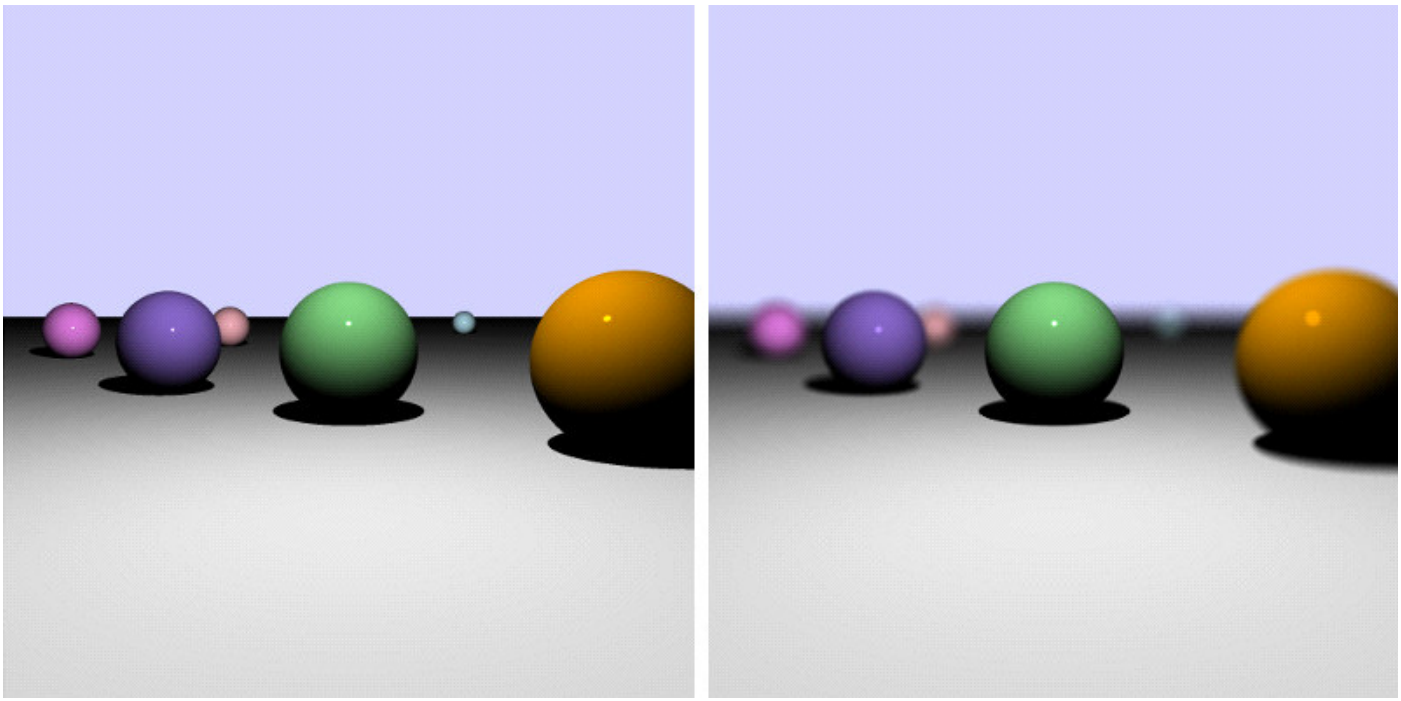
Raytracing: krok po kroku

cz. 9 - depth of field

I. Co/Dlaczego?

Głębia ostrości *(ang. depth of field)* to zakres odległości na których obiekty widziane w kamerze wydają się być wyraźnie widoczne.



Jak widać, na ilustracji po lewej stronie wszystkie obiekty są jednakowo ostre. Na ilustracji po prawej stronie wyraźnie widać jedynie zieloną kulę, a inne są mniej lub bardziej rozmyte - to właśnie głębia ostrości. Jest to zjawisko często wykorzystywane w fotografii dla wyodrębnienia najważniejszych detali na zdjęciu i odwróceniu uwagi obserwatora od tła.

W części drugiej stworzyliśmy aparat otworkowy *(ang. pinhole camera)* który zachowuje się tak jakby całe światło skupiało się w nieskończenie małym punkcie - dlatego wszystkie obiekty wydają się ostre, niezależnie od ich odległości od punktu obserwatora. Prawdziwe aparaty posiadają soczewki, które skupiają światło idealnie tylko na pewnej odległości, zwanej ogniskową *(ang. focal length)*. Posiadają również przysłony, regulujące wielkość otworu przez które światło wpada na materiał światłoczuły.
W tym artykule zajmiemy się tworzeniem kamery za pomocą której będziemy mogli symulować głębię ostrości - nazwiemy ją ThinLens (ponieważ udaje działanie prawdziwej soczewki skupiającej światło).

II. Kilka poprawek

Zanim przejdziemy do zasadniczej części czyli implementacji - musimy wprowadzić kilka poprawek do napisanego wcześniej kodu.

Poprawa interfejsu klasy OrthonormalBasis:
Podczas pisania części drugiej zaszła pewna drobna pomyłka, którą teraz trzeba poprawić - parametry do konstruktora klasy OrthonormalBasis zostały dobrane w sposób który pasował do jednego konkretnego przypadku (tworzenia kamery), ale niekoniecznie do wszystkich - chodzi o `Vector3 eye` i `Vector3 lookAt`:

```
public OrthonormalBasis(Vector3 eye, Vector3 lookAt, Vector3 up)
{
    w = eye - lookAt;
    w = w.Normalized;
    u = Vector3.Cross(up, w);
    u = u.Normalized;
    v = Vector3.Cross(w, u);
}
```

