Contents

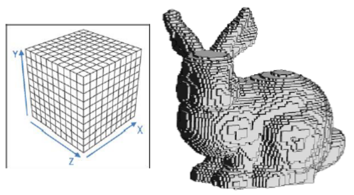
1 3D Concepts and Sensors

1.1 3D representations

In the computer vision field when we talk about **2D representations we are referring to images stored in a 2D array of pixels (with location and color attributes)**.

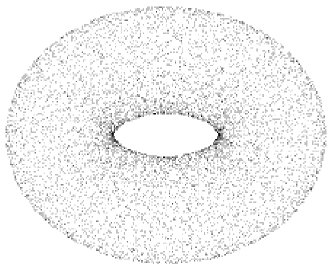
2D representation is straightforward but that's not the case of 3D representation. When we talking about 3D representation we actually have several options:

- **Voxels**. This is a volumetric representation constructed from occupancy grid in 3D: often binary (surface point or non-surface point) or sometimes it can be probabilistic (half occluded, etc.), and we can also store color/texture information. This type of representation is a 3D analog of pixel.

- **Point clouds**. Point clouds are an unordered set of 3D points with (x, y, z)-coordinates. Each of these points have its associated attributes such like **color**, **normal**, etc. **This type of representation are typically "raw" measurements of object as**:
  - There is no inter- or extrapolation.
  - No surface is defined.

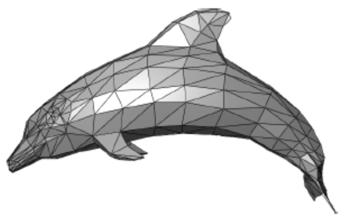
  Point clouds **can be understood as points sampled from the surface** and are often fed as input to other reconstruction algorithms to reconstruct the surface from these points. Also, it requires less storage compared with voxels as voxels stores also the inner body of the object ($O(n^3)$ complexity of storage) while this one only stores points of the surface ($O(n)$ complexity of storage)

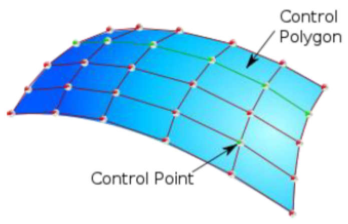
  Point clouds cannot tell you information about unknown areas. **You have no information whether a point of an unknown area is a surface point or not, while voxels grids does tell you**.

- **Polygonal Meshes**. This type of representation is heavily used in computer graphics and it's essentially a collection of **Vertices + Edges + Faces (often triangles)**. **It's a piecewise linear approximation of the surface as every face is planar**.

  We can store different attributes for faces such as **colors, normals, textures**, etc.

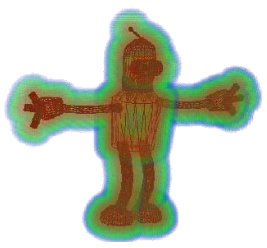
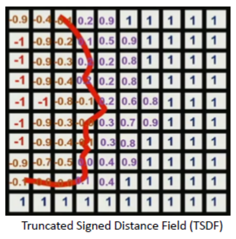


- **Parametric surfaces**. Unlike polygonal mesh representation which is a piecewise linear approximation, in parametric surfaces we can use higher-order functions to approximate the surface and thus get a smoother result. In this type of representation the surface is approximated by a set of control points:



- **Implicit surfaces**. This type of representation is very similar to the voxel grids. Unlike in voxel grids where we store an integer value to define whether there is surface or not, in this representation we store the **signed disctance** to the surface.

  It's called **implicit surface as this method doesn't tells you explicitly which vertices belongs to the surface, but it gives a distance measure which you can use it to check how far is the vertex to the surface.** This type of representation is good for reconstruct high quality surfaces (we'll see it later).

  A quick site note:

  – An explicit function is a function that feed it an input it will give us an output:

$$f(x) = y$$

  – An implicit function is a function that given the expected output we need to find inputs which can generate the expected output:

