NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis

2020 ECCV

3D와 2D

현실 세계는 3차원인데, 카메라로 관측할 때는 2차원이 됩니다.

그러므로, 2차원의 사진들을 합쳐서 3D로 복원하고자 하는 시도들이 많이 있었습니다.

그리고 이 중 가장 성공적으로 거론되는게 NeRF입니다.

Encoding

Voxel : 마인크래프트처럼 3D를 큐브로 표현합니다.

Point : 3차원을 점들의 집합으로 표현합니다.

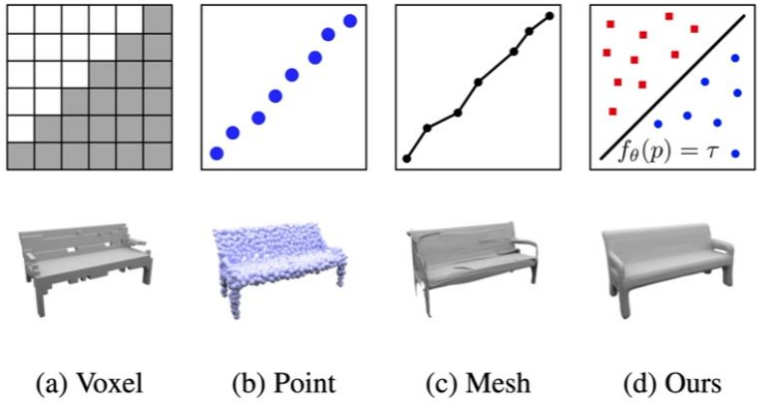
Mesh : 그래픽스에서 많이 쓰는데, 점으로 평면을 만들고, 평면들의 집한으로 3차원을 표현하기

Implicit : 물체를 학습하는게 아니라, 물체들의 경계를 학습하는 방법입니다.

Implicit은 물체의 어디에 뭐가 있는지를 학습하는게 아니라, 물체의 경계선을 학습하는 겁니다.

$$f_\theta : \mathbb{R}^3 \times \mathcal{X} \rightarrow [0, 1]$$

3D coordinate observation out in



NeRF : Neural Radiance Fields

NeRF란 Neural Radiance Fields로, Radiance Field를 를 DNN으로 모델링한 것을 의미합니다.

Radiance Fields란 것은 5가지의 값으로 현실 세계를 나타내는 것입니다.

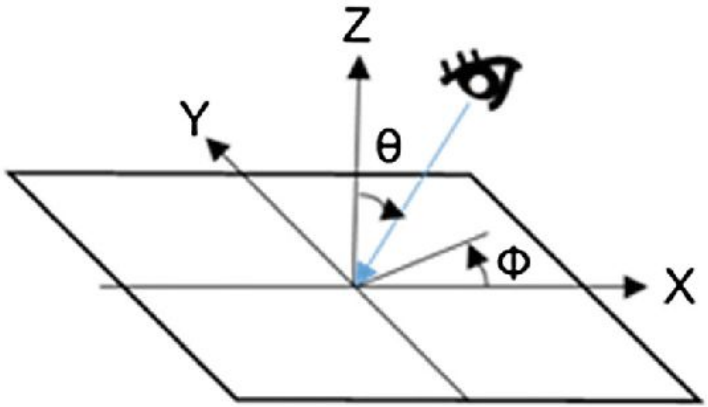
사실 이 단어는 매우 생소했는데, Camera Coordinate나 5D Coordinate라는 단어로도 사람들이 간혹 쓰기도 하더라구요

NeRF : Neural Radiance Fields

- 먼저, Radiance Field에 대해서 먼저 알아보도록 하겠습니다.
- Radiance Field는 **Point**와, **View Direction** 으로 이루어져 있습니다.
  - 관측자가 물체의 어디를 바라보지 방향에 해당하는 정보인 **view direction (θ, ϕ)**
  - 물체로부터 관측자가 존재하는 위치에 대한 정보 **point (x, y, z)**

