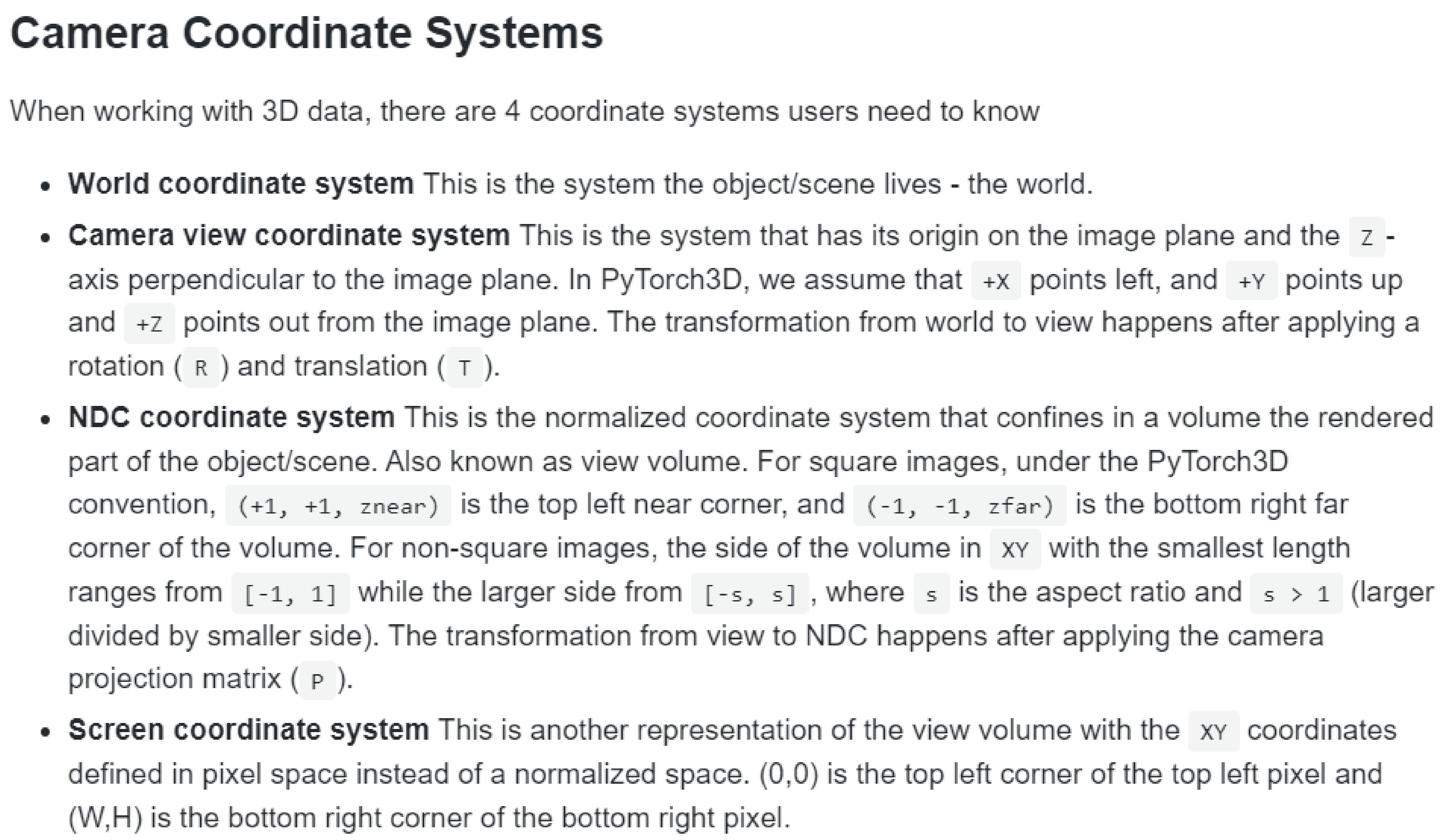
Collective Projection Imaging in PyTorch3D

