Machine learning for inverse graphic

George Tang

August 2023

1 Introduction

The purpose of these notes is to provide a conclusion for the module "Machine Learning for Inverse Graphics," instructed by Vincent Sitzmann. The lectures can be accessed via the following URL: `https://www.scenerepresentations.org/courses/inverse-graphics/`. This online course is publicly available and covers the following topics:

1. Computer vision and computer graphics fundamentals (pinhole camera model, camera pose, projective geometry, light fields, multi-view geometry).
2. Volumetric scene representations for deep learning: Neural fields and voxel grids.
3. Differentiable rendering in 3D representations and light fields.
4. Inference algorithms for deep-learning based 3D reconstruction: convolutional neural networks, auto-decoding.
5. Basics of geometric deep learning: Representation theory, groups, group actions, equivariance, equivariant neural network architectures.
6. Self-supervised learning of scene representations via 3D-aware auto-encoding.
7. Applications of neural scene representations in graphics, robotics, vision, and scientific discovery.

I will summarize the lectures in detail and extend upon these topics with additional knowledge.

2 Image formation

- The first question is: what is a 3D scene?

  A 3D scene refers to a digital representation of a three-dimensional environment or setting that is created and viewed within a computer graphics system. It's a virtual space where objects, characters, textures, lighting, and other elements are arranged and configured to simulate a realistic or stylized visual representation of a physical or imaginary world.

- How do we observe scenes?

  In this case, we are talking about a camera observes a subset of all the light rays in a scene.

We can represent a light ray as a vector. The formula for the light field is $LF = \mathbb{R}^3 \times \mathbb{S}^2 \rightarrow \mathbb{R}^3, LF(x, d) = c$ where we map a light ray with it's direction and position to a color(red,blue,green).

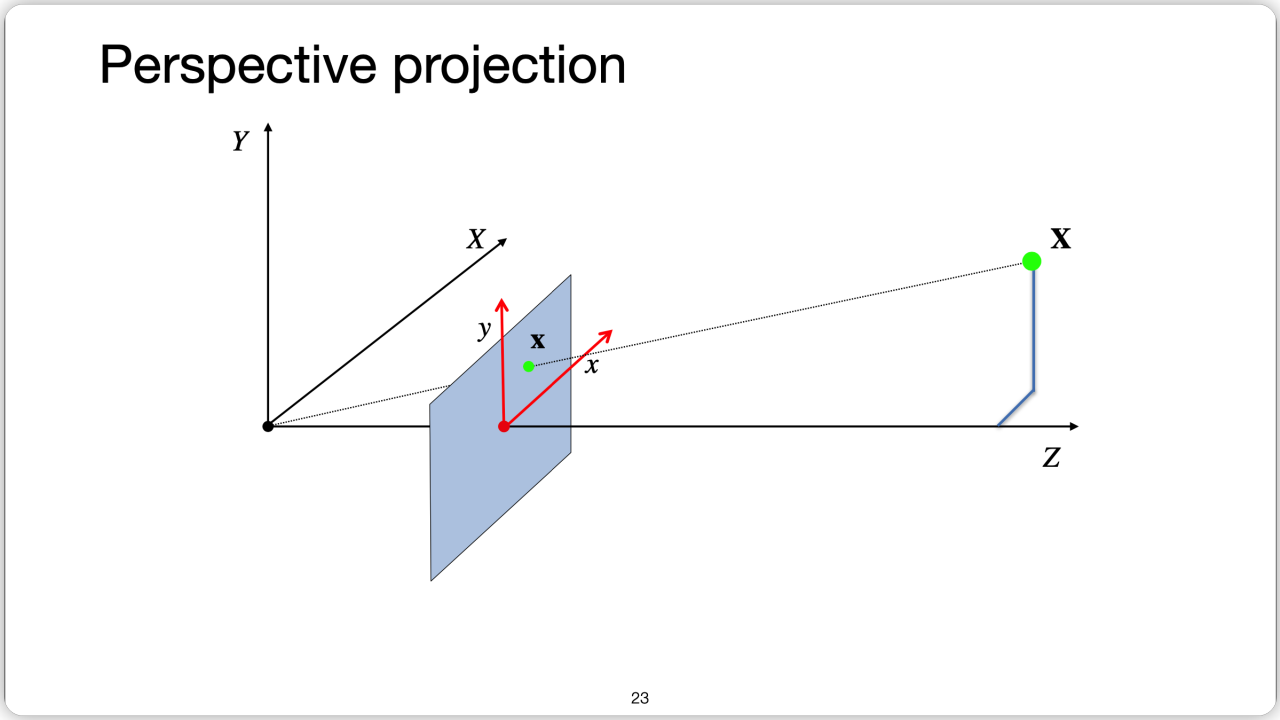

Figure 1: Perspective projection

Now let's consider a Camera Obscura. It is a black box which only allows some part of the light to pass through the hole. We can then represent the image of a 3D object as shown in figure(1). The blue image is modelled in 2D Cartesian coordinate, where as the actual 3D object is modelled in 3D Cartesian coordinate. Notice the projection image point is always along the line connecting the origin and the object. Consider an image point on the image: denote it as $\mathbf{x}$.

$$\mathbf{x} = (x, y) = (X, Y)\frac{f}{Z} \tag{1}$$

However, notice (1) is not linear. Now let's consider the homogeneous coordinates

$$\mathbf{x} = (x, y) \rightarrow \tilde{x} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{2}$$

We can then do the same thing for the world coordinate. We obtain:

$$\tilde{x} = \begin{pmatrix} f & 0 & P_x & 0 \\ 0 & f & P_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \begin{pmatrix} fX \\ fY \\ Z \end{pmatrix} \rightarrow (f\frac{X}{Z}, f\frac{Y}{Z}) \tag{3}$$

where f is the focal length and P_x, P_y are the position of the point $\mathbf{x}$ in respect to the image axis. We can further decompose the matrix into two matrix as below:

$$\begin{pmatrix} f & 0 & P_x & 0 \\ 0 & f & P_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} f & 0 & P_x \\ 0 & f & P_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{4}$$

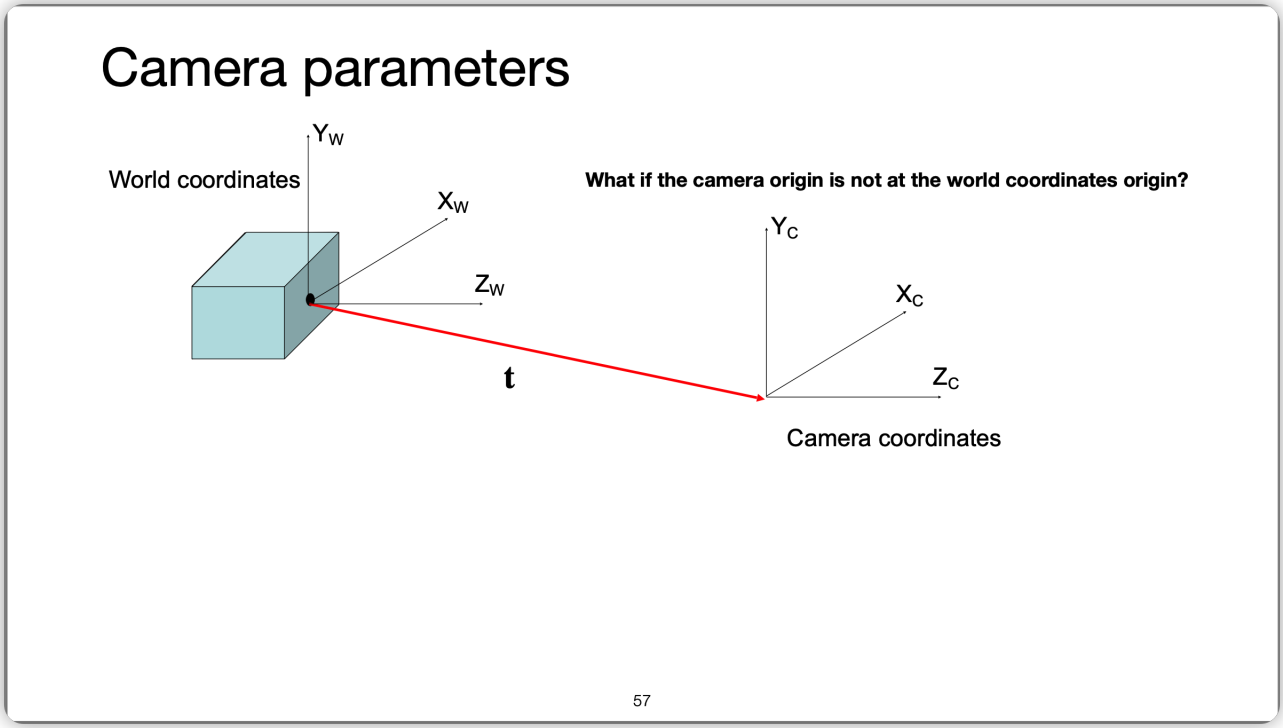

Figure 2: Camera parameters

where we denote the first matrix in the multiplication as $\mathbf{K}$. We can generalise this idea further. Consider when the world coordinates are not inline with the camera coordinate. As shown in figure (2), $X_c = R(X_w - t)$. In homogeneous coordinates, we can write it as:

$$\tilde{\mathbf{X}}_c = \begin{pmatrix} R & -Rt \\ 0 & 1 \end{pmatrix} \tilde{\mathbf{X}}_w \tag{5}$$

Therefore we can conclude:

$$\begin{cases} \tilde{\mathbf{x}} = \mathbf{K}[\mathbf{I}|0]\mathbf{C}^{W2C}\tilde{\mathbf{X}}_w \\ \tilde{\mathbf{x}} = P\tilde{\mathbf{X}}_w \end{cases} \tag{6}$$

3 Multi-View Geometry

Consider a video clip, can we reconstruct a 3D scence just using this video clip? Multi-view geometry will help us to achieve this goal.

3.1 Triangulation

Suppose we know the projection matrix P of the two camera,how could we find the 3D object point?

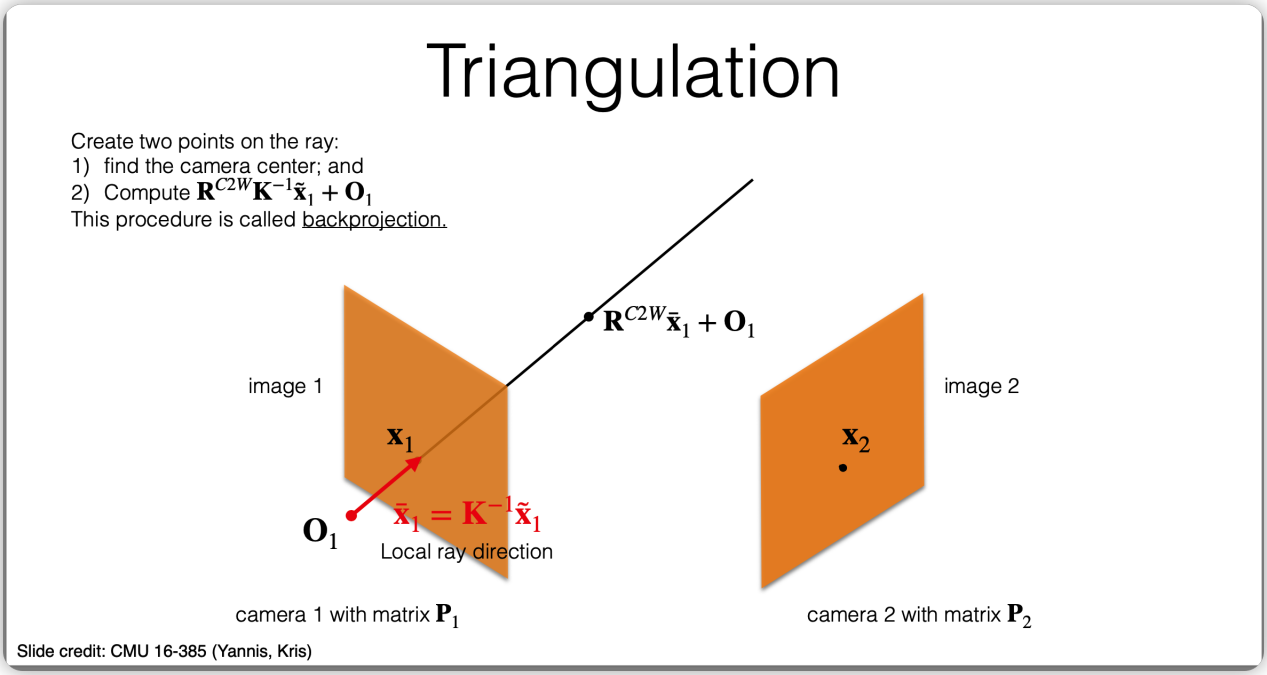

Figure 3: Triangulation

As shown in 3, we can compute the local ray direction in camera view: $\bar{\mathbf{x}}_1 = K^{-1}\tilde{\mathbf{x}}_1$. Then to transfer into world view, we can compute: $\mathbf{R}^{C2W}\bar{\mathbf{x}}_1 + \mathbf{O}_1$. This is called backprojection. This will give us the location of the 3D point when we find out the same vector coming out through x_2. Since we might have minor error when labelling the points on the image, so we can solve a optimisation point instead. We can use the estimate 3D point and find the projection of X into i-th camera:

$$\tilde{\pi}_1(X) = \tilde{K}_i[I|0]\mathbf{C}_i^{W2C}\bar{\mathbf{X}} \tag{7}$$

Then solve the least squares problem, our target is to minimise the sum of error for that point from all cameras:

$$\mathbf{X}^* = \text{argmin}_x \sum_i^N \|\tilde{\pi}_1(X) - x_i\|_2^2 \tag{8}$$

3.2 Epipolar geometry

Suppose we have a point $\mathbf{X}$ and it is corresponding to a point $\mathbf{x_1}$ on the image plane. Now the question is what will the point be on image plane 2.