NeRF : Neural Radiance Fields

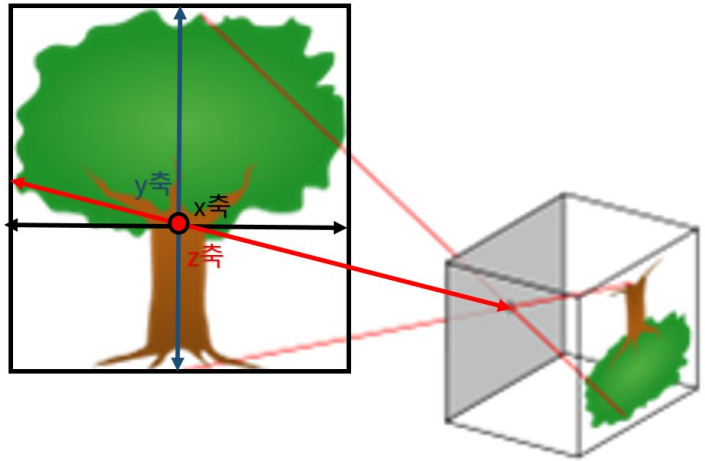
- 이걸 이미지로 표현하게 되면 다음의 그림과 같이 표현할 수 있습니다.



- Point에 해당하는 (X, Y, Z) 좌표값에서 관측자는 있게 됩니다.
- 그리고 관측자는 해당 위치에서 물체가 있는 방향으로 시선을 향하겠죠.
- 특정 위치에서, 바라보는 시선을 **View Direction** 이라고 하고, 바라보고 있는 선을 Ray라고 합니다.
- X,Y 평면에서 Ray와의 각도를 **θ**라고 하고, Z와 Ray가 이루는 각도를 **ϕ**라고 합니다.

NeRF : Neural Radiance Fields

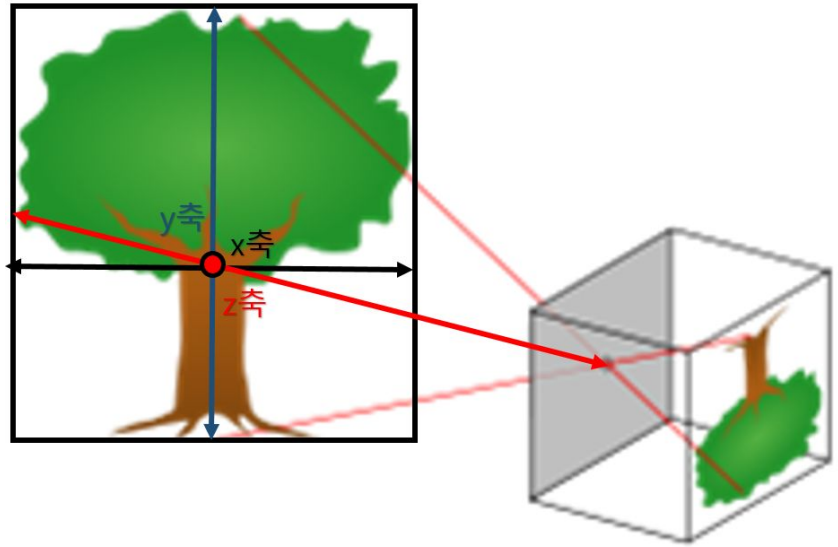
- 이제는 예시를 들어가면서 설명을 해보도록 하겠습니다.
- 우선 다음과 같이 나무가 있습니다.



- 우선은 나무가 렌즈를 기점으로 거꾸로 매달려 있는데, 렌즈보다 위에 있는 물체의 경우에는 위에서 밑으로 Ray가 향하기 때문에, 실질적으로 카메라 상에서는 밑에 상이 맺히고, 렌즈보다 아래에 있는 물체의 경우에는 아래에서 위로 Ray가 향하기 때문에 실질적으로 상이 맺히는 위치는 카메라의 윗부분에 맺히게 됩니다. 이러한 원리로 렌즈를 기점으로 물체는 반전되서 표현됩니다.