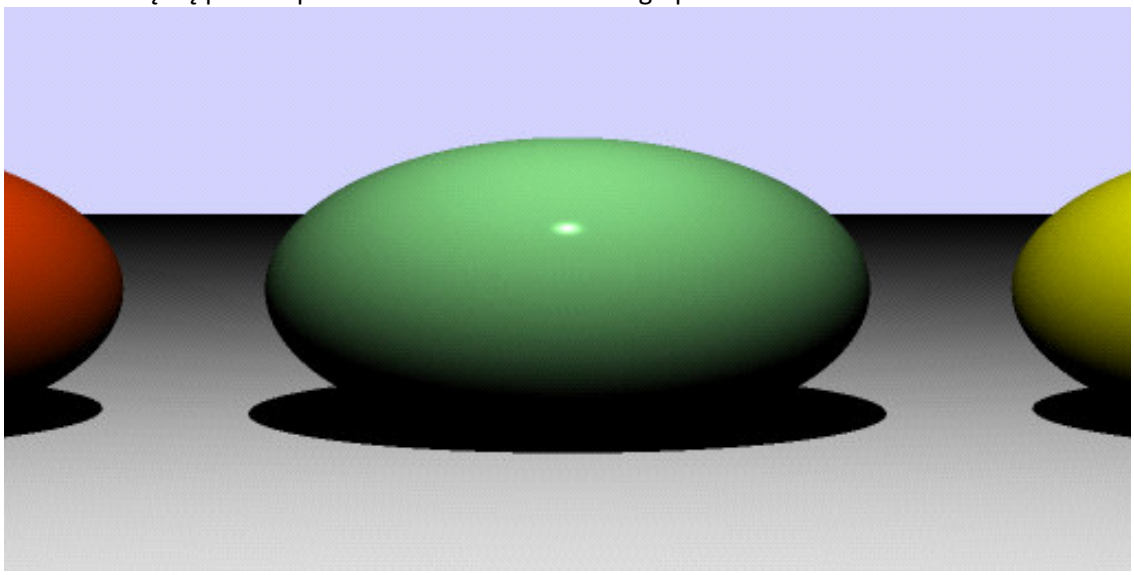
O co chodzi i co można zrobić lepiej?
Otóż `eye` i `lookAt` są wykorzystywane tylko w jednym miejscu, do obliczania wektora `w` (wskazującego `przód` dla lokalnych współrzędnych kamery). Nie tracąc niczego, a zyskując większą elastyczność (w niektórych przypadkach np. definiowanie `lookAt` nie ma sensu) możemy połączyć je w jeden:

```
public OrthonormalBasis(Vector3 front, Vector3 up)
{
    w = front;
    w = w.Normalized;
    u = Vector3.Cross(up, w);
    u = u.Normalized;
    v = Vector3.Cross(w, u);
}
```

Trzeba oczywiście zmienić linijkę w której wywołujemy ten konstruktor (w klasie Pinhole):

```
this.onb = new OrthonormalBasis(origin - lookAt, up);
```

Zoom w kamerze:
Próby wyrenderowania obrazu którego szerokość nie równa się jego wysokości za pomocą klasy Pinhole w jej obecnej postaci skończą się prawdopodobnie stworzeniem takiego potwora:



Jednym z założeń przy tworzeniu kamery była kompletna niezależność od reszty parametrów, jak na przykład rozdzielczość obrazka (naiwne podejście powodowałoby między innymi że renderowanie obrazu w różnych wielkościach zmieniałoby obejmowany obszar).
W tym przypadku, bierzemy obraz obejmujący kwadratowy obszar i `rozciągamy` żeby `upchnąć` go w prostokątnym obrazie - co powoduje widoczne powyżej zniekształcenia.

Mimo wszystko, może wpaść komuś do głowy pomysł wyrenderowania czegoś mającego kształty inne niż jedyny słuszny kwadrat. Na ratunek przychodzi wtedy skala - parametr określający proporcje obrazu oraz jego powiększenie (zoom)

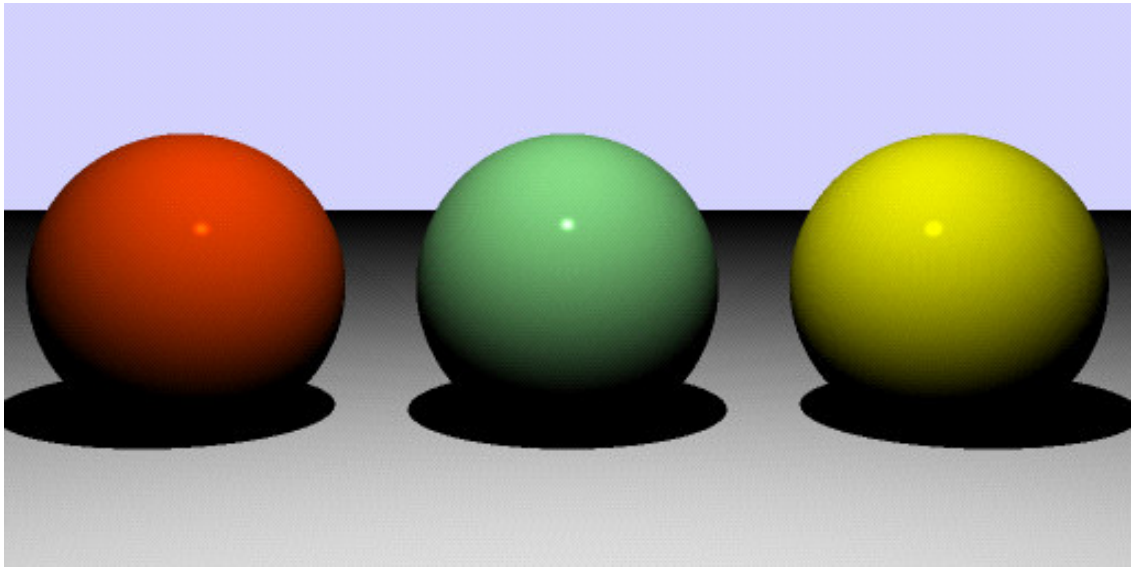
W tym celu, dodajemy kolejny parametr do konstruktora:

```
public Pinhole(Vector3 origin,
    Vector3 lookAt,
    Vector3 up,
    Vector2 scale,
    double distance)
{
    this.onb = new OrthonormalBasis(origin - lookAt, up);
    this.origin = origin;
    this.scale = scale;
    this.distance = distance;
}
```