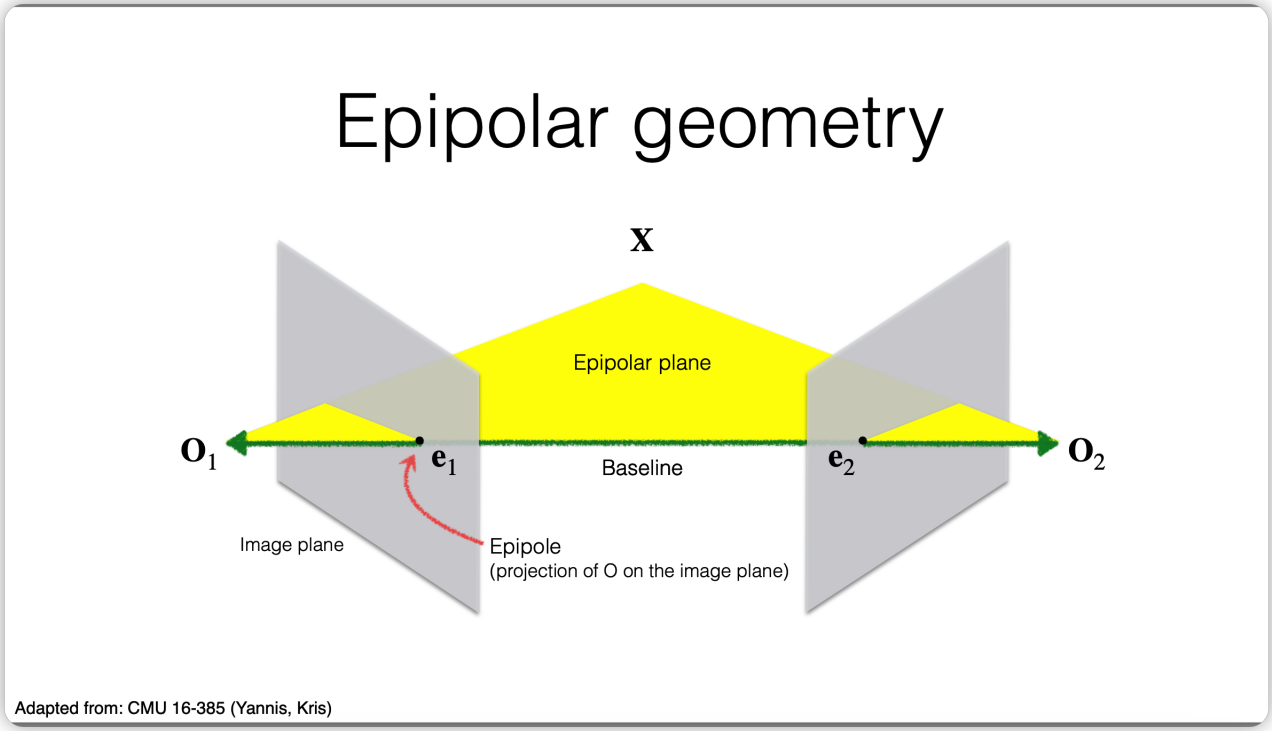

Figure 4: Epipolar geometry

The line connecting O_1 and O_2 is called the baseline. e_1, e_2 are the point on the image plane which correspond to the opposite origin. We call them epipole. The yellow plane connecting the baseline and $\mathbf{X}$ is called the epipolar plane. The line l_1, l_2 on the plane are called the epipolar line. In short, all the possible point x_2 on the image plane 2 will be on the epipolar plane.

When two image plane is parallel to each other, the epipolar line will be parallel as well. This means the epipole will be in infinity.

3.3 Compute epipolar line

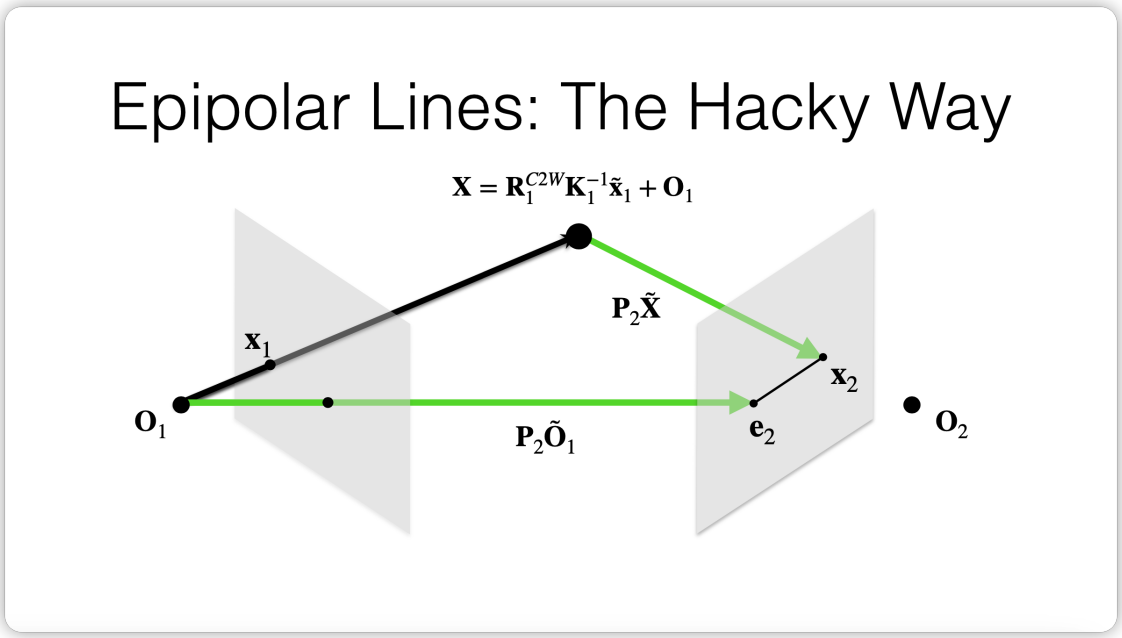

Figure 5: Hackyway

1. Firstly we project O_1 to image plane 2. Since we know O_1 and O_2, this is easy to do.

2. Then we can project any point alone $O_1\vec{x}_1$ to the image plane.

3. Finally connect e_2 and x_2.

This way always work but it has many steps which is complicated to compute. Therefore, we can do the alternative way. Denote the epipolar line $l = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$ Then $ax + by + c = 0$ is a plane contain such line. If the point x is on the epipolar line then that means $\tilde{x}^T l = 0$. This is because we know any point on the line will be on this plane which equal to 0. We then assume there exists a fundamental matrix F such that

$$F\tilde{x_1} = l_2 \tag{9}$$

If such matrix exists, that means the epipolar line on the second image can be found. Combine with the previous equation, we have:

$$\begin{cases} F\tilde{x_1} = l_2 \\ \tilde{x_2}^T l_2 = 0 \end{cases} \tag{10}$$

This implies

$$\tilde{x_2}^T F\tilde{x_1} = 0 \tag{11}$$

we also know the local ray direction $\bar{\mathbf{x}}_1 = K^{-1}\tilde{\mathbf{x}}_1$. By looking at the diagram, we can deduce further

$$\bar{x_2} = R(\bar{x_1} - t) \tag{12}$$

. We need to add the rotation matrix because $\bar{x_2}$ origin is not the same as $\bar{x_1}$, therefore a rotation required.

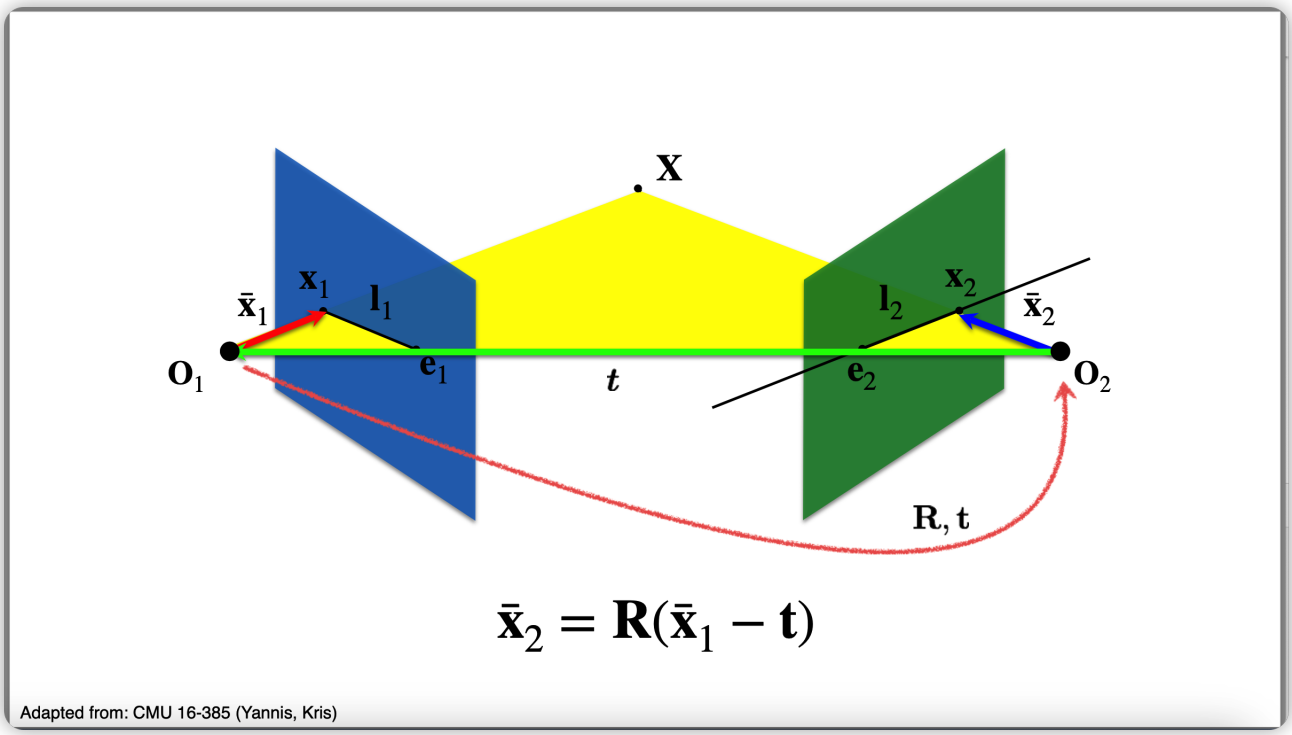

Figure 6: epipolar geometry formal way

Since

$$\bar{x_1}^T(t \times \bar{x_1}) = 0 \tag{13}$$

which means

$$(\bar{x_1} - t)^T(t \times \bar{x_1}) = 0 \tag{14}$$

Combining (13) and (12), we obtain

$$(\bar{x_2}^T R)(t \times \bar{x_1}) = 0 \tag{15}$$

$$(\bar{x_2}^T R)([t]_\times \bar{x_1}) = 0 \tag{16}$$

$$\tilde{x_2}^T(K_2^{-1} R[t]_\times K_1^{-1})\tilde{x_1} = 0 \tag{17}$$

$$\tilde{x_2}^T F\tilde{x_1} = 0 \tag{18}$$

Using (18) and (9), we can find l_2 where all the corresponding image point lying along this vector.

3.4 Finding correspondence

What if we don't know where the location of camera? What if we don't know the rotation relationship between two images?

Then we need to find correspondence: Match features between each pair of images. We can use SIFT for keypoint detection and descriptor computation.

3.5 Eight-point algorithm

Given (18), but now we have eight corresponding points.

$$\begin{pmatrix} x_1x_2 & x_1y_2 & x_1 & y_2x_2 & y_2y_2 & y_2 & x_2 & y_2 & 1 \end{pmatrix} \begin{pmatrix} F_{11} \\ F_{12} \\ F_{13} \\ F_{21} \\ F_{22} \\ F_{23} \\ F_{31} \\ F_{32} \\ F_{33} \end{pmatrix} = 0 \tag{19}$$

We obtain

$$Wf = 0 \tag{20}$$

where $W \in \mathbb{R}^{8\times 9}$ By determine the null-space of W, we can determine f up to scale. Then if K_i is known, we can then back out R and t. If not, additional constraints is required.

3.6 Bundle adjustment

What if we have multiple view? The answer is we could use boundle adjustment.