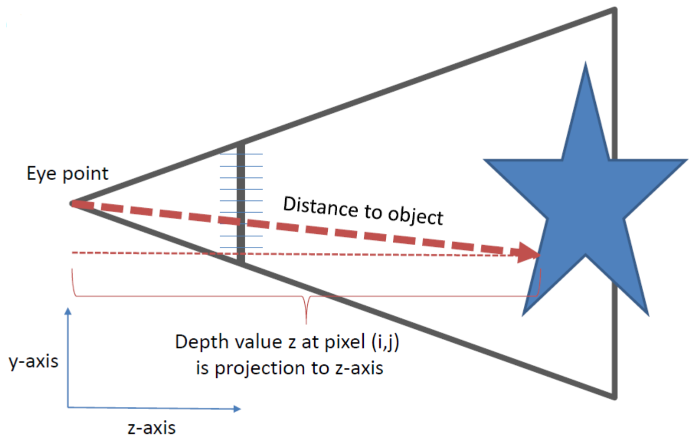
$$x^2 + y^2 - 1 = 0$$

2 3D scanning

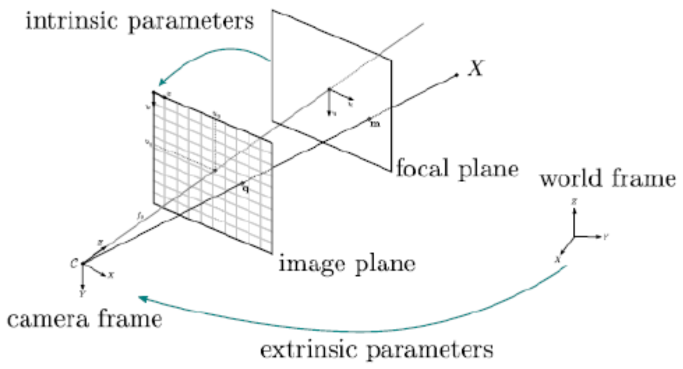
2.1 Depth camera

3D objects could be modeled by artists just as we see in video games and movies. This traditional pipeline is usually time consuming as it's used to be done manually. **What we would like to do is to create 3D objects in our computer terminal just by scanning them**.

3D scanning are often based on **depth images**. **In a depth image, besides the color information every pixel also stores the depth information. In other words, it defines the distance from the camera to that point**:



In order to reconstruct 3D object from a depth image is vital to know how camera works. We need to know how to map the location of a 3D point to a 2D image so we can invert this process to reconstruct 3D object:



Intrinsic parameters will tell us how the projection goes from 3D to 2D in the camera frame while **extrinsic parameters** will tell us how the projection goes from world frame to camera frame, or vice-versa for both cases.

One of the motivations of using depth cameras is that they work very similarly to human's eyes. Human's visual system is essentially depth camera, the **stereo vision** helps our brain to construct a depth map and thus to get the perception of depth.

2.2 Capture devices

There's basically two high level classes:

- **Passive**. Meaning that this type of devices doesn't emits anything to the scene:
  - **RGB**: receives light and the sensor captures the color of the light received from a specific point and encode it to the corresponding pixel.
  - **Stereo and Muiti-view**: same idea as RGB but here one use two (stereo) or even more (multi-view) cameras.

- **Active**. Meaning that this type of devices do emits things to the scene, such as light, in order to get depth information.
  - **Time of Flight (ToF)**: emits light to surface and counts how long it takes to back to sensor once reflected from the surface in order to measure the distance.
  - **Structured Light**: projects a known pattern (often grids or vertical bars) to the scene. The way that these deform when striking surfaces allows vision systems to calculate the depth and get surface information of the objects in the scene.
  - **Lase Scanner, LIDAR**: LIDAR works in similar way to ToF, the functional difference between LiDAR and other forms of ToF is that LiDAR uses pulsed lasers to build a point cloud, which is then used to construct a 3D map of image. ToF applications create "depth maps" based on light detection, usually through a standard RGB camera.

Kinect.v1: structured light

Kinect.v2: time of flight

stereo

active stereo

light detection and ranging

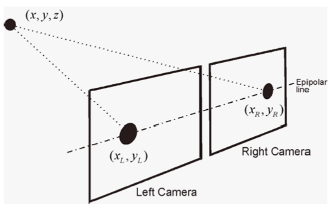
laser scanner (Cyberware)

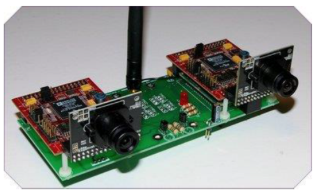
2.2.1 Passive Stereo

The setup for a stereo depth camera is quite easy and straightforward. We just need to have RGB cameras and make sure that the video capture is synchronized, i.e. the capture starts at the same time. Finally, we calibrate these two cameras, that is, determine both the **intrinsic and extrinsic parameters**, and do **stereo matching**.