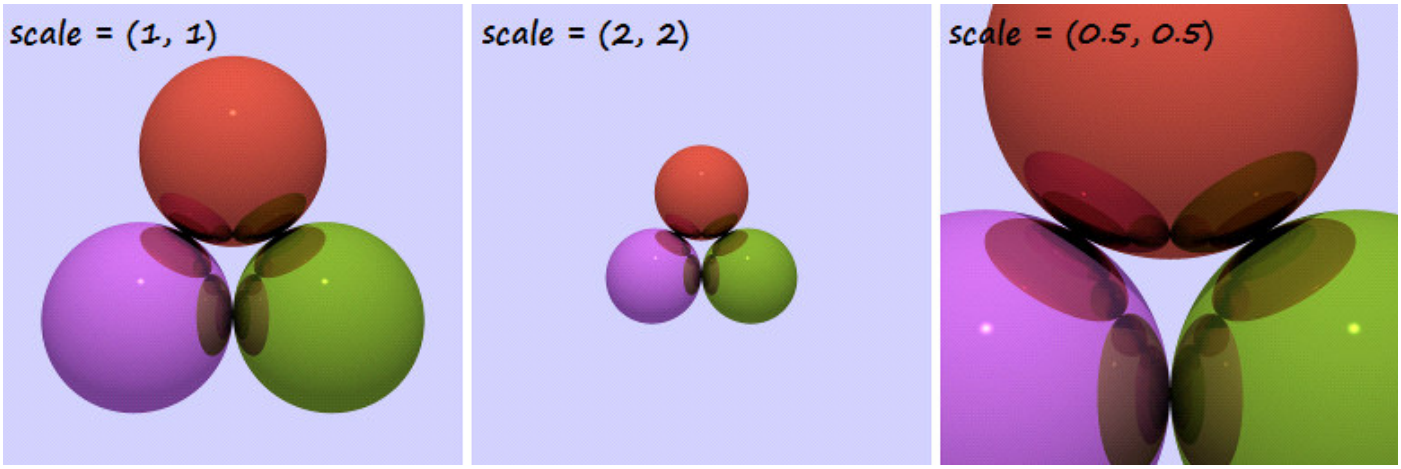
Efekt skalowania możemy uzyskać mnożąc relatywną pozycję każdego rzucanego promienia przez skalę (czyli jeśli skala będzie równa 0.5 maksymalne `wychylenie` promienia będzie dwa razy mniejsze niż dla skali 1 - i konsekwentnie na obrazie zmieści się dwa razy mniej).

```
public Ray GetRayTo(Vector2 relLoc)
{
    Vector2 vpLoc = new Vector2(relLoc.X * scale.X, relLoc.Y * scale.Y);
    return new Ray(origin, RayDirection(vpLoc));
}
```

Gotowe, teraz można renderować obrazy o dowolnym kształcie (ta sama scena co powyżej, scale = (2, 1):



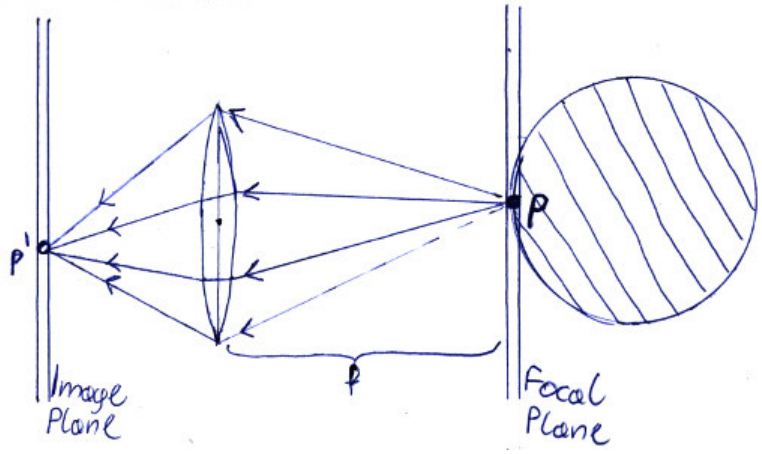
Można też zmieniać obszar obejmowany przez kamerę - być może trochę nieintuicyjnie, ponieważ *zwiększenie* `skali` dwukrotnie powoduje *zmniejszenie* wszystkich obiektów.



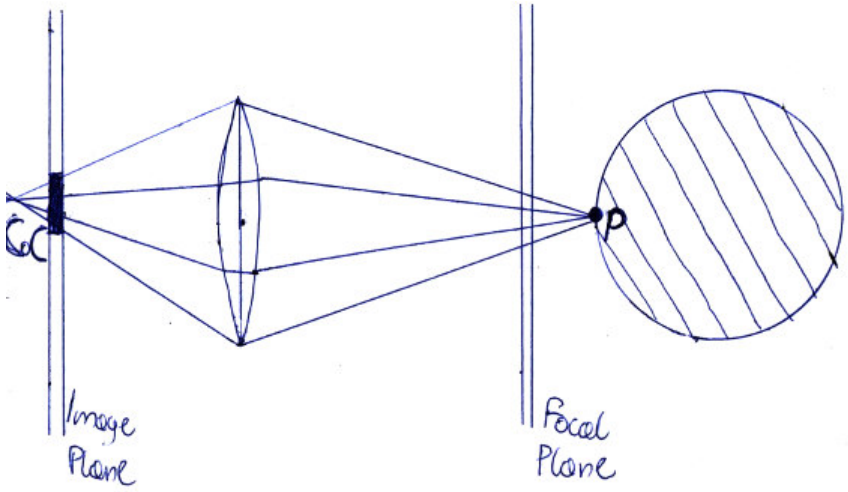
III. Trochę optyki

Będziemy tutaj omawiać kilka właściwości soczewek, więc najpierw załóżmy że bierzemy jedną taką soczewkę i ustawiamy ją w jakimś punkcie w przestrzeni. Teraz wyobraźmy sobie dowolną płaszczyznę prostopadłą do osi optycznej soczewki (w pewnym sensie `równoległą ` do powierzchni soczewki - patrz obrazek) - nazwijmy ją *focal plane* (płaszczyzną odwzorowania obiektywu).

