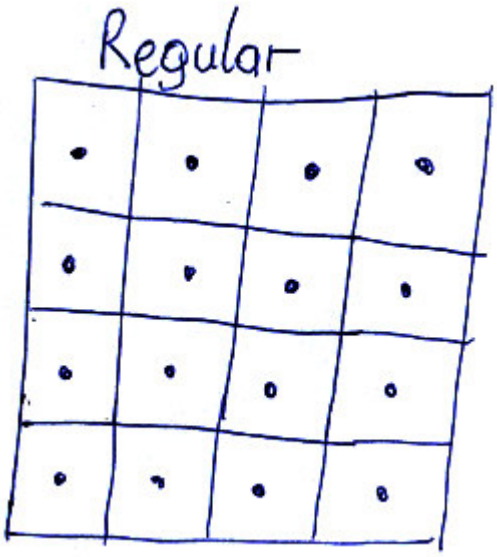
```
class PureRandom : ISampleGenerator
{
    Random r;

    public PureRandom(int seed)
    {
        this.r = new Random(seed);
    }

    public Vector2[] Sample(int sampleCt)
    {
        Vector2[] samples = new Vector2[sampleCt];

        for (int i = 0; i < sampleCt; i++)
        {
            samples[i] = new Vector2(r.NextDouble(), r.NextDouble());
        }

        return samples;
    }
}
```

Nie produkuje on zadowalających wyników (wynik dla 64 sampli):



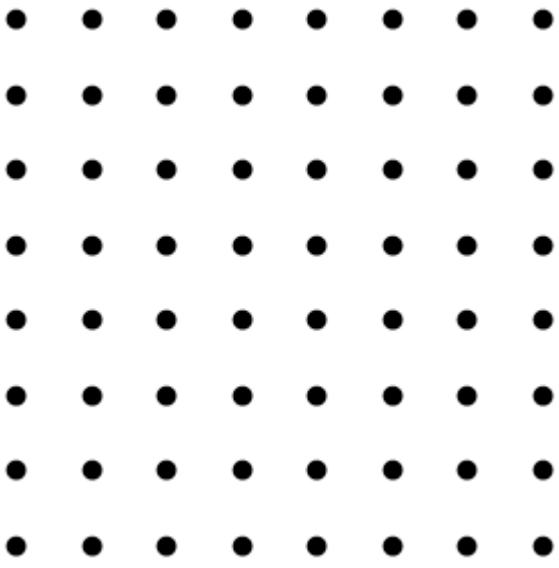
- Zalety: Prosty do napisania
- Wady: Nie mamy żadnej gwarancji na poprawność rozkładu sampli (właściwie to w większości przypadków, jak powyżej, rozkład jest zupełnie niesatysfakcjonujący). Najczęściej sample koncentrują się w jednych obszarach, a inne pozostają niezajęte.
- Werdykt: Nieszczególnie nadaje się do raytracingu - różne techniki powstały po to, żeby dostarczyć lepszego sposobu na generowanie sampli.

Regular: Kolejnym generatorem, bardzo charakterystycznym zresztą, jest Regular - jest jedynym kompletnie deterministycznym samplerem (nie korzysta z (pseudo)losowości w żaden sposób). Dzieli on dostępne kwadratowe pole na *szachownicę* i wstawia po jednej próbce w środek każdego pola. Niestety, ilość sampli musi być w tym przypadku kwadratem liczby naturalnej - zapewnimy to nie do końca uczciwie, poprzez zaokrąglanie wymaganej ilości sampli w dół (inną, chyba jeszcze gorszą, opcją jest rzucanie wyjątku albo przyjmowanie w konstruktorze szerokości szachownicy).




```
class Regular : ISampleGenerator
{
    public Vector2[] Sample(int sampleCt)
    {
        int sampleRow = (int)Math.Sqrt(sampleCt);
        Vector2[] result = new Vector2[sampleRow * sampleRow];

        for (int x = 0; x < sampleRow; x++)
            for (int y = 0; y < sampleRow; y++)
            {
                double fracX = (x + 0.5) / sampleRow;
                double fracY = (y + 0.5) / sampleRow;

                result[x * sampleRow + y] = new Vector2(fracX, fracY);
            }

        return result;
    }
}
```

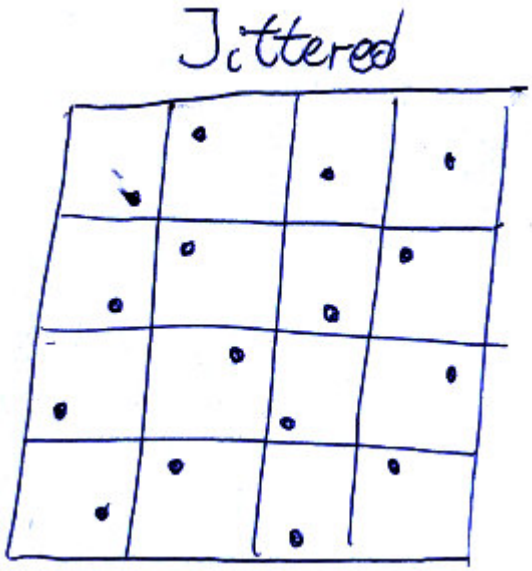
Rozkład dla 64 sampli wygląda tak:



- Zalety: Prosta do napisania, deterministyczna, zawsze dobrze spełnione są zasady 1 (jednorodność) i 3 (minimalna odległość).
- Wady: Zasada 2 (jednorodność po rzutowaniu na osie) nigdy nie spełniona (właściwie to bardziej złamać się jej nie da - sample ułożone w równych liniach na osi X i Y), dodatkowo ilość sampli musi być kwadratem liczby naturalnej. Ale przede wszystkim - tą metodą nie można wygenerować różnych zestawów sampli - wszystkie są identyczne. Jako że to zestawy sampli (o których później) w większości zastosowań ratują wygląd obrazów, ten sposób jest najczęściej niedopuszczalny.
- Werdykt: W niektórych przypadkach jest OK, zazwyczaj się nie nadaje.

Jittered: Pierwszym metodycznym podejściem do samplingu jest Jittered - dzieli on, podobnie jak regular, powierzchnię na szachownicę ale każda próbka jest umieszczana w losowym miejscu wewnątrz swojego pola (a nie zawsze na środku).

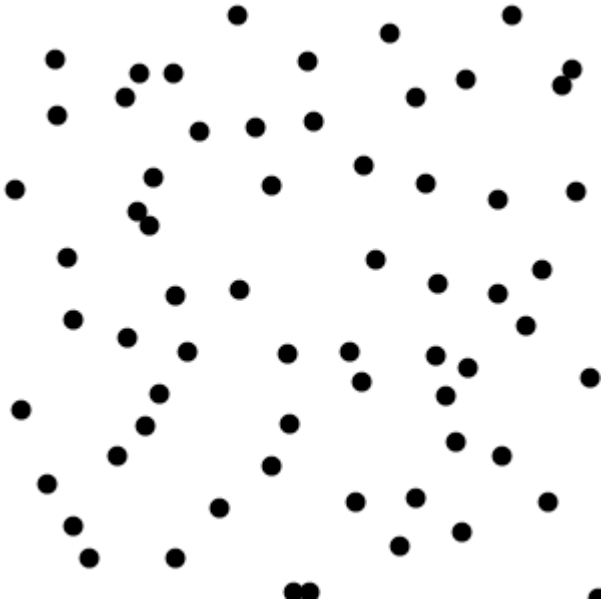
Zasada działania:



Kod:

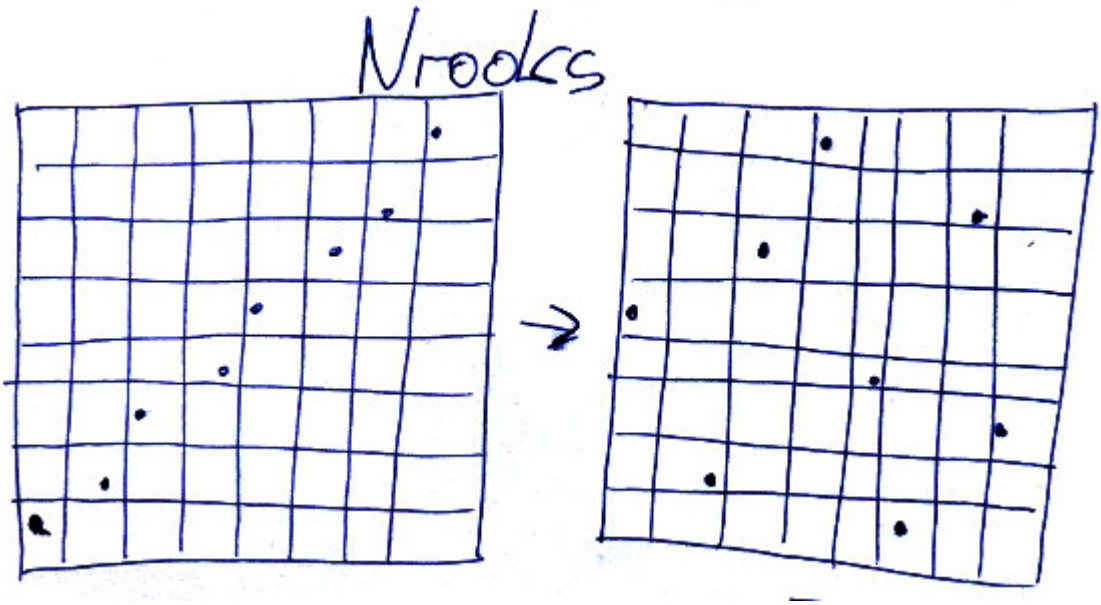
```
class Jittered : ISampleGenerator
{
    Random r;

    public Jittered(int sampleCt, int seed)
    {
        this.r = new Random(seed);
    }

    public Vector2[] Sample(int count)
    {
        int sampleRow = (int)Math.Sqrt(count);

        Vector2[] result = new Vector2[sampleRow * sampleRow];

        for (int x = 0; x < sampleRow; x++)
            for (int y = 0; y < sampleRow; y++)
            {
                double fracX = (x + r.NextDouble()) / sampleRow;
                double fracY = (y + r.NextDouble()) / sampleRow;

                result[x * sampleRow + y] = new Vector2(fracX, fracY);
            }

        return result;
    }
}
```