The stereo matching works in the following way:

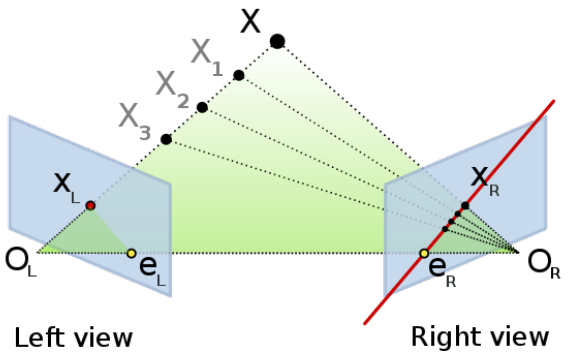
1. Given frames F_A, F_B, We pick a point (x_A, y_A) from F_A taken by camera A and we find it's corresponding point (x_B, y_B) from F_B taken by camera B.
2. Once we have found the above two points we can estimate the depth of its corresponding surface point by **triangulation using epipolar geometry**.





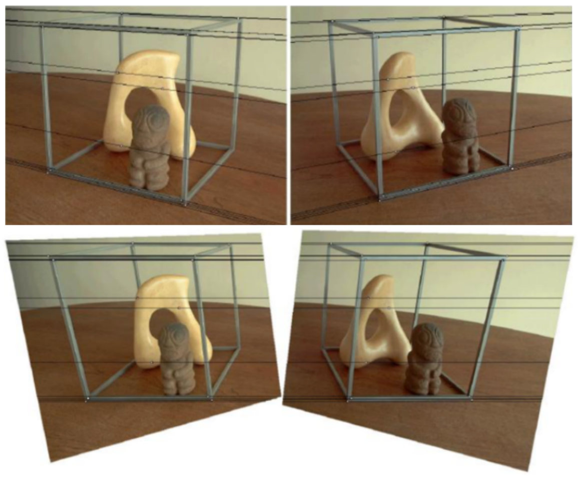
Choosing random points is clearly not ideal as the search space would be the entire image. Instead, we should looking for **epipolar points**. To understand the concept of **epiolar points** we need to introduce first what is the **epipolar geometry**.

Epipolar geometry is the geometry of stereo vision. When two cameras view a 3D scene from two distinct positions, there are a number of geometric relations between the 3D points and their projections onto the 2D images that lead to constraints between the image points. Assume the center of our cameras are O_L, O_R and a point X, we can define the **epipolar geometry** between them in the following way:

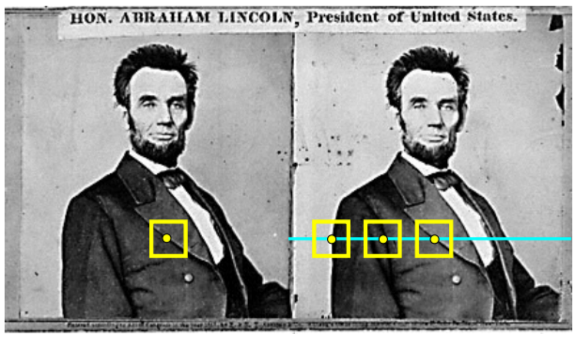


The plane $O_L O_R X$ is called **epipolar plane**. e_L, e_R, which are intersection points between the **baseline** $O_L O_R$ and the two camera planes are called **epipolar points**. And finally lines $X_L - e_L, X_R - e_R$ are called **epipolar lines**. X_L, X_R are intersection points between the image plane and the line defined from camera center to P. This looks nice, but, how can we extract the depth information from this construction?

In order to extract the depth information using the **triangulation technique we first need to rectify our images**. Rectification means **making the epipolar lines horizontal**. The reason is, **from the above plane is clear to see that finding corresponding points in epipolar lines $X_L - e_L, X_R - e_R$ is much easier as the search space is reduced to a single line**. But if epipolar line pairs are not parallel (like in the above example) then it would make the searching much harder. So in order to make the process easier we will first rectify our images, which it will look like following:

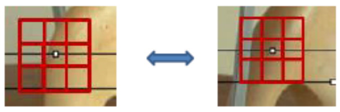


Once images are rectified then for each pixel **we just need to search along epipolar line to find match**. We should use 2D feature descriptors to determine the best match:



Local dense stereo matching One simple 2D descriptor could be just computing the L2 distance between neighbors of each point:

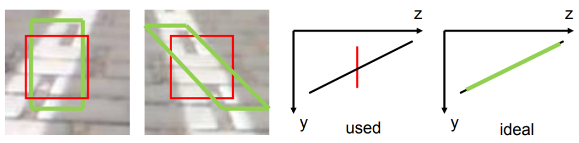
$$SSD(u,v) = \sum_{(u,v}(I_{left}(i,j) - I_{right}(i,j))^2$$