I teraz dzieje się ciekawa rzecz - wszystkie promienie wychodzące z jednego punktu na *focal plane* i przechodzące przez soczewkę przecinają się w jednym punkcie. Co więcej, wszystkie te punkty przecięcia leżą na innej płaszczyźnie, zwanej *image plane* (płaszczyzną obrazu).



Każdy punkt na *focal plane* jest odwzorowywany na dokładnie jeden punkt na *image plane* (i odwrotnie). Co się jednak stanie jeśli weźmiemy punkt nie leżący na płaszczyźnie odwzorowania?



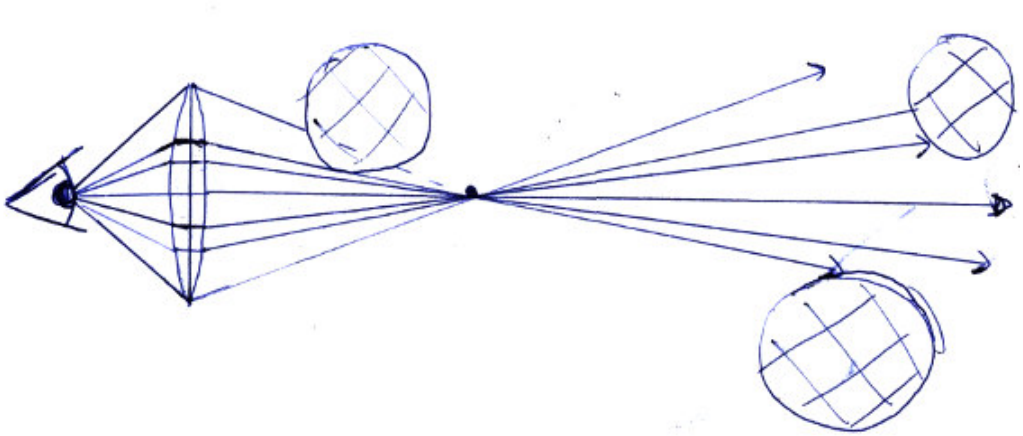
Okazuje się, że teraz promienie wychodzące z jednego punktu na *focal plane* trafiają na różne punkty na *image plane*. Obszar na które padają jest mniej-więcej okrągły i jest zwany *circle of confusion* (krążkiem rozmycia).

No dobrze, ale jaki ma to związek z raytracingiem? Otóż płaszczyzna obrazu jest odpowiednikiem materiału światłoczułego w aparatach fotograficznych oraz obrazu wynikowego w raytracingu. Płaszczyzna odwzorowania jest parametrem soczewki który będziemy mogli ustawiać w dowolny sposób.

Dla naszych zastosowań w raytracingu możemy zdefiniować głębię ostrości jeszcze raz - jest to zakres odległości od soczewki, na których *circle of confusion* jest mniejsze od jednego piksela.

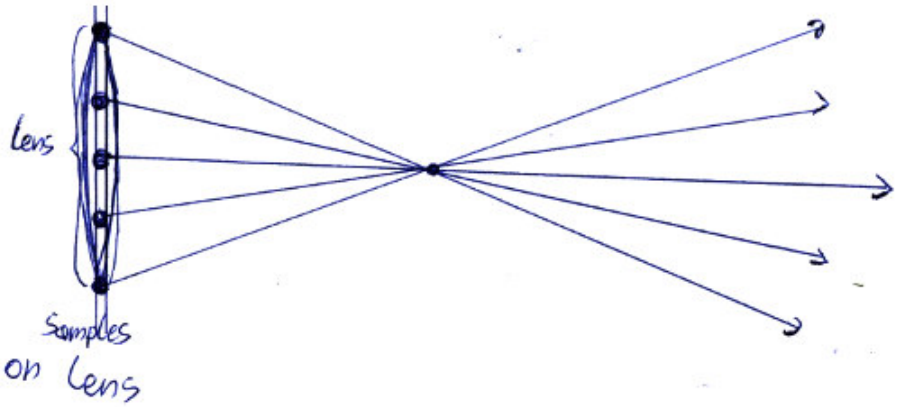
IV. Pomysł na implementację

Tak więc chcemy zamodelować *depth of field* w sposób jak najbardziej zbliżony do przedstawionego powyżej fizycznego modelu. Nie możemy tego niestety zrobić bezpośrednio, ponieważ licząc w ten sposób, dla każdego punktu na *image plane* dochodzi *nieskończona* ilość promieni, z różnych kierunków - a tego sumować niestety nie damy rady - nieco przesadzona ilustracja tego faktu:



Na szczęście, okazuje się że możemy w prosty sposób oszacować ten efekt - za pomocą naszej nowej ulubionej sztuczki, czyli samplingu (tak, nie bez powodu ostatnio się męczyliśmy z jego implementacją).

Będziemy symulować działanie soczewki, śledząc skończoną ilość promieni spośród tych które trafiają do kamery i mając nadzieję, jak zwykle, że wynik jest dokładny:



Punkt początkowy promienia może przesunąć się od początkowego położenia o odległość maksymalnie równą promieniowi soczewki. Za każdym razem przed rzuceniem promienia będziemy losować jego punkt początkowy na powierzchni soczewki.