Bill Worstell

PicoRad -> MGH

2/2/2024

https://pytorch3d.org/docs/cameras#camera-coordinate-systems

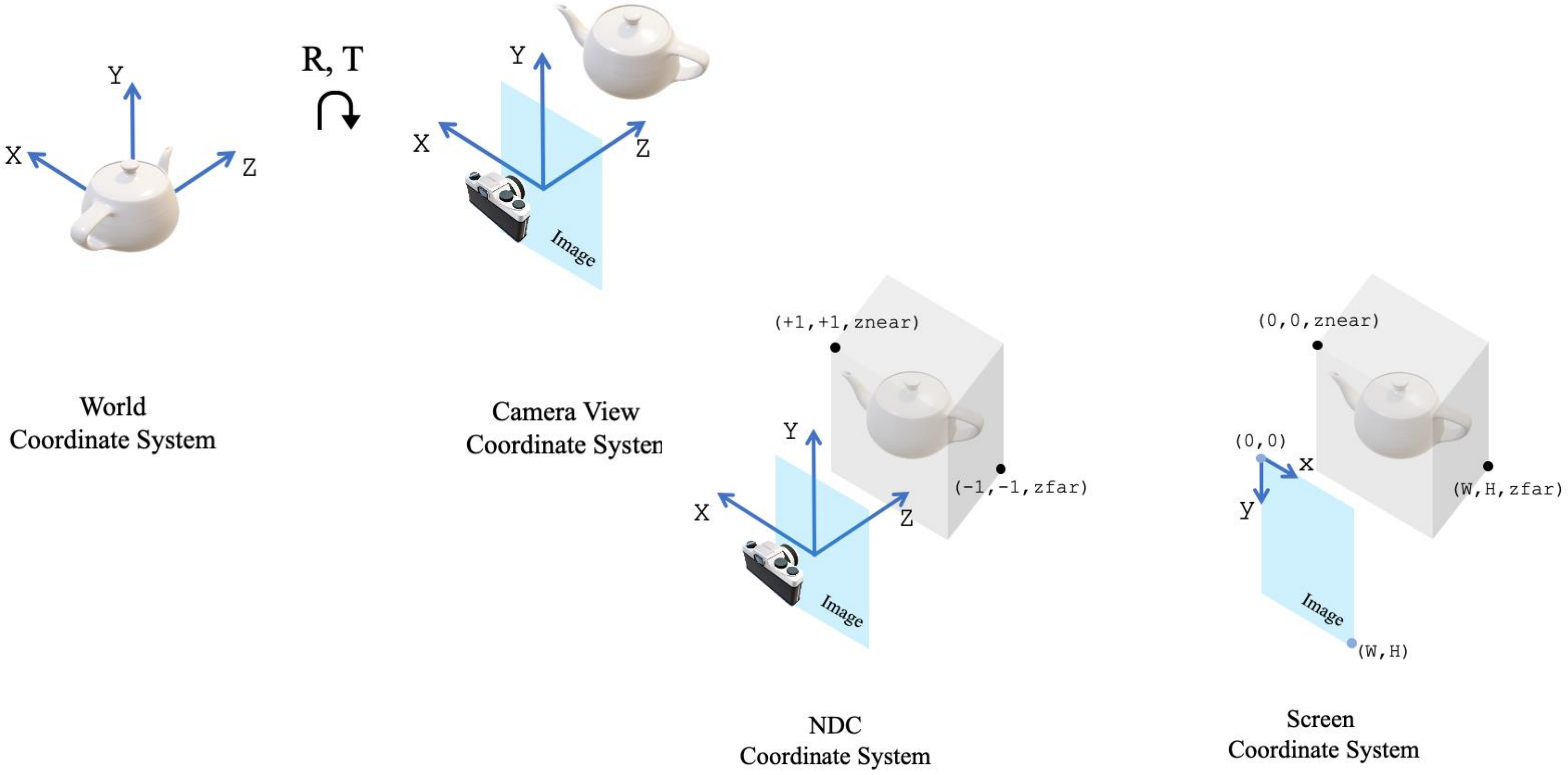


Camera Coordinate Systems

When working with 3D data, there are 4 coordinate systems users need to know

- **World coordinate system** This is the system the object/scene lives - the world.
- **Camera view coordinate system** This is the system that has its origin on the image plane and the `Z`-axis perpendicular to the image plane. In PyTorch3D, we assume that `+X` points left, and `+Y` points up and `+Z` points out from the image plane. The transformation from world to view happens after applying a rotation (`R`) and translation (`T`).
- **NDC coordinate system** This is the normalized coordinate system that confines in a volume the rendered part of the object/scene. Also known as view volume. For square images, under the PyTorch3D convention, `(+1, +1, znear)` is the top left near corner, and `(-1, -1, zfar)` is the bottom right far corner of the volume. For non-square images, the side of the volume in `XY` with the smallest length ranges from `[-1, 1]` while the larger side from `[-s, s]`, where `s` is the aspect ratio and `s > 1` (larger divided by smaller side). The transformation from view to NDC happens after applying the camera projection matrix (`P`).
- **Screen coordinate system** This is another representation of the view volume with the `XY` coordinates defined in pixel space instead of a normalized space. (0,0) is the top left corner of the top left pixel and (W,H) is the bottom right corner of the bottom right pixel.

https://pytorch3d.org/docs/cameras#camera-coordinate-systems



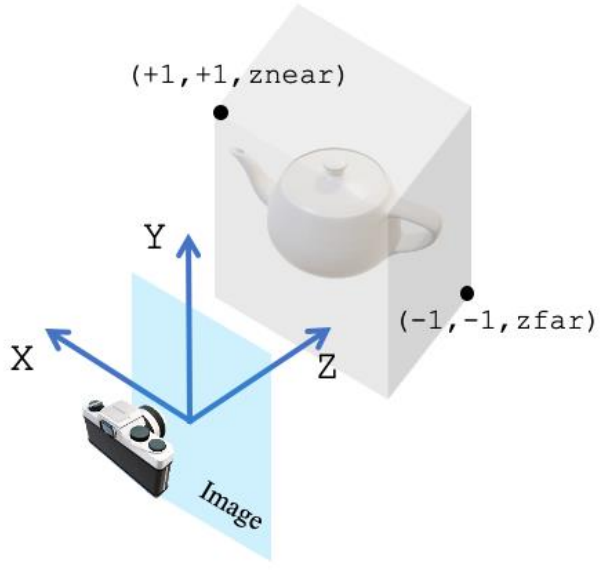
https://pytorch3d.org/docs/cameras#defining-cameras-in-pytorch3d

Defining Cameras in PyTorch3D

Cameras in PyTorch3D transform an object/scene from world to view by first transforming the object/scene to view (via transforms `R` and `T`) and then projecting the 3D object/scene to a normalized space via the projection matrix `P = K[R | T]`, where `K` is the intrinsic matrix. The camera parameters in `K` define the normalized space. If users define the camera parameters in NDC space, then the transform projects points to NDC. If the camera parameters are defined in screen space, the transformed points are in screen space.