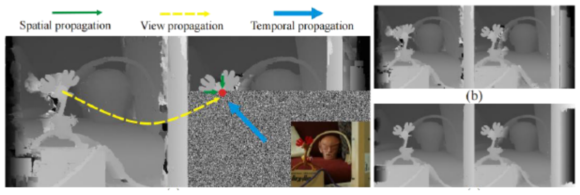
This is an example of **local dense stereo matching**, because it's a brute-force based matching by comparing every pair. The native search window has several problems (same as other approaches):

- **A constant depth within the window is assumed** - this implicit assumption is violated at:
  - Depth discontinuities
  - Slanted/non-planar surfaces

  For example the following picture illustrates a possible scene in which we have a slanted surface and the depth is wrong estimated:

  Ideally, the matching window should take the perspective distortion into account, but this is a "chicken and egg" problem as we need depth for that. One approach to solve this issue is to use **PatchMatch Stereo**. The key idea is to use **slanted support windows**.

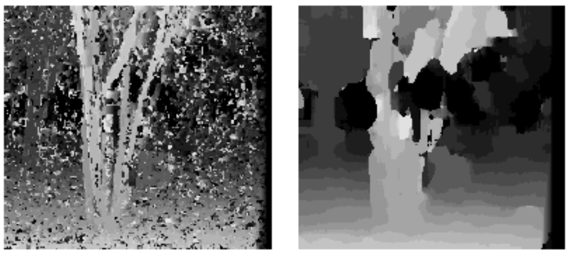
- **Matching time correlates w/ window size**. This problem can also be solved with the PatchMatch algorithm as it can quickly find correspondences between image patches. It is based on the observations that some of the good matches can be found randomly and that these results can be propagated to the neighbour areas due to natural coherence of images.

  This algorithm consists of 3 steps:

  1. **Initialization**: We first establish correspondence between pixels by randomly assign an offset to each pixel.
  2. **Propagation**: Each pixel checks if the offsets from neighboring patches give a better matching patch. If so, adopt neighbor's patch offset.
  3. **Search**: Each pixel searches for better patch offsets within a concentric radius around the current offset. The search radius starts with the size of the image and is halved each time until it is 1.
  4. Iterate between steps 2 and 3.

- **Matching in texture-less and specular surfaces is extremely difficult.**
- **Matching for repetitive textures is ambiguous**.

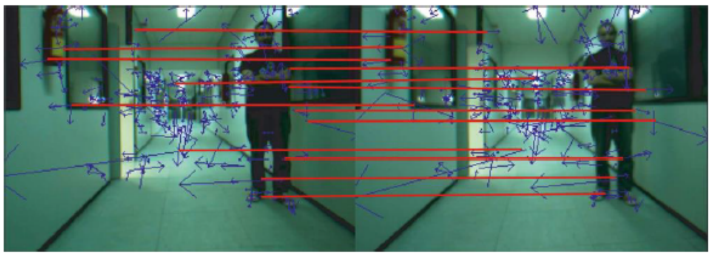
- **Matching in uniform areas is ambiguous**.
- **The size of the searching window matters and is hard to find the optimal** size as:
  - Smaller neighborhood would describe better the local structure but it leads to sharper changes in reconstruction.
  - Larger neighborhood decreases the level of detail but we get a smoother result.



Global dense stereo matching On the other hand we have **global dense stereo matching**. Here the objective function will try to find the optimal global matching cost rather than local optimal for each point. A common approach to find the global optimal is by using **dynamic programming**.

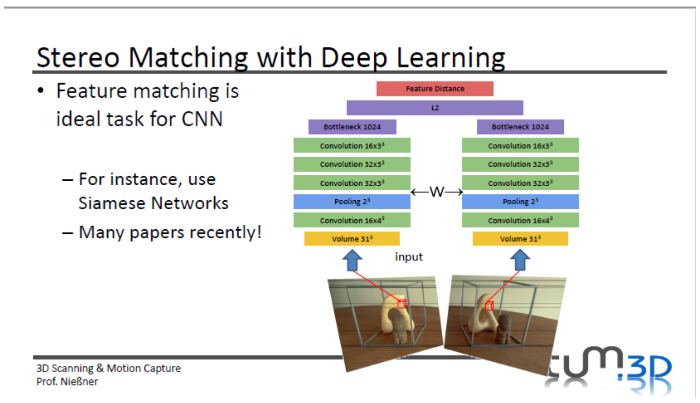
Sparse stereo matching Another choice is **sparse stereo matching**, in this type of matching we first pre-select points which have some desired properties. As both images defines the same object it should be possible to find the same (more or less) set of points (with the same properties) in both images. Once we have found them then we just need to compare those which falls at the same epipolar line:

Sparse features: SIFT, SURF, ORB, …



Matched SIFT features in a stereo pair (red lines indicate matches)