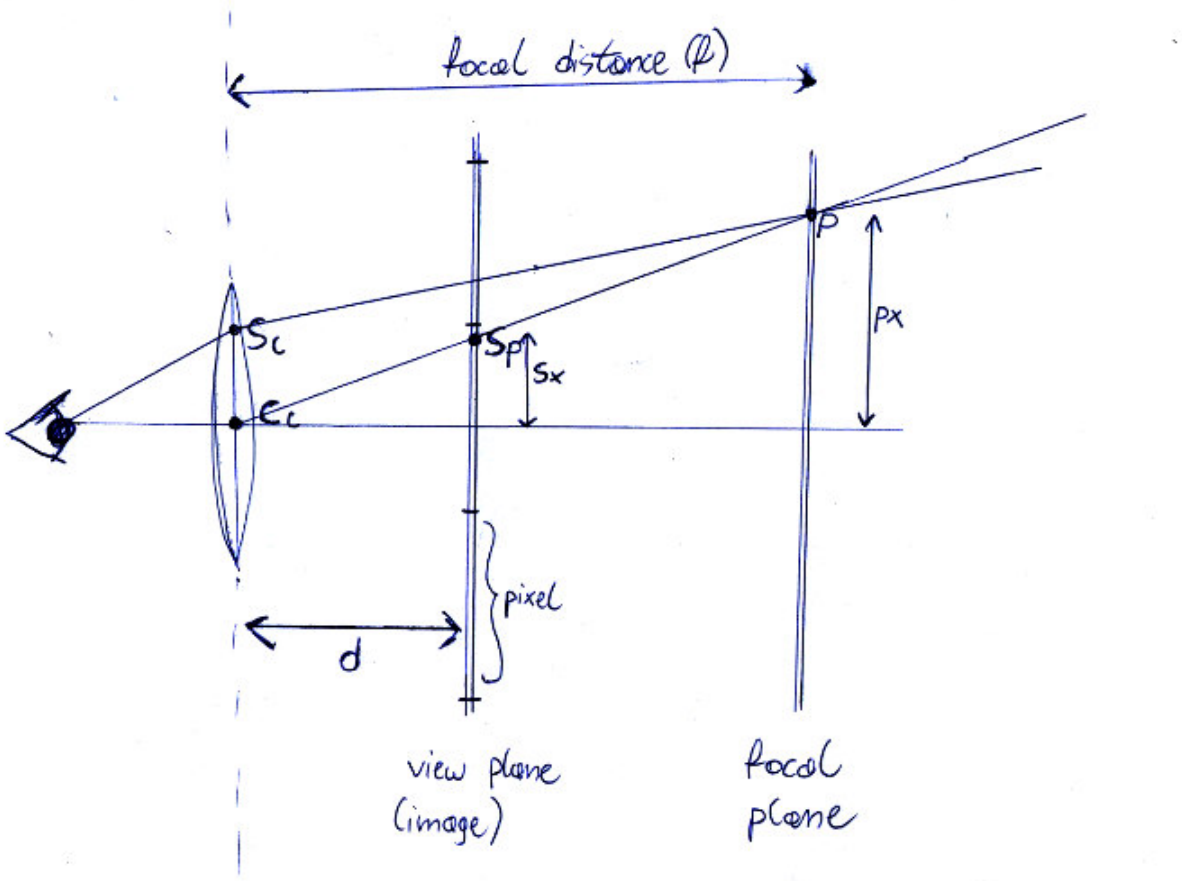
Wiemy dodatkowo że 'przesunięty' promień przecina się z 'oryginalnym' promieniem w odległości równej *`focal distance`*, czyli odległości do *focal plane* (*wszystkie* promienie kamery przecinają się na tej odległości). Okazuje się że tyle informacji w zupełności wystarcza.

V. Implementacja

Obliczenia które wykonamy dla każdego rzucanego z ThinLens promienia można podzielić na kilka kroków:

1. Wylosowanie punktu na powierzchni 'soczewki'.
2. Znalezienie punktu p, gdzie modyfikowany promień przecina *focal plane*.
3. Zmodyfikowanie punktu w którym rozpoczyna się promień za pomocą wylosowanego punktu, oraz wyliczenie dla niego nowego kierunku tak, żeby przechodził przez punkt p.
4. Wykonanie zwykłego śledzenia promienia wgłąb sceny.

Teraz ważna ilustracja:



Przedstawia ona sposób w jaki będziemy `losować` promienie wychodzące - pojawia się tutaj dużo oznaczeń, więc warto dokładnie się przyjrzeć bo może być trudno:

- c_l - centrum soczewki skupiającej promienie
- s_l - próbka losowana *na powierzchni soczewki* - dostajemy ją za pomocą samplera rozkładającego sample na powierzchni dysku (i mnożymy przez promień soczewki).
- d - odległość od soczewki do viewPlane, zmienna *distance* w kamerze
- s_p - próbka losowana *na powierzchni piksela* - dokładnie tak samo jak przy antyaliasingu.
- s_x - odległość od punktu na pikselu do kierunku w którym `patrzy` kamera
- p - punkt w którym zbiegają się wszystkie promienie (czyli ognisko)
- p_x - odległość od ogniska do kierunku w którym `patrzy` kamera

I teraz jeszcze raz kroki które wykonujemy.

Wylosowanie punktu na powierzchni soczewki jest najprostsze - używamy samplera generującego próbki na obszarze dysku:

```
Vector2 lensPoint = distributor.Single() * lensRadius;
```

Wiemy też od razu gdzie finalny promień się zaczyna - musimy zmodyfikować origin (pozycja na której znajduje się kamera) o lensPoint.
Ale - lensPoint jest wyrażony we współrzędnych lokalnych dla kamery (podobnie jak na ilustracji). Przed dodaniem do go `origin` musimy wykonać jego konwersję do współrzędnych absolutnych - za pomocą ONB:

```
Vector3 rayOrigin = origin + onb * new Vector3(lensPoint.X, lensPoint.Y, 0);
```

Głównym problemem jest znalezienie punktu p - wykorzystamy do tego cechy podobieństwa trójkątów.