Note that the base `CamerasBase` class makes no assumptions about the coordinate systems. All the above transforms are geometric transforms defined purely by `R`, `T` and `K`. This means that users can define cameras in any coordinate system and for any transforms. The method `transform_points` will apply `K`, `R` and `T` to the input points as a simple matrix transformation. However, if users wish to use cameras with the PyTorch3D renderer, they need to abide to PyTorch3D's coordinate system assumptions (read below).

We provide instantiations of common camera types in PyTorch3D and how users can flexibly define the projection space below.



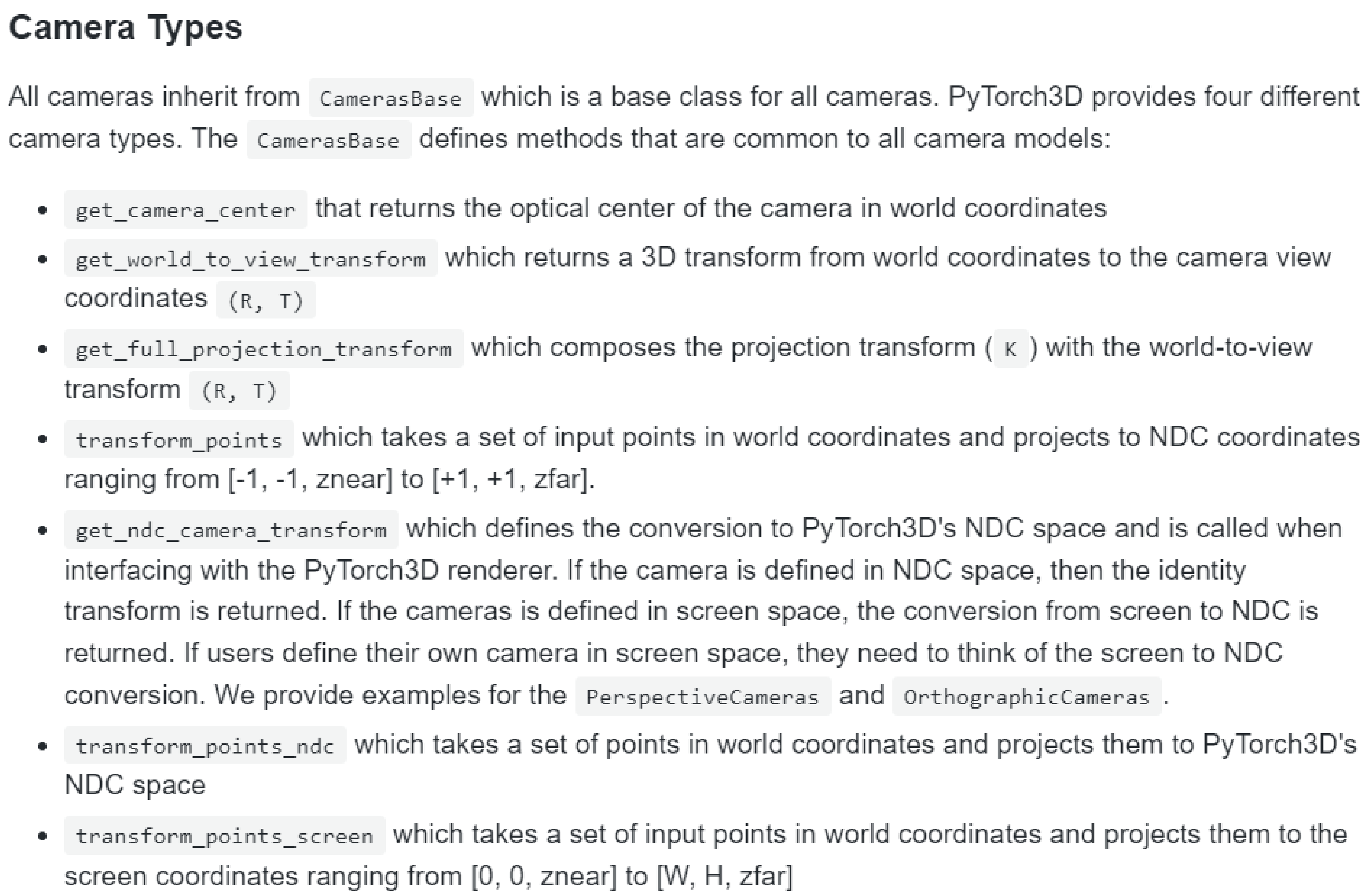
NDC
Coordinate System

Interfacing with the PyTorch3D Renderer

The PyTorch3D renderer for both meshes and point clouds assumes that the camera transformed points, meaning the points passed as input to the rasterizer, are in PyTorch3D's NDC space. So to get the expected rendering outcome, users need to make sure that their 3D input data and cameras abide by these PyTorch3D coordinate system assumptions. The PyTorch3D coordinate system assumes `+X:left`, `+Y: up` and `+Z: from us to scene` (right-handed) . Confusions regarding coordinate systems are common so we advise that you spend some time understanding your data and the coordinate system they live in and transform them accordingly before using the PyTorch3D renderer.

Examples of cameras and how they interface with the PyTorch3D renderer can be found in our tutorials.

https://pytorch3d.org/docs/cameras#camera-types



Camera Types

All cameras inherit from `CamerasBase` which is a base class for all cameras. PyTorch3D provides four different camera types. The `CamerasBase` defines methods that are common to all camera models:

- `get_camera_center` that returns the optical center of the camera in world coordinates
- `get_world_to_view_transform` which returns a 3D transform from world coordinates to the camera view coordinates `(R, T)`
- `get_full_projection_transform` which composes the projection transform (`K`) with the world-to-view transform `(R, T)`
- `transform_points` which takes a set of input points in world coordinates and projects to NDC coordinates ranging from [-1, -1, znear] to [+1, +1, zfar].
- `get_ndc_camera_transform` which defines the conversion to PyTorch3D's NDC space and is called when interfacing with the PyTorch3D renderer. If the camera is defined in NDC space, then the identity transform is returned. If the cameras is defined in screen space, the conversion from screen to NDC is returned. If users define their own camera in screen space, they need to think of the screen to NDC conversion. We provide examples for the `PerspectiveCameras` and `OrthographicCameras`.
- `transform_points_ndc` which takes a set of points in world coordinates and projects them to PyTorch3D's NDC space
- `transform_points_screen` which takes a set of input points in world coordinates and projects them to the screen coordinates ranging from [0, 0, znear] to [W, H, zfar]

https://pytorch3d.org/docs/cameras#camera-types

FoVPerspectiveCameras, FoVOrthographicCameras

These two cameras follow the OpenGL convention for perspective and orthographic cameras respectively. The user provides the near `znear` and far `zfar` field which confines the view volume in the `Z` axis. The view volume in the `XY` plane is defined by field of view angle (`fov`) in the case of `FoVPerspectiveCameras` and by `min_x, min_y, max_x, max_y` in the case of `FoVOrthographicCameras`. These cameras are by default in NDC space.

PerspectiveCameras, OrthographicCameras

These two cameras follow the Multi-View Geometry convention for cameras. The user provides the focal length (`fx`, `fy`) and the principal point (`px`, `py`). For example, `camera = PerspectiveCameras(focal_length=((fx, fy),), principal_point=((px, py),))`

The camera projection of a 3D point `(X, Y, Z)` in view coordinates to a point `(x, y, z)` in projection space (either NDC or screen) is

```
# for perspective camera
x = fx * X / Z + px
y = fy * Y / Z + py
z = 1 / Z

# for orthographic camera
x = fx * X + px
y = fy * Y + py
z = Z
```

The user can define the camera parameters in NDC or in screen space. Screen space camera parameters are common and for that case the user needs to set `in_ndc` to `False` and also provide the `image_size=(height, width)` of the screen, aka the image.

The `get_ndc_camera_transform` provides the transform from screen to NDC space in PyTorch3D. Note that the screen space assumes that the principal point is provided in the space with `+X left`, `+Y down` and origin at the top left corner of the image. To convert to NDC we need to account for the scaling of the normalized space as well as the change in `XY` direction.

Below are example of equivalent `PerspectiveCameras` instantiations in NDC and screen space, respectively.

```
# NDC space camera
fcl_ndc = (1.2,)
prp_ndc = ((0.2, 0.5),)
cameras_ndc = PerspectiveCameras(focal_length=fcl_ndc, principal_point=prp_ndc)

# Screen space camera
image_size = ((128, 256),)    # (h, w)
fcl_screen = (76.8,)          # fcl_ndc * min(image_size) / 2
prp_screen = ((115.2, 32), )  # w / 2 - px_ndc * min(image_size) / 2, h / 2 - py_ndc * min(image_size)
cameras_screen = PerspectiveCameras(focal_length=fcl_screen, principal_point=prp_screen, in_ndc=False,
```

https://pytorch3d.org/docs/cameras#camera-types

Below are example of equivalent PerspectiveCameras instantiations in NDC and screen space, respectively.

```
# NDC space camera
fcl_ndc = (1.2,)
prp_ndc = ((0.2, 0.5),)
cameras_ndc = PerspectiveCameras(focal_length=fcl_ndc, principal_point=prp_ndc)

# Screen space camera
image_size = ((128, 256),)    # (h, w)
fcl_screen = (76.8,)          # fcl_ndc * min(image_size) / 2
prp_screen = ((115.2, 32), )  # w / 2 - px_ndc * min(image_size) / 2, h / 2 - py_ndc * min(image_size) / 2
cameras_screen = PerspectiveCameras(focal_length=fcl_screen, principal_point=prp_screen, in_ndc=False,
image_size=image_size)
```

The relationship between screen and NDC specifications of a camera's focal_length and principal_point is given by the following equations, where s = min(image_width, image_height). The transformation of x and y coordinates between screen and NDC is exactly the same as for px and py.

```
fx_ndc = fx_screen * 2.0 / s
fy_ndc = fy_screen * 2.0 / s

px_ndc = - (px_screen - image_width / 2.0) * 2.0 / s
py_ndc = - (py_screen - image_height / 2.0) * 2.0 / s
```

https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/orthographic-projection-matrix.html

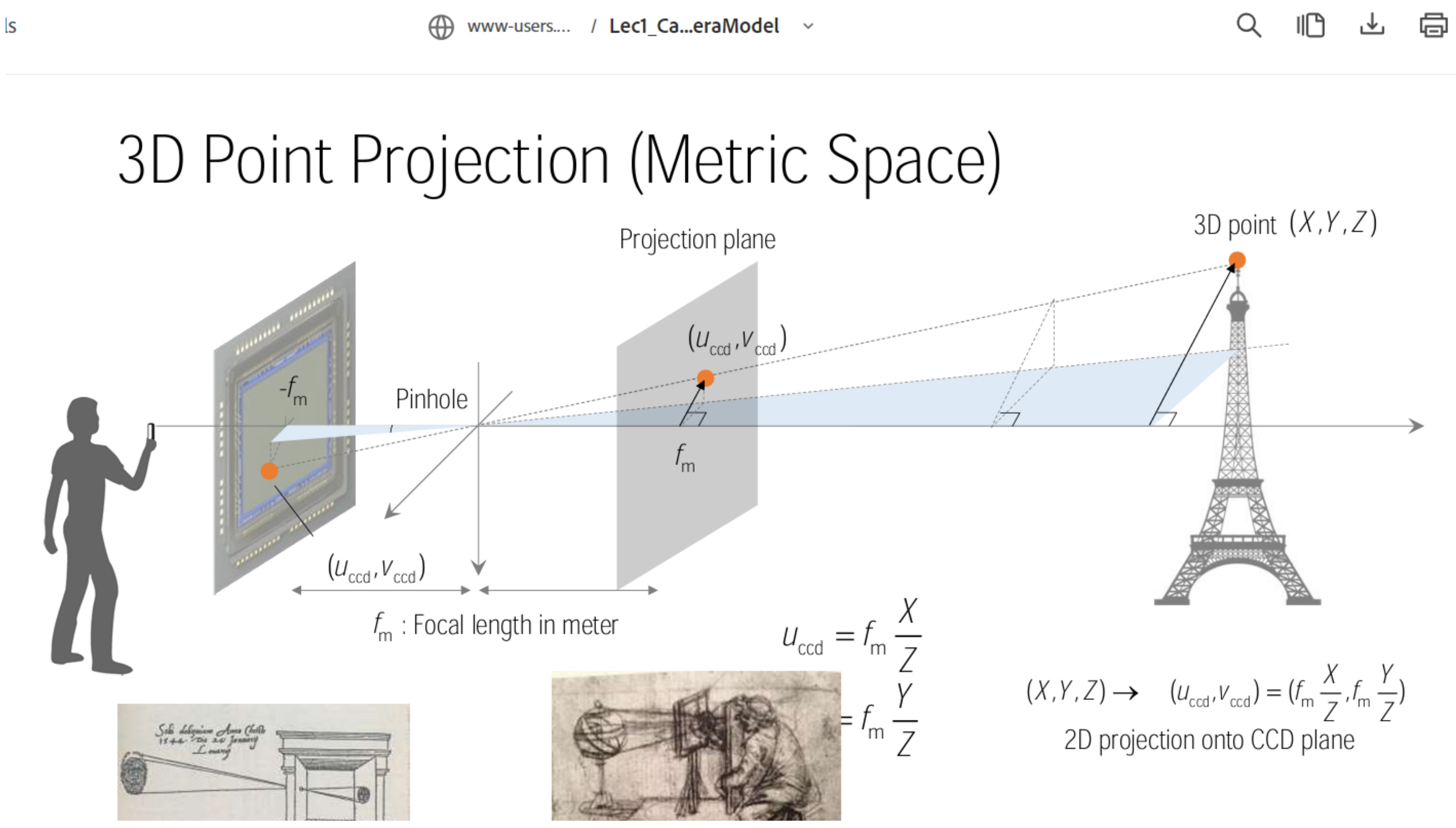
The Perspective and Orthographic Projection Matrix

https://www-users.cse.umn.edu/~hspark/CSci5980/csci5980_3dvision.html

Spring 2018 CSCI 5980 Multiview 3D Geometry in Computer Vision

Mon/Wed 4:00pm-5:15pm @ Ford Hall B15

Lecture	Supplementary
Camera model	Dolly zoom
Camera projection matrix	Affine camera model Lens distortion
Projective line	Point and line at infinity
Localization using vanishing points	Celestial navigation
Single view metrology	
Image transform	Image warping code Cylindrical panorama
Linear estimation	Least squares via vector derivative SVD on Mondrian Painting
Camera calibration	MATLAB calibration toolbox Physical focal point
Where am I via homography?	
3D Spatial Rotation	
Epipolar geometry (fundamental matrix)	
Relative pose estimation	Local visual feature matching
Robust estimation (RANSAC)	
3D point triangulation	



3D Point Projection (Metric Space)

$$u_{ccd} = f_m \frac{X}{Z}$$

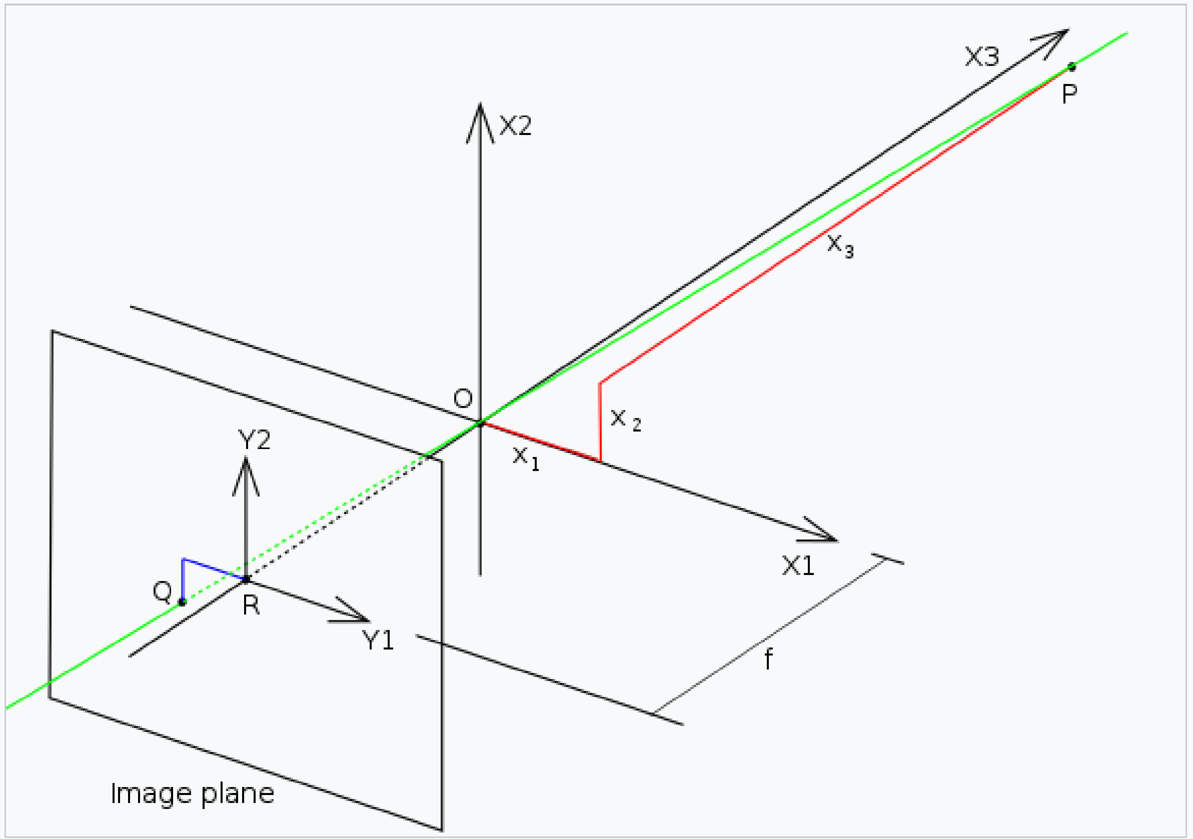
$$= f_m \frac{Y}{Z}$$

$$(X,Y,Z) \rightarrow (u_{ccd}, v_{ccd}) = (f_m \frac{X}{Z}, f_m \frac{Y}{Z})$$

https://en.wikipedia.org/wiki/Pinhole_camera_model#Geometry

The geometry related to the mapping of a pinhole camera is illustrated in the figure. The figure contains the following basic objects:

- A 3D orthogonal coordinate system with its origin at **O**. This is also where the *camera aperture* is located. The three axes of the coordinate system are referred to as X1, X2, X3. Axis X3 is pointing in the viewing direction of the camera and is referred to as the *optical axis*, *principal axis*, or *principal ray*. The plane which is spanned by axes X1 and X2 is the front side of the camera, or *principal plane*.
- An image plane, where the 3D world is projected through the aperture of the camera. The image plane is parallel to axes X1 and X2 and is located at distance f from the origin **O** in the negative direction of the X3 axis, where *f* is the focal length of the pinhole camera. A practical implementation of a pinhole camera implies that the image plane is located such that it intersects the X3 axis at coordinate *-f* where *f > 0*.
- A point **R** at the intersection of the optical axis and the image plane. This point is referred to as the *principal point* [1] or *image center*.



The geometry of a pinhole camera. Note: the $x_1x_2x_3$ coordinate system in the figure is left-handed, that is the direction of the OZ axis is in reverse to the system the reader may be used to.

https://en.wikipedia.org/wiki/Pinhole_camera_model

Formulation [edit]

Next we want to understand how the coordinates (y_1, y_2) of point **Q** depend on the coordinates (x_1, x_2, x_3) of point **P**. This can be done with the help of the following figure which shows the same scene as the previous figure but now from above, looking down in the negative direction of the X2 axis.

In this figure we see two similar triangles, both having parts of the projection line (green) as their hypotenuses. The catheti of the left triangle are $-y_1$ and *f* and the catheti of the right triangle are x_1 and x_3. Since the two triangles are similar it follows that

$$\frac{-y_1}{f} = \frac{x_1}{x_3} \text{ or } y_1 = -\frac{f\,x_1}{x_3}$$

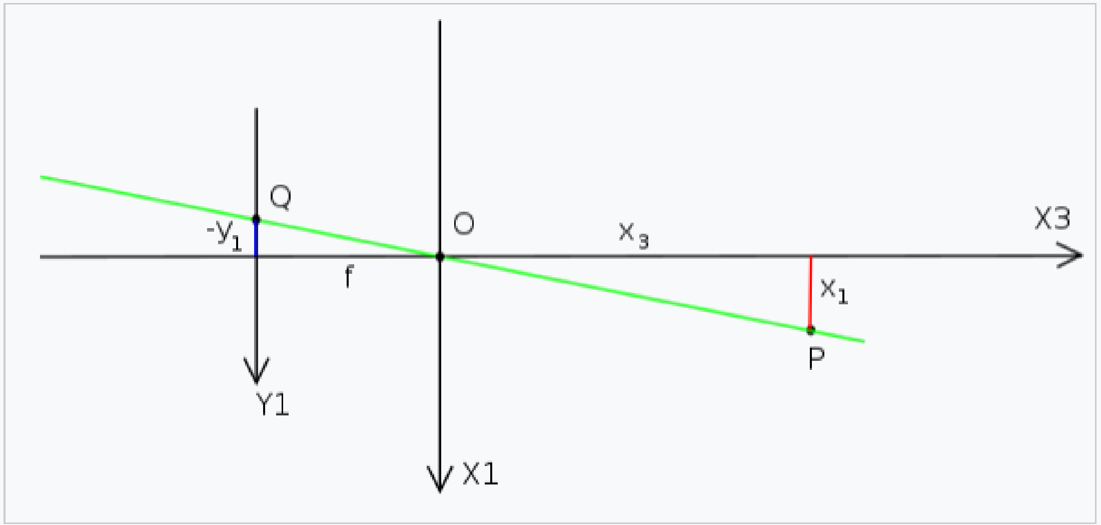
A similar investigation, looking in the negative direction of the X1 axis gives

$$\frac{-y_2}{f} = \frac{x_2}{x_3} \text{ or } y_2 = -\frac{f\,x_2}{x_3}$$

This can be summarized as

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = -\frac{f}{x_3}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

which is an expression that describes the relation between the 3D coordinates (x_1, x_2, x_3) of point **P** and its image coordinates (y_1, y_2) given by point **Q** in the image plane.



The geometry of a pinhole camera as seen from the X2 axis https://pytorch3d.readthedocs.io/en/latest/modules/transforms.html

pytorch3d.transforms.axis_angle_to_matrix(axis_angle: Tensor)→ Tensor[source]
Convert rotations given as axis/angle to rotation matrices.

Parameters
:
axis_angle – Rotations given as a vector in axis angle form, as a tensor of shape (…, 3), where the magnitude is the angle turned anticlockwise in radians around the vector's direction.

Returns
:
Rotation matrices as tensor of shape (…, 3, 3).

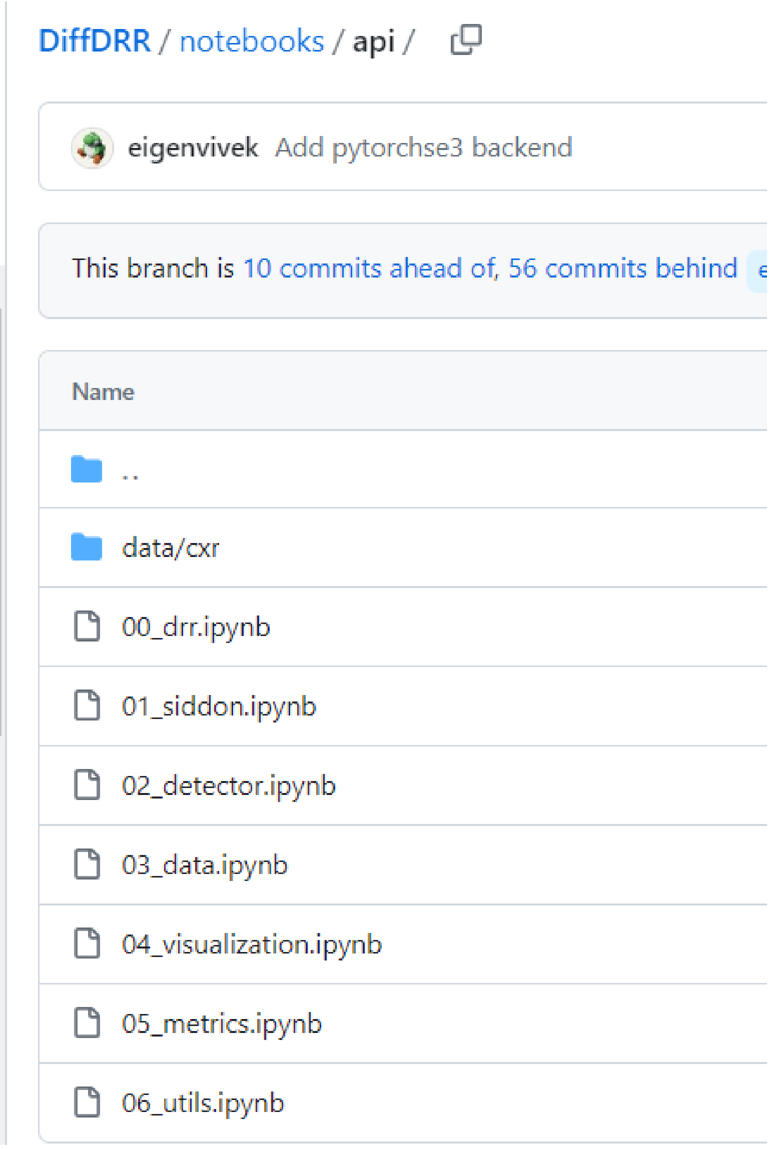
https://github.com/BillWorstell/DiffDRR/blob/main/diffdrr/detector.py

https://github.com/BillWorstell/DiffDRR/tree/main/notebooks/api

```
# %% ../notebooks/api/02_detector.ipynb 6
class Detector(torch.nn.Module):
    """Construct a 6 DoF X-ray detector system. This model is based on a C-Arm."""

    def __init__(
        self,
        sdr: float,  # Source-to-detector radius (half of the source-to-detector distance)
        height: int,  # Height of the X-ray detector
        width: int,  # Width of the X-ray detector
        delx: float,  # Pixel spacing in the X-direction
        dely: float,  # Pixel spacing in the Y-direction
        x0: float,  # Principal point X-offset
        y0: float,  # Principal point Y-offset
        n_subsample: int | None = None,  # Number of target points to randomly sample
        reverse_x_axis: bool = False,  # If pose includes reflection (in E(3) not SE(3)), reverse x-axis
    ):
        super().__init__()
        self.sdr = sdr
        self.height = height
        self.width = width
        self.delx = delx
        self.dely = dely
        self.x0 = x0
        self.y0 = y0
        self.n_subsample = n_subsample
        if self.n_subsample is not None:
            self.subsamples = []
        self.reverse_x_axis = reverse_x_axis

        # Initialize the source and detector plane in default positions (along the x-axis)
        source, target = self._initialize_carm()
        self.register_buffer("source", source)
        self.register_buffer("target", target)
```


DiffDRR / notebooks / api /

eigenvivek Add pytorchse3 backend

This branch is 10 commits ahead of, 56 commits behind

Name
..
data/cxr
00_drr.ipynb
01_siddon.ipynb
02_detector.ipynb
03_data.ipynb
04_visualization.ipynb
05_metrics.ipynb
06_utils.ipynb

```
#| exporti
from pytorch3d.transforms import (
    axis_angle_to_matrix,
    euler_angles_to_matrix,
    quaternion_to_matrix,
    rotation_6d_to_matrix,
)
from pytorchse3.so3 import so3_exp_map


def _convert_to_rotation_matrix(rotation, parameterization, convention, **kwargs):
    """Convert any parameterization of a rotation to a matrix representation."""
    if parameterization == "axis_angle":
        R = axis_angle_to_matrix(rotation)
    elif parameterization == "euler_angles":
        R = euler_angles_to_matrix(rotation, convention)
    elif parameterization == "matrix":
        R = rotation
    elif parameterization == "quaternion":
        R = quaternion_to_matrix(rotation)
    elif parameterization == "rotation_6d":
        R = rotation_6d_to_matrix(rotation)
    elif parameterization == "rotation_10d":
        R = quaternion_to_matrix(rotation_10d_to_quaternion(rotation))
    elif parameterization == "quaternion_adjugate":
        R = quaternion_to_matrix(quaternion_adjugate_to_quaternion(rotation))
    elif parameterization == "so3_log_map":
        R = so3_exp_map(rotation, **kwargs)
    else:
        raise ValueError(
            f"parameterization must be in {PARAMETERIZATIONS}, not {parameterization}"
        )
    return R
```

```
from pytorch3d.transforms import (
    matrix_to_axis_angle,
    matrix_to_euler_angles,
    matrix_to_quaternion,
    matrix_to_rotation_6d,
)
from pytorchse3.so3 import so3_log_map


def _convert_from_rotation_matrix(matrix, parameterization, convention=None, **kwargs):
    "Convert a rotation matrix to any allowed parameterization."
    if parameterization == "axis_angle":
        rotation = matrix_to_axis_angle(matrix)
    elif parameterization == "euler_angles":
        rotation = matrix_to_euler_angles(matrix, convention)
    elif parameterization == "matrix":
        rotation = matrix
    elif parameterization == "quaternion":
        rotation = matrix_to_quaternion(matrix)
    elif parameterization == "rotation_6d":
        rotation = matrix_to_rotation_6d(matrix)
    elif parameterization in ["rotation_10d"]:
        q = _convert_from_rotation_matrix(matrix, "quaternion")
        rotation = quaternion_to_rotation_10d(q)
    elif parameterization == "quaternion_adjugate":
        q = _convert_from_rotation_matrix(matrix, "quaternion")
        rotation = quaternion_to_quaternion_adjugate(q)
    elif parameterization == "so3_log_map":
        rotation = so3_log_map(matrix, **kwargs)
    else:
        raise ValueError(
            f"parameterization must be in {PARAMETERIZATIONS}, not {parameterization}"
        )
    return rotation
```

https://github.com/BillWorstell/DiffDRR/blob/main/notebooks/api/06_utils.ipynb