Stereo matching with Deep Learning Finally stereo matching with Deep Learning is also a hot topic nowadays:

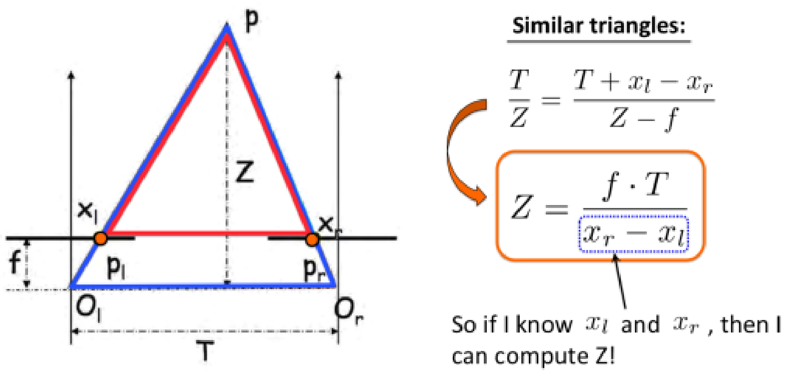


Stereo Matching with Deep Learning

- Feature matching is ideal task for CNN
  - For instance, use Siamese Networks
  - Many papers recently!

3D Scanning & Motion Capture
Prof. Nießner

Once we have found matches then computing the depth is straightforward by triangulation using epipolar geometry:


p
Z
x_l
x_r
f
p_l
p_r
O_l
O_r
T
Similar triangles:
$\frac{T}{Z} = \frac{T + x_l - x_r}{Z - f}$
$Z = \frac{f \cdot T}{x_r - x_l}$
So if I know x_l and x_r, then I can compute Z!

The choice of baseline O_lO_r is very important as:

- If we choose a wide baseline then we will make the matching harder because:
  - There will be more distortion.
  - There will be more occlusions.

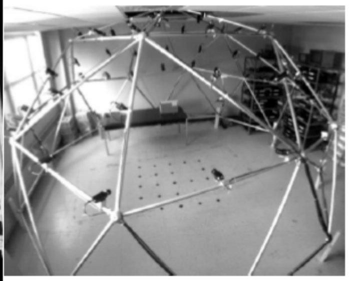
  But if we are able to find the matches **then the depth estimation would be very accurate**.
- If we choose a small baseline then matches will be less accurate because:
  - Even small disparity error results in large distance.
  - Disparity needs to be sub-pixel accurate as the triangle might becomes very skinny.

  But the advantage is the matching process will be much easier.

2.2.2 Multi-view Stereo

This is just an extension of Stereo except that instead of using 2 cameras we are using multiple cameras:

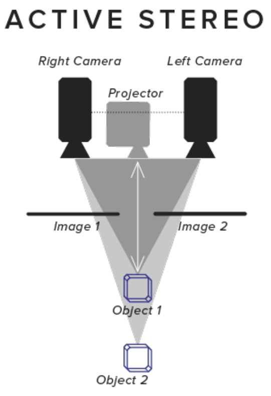
Stanford multi-camera array


CMU multi-camera stereo

2.2.3 Active Stereo

Active stereo is same as Stereo but with one more component, **projector**. The aim of the projector is to emit random patterns (typically IR).

This is very useful when we are dealing with for example texture-less objects. We can project some random patterns into it's surface and then making the matching process possible.

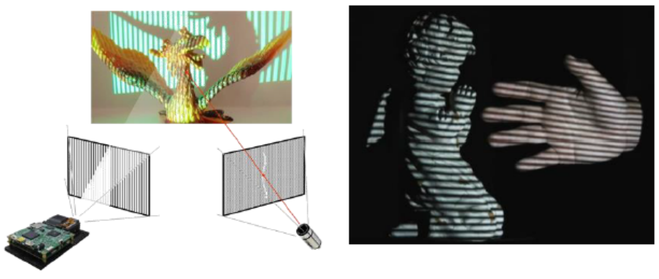


We should use:

- Active mode when the range is very close or in a indoor scene in order to get a better reconstruction.
- Passive mode when we are at outdoor as in this case we would have natural light and the pattern wouldn't be visible.