Patrząc na ilustrację, z prawa Talesa wynika że:

$$\frac{s_x}{d} = \frac{p_x}{f}$$

Analogicznie dzieje się dla współrzędnej y:

$$\frac{s_y}{d} = \frac{p_y}{f}$$

Jako że szukamy współrzędnych punktu p, przekształcamy:

$$p_x = s_x(f / d)$$
$$p_y = s_y(f / d)$$

```
Vector2 p = pixelPosition * focal / distance;
```

Mając punkt p przez który przechodzi promień i punkt l w którym się zaczyna, możemy łatwo znaleźć jego kierunek:

$$d = p - l$$

```
Vector3 d = new Vector3(p.X - lensPoint.X, p.Y - lensPoint.Y, -focal);
```

I znowu - wektor `d` jest wyrażony we współrzędnych lokalnych, musimy je przenieść do absolutnych - więc kierunek promienia będzie równy:

```
Vector3 direction = onb * new Vector3(p.X - focalLoc.X, p.Y - focalLoc.Y, -focal);
```

Nie pozostaje nic innego niż rzucić go w świat:

```
return new Ray(rayOrigin, direction);
```

Implementacja napisanego tutaj kodu - poza tym że klasa przyjmuje chorą wręcz ilość parametrów w konstruktorze, nie ma tu niczego zaskakującego:

```
class ThinLens : ICamera
{
    OrthonormalBasis onb;
    Vector3 origin;
    Vector2 scale;
    double distance;
    Sampler distributor;
    double lensRadius;
    double focal;

    public ThinLens(Vector3 origin,
        Vector3 lookAt,
        Vector3 up,
        Vector2 scale,
        double distance,
        Sampler distributor,
        double lensRadius,
        double focal)
    {
        this.onb = new OrthonormalBasis(origin - lookAt, up);
        this.origin = origin;
        this.scale = scale;
        this.distance = distance;
        this.distributor = distributor;
        this.lensRadius = lensRadius;
        this.focal = focal;
    }

    public Ray GetRayTo(Vector2 relativePosition)
    {
        Vector2 pixelPosition = new Vector2(relativePosition.X * scale.X,
```

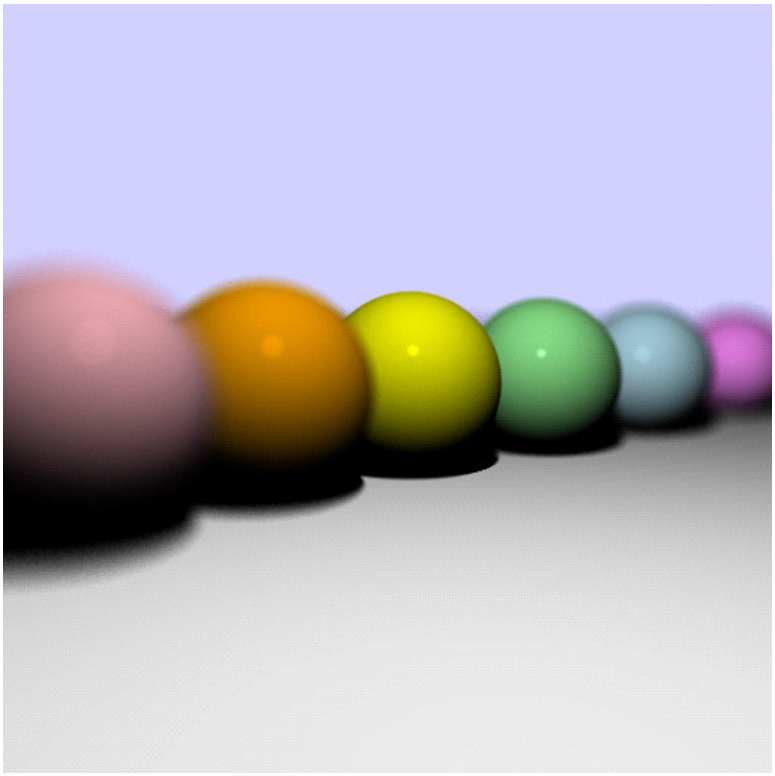
```
relativePosition.Y * scale.Y);
        Vector2 lensPoint = distributor.Single() * lensRadius;
        Vector3 rayOrigin = origin + onb * new Vector3(lensPoint.X, lensPoint.Y, 0);

        return new Ray(rayOrigin, RayDirection(pixelPosition, lensPoint));
    }

    Vector3 RayDirection(Vector2 pixelPosition, Vector2 lensPoint)
    {

        Vector2 p = pixelPosition * focal / distance;
        return onb * new Vector3(p.X - lensPoint.X, p.Y - lensPoint.Y, -focal);
    }
}
```

VI. Sampling a jakość

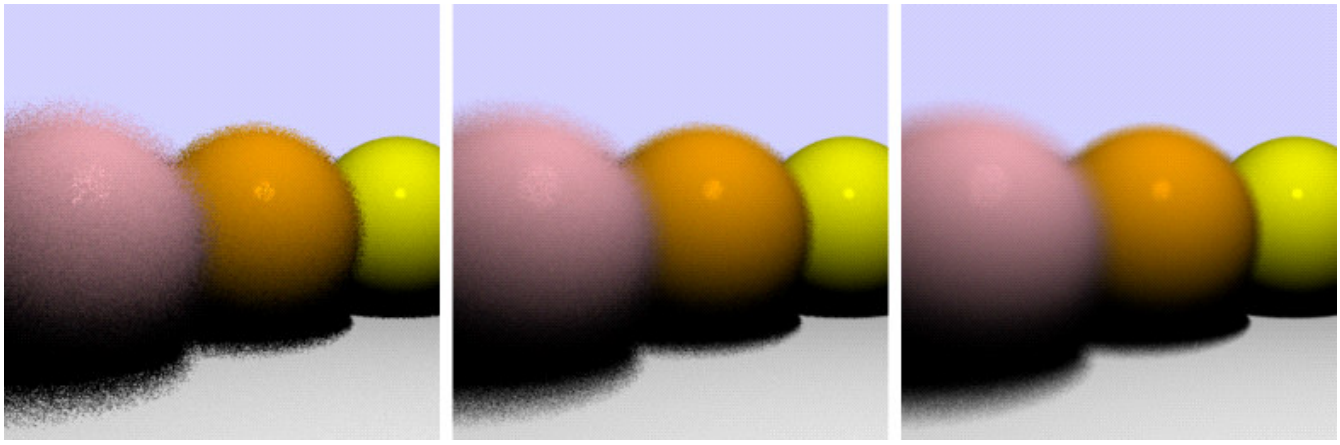
Mamy teraz w kodzie sztuczkę korzystającą intensywnie z samplingu - możemy wrócić do tematu który zaniedbaliśmy w poprzedniej części. Dlaczego właściwie implementowaliśmy tak wiele różnych sposobów generowania sampli, skoro ostatecznie wszystkie wydają się dawać taki sam efekt?