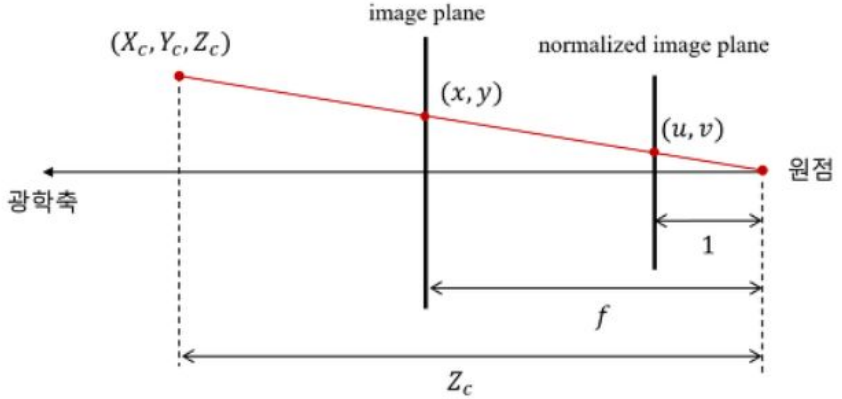
NeRF : Neural Radiance Fields

- 카메라가 물체를 바라볼 때 X,Y 좌표는 물체의 위치를 나타내고, Z축의 경우에는 카메라와 물체간의 거리를 의미하게 됩니다.
- 무슨 의미냐면, 현실에서 어떤 물체를 카메라로 담을 때에, Z가 Depth의 역할을 한다는 것입니다.
- 그리고 Z축의 거리에 따라서 부르는 명칭이 달라지게 됩니다.



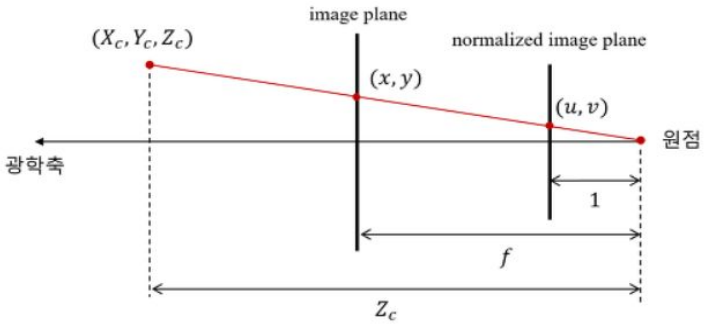
NeRF : Neural Radiance Fields

- 광학축에서는 따로 부르는 명칭이 있는지는 모르겠지만, 어떤 물체의 (X,Y,Z)좌표에서 카메라로 빛이 반사되서 들어오는 상황에서 예시를 들어보겠습니다.
- 카메라와 물체간의 거리인 **Z축 값이 1되는 지점을 normalized image plane** 이라고 부릅니다.
  여기서 각 좌표 값은 (X/Z, Y/Z, Z/Z) = (u, v, 1)이며 해당 지점에 임의의 값을 곱해서 image plane을 만들거나 Z를 곱해서 다시 광학좌표계로 만들 수 있습니다.
- image plane에서는 우리가 실질적으로 카메라로부터 얻어지는 image들이 얻어지게 되고, 여기서 각 좌표는 x, y로 표현됩니다. (z축은 normalized image plane을 만들 때, 1로 고정하므로 없어져도 무방하고, 제거하게 됩니다.)



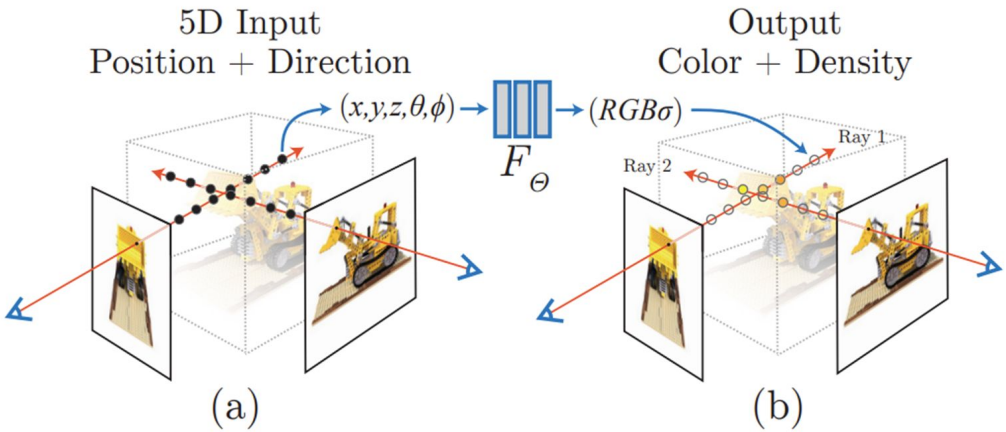
NeRF : Neural Radiance Fields

- 여기서 한 가지 주의깊게 보셔야 할 점은, 광학축에서 image plane으로 만들 때, Z축이 없어진다는 겁니다.
- 없어지는 건 아니고 음.. Z축의 있는 모든 값들이 projection되는 과정에서 합쳐진다고 표현해야겠네요.
- 이는 일종의 weighted sum이라고 볼 수 있는데, 3D의 이미지를 2D로 표현하는 과정에서 관찰자와 물체간의 일직선상의 모든 점을 weighted sum해서 하나의 점으로 mapping해야 그제서야 우리가 알고 있는 pixel이 되는 겁니다.
- 여기서 weighted는 물체의 빽빽함이 됩니다.(density라고 부르는데, 투명한것의 반대개념입니다.)
  예를 들자면, 내가 나무를 바라보고 있는데, 중간에 자동차가 있다면, 자동차에 가려서 나무가 보이지 않겠죠?
  이건 관측자가 물체를 바라보고 직선상의 경로인 (Camera ray) 사이에 자동차에 해당하는 픽셀의 density가 높아서 그렇습니다.



NeRF : Neural Radiance Fields

- 결국 pixel value = camera ray위에 존재하는 모든 point들의 RGB * density의 값 이라고 정의할 수 있습니다.
- 이는 밑의 그림을 보시면 좀 쉽게 이해할 수 있습니다.



- 특정 시점에서 물체를 바라볼 때, 시선에 해당하는 Camera Ray위의 있는 모든 점들이 합쳐져야 하나의 픽셀로서 나타난다는 겁니다.
- 공기에 해당하는 부분은 픽셀값이 없을 것이고, 물체에 시선이 닿았는데 투과를 한다면 해당 물체의 값 + 투명도를 곱해주면서 계속 더해지게 됩니다.
- 이런 방식으로 우리는 특정 방향에서 오브젝트를 바라봤을 때의 초점에서 물체의 픽셀값을 구할 수 있게 됩니다.
- 그런데 한가지 고려하지 않은게 있는데, ray 위에 존재하는 모든 point, 픽셀들…은 사실 연속된 값들이기 때문에, 컴퓨터 입장에서는 Sampling을 해줘야만 계산을 할 수가 있습니다.

NeRF : Neural Radiance Fields

- 무슨 얘기냐면, Camera Ray의 기댓값 C(r)은 다음과 같이 나타납니다.
- 여기서 ray [(x, y, z)에서 (θ, ϕ)방향으로 바라본]의 기댓값 C(r)은 ray의 range(near, far)사이의 누적된 픽셀값입니다.

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt$$

Camera Ray(r) 의 기댓값

near

far

t시점까지 누적된 density

t시점의 density

t지점의 픽셀 값

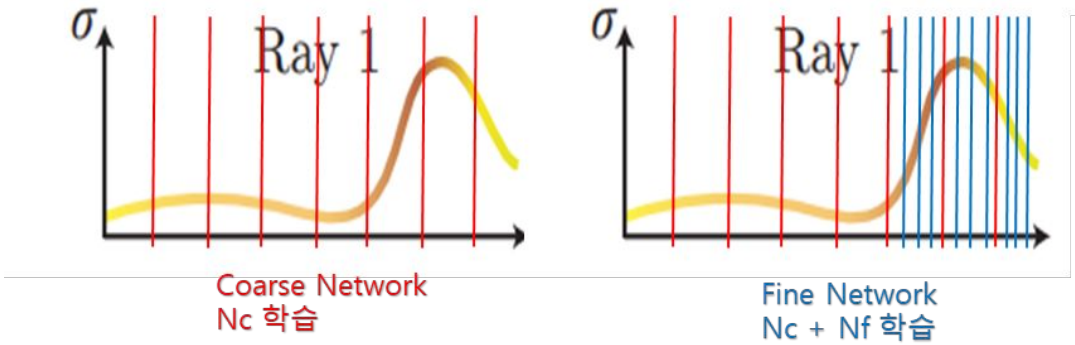
- 그리고 각 포인트들은 밀도 혹은 불투명도에 해당하는 density값들을 가지고 있으므로, 특정 포인트에서의 픽셀값은 density를 곱해줘야 합니다. t시점까지 누적된 density인 T(t)는 다음과 같이 표현됩니다.

$$T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right)$$

NeRF : Neural Radiance Fields

- 즉, 카메라에서 물체를 바라봤을 때의 픽셀값은 누적분포함수를 통해서 구해야 하는데, Continual한 값은 컴퓨터에서 처리할 수 없으므로, Sampling을 해줘야 픽셀값을 계산할 수가 있습니다.
- Sampling의 가장 간단한 방법은 ray를 Uniform Sampling하는 것일겁니다.
- 하지만 Uniform Sampling은 성능 좋게 나오기가 어렵겠죠.
- 왜냐면, Radiance Field내에서, 물체를 만나기 전까지는 pixel값도 0이고, density도 0인 point들이 ray에서 분포해있다가, 물체를 만나게 되면서, 물체가 가진 density에 의해서 픽셀값들이 정해지게 되므로, 특정 시점 전까지는 픽셀값에 영향력이 거의 없기 때문이죠.
- 그래서, 대안으로 네트워크를 두개를 만들어서 사용합니다.
- Ray에서 point들의 density가 다음과 같을 때, Uniform하게 point들을 뽑으면 비효율 적이니,
- density를 예측하기 위해서 사용하는 네트워크와
- 예측된 density를 기반으로 세밀하게 point를 뽑는 네트워크를 만드는 겁니다.

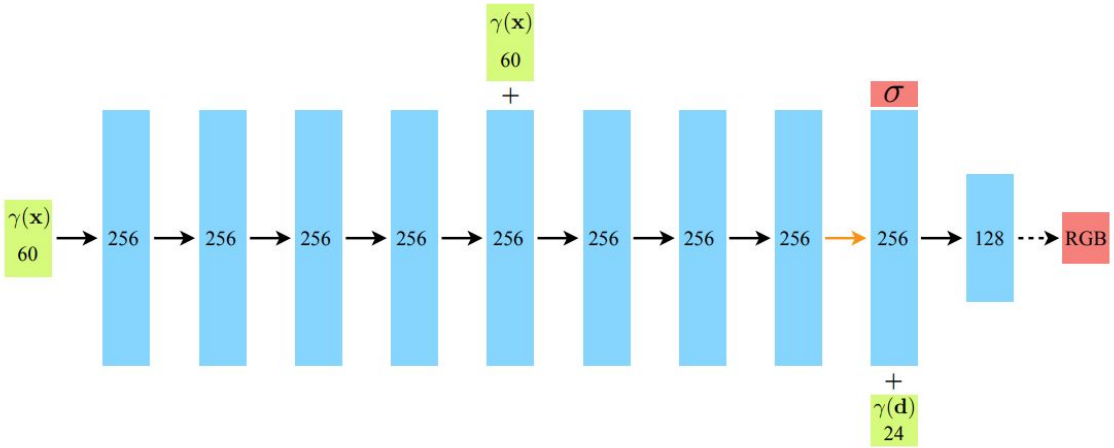
NeRF : Volume Density



- ray를 n개로 Uniform하게 나눈 후(Nc 개의 Sample) 해당 Sample로 학습한 모델을 만듭니다. 해당 모델은 Coarse Network라고 하고 해당 네트워크에서는 density를 나타내는 T(t), volume density를 구합니다.
- volume density의 분포에 따라, density가 높은것들이 분포하는 구간에서 세밀하게 Sampling을 한 후(Nf개의 Sample), Nc + Nf개의 Sample로 학습한 Fine Network를 만듭니다.
- 다음으로 학습 방법에 대해서 알아보겠습니다.

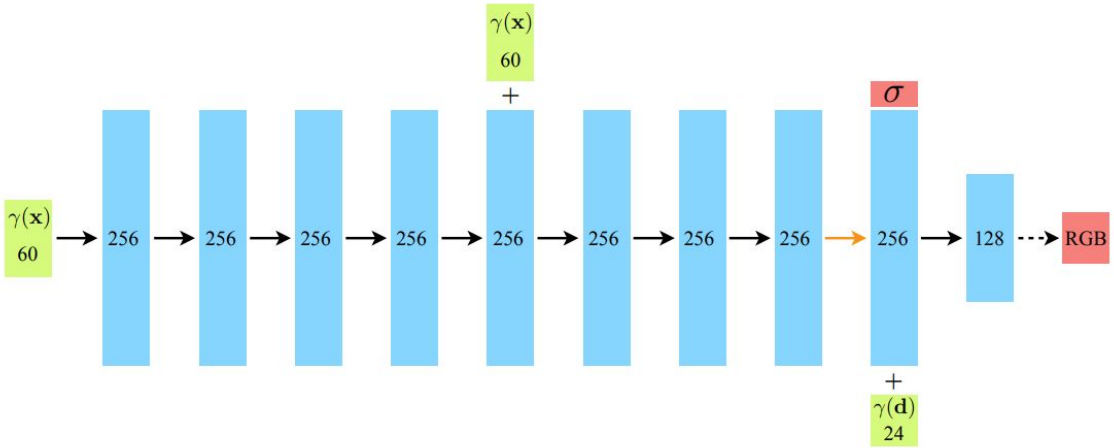
Volume Rendering

NERF : Network Architecture



- 네트워크는 단순히 MLP를 여러개 사용한 모델입니다.
- 하지만, MLP에서는 입력값이 3(x,y,z)개 밖에 안되면, 학습이 어려워지므로, 입력값들에 Positional Encoding을 더해서 입력값 x(3개의 feature)를 x`(60 개의 feature)로 변환해줍니다.
- Positional Encoding을 하는 이유는, x의 값 자체가 너무 적으면 모델이 학습할 때 오버피팅되는? 경향이 좀 있다고 합니다.
- 보지 못한 방향에 대한 값이 많이 안 좋게 나온다고 합니다.

NERF : Positional Encoding



$$\gamma(p) = \left(\sin\left(2^0\pi p\right), \cos\left(2^0\pi p\right), \cdots, \sin\left(2^{L-1}\pi p\right), \cos\left(2^{L-1}\pi p\right)\right).$$

- Positional Encoding을 할 때에, 카메라의 좌표는 (x,y,z)이고, 10배 늘려주는데, sin, cos 값으로 표현하니 60개의 값으로 표현이 됩니다.
- Direction 같은 경우에는 원래 **(θ, ϕ)** 지만 네트워크를 학습할 때는 표현하는게 어려워서, (x`,y`,z`)로 바라보는 direction을 나타내서 3개의 값이 되고, 4배 늘려주는데, sin, cos 값으로 표현하니 24개의 값이 만들어지게 됩니다.

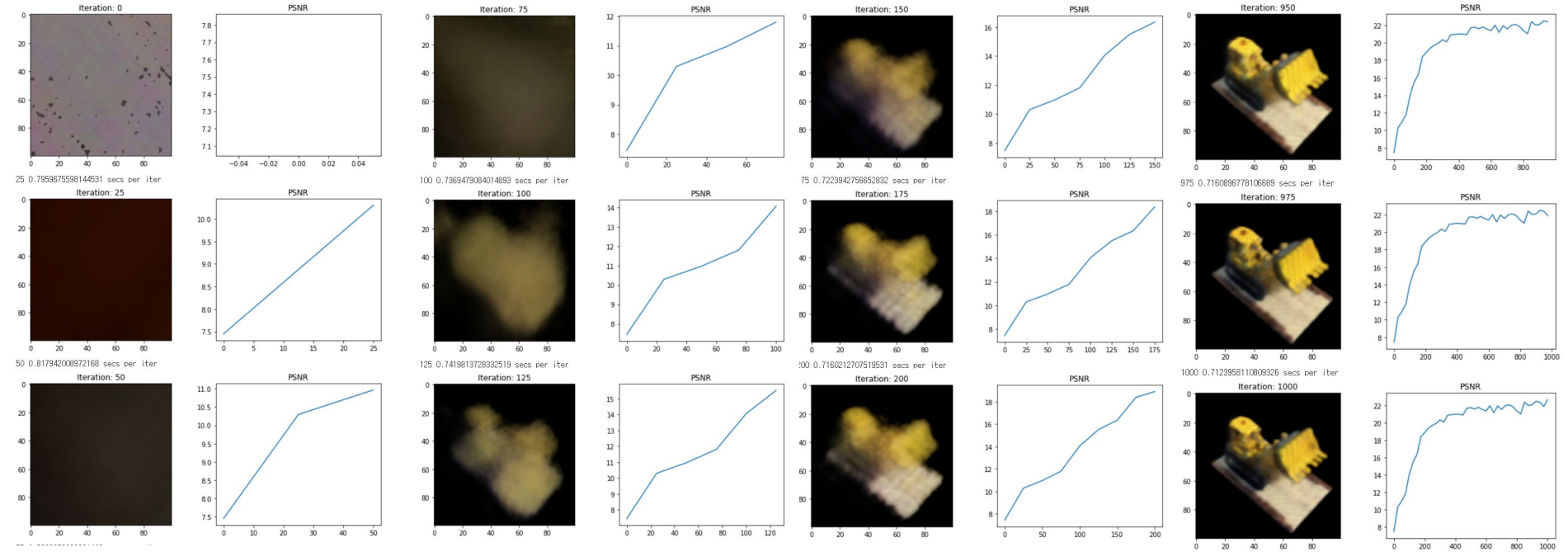
NERF : Radiance Fields

```python
def init_model(D=8, W=256):
    relu = tf.keras.layers.ReLU()
    dense = lambda W=W, act=relu : tf.keras.layers.Dense(W, activation=act)

    inputs = tf.keras.Input(shape=(3 + 3*2*L_embed))
    outputs = inputs
    for i in range(D):
        outputs = dense()(outputs)
        if i%4==0 and i>0:
            outputs = tf.concat([outputs, inputs], -1)
    outputs = dense(4, act=None)(outputs)

    model = tf.keras.Model(inputs=inputs, outputs=outputs)
    return model
```

```python
def posenc(x):
  rets = [x]
  for i in range(L_embed):
    for fn in [tf.sin, tf.cos]:
      rets.append(fn(2.**i * x))
  return tf.concat(rets, -1)

L_embed = 6
embed_fn = posenc
```

- 코드도 생각보다 간단합니다.
- Residual Connection이랑 유사한데, 각 f를 4개의 dense layer라고 보시면 f`(x + f(x))라고 보시면 됩니다.

Result


Iteration: 0
PSNR
25 0.7959675598144531 secs per iter
Iteration: 75
PSNR
100 0.7369479084014893 secs per iter
Iteration: 150
PSNR
75 0.7223942756652832 secs per iter
Iteration: 950
PSNR
975 0.7160896778106689 secs per iter
Iteration: 25
PSNR
50 0.817942000972168 secs per iter
Iteration: 100
PSNR
125 0.7419813728332519 secs per iter
Iteration: 175
PSNR
00 0.7160212707519531 secs per iter
Iteration: 975
PSNR
1000 0.7123958110809326 secs per iter
Iteration: 50
PSNR
Iteration: 125
PSNR
Iteration: 200
PSNR
Iteration: 1000
PSNR

Result

성능의 경우에는 PSNR 같은 수식들을 사용합니다.
NeRF가 가장 좋은 성능을 보여준다고 하네요.

Method	Diffuse Synthetic 360° [41]			Realistic Synthetic 360°			Real Forward-Facing [28]		
	PSNR↑	SSIM↑	LPIPS↓	PSNR↑	SSIM↑	LPIPS↓	PSNR↑	SSIM↑	LPIPS↓
SRN [42]	33.20	0.963	0.073	22.26	0.846	0.170	22.84	0.668	0.378
NV [24]	29.62	0.929	0.099	26.05	0.893	0.160	-	-	-
LLFF [28]	34.38	0.985	0.048	24.88	0.911	0.114	24.13	0.798	**0.212**
Ours	**40.15**	**0.991**	**0.023**	**31.01**	**0.947**	**0.081**	**26.50**	**0.811**	0.250

Result

MSE에서 원본 이미지(I)와 잡음이 포함된 이미지 (K)사이의 픽셀단위 오차를 제곱해서 구합니다.

[신호의 최대 값 / 평균 신호 오차] 값을 PSNR로 사용하게 되는 겁니다.

신호의 오차값이 0이라면, PSNR은 커지고, 신호의 오차값이 커질수록 PSNR은 작아지는 방식입니다.

$$PSNR = 10\log_{10}\left(\frac{MAX_I^2}{MSE}\right)$$

$$= 20\log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right)$$

$$= 20\log_{10}(MAX_I) - 10\log_{10}(MSE)$$

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,\ j) - K(i,\ j)]^2$$

Appendix

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt$$

Camera Ray(r)의 기댓값

far / near

t시점까지 누적된 density

t시점의 density

t지점의 픽셀 값

r(t) = o + td

Camera Ray인 r(t)는 다음과 같이 정의됩니다.
초점 o에서 특정 방향(d)만큼 t거리까지의 점들
즉, ray는 (x, y, z, θ, φ)로 정의됩니다.

$\mathbf{c}(\mathbf{r}(t), \mathbf{d})$ 특정 지점 t에서의 픽셀 값(ray와 density에 따른)

$$T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right)$$

T(t)는 t_n부터 t까지의 ray의 density를 누적한 함수입니다.
뭔 얘기냐면, 내가 바라보는 위치에서 바라보고자 하는 위치까지
Z축의 density의 값들입니다.

$\sigma(\mathbf{r}(t))$ 주어진 ray의 t지점에서의 투과율을 의미합니다.