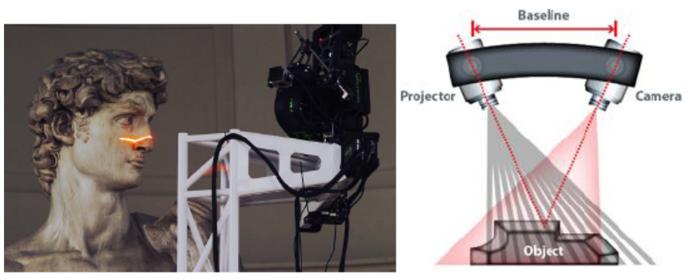
2.2.4 Structured Light

The structured light depth estimation approach uses a **projector** and a **camera**. The projector projects some patterns on the surface of the object and the camera records these patterns from another different angle. The high-level idea is that, based on the difference between distorted patterns caused by the surface of the object and the pre-calibrated pattern we can use triangulation to estimate depth. One important thing to keep in mind is that the camera must be able to recognize these patterns (the pattern must be known) otherwise the match won't be possible.



2.2.5 Laser Scanning

Laser scanning is just a simple version of the structured light approach. Instead of using complex patterns we use a laser pointer and scanline-by-scanline:

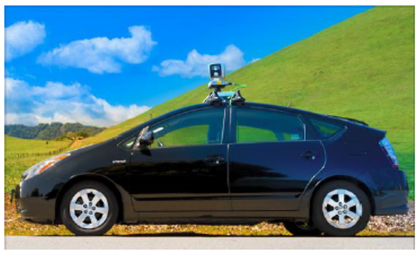


The reconstruction process may take a while but we will get very high quality results using this approach.

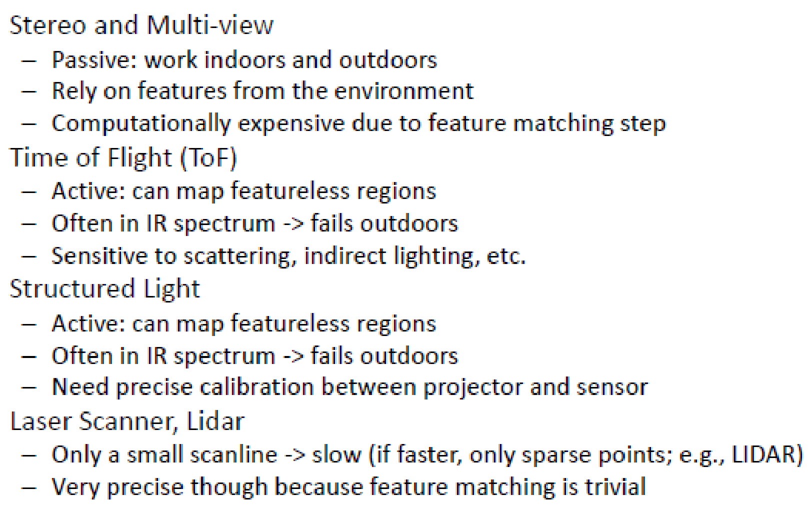
2.2.6 LIDAR

LIDAR determines ranges (variable distance) by targeting an object or a surface with a laser and measuring the time for the reflected light to return to the receiver.

Light Imaging, Detection, And Ranging (LIDAR)



2.3 Summary of Depth Cameras



Stereo and Multi-view
- Passive: work indoors and outdoors
- Rely on features from the environment
- Computationally expensive due to feature matching step

Time of Flight (ToF)
- Active: can map featureless regions
- Often in IR spectrum -> fails outdoors
- Sensitive to scattering, indirect lighting, etc.

Structured Light
- Active: can map featureless regions
- Often in IR spectrum -> fails outdoors
- Need precise calibration between projector and sensor

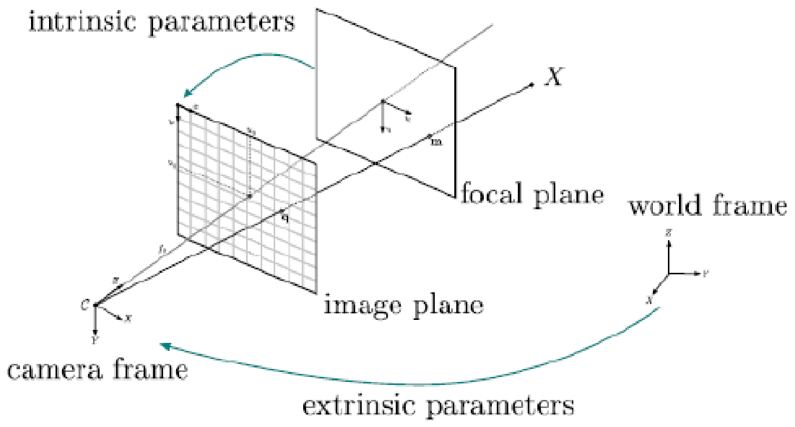
Laser Scanner, Lidar
- Only a small scanline -> slow (if faster, only sparse points; e.g., LIDAR)
- Very precise though because feature matching is trivial

3 Camera Concepts

3.1 Camera parameters

We can divide the parameters of a camera into **Intrinsic** and **Extrinsic**:

- Intrinsic parameters are parameters which describes the internal setting of a camera. It allow a mapping between camera coordinates and pixel coordinates in the image frame:
  - Focal length
  - Principal point
  - (Skew)
  - (Distortion params.)
- Extrinsic parameters are parameters which describes the external setting of a camera, it defines the location and orientation of the camera with respect to the world frame:
  - 6 degrees of freedom (DoF): aka 'pose', which is **rotation + translation**.