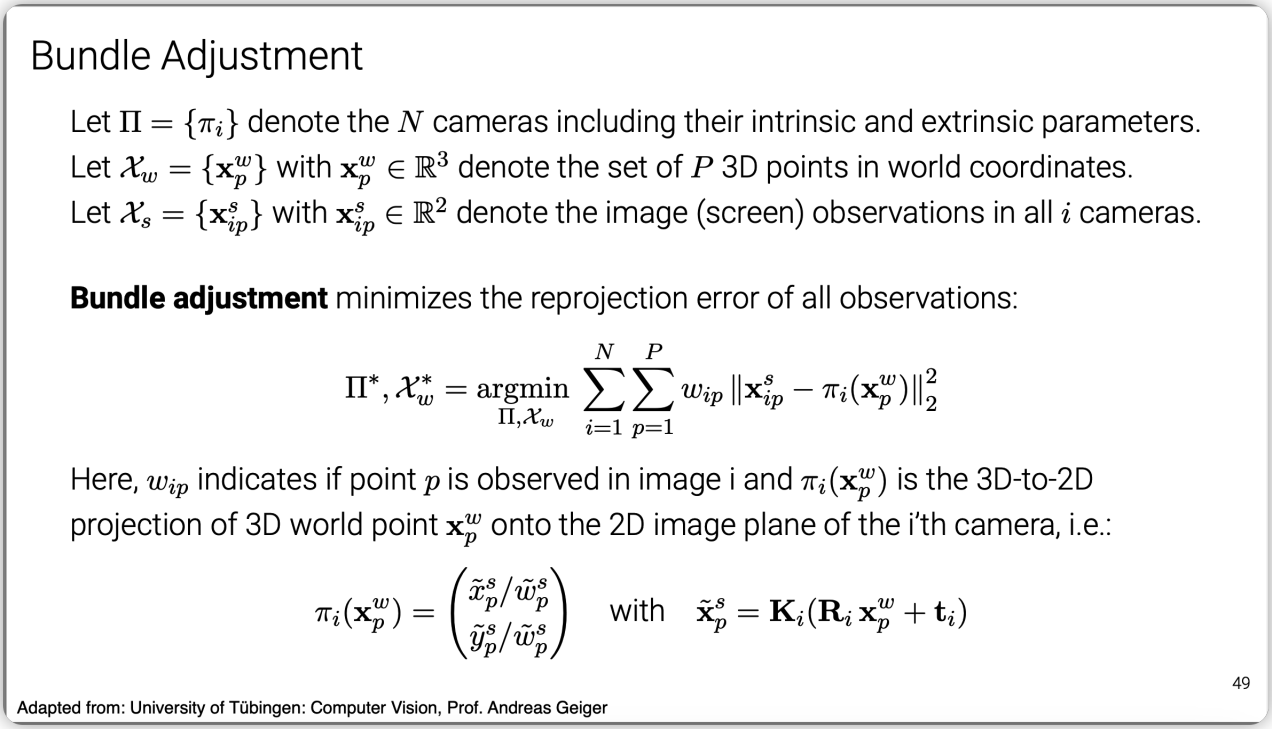

Figure 7: Bundle adustment formula

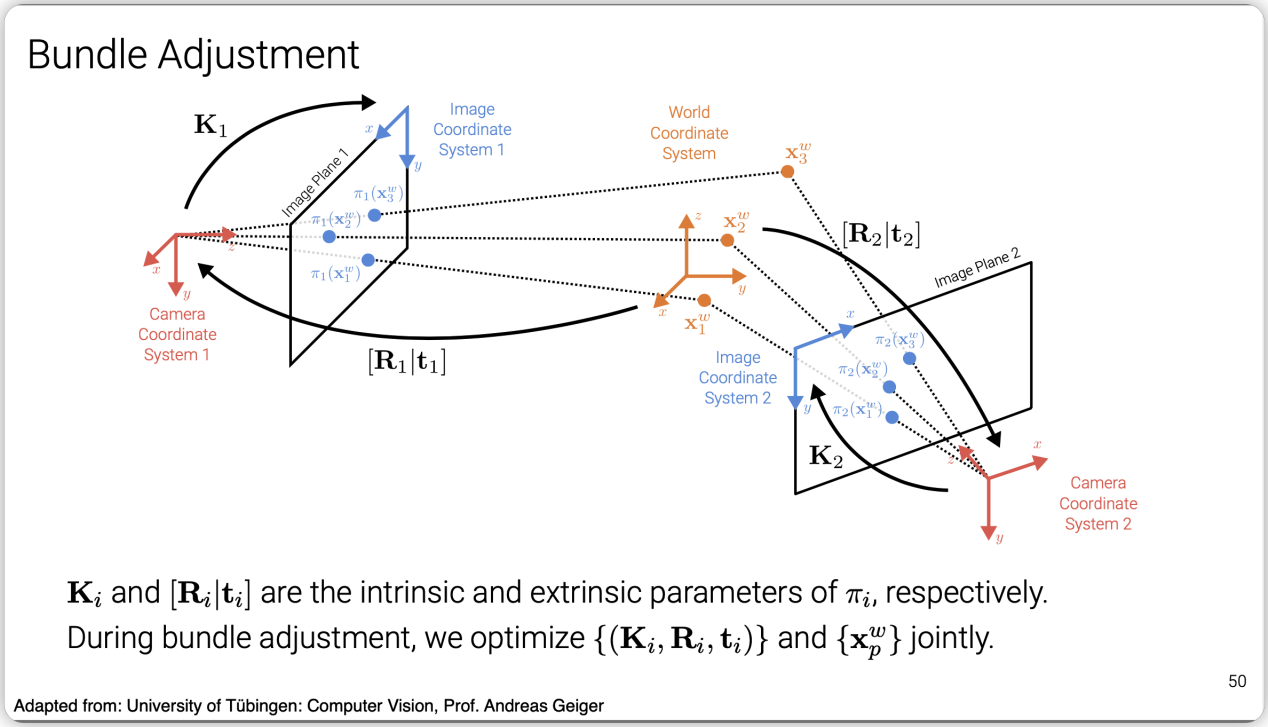

Figure 8: Bundle adjustment diagram

As shown in the diagram, we are minimising all the points p in world coordinates for all the cameras.

3.6.1 Challenge in initialisation

1. The energy landscape of the bundle adjustment problem is highly non-convex
2. A good initialisation is crucial to avoid getting trapped in bad local minima
3. incremental bundle adjustment initialises with a carefully selected two-view reconstruction and iteratively adds new images/cameras since initialising all 3D points and cameras is difficult.

3.6.2 Challenge in optimization

1. Given millions of features and thousands of cameras, large-scale bundle adjustment is computationally demanding
2. Since not all points are observed in every camera, the problem is sparse.

3.7 What has change since deep learning?

For any topic which is related to differential render, we only use Colmap. Any task which is related to underlying 3D scenes, reconstruct 3D scenes.

3.7.1 monocular depth estimation

Firstly it build a convolution neural network, take an input image and map every pixel of the image to a depth map. Notice it doesn't using multi-view geometry. It uses dependence, eg. a car should be some size relative to the background which means it should be further away. These ideas are passed to a convolution neural network. Notice these are supervised learning because we are training it with annotated ground truth depth map. If there is a way to find out the depth map only looking at the video (unsupervised learning).

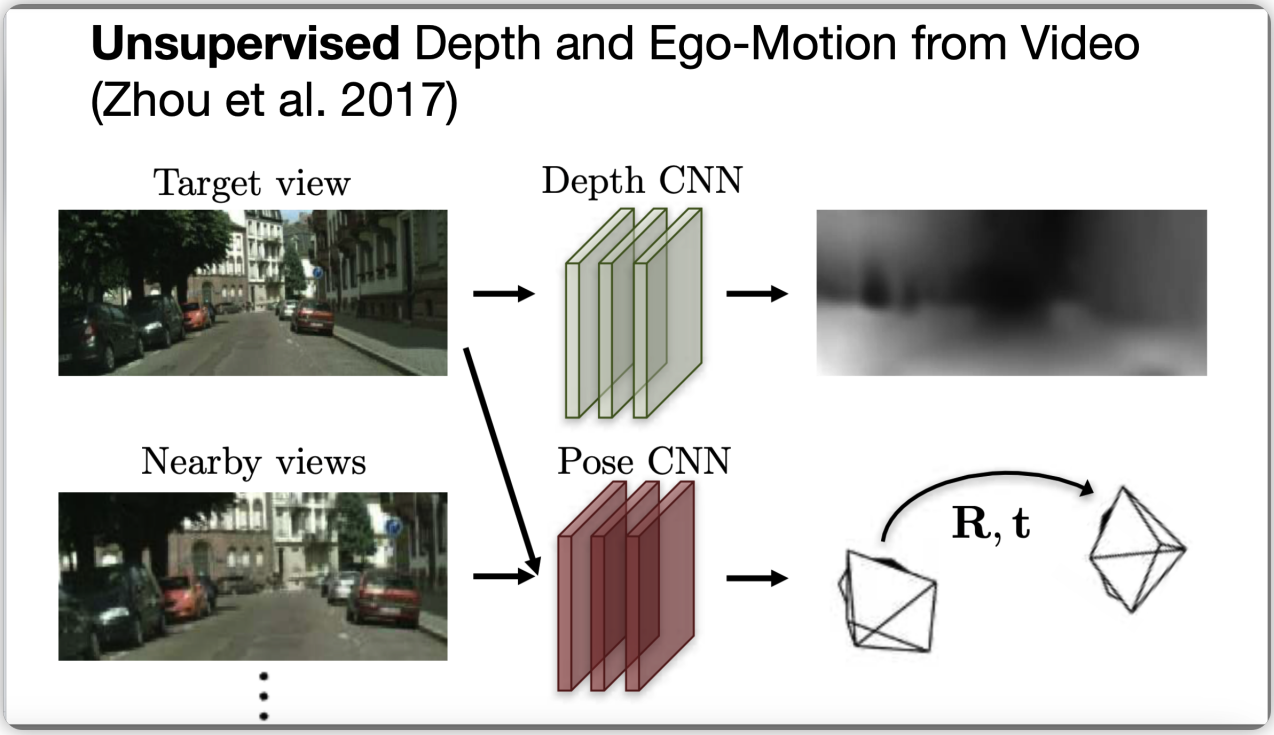

Figure 9: unsupervised depth and ego-motion from video

The idea is simple. As shown in figure 9, we firstly put a target view into the depth CNN, usually it will give us nothing. Then we put a close view picture with the original target view picture into the CNN. Then it will give us the R, t.

Then take the two camera pose picture with the predicted depth and ask one of them to twist into the view of the other camera. Then we could compare the picture colour that it has been converted back into 3D scene with the ground true colour we have captured. This will gives us the re-projection error. Notice this only work for picture which static and it is domain specific. Synthetic data is not necessary for this problem because CNN is usually good at over-fitting, which means it might be extremely good for the synthetic data but not with picture in real life.

4 Scene Representations

4.1 2.5D representation

Firstly, what is 2.5D representation? An example of 2.5D representation is the depth map I have mentioned in previous chapter. It is different to 3D because it doesn't capture the 'full' 3D structure.

4.1.1 Normal Maps

Normal maps is used to capture surface normal. It maps every pixel to the orientation of the pixel.

$$\mathcal{D} : \mathbb{R}^2 \to \mathbb{R}^+ \qquad \mathcal{N} : \mathbb{R}^2 \to \mathbb{R}^3 \tag{21}$$

$$\mathcal{D}(\mathbf{x}_{pix}) = d \qquad \mathcal{N}(\mathbf{x}_{pix}) = \nabla_x \mathcal{D}(\mathbf{x}_{pix}) = \mathbf{n} \tag{22}$$

Surface normal is sometimes more efficient to measure depth. Suppose we have an elephant, I shrink the elephant so that it fits within a bathroom background. However, in this case, depth maps doesn't work. It compare the size of elephant with a size of bathroom in real life to predict how far away the elephant is. Since the size of elephant is different to the original ones, therefore, this doesn't work anymore.

4.2 Surface representations

In order to capture the 3D representation of an object, we can use many method, one of them is points cloud.

4.2.1 Points cloud

A point cloud (figure 10)is a discrete set of data points in space. The points may represent a 3D shape or object. Each point position has its set of Cartesian coordinates (X, Y, Z). It measure many points on the external surfaces of objects around them.

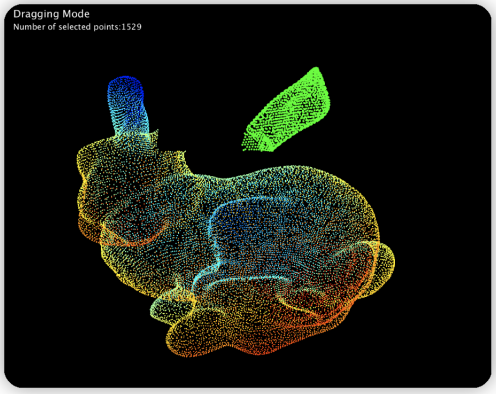

Figure 10: Points cloud

The disadvantage of points cloud representation is they are a group of unordered set of points. That means if we want to find the neighbourhood of some certain point, it require extra computation such as nearest neighbour to accomplish.

It is also difficult to fit a point cloud with a fully connected layer. There are several reasons. Dense layer do not consider the spatial arrangement between points. It also require a fixed-size input, but point clouds can have varying numbers of points. Point clouds are invariant to permutations of points, meaning that the order of points should not affect the network's predictions, but dense layer do consider this.

We also doesn't have any idea about the surface normal for points cloud, so assumption must be made.

4.2.2 Mesh

Meshes are different. We have vertices with edges to connect the points. Therefore, surface have been created. However, one thing to notice is that the way we connect the vertices matters. Therefore, we usually restrict to Triangular meshes which is also super easy for rendering. This means we can create a 'watertight' surface.i.e. if you filled the mesh with water, nothing would leak out.