Rozkład dla 64 sampli:



- Zalety: Dalej dość prosta do napisania, nieźle sprawuje się w praktyce, zasada 1 (jednorodność) jest spełniona.
- Wady: Nie mamy pewności co do spełnienia reguł 2 (jednorodność na osiach) i 3 (minimalna odległość), ilość sampli musi być kwadratem liczby naturalnej.
- Werdykt: Całkiem niezły wbrew pozorom, przez długi czas był używany jako najlepsza istniejąca technika. Powstały różne próby ulepszenia tej techniki jak HalfJittered (który losuje punkty na obszarze połowy dostępnego pola) i MultiJittered.

NRooks: Innym podejściem do zapewniania dobrego rozkładu jest sampler NRooks. Działa trochę inaczej niż poprzednie generatory - najpierw tworzy regularne punkty na przekątnej kwadratu (co zapewnia (2) jednorodność na osiach) a następnie miesza ich współrzędne ze sobą. Nie ma tu żadnych dodatkowych, ukrytych warunków na ilość sampli.

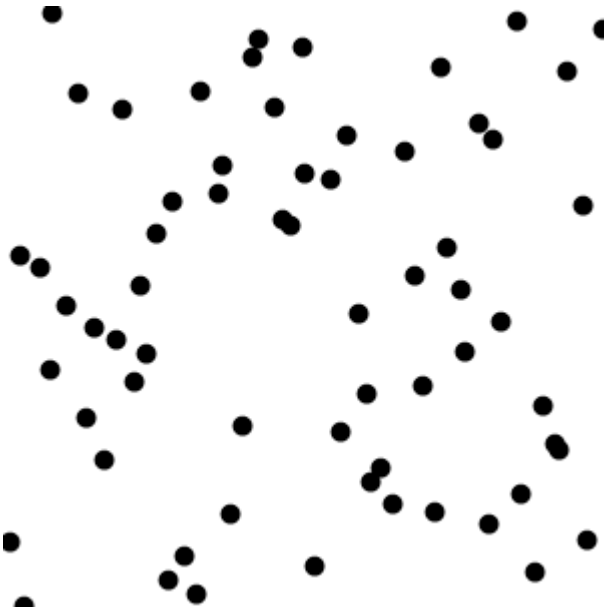


Kod:

```
class NRooks : ISampleGenerator
{
    Random r;

    public NRooks(int seed)
    {
        this.r = new Random(seed);
    }

    public Vector2[] Sample(int sampleCt)
    {
```

```
        Vector2[] samples = new Vector2[sampleCt];

        for (int i = 0; i < sampleCt; i++)
        {
            samples[i] = new Vector2(
                (i + r.NextDouble()) / sampleCt,
                (i + r.NextDouble()) / sampleCt);
        }

        ShuffleX(samples, sampleCt);

        return samples;
    }

    void ShuffleX(Vector2[] samples, int sampleCt)
    {
        for (int i = 0; i < sampleCt - 1; i++)
        {
            int target = r.Next() % sampleCt;
            double temp = samples[i].X;
            samples[i].X = samples[target].X;
            samples[target].X = temp;
        }
    }
}
```

Rozkład dla 64 sampli:



- Zalety: Proste, zasada 2 (jednorodność) jest spełniona, dowolna ilość sampli (w przeciwieństwie do Jittered i Regular) obsługiwana.
- Wady: Pozostałe wymagania kompletnie leżą i nie jest z nimi lepiej niż dla FullRandom...
- Werdykt: Poprawia kilka wad poprzedniego samplera, ale robiąc ten krok naprzód cofa się o dwa. W praktyce prawie zawsze zachowuje się gorzej niż poprzednik.

Co dalej?: Sample można generować na bardzo wiele sposobów, z których z braku miejsca omówiliśmy tylko kilka, wybierając najprostsze do zrozumienia oraz dające (ostatnie dwa) dość dobre efekty w praktyce.

W razie potrzeby dalszej poprawy jakości, jedną z najlepszych obecnie stosowanych metod samplingu jest MultiJittering, będący właściwie połączeniem Jitteringu i NRooks. Został on przedstawiony przez Petera Shirleya w Graphics Gems IV.

Oryginalny artykuł można znaleźć na przykład tutaj:
http://www.google.pl/books?id=CCqzMm_-WucC&lpg=PA370&ots=msjr33JFae&pg=PA370#v=onepage

IV. Dystrybucja na kwadracie

Jak było wspomniane we wstępie, klasy rozprowadzające próbki będą u nas implementowały ogólny interfejs ISampleDistrbutor:

```
interface ISampleDistributor
{
    Vector2 MapSample(Vector2 sample);
}
```

Dystrybucja na kwadracie to rzecz z naszej perspektywy bardzo prosta - sample są już generowane na powierzchni kwadratu, nie musimy ich w żaden sposób modyfikować.

```
class SquareDistributor : ISampleDistributor
{
    public Vector2 MapSample(Vector2 sample)
    { return sample; }
}
```

Ale brakuje nam jeszcze jednego - potrafimy już tworzyć takie sample jakich potrzebujemy, ale nie mamy gdzie ich przechowywać... Generowanie próbek może zająć chwilę czasu (na pewno tworzenie całego zestawu nie zajmie więcej niż ułamek sekundy, ale robienie tego dla każdego piksela mogłoby znacznie spowolnić renderowanie), więc idealnie byłoby stworzyć je raz - na początku działania programu.
Pojawia się tu nowy problem - jeśli dla wszystkich pikseli zastosujemy takie same próbki, szum który powstanie na wyrenderowanym obrazie (a w praktycznych warunkach zawsze jakiś powstaje) będzie dużo bardziej widoczny (intuicyjnie, jeśli dla pewnego zestawu próbek dokładny kolor piksela jest źle przybliżony, to istnieje duża szansa że dla sąsiadujących pikseli wynik będzie podobnie niedokładny).

Więc musimy przechowywać wiele zestawów sampli. Dodajmy do tego ISampleGenerator, ISampleDistributor oraz wymaganą ilość sampli w zestawie i mamy gotową klasę Sampler zajmującą się zarządzaniem (tworzeniem i wybieraniem kolejnych) próbkamip:

```
class Sampler
{
    Random r;
    List<Vector2[]> sets;
    int sampleNdx;
    int setNdx;

    public Sampler(ISampleGenerator sampler,
        ISampleDistributor mapper,
        int sampleCt,
        int setCt)
    {
        this.sets = new List<Vector2[]>(setCt);
        this.r = new Random(0);
        this.SampleCount = sampleCt;

        for (int i = 0; i < setCt; i++)
        {
            var samples = sampler.Sample(sampleCt);
            var mappedSamples = samples.Select((x) => mapper.MapSample(x)).ToArray();
            sets.Add(mappedSamples);
        }
    }

    public Vector2 Single()
    {
        Vector2 sample = sets[setNdx][sampleNdx];

        sampleNdx++;
        if (sampleNdx >= sets[setNdx].Length)
        { sampleNdx = 0; setNdx = r.Next(sets.Count); }

        return sample;
```

```
    }

    public int SampleCount { get; private set; }
    public int SetCount { get { return sets.Count; } }
}
```

Funkcja Single() ma za zadanie zwracać po każdym wywołaniu kolejną próbkę - będziemy jej stosować jako wygodnego interfejsu do losowania kolejnych sampli.

Uwaga: Została tu użyta funkcja IEnumerable<T>.Select (w samples.Select((x) => ...)), wchodząca w część LINQ. LINQ jest biblioteką (a właściwie trochę częścią języka) wzorowaną na językach funkcyjnych i służącą do obsługi różnych kolekcji i *bardzo* upraszczającą różne operacje na zbiorach danych. Ponieważ znajomość tej biblioteki jest właściwie podstawą operowania na kolekcjach w .NET oraz ponieważ znacznie upraszcza to kod zdecydowałem się na używanie jej w tym i następnych źródłach. Inne języki często zawierają podobne konstrukcje, a jeśli nie - prawie wszystkie funkcje LINQ można zastąpić pojedynczą pętlą for/foreach.

Funkcja rozszerzająca *`Select`* (w większości języków tradycyjnie nazywana *`map`*) jest jedną z najprostszych zawartych tam metod i zwraca kolekcję powstałą po zastosowaniu przekazanego wyrażenia lambda na każdym elemencie wejściowego zbioru - czyli odpowiednikiem

```
List<int> ints = new List<int> { 1, 6, 2, 5, 2 };
List<string> result = ints.Select((x) => "--> " + x.ToString() + " <--").ToList();
```

Jest:

```
List<int> ints = new List<int> { 1, 6, 2, 5, 2 };
List<string> result = new List<string>();
foreach (var i in ints)
{
    result.Add("--> " + i.ToString() + "<--");
}
```

Do użycia tej i innych funkcji potrzeba:

- .C# w wersji 3.5
- using System.Linq; na górze pliku
- dołączyć referencję do System.Core (jeśli nie została dołączona automatycznie przez kompilator)

Niniejszy artykuł nie jest zamierzony jako wprowadzenie do LINQ, ale używane funkcje będą pobieżnie omawiane.

V. Dystrybucja na dysku

Kolejny dystrybutor który zaimplementujemy to DiskDistributor - tutaj musimy już wykonać pewne mapowanie wartości. Podstawową cechą wymaganą od dobrej dystrybucji jest to, że jeśli sample są dobrze rozmieszczona na kwadracie, będą również dobrze rozmieszczone na dysku.
Porównywanie różnych algorytmów mapowania jest moim zdaniem ciekawym zagadnieniem, dlatego zostanie tutaj omówione znacznie więcej sposobów niż wykorzystamy (bo będziemy używać dokładnie jednego) - postąpimy tak tylko dla dysku, przy półkuli zajmiemy się od razu do gotowym rozwiązaniem. Jeśli komuś szkoda czasu może pominąć opisy poglądowe wszystkich sposobów poza ostatnim którego będziemy używać.

```
class DiskDistributor : ISampleDistributor
{
    public Vector2 MapSample(Vector2 sample)
    {
        // ?
    }
}
```

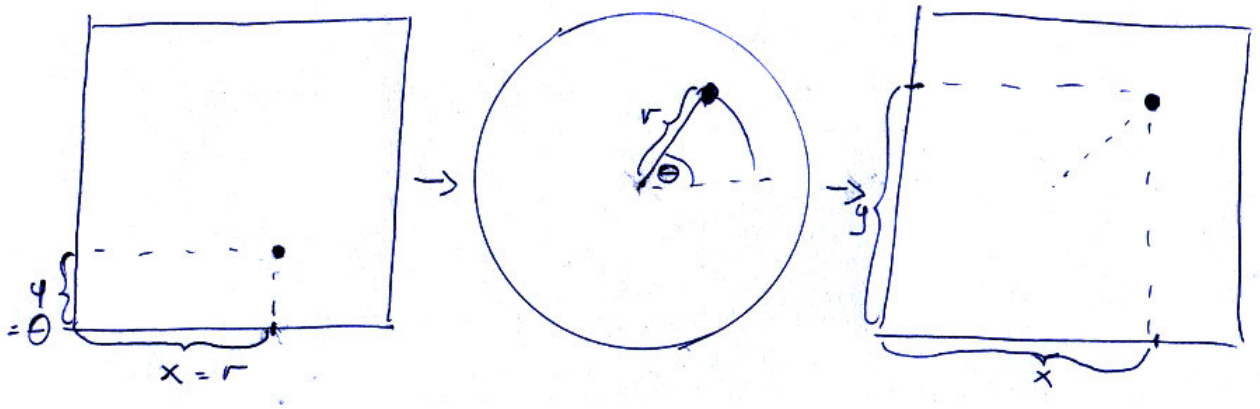
Odrzucanie: Najprostszy sposób - sprawdzamy dla każdego sample czy leży na powierzchni jednostkowego dysku. Jeśli nie - odrzucamy go.

- Zalety: Bardzo prosty do napisania
- Wady: Nie możemy przewidzieć ile sampli nam zostanie! Średnio odrzucimy *n*pi/4* sampli, ale może równie dobrze się okazać że nie odrzucimy żadnego (wszystkie będą dobrze rozłożone), odrzucimy połowę, albo odrzucimy wszystkie!
- Werdykt: Zakopać pod ziemią i zalać betonem.

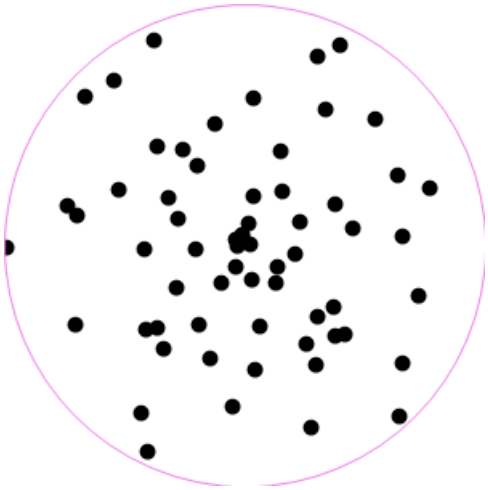
Odrzucanie do skutku: - Modyfikacja poprzedniego sposobu, generujemy i odrzucamy sample aż będziemy mieli ich wystarczającą ilość.

- Zalety: Prosty do napisania
- Wady: Psuje nasz podział odpowiedzialności - do teraz to sampler wiedział ile sampli ma wygenerować, teraz musiałby o tym wiedzieć też dystrybutor. Również psuje rozkład, mimo że jest to mniej widoczne (sample z drugiego generowania mogą trafić dowolnie blisko tych z pierwszego - przykład - rozkład regularny). Ostatecznie, nie mamy gwarancji ile czasu będzie się wykonywał nasz algorytm (złożoność pesymistyczna = $\Theta(+\infty)$).
- Werdykt: Niezbyt elegancki i nie nadaje się do precyzyjnych zastosowań. Przytaczany tutaj tylko jako ciekawostka.

Polar mapping: Potraktowanie pary (x, y) jako (r, θ) (współrzędne polarne, pierwsze to promień hipotetycznego koła a drugie kąt - patrz obrazek), a następnie przeliczenie współrzędnych polarnych z powrotem na kartezjańskie (ważne jest zrozumienie że pierwszy krok to *zmiana sposobu traktowania* - czyli x traktujemy jako r a y jako θ, a drugi krok to *przeliczenie* - czyli x = r cos θ a y = r sin θ) - patrz obrazek:



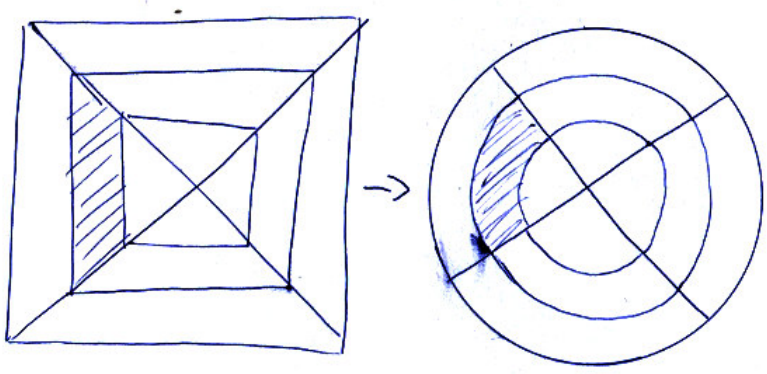

```
Vector2 MapSample(Vector2 sample)
{
    return new Vector2(
        sample.X * Math.Cos(sample.Y * Math.PI),
        sample.X * Math.Sin(sample.Y * Math.PI)));
}
```



- Zalety: Prosty do napisania
- Wady: Mocno niszczy rozkład (nie ma już minimalnych odległości, a większość sampli ląduje w środku dysku).

- Werdykt: Elegancji i bazujący na matematyce - ale to nie wystarczy. Do naszych zastosowań z powodu szatkowania rozkładu się nie nadaje.

Koncentryczna mapa: Technika opracowana i opublikowana przez Petera Shirleya w artykule `A Low Distortion Map Between Disk and Square`. Dla każdego wektora zachowujemy jego kąt od środka kwadratu, ale skalujemy tak, żeby nigdy nie wyszedł poza jednostkowy dysk (patrz - obrazek).

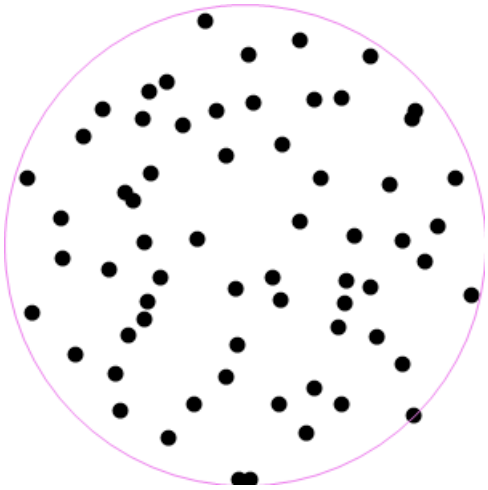


W kodzie rozpatrujemy osobno każdą ćwiartkę wejściowego kwadratu:

Ćwiartka	X i Y	Równania dla r i phi
1	X > Y	r = x
	X > -Y	phi = (pi/4) * (y/x)
2	X < Y	r = y
	X > -Y	phi = (pi/4) * (2 - y/x)
3	X < Y	r = -x
	X < -Y	phi = (pi/4) * (4 + y/x)
4	X > Y	r = -y
	X < -Y	phi = (pi/4) * (6 - y/x)

Kod jest implementacją przekształceń przedstawionych w tabeli:

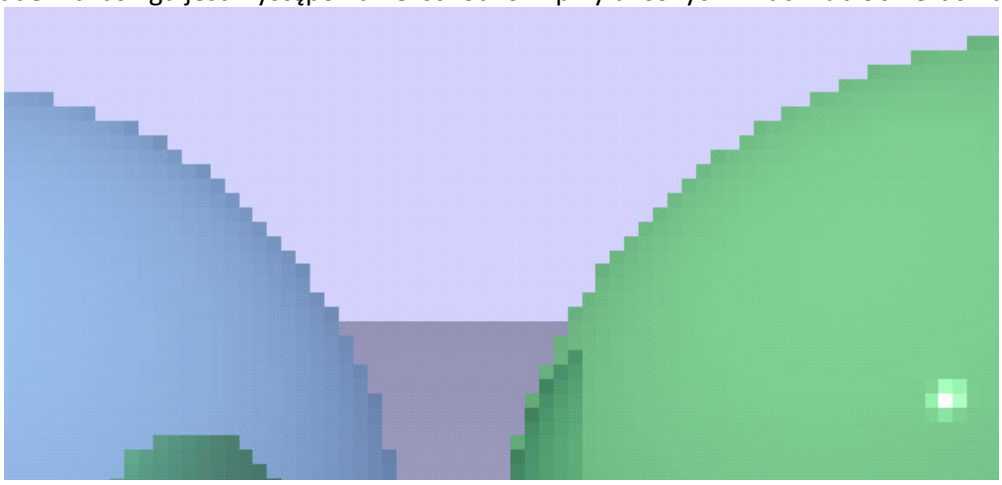
```
public Vector2 MapSample(Vector2 sample)
{
    sample.X = sample.X * 2 - 1;
    sample.Y = sample.Y * 2 - 1;

    double r;
    double phi;

    if (sample.X > -sample.Y)
        if (sample.X > sample.Y) { r = sample.X; phi = sample.Y / sample.X; }
        else { r = sample.Y; phi = 2 - sample.X / sample.Y; }
    else
        if (sample.X < sample.Y) { r = -sample.X; phi = 4 + sample.Y / sample.X; }
        else { r = -sample.Y; phi = 6 - sample.X / sample.Y; }

    if (sample.X == 0 && sample.Y == 0) { phi = 0; }

    phi *= Math.PI / 4;

    return new Vector2(
        r * Math.Cos(phi),
        r * Math.Sin(phi));
}
```



- Zalety: Zachowuje rozkład całkiem nieźle, bliskie sample pozostają bliskie po transformacji.
- Wady: Rozkład nie jest idealny, ale niedokładności są bardzo mało znaczące, dość długa w pisaniu.
- Werdykt: Najlepsza z przedstawionych tutaj technik - właśnie ona będzie używana w dalszych częściach.

VI. Aliasing i Anty-aliasing

Aliasing to niekorzystne zjawisko w grafice powodowane zbyt małą częstotliwością próbkowania podczas renderowania.
Typowym przykładem aliasingu jest występowanie 'schodków' przy ukośnych liniach lub obrzeżach brył:



Do walki z aliasingiem w raytracingu korzysta się z techniki zwanej supersamplingiem - dla każdego piksela testujemy nie jeden ale wiele promieni, każdy w trochę innym kierunku. Brzmi to trochę jak bruteforce (i słusznie), ale jest to jedyny sposób na antyaliasing bez utraty precyzji.

Kod umieścimy w klasie Raytracer (zaczyna się nam ta klasa niebezpiecznie rozrastać - ma już aż trzy niezwiązane za sobą odpowiedzialności. Będziemy się tym musieli wkrótce zająć).

Poprzednio kolor piksela był wyznaczany przez pojedyncze wywołanie funkcji ShadeRay, tym razem będziemy wywoływać tą funkcję wielokrotnie, a kierunek promienia będziemy zmieniać za pomocą sampli. Tak więc zamiast:

```
Vector2 pictureCoordinates = new Vector2(
    ((x + 0.5) / (double)imageSize.Width) * 2 - 1,
    ((y + 0.5) / (double)imageSize.Height) * 2 - 1);

Ray ray = camera.GetRayTo(pictureCoordinates);

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

Podstawiamy:

```
ColorRgb totalColor = ColorRgb.Black;
for (int i = 0; i < sampler.SampleCount; i++)
{
    Vector2 sample = sampler.Single(); // pobierz próbkę

    Vector2 pictureCoordinates = new Vector2( // oblicz kierunek
        ((x + sample.X) / (double)imageSize.Width) * 2 - 1,
        ((y + sample.Y) / (double)imageSize.Height) * 2 - 1);

    Ray ray = camera.GetRayTo(pictureCoordinates);

    totalColor += ShadeRay(world, ray, 0) / (double)sampler.SampleCount;
}

bmp.SetPixel(x, y, StripColor(totalColor));
```

Pozostaje połączyć to z resztą kodu (żeby wykorzystać nowe możliwości potrzebujemy kolejnego parametru, zawierającego zestaw próbek używany do antyaliasingu) - dla kompletności zostaje wklejony cały kod:

```
public Bitmap Raytrace(World world, ICamera camera, Size imageSize)
{
    Sampler singleSample = new Sampler(
        new Regular(), new SquareDistributor(), 1, 1);
    return this.Raytrace(world, camera, imageSize, singleSample);
}

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

    for (int y = 0; y < imageSize.Height; y++)
    {
        for (int x = 0; x < imageSize.Width; x++)
        {
            ColorRgb totalColor = ColorRgb.Black;
            for (int i = 0; i < sampler.SampleCount; i++)
            {
                Vector2 sample = sampler.Single();

                Vector2 pictureCoordinates = new Vector2(
                    ((x + sample.X) / (double)imageSize.Width) * 2 - 1,
                    ((y + sample.Y) / (double)imageSize.Height) * 2 - 1);

                Ray ray = camera.GetRayTo(pictureCoordinates);

                totalColor += ShadeRay(world, ray, 0)
                    / (double)sampler.SampleCount;
            }

            bmp.SetPixel(x, y, StripColor(totalColor));
        }

    }

    return bmp;
}
```

Na koniec, w funkcji Program.Main zmieniamy:

```
Bitmap image = tracer.Raytrace(world, camera, new Size(256, 256));
```

Na:

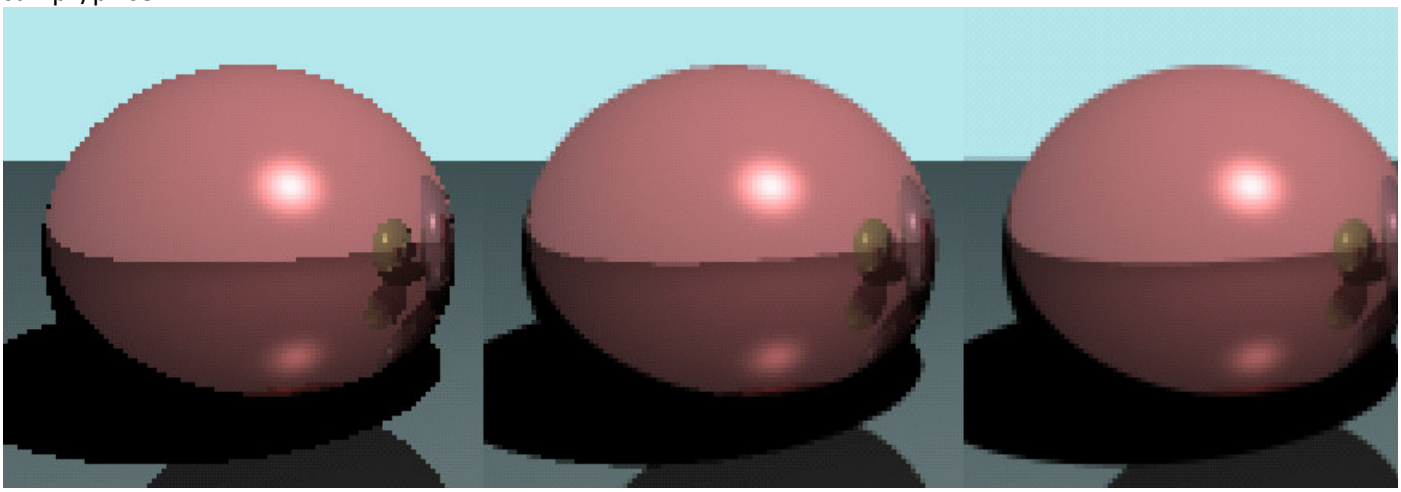
```
const int SampleCt = 9;

// Raytracing!
```

```
Sampler antiAlias = new Sampler(
    new Regular(),
    new SquareDistributor(),
    SampleCt,
    1); // na razie wystarczy jeden set

Bitmap image = tracer.Raytrace(world, camera, new Size(300, 300), antiAlias);
```

I pisanie kodu w tej części skończone.

Im więcej sampli dla piksela używamy tym wolniejszy raytracing (każda czynność musi zostać powtórzona dla każdej próbki). Różnice wydajności potrafią być bardzo duże! Na przykład antyaliasing z 9 próbkami => 9 razy więcej promieni dla każdego piksela => 9 razy wolniejsze renderowanie.

Co daje antyaliasing? Oto jak zmienia się wygląd renderowanej kuli (w trzykrotnym powiększeniu) dla różnych ilości sampli/piksel:



Ilość sampli: lewy obrazek - 1, środkowy - 4, prawy - 64.

Wygląd całej sceny po użyciu antyaliasingu - widać poprawę jeśli porównać to z poprzednim, poszarpanym obrazkiem.