3.2 Spaces

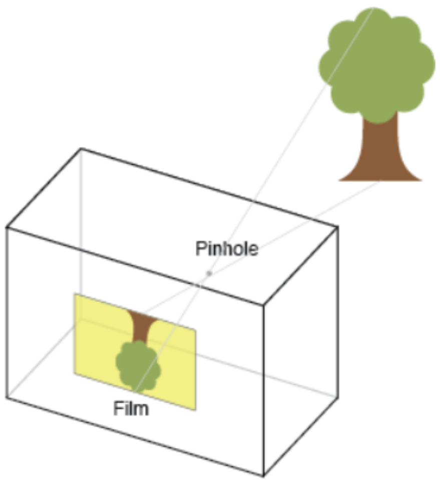
Just like in computer graphics here we also have different **spaces**:

- **World space**: global coordinate system.
- **Camera space**: coordinate system within current frame.
- **Screen/Pixel coordinates**: pixel position (x, y) + depth value (z)
- **Normalized Device coordinates (NDC)**:
  - In OpenGL $[-1;1]^3$
  - In DirectX $[-1;1]^2$ x$[0;1]$
- **Model space**: local coordinate frame for a given model (within a scene)

In computer vision we will mainly dealing with: **world space, camera space, Screen/Pixel coordinates and model space**.

3.3 Intrinsic Parameters

Intrinsic parameters defines the intrinsic matrix which can be used to transform 3D camera coordinates to 2D homogeneous image coordinates. This perspective projection is modeled by the ideal pinhole camera, illustrated below:

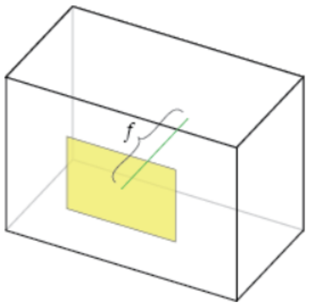


The intrinsic matrix is parameterized by **Hartley and Zisserman** as:

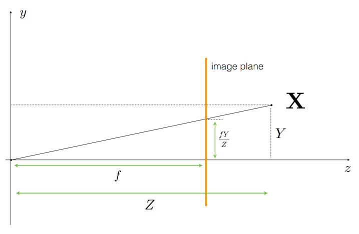
$$K = \begin{pmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{pmatrix}$$

3.3.1 Focal length, f_x, f_y

The focal length is the distance between the pinhole and the film (a.k.a. image plane) and is measured in pixels. In a true pinhole camera, both f_x and f_y have the same value, which is illustrated as f below:



The following picture illustrates how the 2D image coordinate can be computed from 3D camera coordinates:

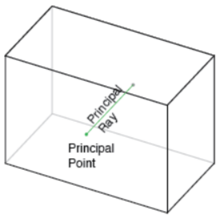


$$[X \quad Y \quad Z]^\top \mapsto [fX/Z \quad fY/Z]^\top$$

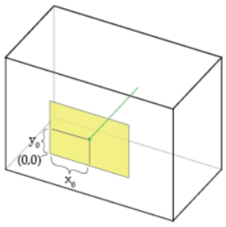
The reason why we use a separate focal length for x and y is because the **resulting image might not always be in a square format**, and the ratio $\frac{f_x}{f_y}$ defines the aspect ratio of our image

3.3.2 Principal Point Offset, m_x, m_y or (x_0, y_0)

The camera's "principal axis" is the line perpendicular to the image plane that passes through the pinhole. Its intersection with the image plane is referred to as the "principal point," illustrated below.



The "principal point offset" is the location of the principal point relative to the film's origin. The exact definition depends on which convention is used for the location of the origin; the illustration below assumes it's at the bottom-left of the film.



3.3.3 Projection from Camera space to Pixel coordinates

$$\begin{pmatrix} f_x & \gamma & m_x \\ 0 & f_y & m_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix} = z_c \begin{pmatrix} u \\ v \\ 1 \end{pmatrix}$$

$$K \begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix} = z_c \begin{pmatrix} u \\ v \\ 1 \end{pmatrix}$$

Need division by z_c to obtain pixel coordinates

```
// pos.xyz in meters; output in pixels/meters
static inline float3 cameraToScreen(const float3& pos)
{
        return make_float3(
                pos.x*c_depthCameraParams.fx/pos.z + c_depthCameraParams.mx,
                pos.y*c_depthCameraParams.fy/pos.z + c_depthCameraParams.my,
                pos.z);
}
```

```
// ux, uy in pixels; depth in meters
static inline float3 screenToCamera(uint ux, uint uy, float depth)
{
        const float x = ((float)ux-c_depthCameraParams.mx) / c_depthCameraParams.fx;
        const float y = ((float)uy-c_depthCameraParams.my) / c_depthCameraParams.fy;
        return make_float3(depth*x, depth*y, depth);
}
```

3.4 Extrinsic Parameters

6 Degrees of Freedom (DoF)

– 3 for rotation

– 3 for translation

4x4 or 3x4 matrix

$$[R,T] = \begin{pmatrix} R_{00} & R_{01} & R_{02} & t_x \\ R_{10} & R_{11} & R_{12} & t_y \\ R_{20} & R_{21} & R_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We can see the above matrix is a composition of a **translation matrix,** T, and a **Rotation matrix**, R. The translation matrix is straightforward while the computation of the rotation matrix is a little bit complex:

$$R = R_z(\gamma) \cdot R_y(\beta) \cdot R_x(\alpha)$$

$$R^T = R^{-1}$$

$$|\det R\,| = 1$$

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

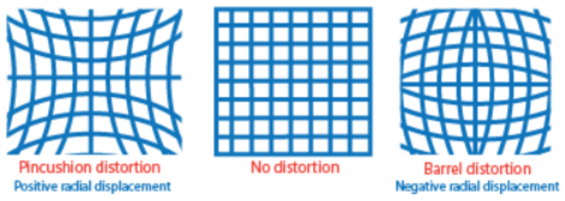
It's often more natural to specify the camera's pose directly rather than specifying how world points should transform to camera coordinates. Luckily, building an extrinsic camera matrix this way is easy: **just build a rigid transformation matrix that describes the camera's pose and then take it's inverse**.

3.5 Camera calibration

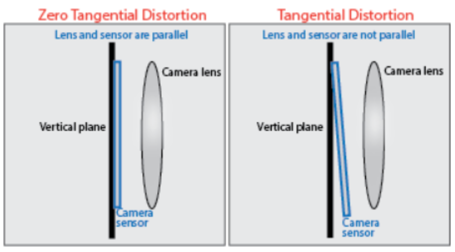
A pinhole camera has no lens but that's not the case of real cameras, they do contain lens. So in order to perform camera calibration we need to take into account the distortion effect as well.

There're two most common distortions:

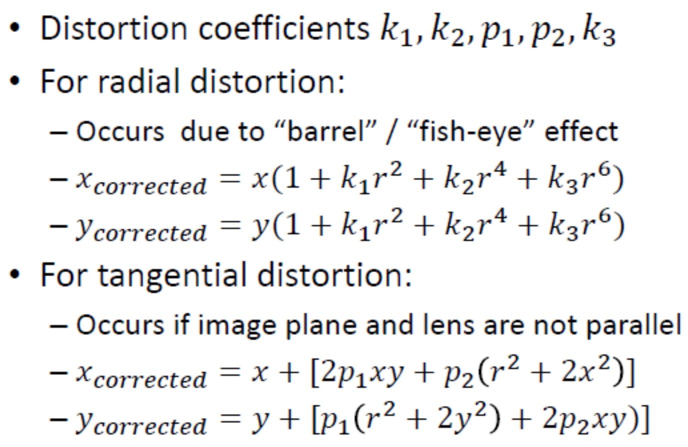
- **Radial distortion**



- **Tangential distortion**



We can approximate the distortion effect in the following way:



- Distortion coefficients k_1, k_2, p_1, p_2, k_3
- For radial distortion:
  - Occurs due to "barrel" / "fish-eye" effect
  - $x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$
  - $y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$
- For tangential distortion:
  - Occurs if image plane and lens are not parallel
  - $x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$
  - $y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy)]$

4 3D reconstruction

The general pipeline of 3D reconstruction works in the following way:

1. Get the point cloud for the current frame using the **intrinsic matrix** (with a 3D scanner).

2. Then move the camera (or object) to obtain next frame (i.e, next view).

3. Then we compute the alignment between the captured frames, i.e., compute the extrinsic (6DoF) and align them.

4. Finally we merge the point clouds and do surface reconstruction from point cloud.