4.2.3 Parametric surfaces

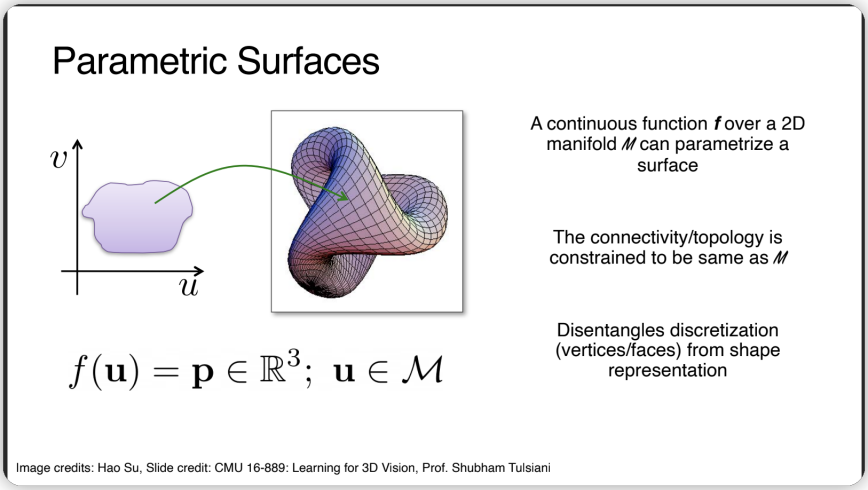

Figure 11: Parametric Surfaces

As shown in figure 11, we can create a function to map from a 2D shape to a 3D function. We also could train a supervised neural network to do the things for me. On the other hand, just like the mesh, when you create an

3D object by using parametric surfaces method, it is hard to query whether a given arbitrary point lies inside or outside the 3D object.

4.3 Field Parameterizations

Definition 4.1 (Field)**.** *Field is a physical quantity represented by a scalar,vector, or tensor, that has a value for each point in space and time.*

The world is a field. We could use a 3D function map to map each space-time coordinate to the quantity (property) that point contains. All property in world can be describe by function in space time.

4.3.1 Occupancy field

In the field representation, how do we represent surface? Since we know in surface representation, the vertices always lies on the surface. First method, we could define a occupancy function, shown in the figure (12) below. We could define the surface as when the occupancy function evaluate to 0.5.

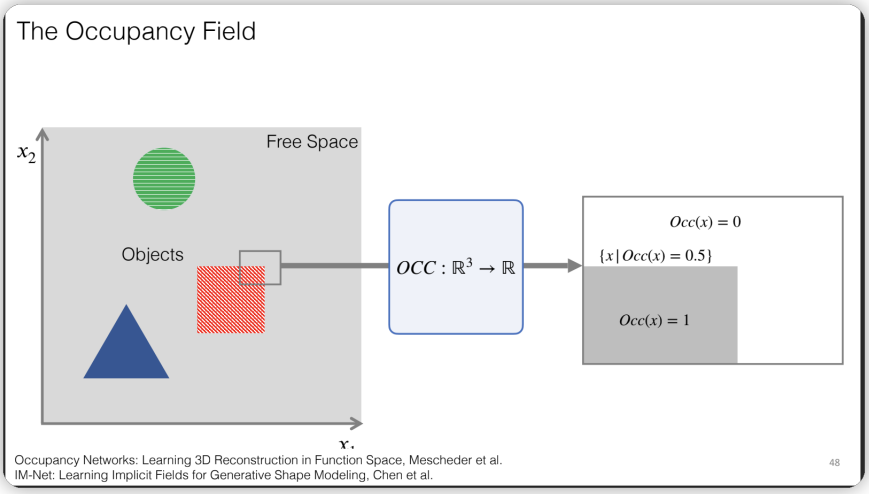

Figure 12: Occupancy field

4.3.2 Signed Distance Field

This is just another function which is used to defined distances. Notice we defined the surface are at 0.

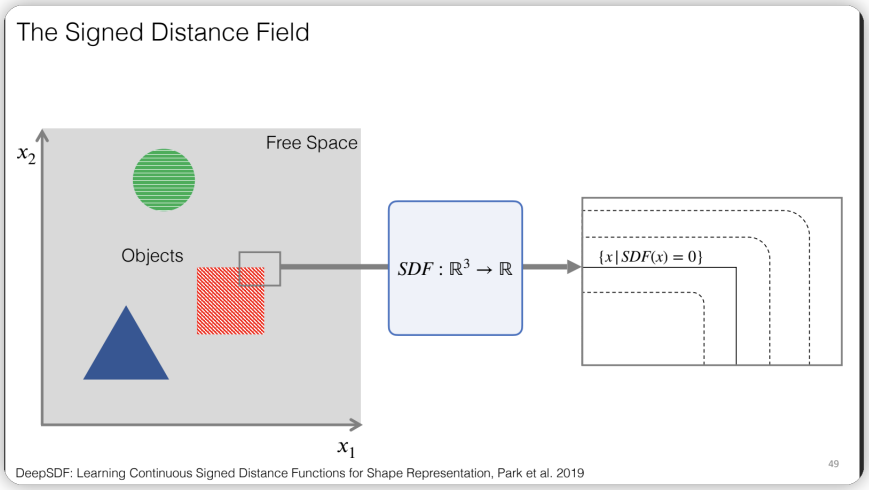

Figure 13: Signed distance field

Now with these function,it is easy to defined whether the point is inside or outside the shape. However, it is hard to find the surface first! We could use marching cube algorithm to exhaustively find the surface of the object by trying all possibilities.

4.4 Parameterised Fields

Now the question comes to how could we parameterise the occupancy function.

4.4.1 Voxel grids

This is the method which we draw grids/cubes around the object and within each cubes it takes the 3D coordinate as input and find out the feature within that 3D coordinate.

For example, shown in figure (14), for each grids, we store some feature which is related to that grid. It is like a lookup table.

What need to be caution is that we don't store every points. We only store some value in the corner point. Then the question comes what values we obtain if the point is in the middle of the gird. We could use linear interpolation with some kernel to querying continuous values.

For d spatial dimensions and resolution n, the memory grows $O(n^d)$. However, it is very easy to query any point. It is also convenient processing as it exposes locality which means I could run convolutions with it.

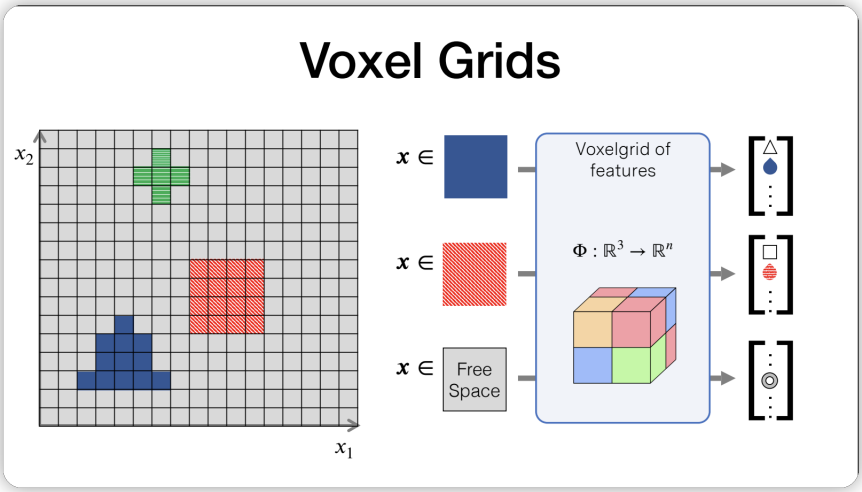

Figure 14: Voxel Grids

The solution to reduce memory requirments is to use a octrees. If the space is empty, we could have less grids. However, this method uses the fact that if there is an object in that grid, we need more grid. The sub-division is not differentiable which means neural network fail to produce an octrees.

4.4.2 Neural fields

It took a 3D points as input and output the property of that given point. For example, one neural network could be used for produce the occupancy function of the given point. One way we could do to find it is create a supervised neural network. Unlike voxel grid, neural network will dynamically allocate more weights to the part of the scenes which requires more detail.

One important neural network called Sinusoidal Representation networks uses sin function as activation function which works very well for task shown in figure (15).

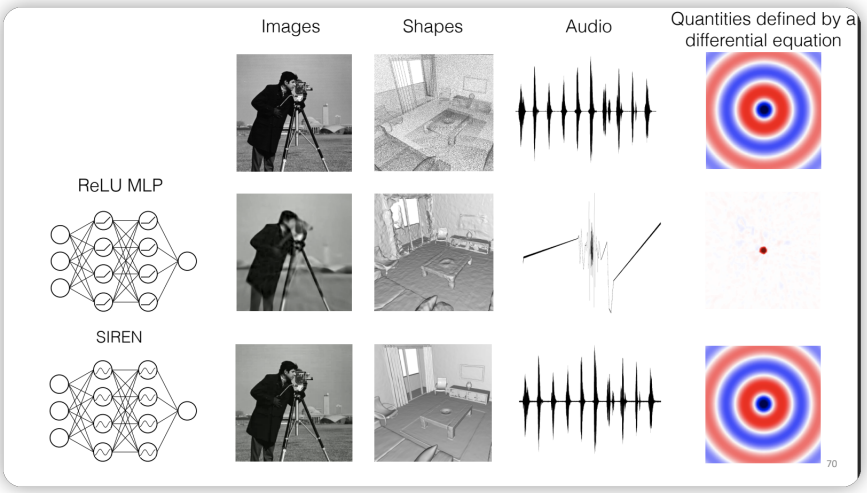

Figure 15: SIREN

4.4.3 Relation of kernel regression with Neural fields

Why does relu works so much worse than siren function?
The kernel function is a critical element of kernel machines. It computes a measure of similarity or a dot product between two data points in the original feature space. It makes predictions on a point x by interpolating labels y_i in the training set according to pariwise weights between x_i to x as measured by a kernel function:

$$f(x) = \sum_{i=1}^{n} (\mathbf{K}^{-1} y)_i k(x_i, x) \tag{23}$$

where $\mathbf{K}_{ij} = k(x_i, x_j)$, k is kernel function with $k \geq 0$ Recently, with the assumption to make the neural network infinitely wide, then the neural network will make the prediction according to such kernel machine. That means the test-time predictions are not 'learned' but interpolate by the training set.

Neural fields are

1. Storage memory does not grow with spatial resolution or number of spatial dimensions.
2. Slow sampling: Each query takes a forward pass through the neural net. Means GPU-memory intensive forward passes.
3. Does not expose locality: Can't identify set of parameters / direction in parameter space that encodes particular spatial location.
4. Inconvenient processing: cannot run convolutions.

4.5 Hybrid parameterizations

We firstly have our voxel grid. If I want to query a 3D coordinate, I firstly query the voxel grid. Instead of storing occupancy, we store some deep learning feature about that point. Then I give such value to the neural network and ask the neural network to predict the feature. In this case, for each voxel, I will have a neural network. By this arrangement, I will have store the local geometry.

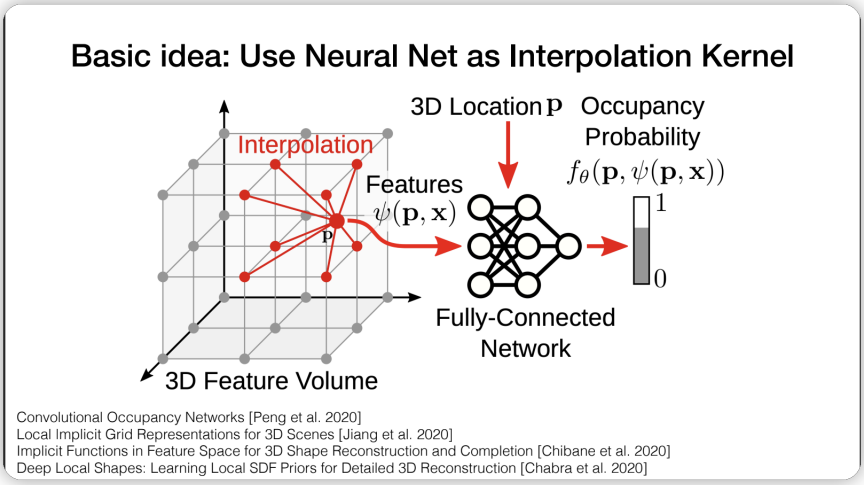

Figure 16: Hybrid parametrisation basic idea

Another idea is instead of define a voxel grid, I define a 2D ground plan. For each points, I project the point to the ground plan, I will get the feature which is stored in that location. Then the neural network decode that feature with the height also as input. This means I have factor one part into a discrete representation and the other part is still store in the neural field. This method is like chooing a trade off between discrete representation and neural field.

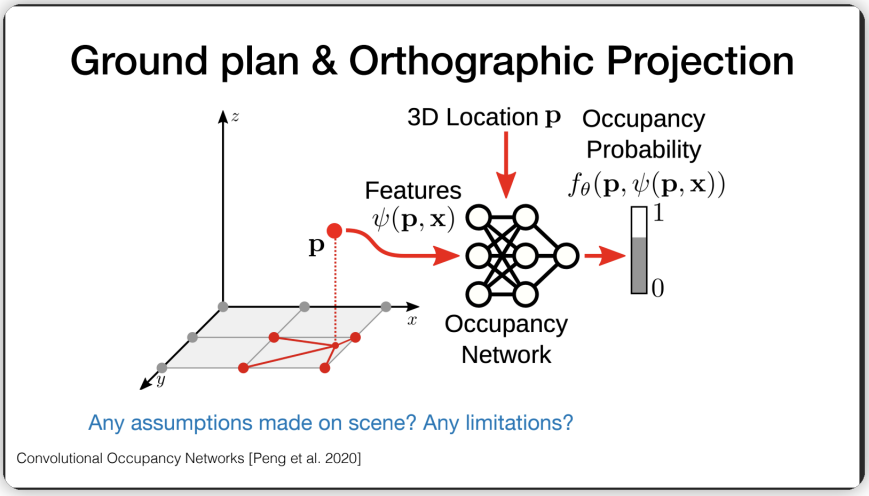

Figure 17: ground plan

4.5.1 Parameterised unbounded scenes

Suppose I have a scene which contains unbounded scene. Then I could use a non-linear function to map the infinity to some finite value and so on. This allows us to turn the input into discrete representation as before but with lower resolution for big values.

5 Light transportation

5.1 Render equation

Firstly what is Rendering? Rendering in computer graphics refers to the process of generating a 2D image or animation from a 3D model or scene. The primary goal of rendering is to simulate the interaction of light with objects in a virtual environment and produce a realistic or stylized representation of that scene. The rendering equation is the following:

$$\mathbf{L}_o(\mathbf{x}, \omega_o, \lambda) = \mathbf{L}_e(\mathbf{x}, \omega_o, \lambda) + \int_\Omega f(\mathbf{x}, \omega_i, \omega_O, \lambda)\mathbf{L}_i(\mathbf{x}, \omega_i, \lambda)(\omega_i \cdot \mathbf{n})d\omega_i \tag{24}$$

5.1.1 Radiance

Before I unpack the definition of the rendering equation, let's discuss what is radiance. Radiance is a fundamental quantity of light. It is density of photons at a point, travelling in the same direction, at given wavelength. It is measure by energy per time, per unit area, per unit solid angle, per wavelength.

1. Radiance along an unblocked ray is constant.
2. Response of sensors is proportional to of visible surface.
3. Radiance of reflected light is proportional to that of incoming light:

$$\mathbf{L}(\mathbf{x}, \omega_{\mathbf{O}}, \lambda) \propto \mathbf{L}(\mathbf{x}, \omega_{\mathbf{i}}, \lambda) \tag{25}$$

5.1.2 BRDF

It is the constant of proportionality of:

$$f(\mathbf{w}_i, \mathbf{w}_o) = \frac{\textbf{outgoing light in direction } \mathbf{w}_o}{\textbf{incoming light in direction } \mathbf{w}_i} \tag{26}$$

It describes how a material reflects light and it satisfies the Helmholtz reciprocity $f(\mathbf{w}_1, \mathbf{w}_2) = f(\mathbf{w}_2, \mathbf{w}_1)$

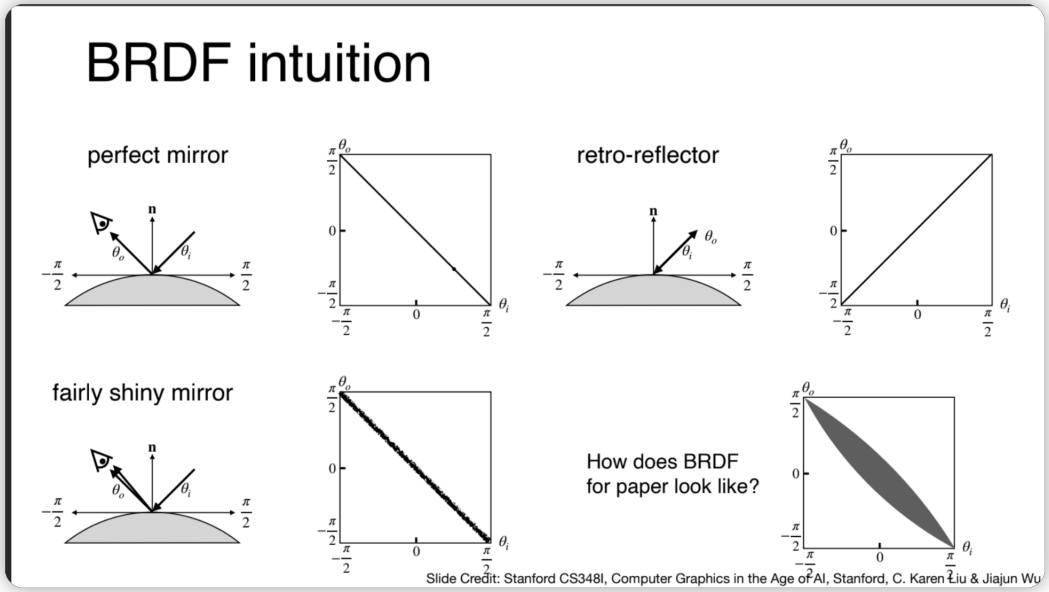

Figure 18: BRDF

5.1.3 Angle Weight of photon density

Suppose we have two point light source shine on one single point with different angles. Which incoming light results in more photon density at the surface? For example, in figure (19), the top one will be denser because the area is less.

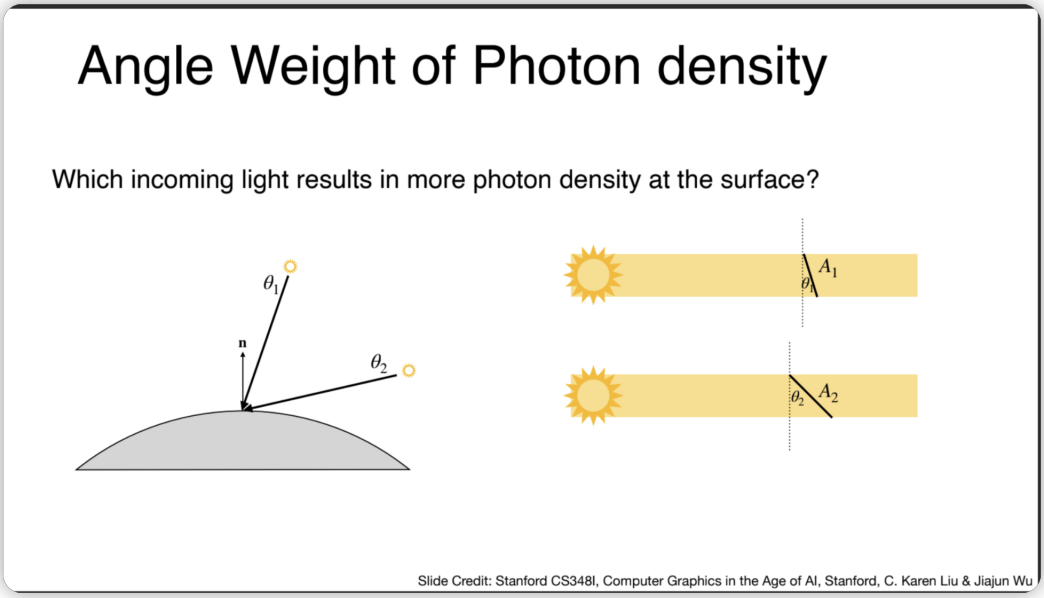

Figure 19: Angle weight of photon density

Back to the rendering equation,

1. $\mathbf{L}_O(\mathbf{x}, \omega_o, \lambda)$ is the outgoing radiance in ω_o.
2. $\mathbf{L}_e(\mathbf{x}, \omega_o, \lambda)$ is the emitted radiance
3. $f(\mathbf{x}, \mathbf{w}_i, \omega_O, \lambda)$ is the fraction of incoming light that is reflected in ω_o
4. $\mathbf{L}_i(\mathbf{x}, \omega_i, \lambda)$ is the incoming radiance
5. $(\omega_i \cdot \mathbf{n}$ is the angle weight of photon density
6. The first term on the right is the emited light source. That measure how much light source that point is producing. Then the second term is integrated over a sphere that measure all reflected light source.

At the end, we could have three of these equation for RGB. Each equation will tell us how many such colour our camera should receive.

5.2 Phong model with rasterization

So how do we render? Before we reach out to this question, we firstly need to look at the phong model. Firstly we assume the material of the surface is Lambertian. That means the light is deflected equally out of the space. Then (24) will become:

$$\mathbf{L}_o(\mathbf{x}, \omega_o, \lambda) = \mathbf{L}_e(\mathbf{x}, \omega_o, \lambda) + \mathbf{C}_{\text{surf}} \int_\Omega \mathbf{L}_i(\mathbf{x}, \omega_i, \lambda)(\omega_i \cdot \mathbf{n}) d\omega_i \tag{27}$$

Further we assume all the light source are point light source. That means:

$$\mathbf{L}_o(\mathbf{x}, \omega_o, \lambda) = \mathbf{L}_e(\mathbf{x}, \omega_o, \lambda) + \mathbf{C}_{\text{surf}} \sum_i \mathbf{L}_i(\mathbf{x}, \omega_i, \lambda)(\omega_i \cdot \mathbf{n}) \tag{28}$$

5.2.1 Diffuse light

We can then conclude the diffuse light is given by:

$$\mathbf{C} = \mathbf{C}_{\text{surf}} \cdot \sum_{i=1}^{N} \mathbf{C}_{l_i} \max(0, \mathbf{n} \cdot \mathbf{l}_i) \tag{29}$$

A diffuse light source is a type of lighting that emits light in multiple directions, scattering it evenly in all directions. This creates soft, uniform illumination without harsh shadows or distinct beams of light.

5.2.2 Ambient light and specular light

We have model all possible light which clearly come from a given light source. However, we will always have some background light source we could not model. That is the reason why we add an ambient term. It approximates the average color of all surfaces in the scene to make up for indirect lights from every direction.

Specular light is a type of light that is reflected off a surface in a specific, concentrated manner, creating a bright and shiny highlight.

Therefore we could add all these three term together, we obtain:

$$\mathbf{C}_{\text{surf}} \cdot \mathbf{C}_{\text{ambi}} + \mathbf{C}_{\text{surf}} \cdot \sum_{i=1}^{N} \mathbf{C}_{l_i} \max(0, \mathbf{n} \cdot \mathbf{l}_i) + \sum_{i=1}^{N} \mathbf{C}_{l_i} \max(0, e \cdot r_i)^p \tag{30}$$

5.3 Ray tracing

The phong model doesn't do shadows, refraction, direct light or inter-reflection. The very standard ray tracing follow the following procedure:

1. For every pixel (x,y): create ray R from eye through (x,y)
2. For each object O in scene, if R intersects O and is closest so far, record this intersection.
3. Shade pixcel (x,y) based on nearest intersection

The intersection actually happens as following: Given a mesh, we check whether the ray passing through (x,y) intersects each triangular plan exhaustively.

5.3.1 Reflection

We can generate reflection by doing the following: Shoot a ray to the first object the ray intersect. Then reflect the ray by looking at its BRDF. Find the most likely direction the ray could be reflected to. For each intersection, we could find out what light has been contributed to that point.

In order to improve the realistic of reflection, instead of just reflect one single ray, we could reflect several ray at the same time.

5.3.2 Shadow rays

To find shadows, we could do the following: For every intersection point the ray intersects, we could check whether the light source ray will be blocked by any object on its way or not. If yes, then we just not going to use that light contribution.

$$\mathbf{C} = \mathbf{C}_{\text{ambi}} + \sum_{i=1}^{2} \text{visible}(\mathbf{l_1}, \mathbf{x}) \cdot \mathbf{C}_{l_i} \cdot \mathbf{C}_{\text{surf}} \max(0, \mathbf{n}) + \mathbf{C}_{\text{spec}} \tag{31}$$

Notice this is how we perform hard shadows. To find the soft shadow, we could do the following. For the area light(The light source is not ususally a point light source) intersect with the object surface, we shoot a few rays to different direction to check whether the light is blocked or not. Then we could take the mean of them.

5.4 Path tracing

Path tracing simulates the entire path that light takes from the camera to the light sources. It considers the full path of light, including indirect bounces.

1. Send a ray through a pixel, find the first intersection on the surface of an object, and integrate over all the illuminance arriving to the intersection point.
2. Compute illuminance by recursively projecting a ray from the surface to the scene in a bouncing path that terminates when a light source is intersected.
3. The light is then sent backwards through the path and to the output pixel.

6 Differentiable Rendering

In the previous few chapter, we have discuss the idea of rendering which is the method of transfer 3D Scene to images. Now we are looking at the reverse process, how to transfer from images to 3D scene as well as camera poses.

As we have discussed in multi-view geometry, finding out camera poses is done by method such as bundle adjustment. Now the problem become given the camera poses with the images, how could we reconstruct the 3D scene.

Firstly, we have some scene representation with randomly initialised parameters. Then we will use the differential render method such that it will produce rendered images. Now we compare the rendered images with the ground truth images. We can optimise the rendered images by using SGD to twist the parameter in the scene representation. That's why we need to use differential render.

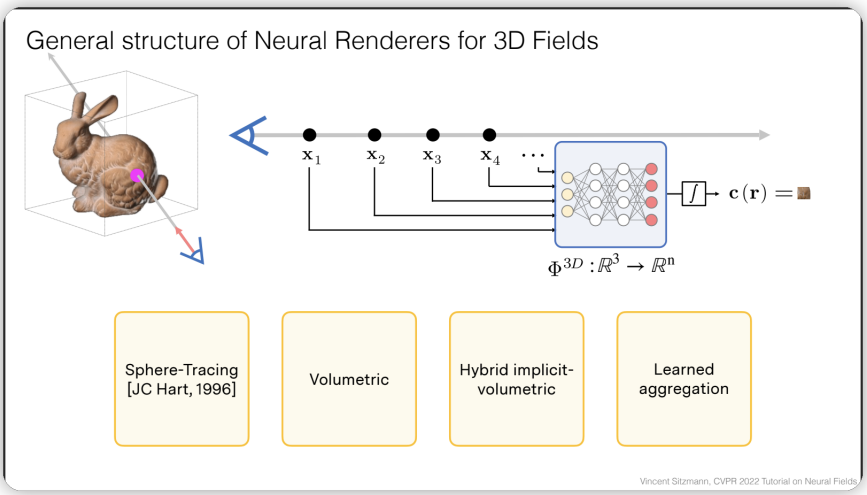

Figure 20: General structure of NR

As shown in figure 20, suppose we have a bunny and a neural field representation. We shoot a ray to the object, and we need to sample points along the ray. Then there will be some function which use the sample point and given out a colour in the end.

6.1 Sphere tracing

Firstly we place the camera and shoot the ray to the object. We want to find when the point intercept with the geometry of the scene. Then the later question will be how to find out the colour.

Firstly how could we parameterize scene geometry?

We could use the signed distance function. We could set up a neural network with a 3D coordinate as input and a scalar as signed distance function output.

Now suppose we have our the solution of our signed distance function. **How do we find the intersection?**

We start with a point on the ray which is close to our camera. We know its signed distance function, which means we just need to advance along the ray according to such value. It guarantees we don't step into the surface. Then we need to iterate this process because the ray direction is fixed which means moving along the ray may doesn't mean we are on the surface. For every ray, we can repeat this process. To find the colour, since we are already on the surface of the object, we can sample out the colour easily.

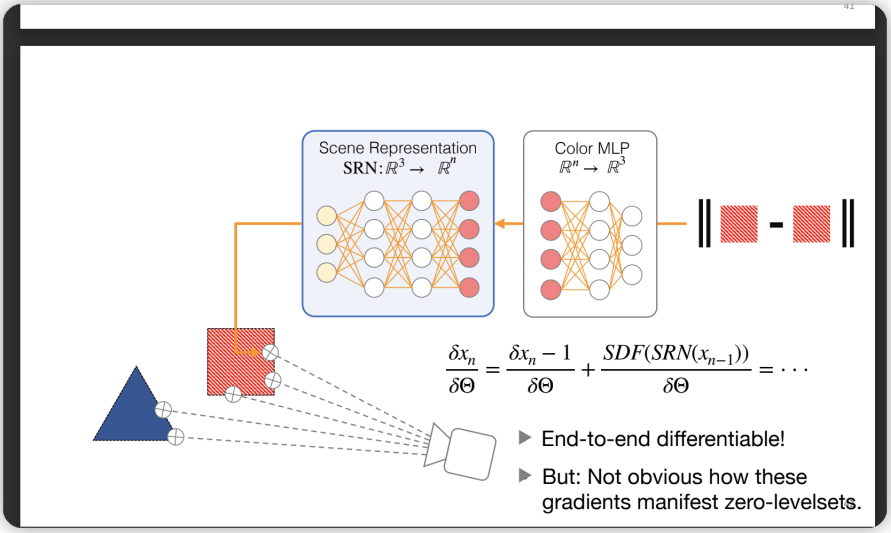

Figure 21: Shpere tracing

We can compute the gradient of the colour easily. However, the gradient of the signed distance function is not easy to define. In the start, I randomly initialise the signed distance function. Suppose after initialisation, I move the initial point closer to the surface along the ray. I can find its colour and compare to the ground truth colour and compute the gradient. If I have another ray with another point close the point I just sampled. If it need to get the colour right, it is trying to get closer to the point I just sampled. That is why I will have a gradient in this case. If we run the above algorithm to find the intersection too much, we could have problem such as vanishing gradient and exploding gradient.

6.2 Volume rendering

Firstly we assume the scene is consist of infinitely small particles and then let's shoot a ray through the camera into the scene. If a ray traveling through the scene hits a particle at t, we return its color c(t). We denote the probability that ray stops in a small interval around t is $\sigma(t)\ dt$. Then we can model our field as the following:

$$\Phi : \mathbb{R}^3 \rightarrow \mathbb{R}^3 \times \mathbb{R}^+ : \Phi(x) = (\sigma, c) \tag{32}$$

To determine if t is the first hit, need to know T(t): probability that the ray didn't hit any particles earlier. T(t) is called Transmittance.

$$T(t + dt) = T(t)(1 - \sigma(t)dt) \tag{33}$$

This is because the probability of we don't have a hit until t+dt is equivalent to say we don't have a hit until t as well as around dt.

$$T(t + dt) = T(t)(1 - \sigma(t)dt) \tag{34}$$

$$T'(t) = -T(t)\sigma(t) \tag{35}$$

$$\log T(t)|_a^b = -\int_a^b \sigma(t)dt \tag{36}$$

$$\frac{T(b)}{T(a)} = \exp\left(-\int_a^b \sigma(t)dt\right) \tag{37}$$

$$T(t) = \exp\left(-\int_0^b \sigma(a)da\right) \tag{38}$$

We can define T to be the distance until the first accident in a poison process. T(t) means until t, there will be no hit before it. Therefore, this follows the exponential distribution. $1 - T(t)$ will be the equivalent to $P(T \leq t)$ which is the probability that ray hits something before reaching t.

Therefore the expected color returned by the ray will be

$$\int T(t)\sigma(t)c(t)dt \tag{39}$$

Notice we estimate the probability density function of T(t) to $T(t)\sigma(t)$.

Since the function is complicated so solve, we are going to numerical approximating. We use quadrature to approximate the nested integral, splitting the ray up into n segments with endpoints $\{t_1, t_2, \cdots, t_n\}$. How do we choose these points along the ray? Firstly we define the near plane and the far plane. These are the plane we start and stop sampling, because we will always have a large empty space which doesn't require sampling. We assume volume density and color are roughly constant within each interval. (39) could be modify to the following:

$$\sum_{i=1}^{n} \int_{t_i}^{t_{i+1}} T(t)\sigma_i c_i \, dt \tag{40}$$

However notice T(t) is not a constant value.

$$T(t) = \exp\left(-\int_{t_1}^{t_i} \sigma_i ds\right) \exp\left(-\int_{t_i}^{t} \sigma_i ds\right) \tag{41}$$

$$T(t) = \underbrace{\exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right)}_{T_i} \exp(-\sigma_i(t - t_i)) \tag{42}$$

Therefore, (39) becomes

$$\int T(t)\sigma(t)c(t)dt \approx \sum_{i=1}^{n} \int_{t_i}^{t_{i+1}} T(t)\sigma_i c_i \, dt \tag{43}$$

$$= \sum_{i=1}^{n} T_i \sigma_i c_i \int_{t_i}^{t_{i+1}} \exp(-\sigma_i(t - t_i)) \tag{44}$$

$$= \sum_{i=1}^{n} T_i \sigma_i c_i \frac{\exp(-\sigma_i(t - t_i)) - 1}{-\sigma_i} \tag{45}$$

$$= \sum_{i=1}^{n} T_i c_i (1 - \exp(-\sigma_i \delta_i)) \tag{46}$$

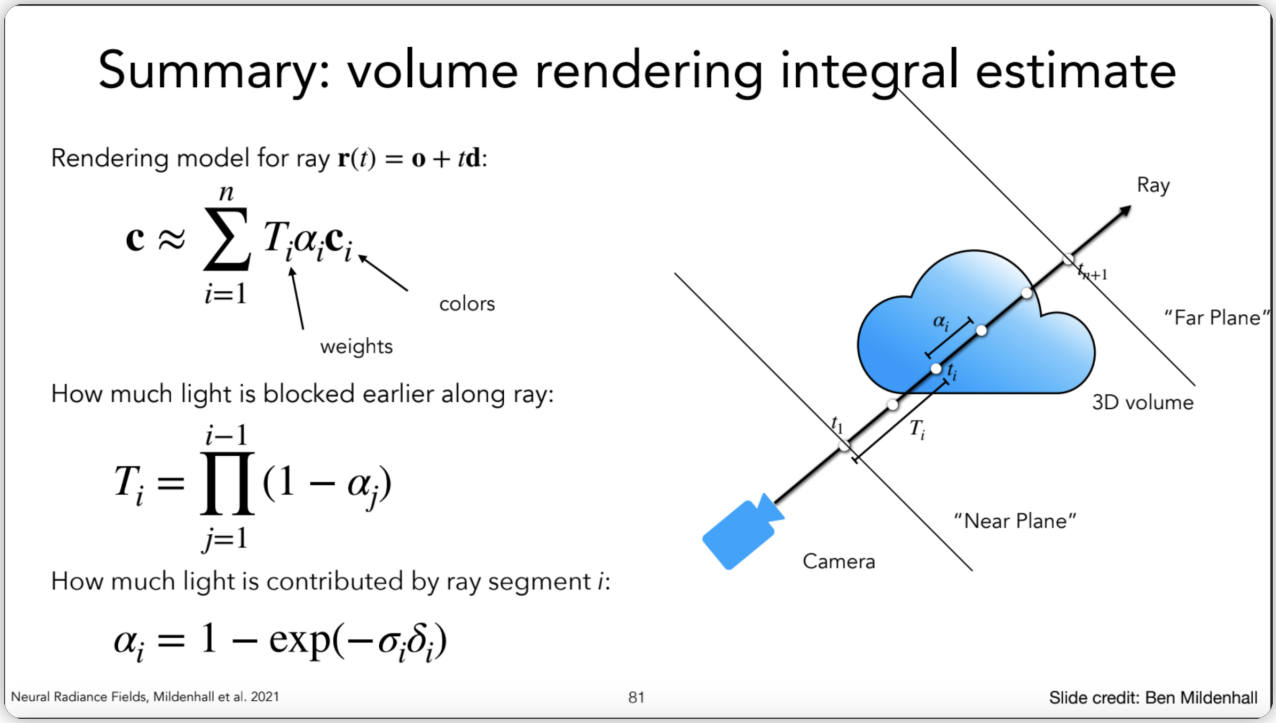

Figure 22: Volume rendering

Even using the differential render, we are still trying to fit the render equation. However, the problem is we have just sampled the 3D coordinate with one single associated colour. We have not added any reflection or

other effect.

$$\mathcal{L}_o(x, w_o, \lambda) = \mathcal{L}_e(x, \lambda) \tag{47}$$

We can add a direction (view dependent), $c = c(x, w_o)$. In this case, we can model the reflection easily, however, it is still not good enough to capture realistic model.

7 Neural Networks as Prior based inference algorithm

Now the problem becomes what if we just have one single image. Or more clearer what if we have lots of images but with a lot of uncertainties. e.g. We only take pictures on one side of the object but not the back side of the object. We could regenerate the image but everything else such as normal map will not work, as shown in figure 23.

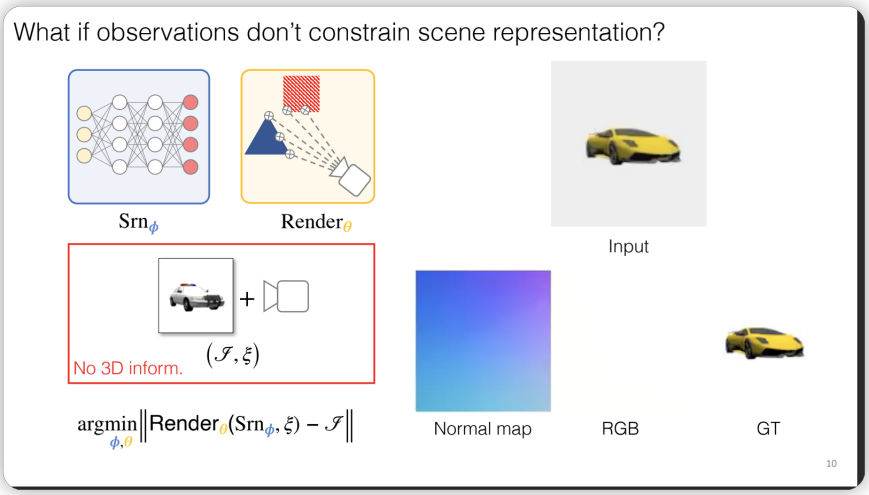

Figure 23: Single image

If we could just use videos to model the actual 3D scene that will be amazing because there are so many images on Youtube and such algorithm will allow us to reform the 3D scene just by videos.

7.1 Auto-encoding

As shown in figure 24, we have a encoder part which down sample the resolution of the image. We hope the latent space will learn the key detail about the image. This is key because now the MLP nows what object we are dealing with. Then we decode it to get back to the original shape.

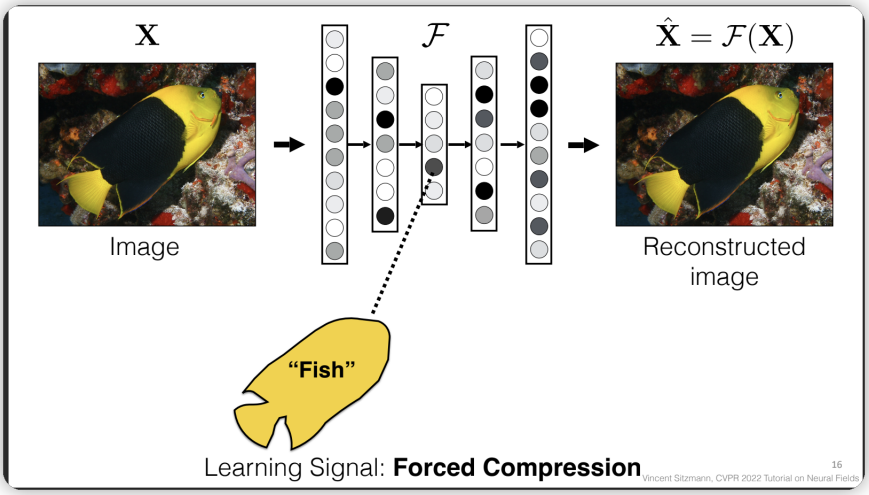

Figure 24: Encoder

How could we generate it to 3D object? We can have a single image and put into a convolutional encoder to obtain a latent code. Notice the encoder will encode every image to a latent code and the latent code is fixed (It will not be changed in during the decoding process). Then we concatenate the output pose(camera pose) and through the convolutional decoder to render a image in different prospective view. We could train this MLP by compare the image with the ground truth image.

However, it doesn't work very well. This is because there are infinite number of possible image-to-image functions to explain training set but only one of them is correct. CNN does not have any 3D inductive bias - works by "matching" 2D features.

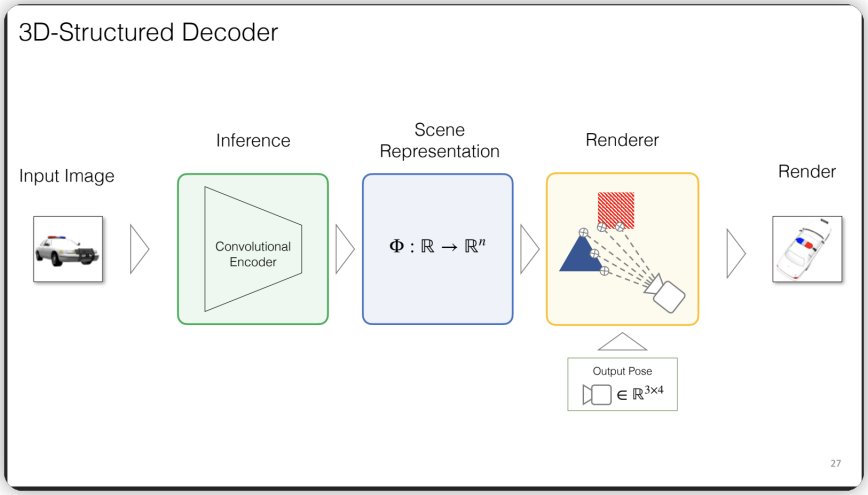

Figure 25: 3D structured decoder

We can use a auto-encoder like model, but instead of using a convolutional network for decoder, we could use a renderer as shown in figure 25.

What happens between the inference and the scene representation?

1. Firstly we will have our latent code by convolutional network. Denoted as $\{z_i\}_{i=1}^n$.
2. We will condition Scene Representation on latent code. Any Scene Representation has parameters, denoted as θ: Voxelgrid has voxel contents, Neural Field (MLP) has weights and biases. We build a neural network which will take latent code as input and produce these parameters.
3. In fact, the Latent Codes are what's actually encoding the scene! What we previously called "Scene Representation" is now simply a way of decoding the latents, just as the 2D convolutions were before.

The end to end training will look like the following: We take two images with camera pose. (All images need to take at the same distance to the object) We pass one to the inference algorithm and produce latent code as scene representation. Using this scene representation, we could render the image with the associated camera pose. Then we compare the image with the ground truth images and modify the loss.

7.2 Auto-decoding

If we have problems with out-of-distribution images, the previous network doesn't work. Instead we could use the auto decoding method. For every example in my training set, we will have a embedding for each of the example. During training, we could use a hypernetwork to code the embedding. This time we are both training the embedding and the decoder. Therefore, the model could capture the 3D structure as we are not just chaning the latent code. At test time, we will only optimse the embedding via SGD to produce the new prosective view of a object. That means my model(decoder/scene representation) will be learned during training time, and freeze during test time.

The difference is instead of using latent code to determine the model, we use latent code to determine the training example and pass it to the model we need to train. Notice for every 3D scene we use the same latent code for auto-decoder network. However, this is different for auto-encoder as the encoder will produce different latent for different images. We basically registered the geometric of the object because it is not a parametric render like the encoding example.

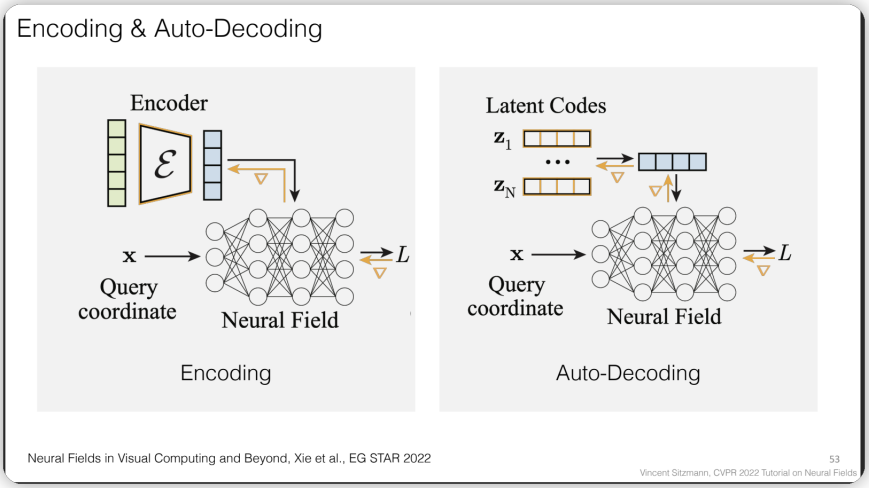

Figure 26: Encoding and Auto-decoding

Notice all we doing is to input a unique latent code for all object. We need to make sure all object have similar structure. e.g cars all have wheels. Then they could be modeled using single latent code.

All we have done is with one-single object. However, such model will not generalise well for multiple number of objects. This is because the dimensionality increases exponentially with number of object. This is also called the cursed dimensionality.

7.3 Local conditioning

We introduce the idea local latent code. We have a discrete data structure been introduced into out networks. The simple idea is we have points cloud as input in this example. Then we encode the point-cloud which contains some 3D structure as shown in figure27 (as a ground plane). Then follow the procedure as before.

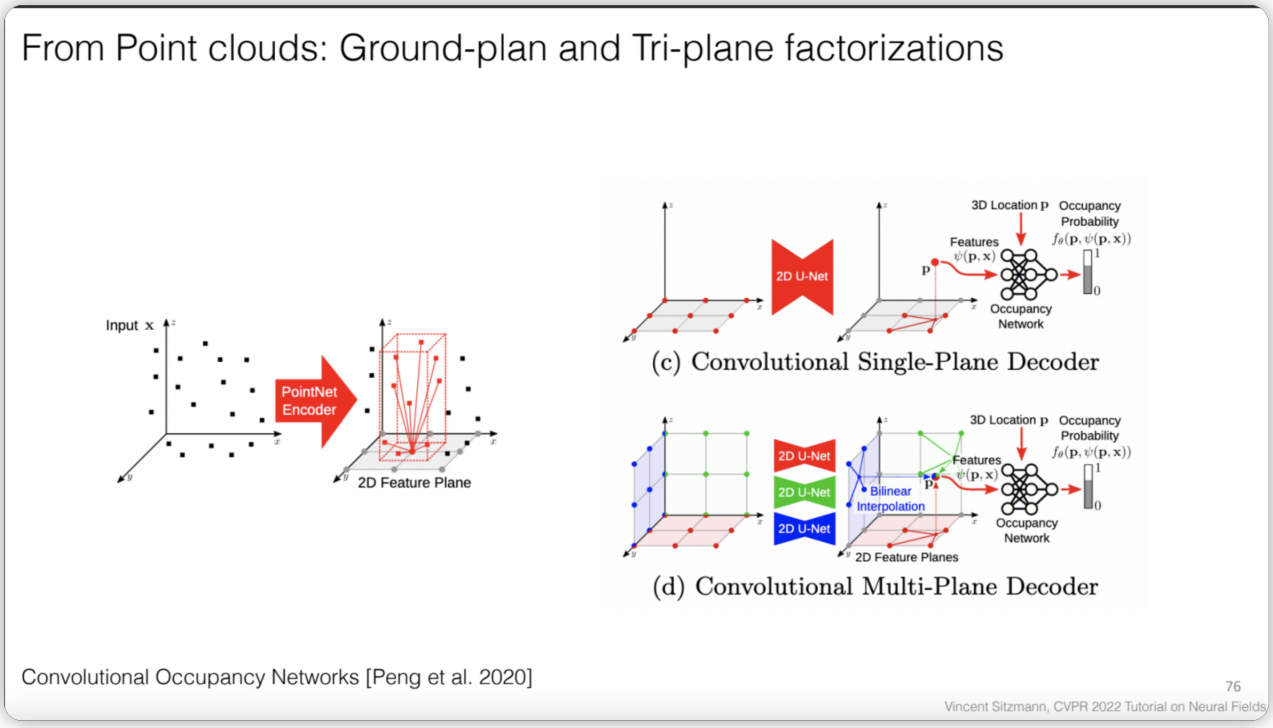

Figure 27: Discrete data structure

During the global case, the latent code need to understand the whole images. However, in this case, we dipatch the 3D geometry. That means each latent code will understand only a part of the images. That gives more flexibility.

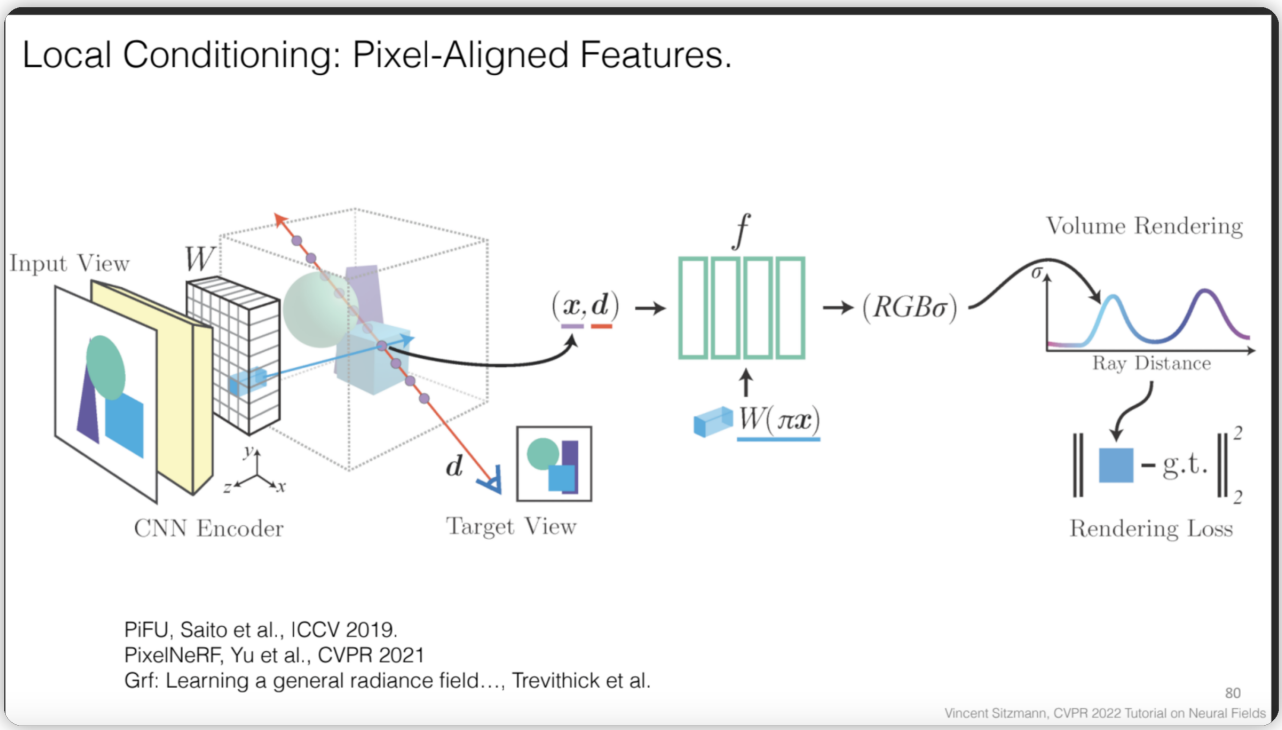

Figure 28: Local conditioning: Pixel-aligned features

Figure 28 shows an example of local conditioning. Firstly we have our input picture, we put it into a convolutional network. This makes sure later we can query the wanted location not the whole image. Then we can do volume rendering like what we did before. By back-propagation to find the rendering loss, we can find the object's RGB just query the local feature through the convlutional network.

7.4 Conditional Ground Plans for Single-Image 3D Reconstruction

We now introduce a new idea called unproject which project from a lower dimension to a higher dimension space. For example, we could project the 1D feature map to a 2D world grid.

Now we could reconstruct a 3D object using the following idea. Suppose I am a 2D image, I could use unprojection method to unproject the 2D image to a 3D scene. Then I could use the 3D scene to project it down to a ground plan. Both of them we could use neural network method.

8 Advance inference

8.1 Light field network

Suppose we only care about the colour of the ray when we do rendering, then what if instead of sample points along the ray, we sample ray. We call this a light field network.

8.1.1 Ray parametrisation

How do we parametrise the ray? For example, we could use the two plane representation. We record the 2D coordinate when the ray intersect the planes (We have a far plane and a close plane). We also have cylindrical representation where it intersect the cylindrical. We also have sphere and Lumigraph. However, they all have disadvantage. For example the 2D plane representation is not 360 degree covered.

Why we don't use the normal ray presentation $(\mathbf{x}, d)$. This is because the position $\mathbf{b}$ of the ray doesn't matter as long as they are along the same direction.

There are also limitation to this model. Suppose we have two different colour ball along the plane. Then the ray will only see the very front ball because the ray is independent to the position of the ray.

Why in this method the ray can represent the 3D structure of the light field? Suppose we have two object shown in figure, then we can use the two plane method but instead this case we use two line. Then we record the coordinate (on the two line) with the associated ray. By plot the s,t coordinate, we obtain a graph on the right. It shows if the point p moves closer to us, the line will move up. Therefore the slope of the line tell us

how far the point is away. Even the representation is not in 3D. It turns out if we have render out enough of the scene which is multi-view consistent, that always mean we have capture the geometry of the scene.

8.1.2 The network structure

As shown in figure, we known instead of having 3D scene representation with 3D render, we now only have one single thing. We now sample the ray which will straight away give us the colour. It is like we already rendered during the scene representation.

The advantage of it is instead of doing thousands of sample points along the ray, we only need to have one single ray which means the process will be extremely fast.

However, if we would like the model to learn prior over light field which will be extremely hard. This is because if we want to discretise the ray, since our ray is 5 dimensional object, it means it is not possible to fit it on any GPU. Therefore, we clearly need to have another way to model the field instead of local conditioning.

8.2 Attention based conditioning

We could use an attention based model to solve such problem. Firstly our query will be our 3D coordinate. Instead of having voxel grid to store the local information, we have a key and a value which store the 3D information of the object (This information could come from a CNN network to produce image embedding. Then the transformer encoder part will produce some latent scene. With these latent scene, I could produce an output colour.

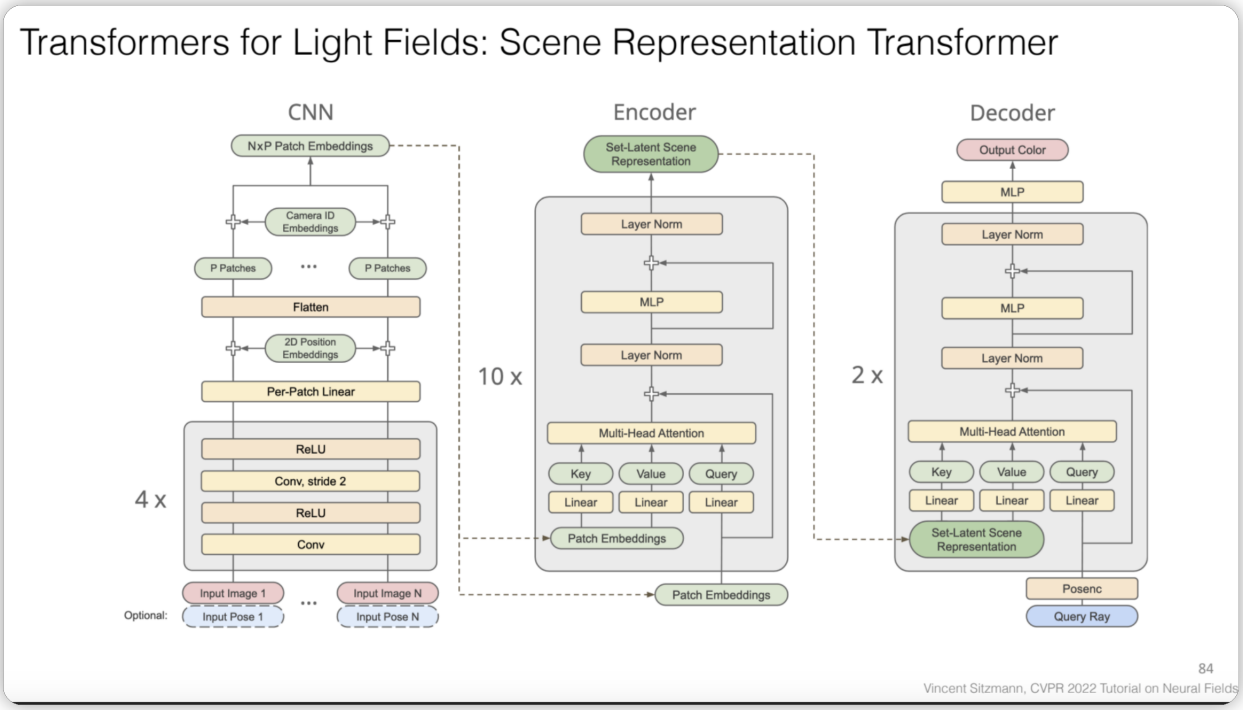

Figure 29: Scene representation transformer

9 Uncoditional generative model

10 Geometric deep learning

One important feature about convolution network is it is "Shift Equivariance". Suppose I am doing a segmented segmentation on a cat image, if I shift the image, the CNN will still work but not the fully-connect neural network. However, CNN are not rotation equivarient. However, the group equivarient CNN is.

A group is a set of elements equipped with a group product, a binary operator, that satisfies the following four axioms:

Definition 10.1. *1. **Closure Property:** For all a, b in the group, the product ab is also in the group, denoted as $ab \in G$.*

2. ***Associative Property:*** *For all a, b, c in the group, the product is associative, i.e., $(ab)c = a(bc)$.*

3. ***Identity Element:*** *There exists an identity element e in the group such that for all a in the group, $ea = ae = a$.*

4. ***Inverse Element:*** *For each element a in the group, there exists an inverse element a^{-1} in the group such that $aa^{-1} = a^{-1}a = e$, where e is the identity element.*

Definition 10.2. *A translation group is a mathematical structure that consists of translations in one or more dimensions and is equipped with a group product, which is typically composition of translations, and it satisfies the following properties:*

1. ***Closure Property:*** *For any two translations T_1 and T_2 in the group, their composition $T_1 \circ T_2$ (i.e., applying one translation followed by the other) is also a translation in the group.*

2. ***Associative Property:*** *The composition of translations is associative. For any translations T_1, T_2, and T_3 in the group, $(T_1 \circ T_2) \circ T_3 = T_1 \circ (T_2 \circ T_3)$.*

3. ***Identity Element:*** *There exists an identity element, denoted as I, which represents the absence of translation. For any translation T in the group, $I \circ T = T \circ I = T$.*

4. ***Inverse Element:*** *For each translation T in the group, there exists an inverse translation, denoted as T^{-1}, such that $T \circ T^{-1} = T^{-1} \circ T = I$, where I is the identity element.*

A translation group is often encountered in geometry and physics when dealing with spatial translations, where elements of the group represent displacements or shifts in space along one or more dimensions.

Definition 10.3. *A representation $\rho : G \to GL(v)$ is a group homomorphism from G to the general linear group $GL(v)$.*
That is $\rho(g)$ is a linear transformation that is parameterized by group elements $g \in G$ that transforms some vector $v \in V$ such that $\rho(g') \circ \rho(g)[v] = \rho(g' \cdot g)[v]$

The reason we need this definition can be considered by the following example. Suppose I have a image of smiley emoji. It can be think of as a function that map from a 2D coordinate to the property of that 2D coordinate. Then we cannot use elements in roto-translation group to functions straight away, so we need another representation of the group, such as a matrix representation.

Definition 10.4. *A left-regular representation $\mathcal{L}_g$ is a representation that transforms functions f by transforming their domains via the inverse group action $\mathcal{L}_g[f](x) := f(g^{-1} \cdot x)$*

Definition 10.5. *Equivariance is a property of an operator $\Phi \circ \rho^X(g) = \rho^Y \circ \Phi(g)$*

Convolutions are translation equivariant.

11 Dynamic Nerual scene representations & Physics

Our world is not static, but dynamic. We can reason about dynamic processes (rigid and non-rigid), such as tracking a car over time or realizing when something is deforming. We need mathematical tools for this.

One of the idea is we could have time-based latent variable. Our scene representation take a 3D coordinate as input and conditioned over some latent code on how scene evolve over time step. The world is constraint over time because the world follow some physics of motion.

11.1 Correspondence via Flow Field

A 2D flow field is a function that tells how much a pixel have shifted during the motion. An image: $\mathcal{F} : \mathbb{R}^2 \to \mathbb{R}^3$
Optical flow: $\Phi(x) : \mathbb{R}^2 \to \mathbb{R}^{\not{2}}; \Phi(x) = \Delta x$
An image at time frame t+1: $\widetilde{\mathcal{F}(x)} = \mathcal{F}(x + \Phi(x))$
If we want to warp over multiple time steps we can do the following:

$$\mathcal{F}_2(x) = \mathcal{F}_1(x + \Phi_{21}(x)) \tag{48}$$
$$= \mathcal{F}_0(x + \Phi_{21}(x) + \Phi_{10}(x + \Phi_{21}(x))) \tag{49}$$

How do we compute optical flow? Correlate every pixel in image 1 with every pixel in image 2. Optical flow can not be computed as a smoothed minimization problem on this cost volume. Suppose we have a 1D picture with binary colour. Suppose there are a 1 in position 2 and in time frame 1, it moves to position 10. We create

a matrix which have rows represent time frame 0 and column representing time frame 1. Then we will draw a 1 at position (2,10) in this case. How we find the optical flow is to find how much does the position change? Originally, if nothing moves, then we will have a diagonal matrix. We can compute the loss on how much it moves from the diagonal to (2,10) in this case. Notice if it moves a bit when we compute the loss, the loss will not change. Therefore, we can create a decent colour change which lead the loss change.

There is a paper called RAFT which introduces the idea on how to estimate optical flow. The idea is instead of comparing pixel by pixel, we extract deep learning feature. We have a RNN which in each layer, we repeat the process of correspondence checking with a down sample of resolution. We firstly start assume both image are identical. Then in each RNN layer, we use correspondence volume as input which predict the update (where the feature is moving).

This method is very expansive because if we are comparing the pixels, we need to store a 4D information to model the changes. 2D flow constrains possible 3D flow (for visible points), and optical flow can be computed given 3D scene flow plus depth. 3D flow in some sense is easier to compute this is because we can have a boundary condition for object in 3D but it is harder to do for 2D object. For example, if an object come close to us, to model optical flow with 2D object is extremely hard, because the image will just blow up. However, for 3D flow, we have a rough idea on how fast an object is moving.

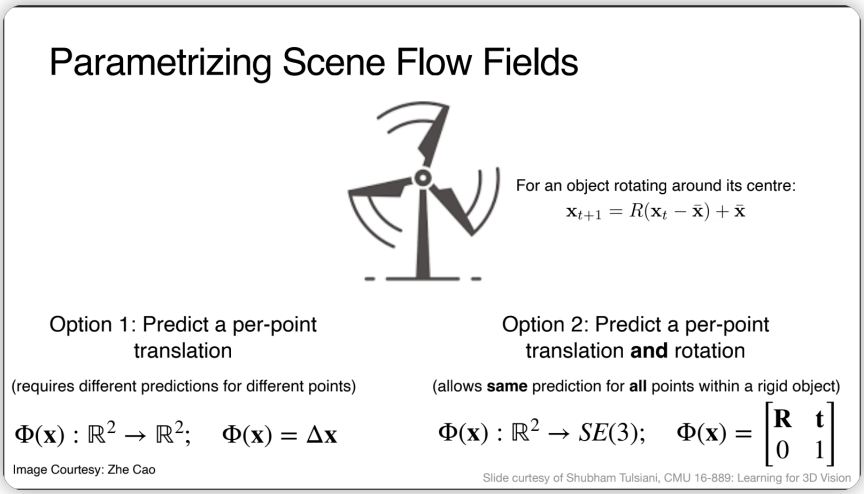

Figure 30: Parametrizing scene flow fields

Consider the example in figure 30, if could not parameterise a per-point translation because every point on the turbine moves a different amount of distance. However, if we add another degree of freedom, the rotation, then it is easy to model.

1. We use scene flow to predict optical flow. Everything is scene flow and optical flow is just the projected version of scene flow.
2. Typical motions are piecewise rigid.
3. Possible parametrization as pointwise SE3 field instead of pointwise translation.

$$(c_i, \sigma_i, \mathcal{F}_i, \mathcal{W}_i) = F_\theta(x, d, i) \tag{50}$$

Where $\mathcal{F}_i$ indicates the 3D flow to next frame for point x. For every point in 3D, we find out how it flow forwards and backwards in time. Basically, a time-dependent NeRF representation. Then we need to add some constraint on the flow. For example, the flow goes from time i to time j should be identical to the flow from time j to time i. If we take the 3D flow and project it into the image plane. If the projected flow should match the sudo-ground truth optical flow. Remember we only have the video at the start, so since optical flow can be computed quite easily by using RAFT. Therefore, we can compare the scene flow with the optical flow. We need to add constraint to the flow field. Otherwise we could just have a look up table and the flow moves from one colour to the other colour along this look up table.