Tak więc krótkie, obrazkowe porównanie jakości obrazów wygenerowanych za pomocą różnych technik.

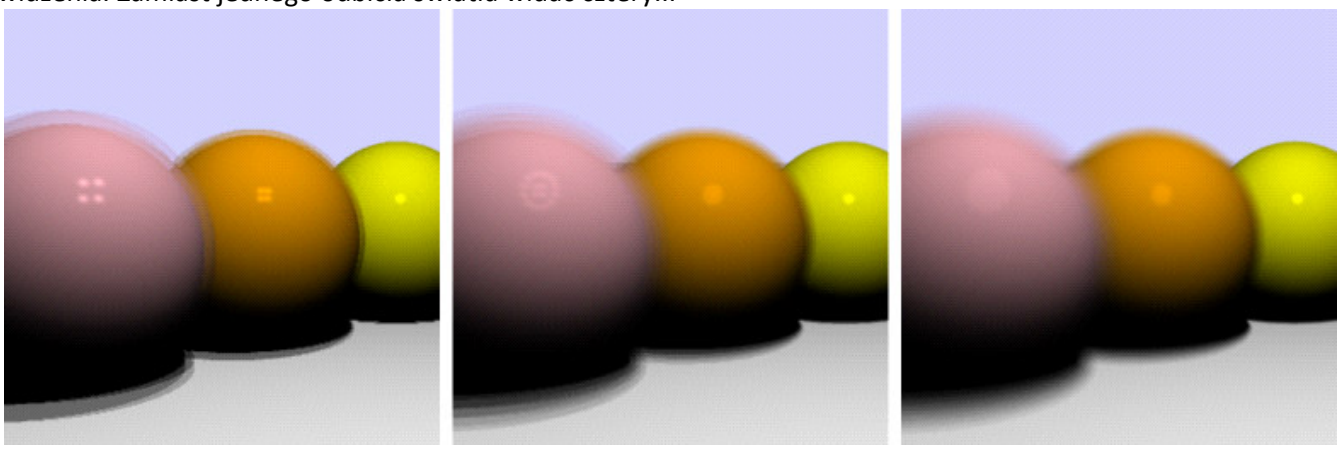
Wszystkie testy wykonywane na takiej oto scenie, generowane kolejno dla 4, 16 i 64 sampli (niestety, do artykułu nie można było wrzucić obrazów o większych rozdzielczościach więc różnice nie rzucają się w oczy tak jak powinny).



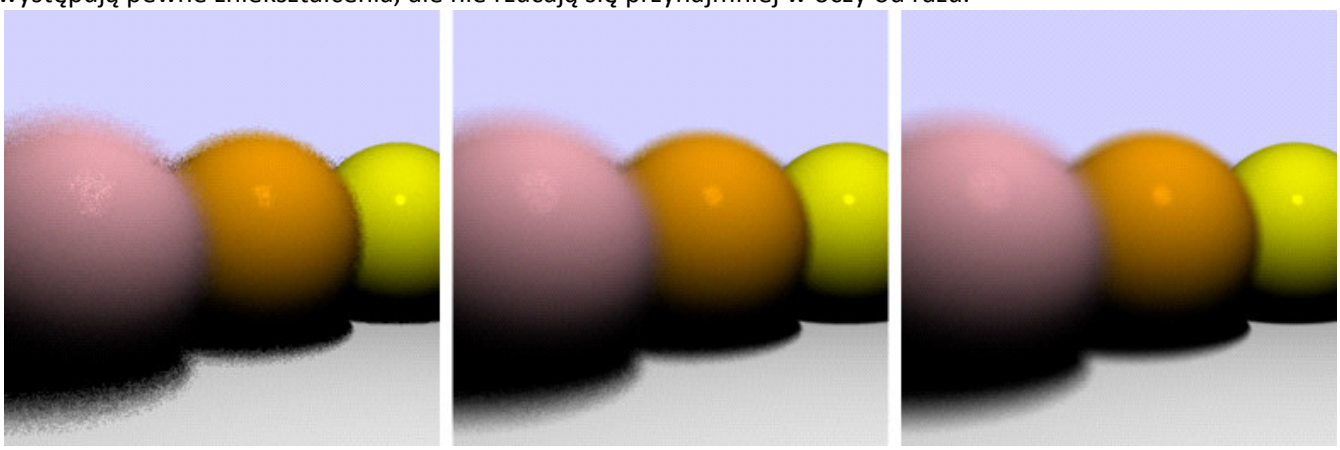
PureRandom:
Nawet dla największej ilości, 64 sampli, widać wyraźnie szum (szczególnie przy cieniu, z powodu dużego kontrastu):



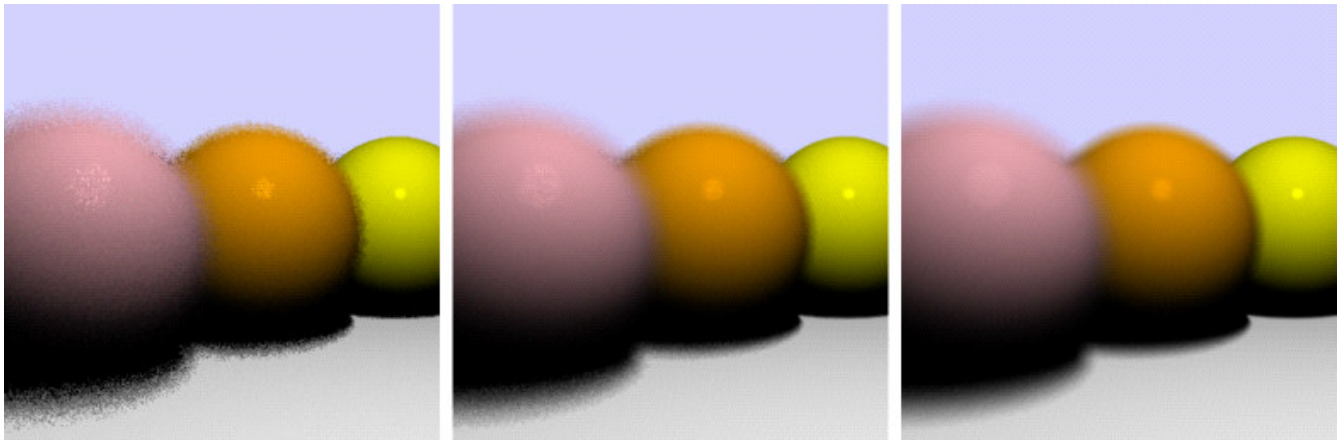
Regular:
Efekty są dość specyficzne, ponieważ nie ma tutaj szumu, ale pojawia się efekt podwójnego (poczwórnego) widzenia. Zamiast jednego odbicia światła widać cztery...



NRooks:
Znacznie lepszy od PureRandom, nie ma również dziwnych artefaktów jak poprzednik. Przy 64 samplach dalej występują pewne zniekształcenia, ale nie rzucają się przynajmniej w oczy od razu.



Jittered:
Daje bardzo podobne wyniki do poprzedniego - przy 4 samplach zachowuje się gorzej, ale już dla 16 wyniki są bardzo podobne. Przy 64 próbkach osiąga najlepsze wyniki spośród wszystkich i właściwie tak wygenerowany obraz nadaje się do publikacji.

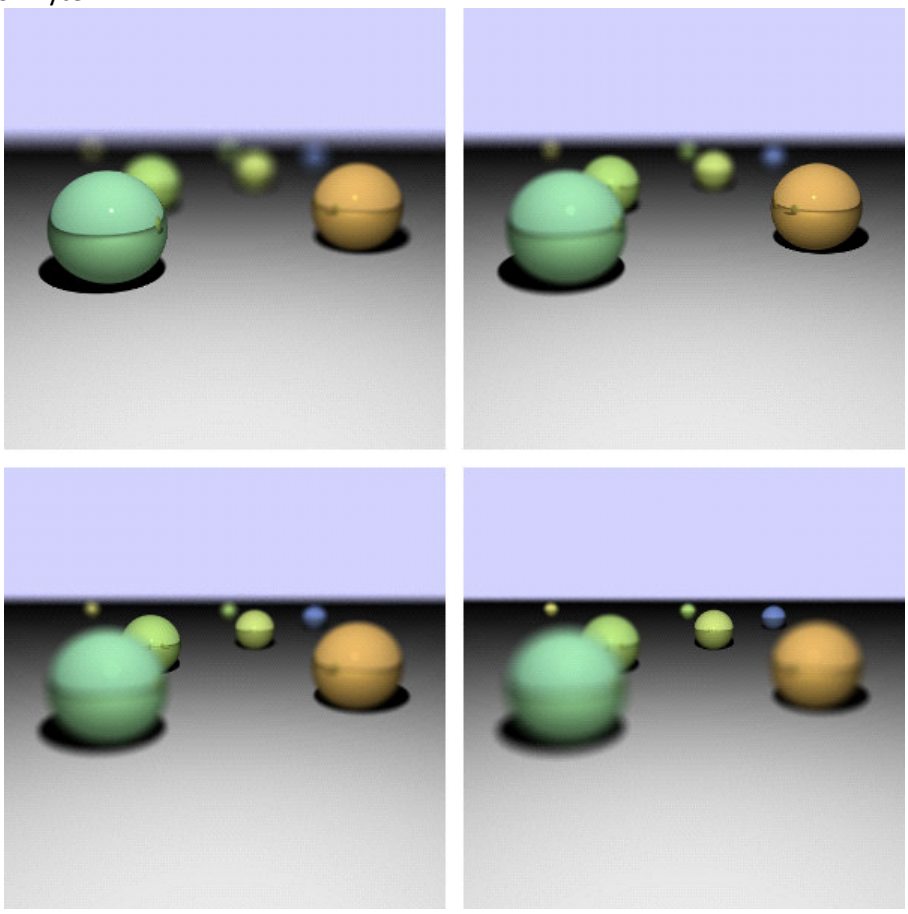


Tak naprawdę, praktycznie każda metoda (oprócz sposobów niejednorodnie dzielących sample, oraz pomijając ograniczenia liczb pseudolosowych) dąży do `idealnej` dokładności przy zwiększaniu ilości sampli - ale różnią się tym jak szybko polepsza się jakość obrazu. A jako że różnice są znaczne, a często lepszą jakość osiąga się dla 100 dobrze wygenerowanych sampli i ponad 1000 czysto losowych...

Jak widać, żeby osiągnąć sensowne efekty przy głębi ostrości często trzeba śledzić bardzo dużo promieni na pojedyńczy piksel.
Na szczęście, okazuje się że praca zużyta na testowanie wielu sampli dla każdego piksela musi być wykonana tylko raz - użycie, powiedzmy, 16 promieni/piksel pozwala otrzymać jednocześnie antyaliasing x16, depth of field x16 i dowolną ilość różnych efektów z taką samą jakością praktycznie bez żadnego kosztu wydajnościowego.

VII. Wyniki

Teraz możemy używać głębi ostrości do wyostrzania najważniejszych obiektów na scenie i rozmywania tych które nas nie interesują. Przykład - kamera na każdym z tych obrazków jest wyostrzona na jedną kulę podczas gdy pozostałe są rozmyte:



Zostało to osiągnięte przez zmienianie ogniskowej kamery (parametr *`focal`*). Promień soczewki (czyli siła rozmycia w tym przypadku) jest stały i wynosi 0.5, ale nic nie stoi na przeszkodzie żeby go zmieniać.

Zwyczajowa scena przedstawiana zawsze na koniec artykułu nie do końca pasuje, więc została tymczasowo odświeżona: