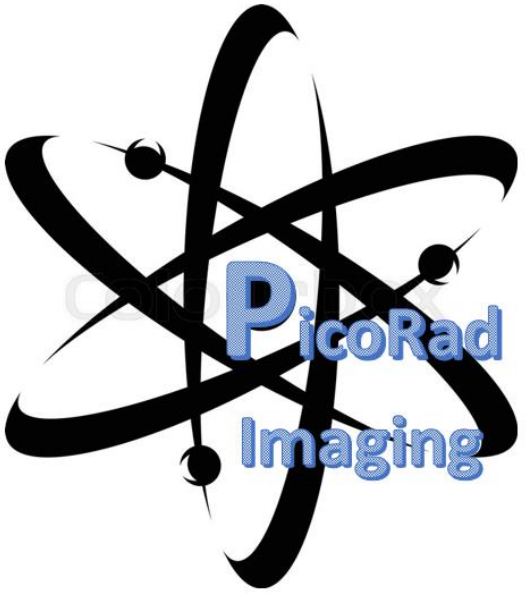


Homogeneous Coordinate Projection Imaging using diffDRR

Bill Worstell

PicoRad -> MGH

4/9/2024

https://vivekg.dev/DiffDRR/api/drr.html

DRR

```
DRR (volume:numpy.ndarray, spacing:numpy.ndarray, sdr:float, height:int,
     delx:float, width:int|None=None, dely:float|None=None, x0:float=0.0,
     y0:float=0.0, p_subsample:float|None=None, reshape:bool=True,
     reverse_x_axis:bool=False, patch_size:int|None=None,
     bone_attenuation_multiplier:float=1.0, renderer:str='siddon',
     **renderer_kwargs)
```

PyTorch module that computes differentiable digitally reconstructed radiographs.

	Type	Default	Details
volume	np.ndarray		CT volume
spacing	np.ndarray		Dimensions of voxels in the CT volume
sdr	float		Source-to-detector radius for the C-arm (half of the source-to-detector distance)
height	int		Height of the rendered DRR
delx	float		X-axis pixel size
width	int | None	None	Width of the rendered DRR (default to `height`)
dely	float | None	None	Y-axis pixel size (if not provided, set to `delx`)
x0	float	0.0	Principal point X-offset
y0	float	0.0	Principal point Y-offset
p_subsample	float | None	None	Proportion of pixels to randomly subsample
reshape	bool	True	Return DRR with shape (b, 1, h, w)
reverse_x_axis	bool	False	If pose includes reflection (in E(3) not SE(3)), reverse x-axis
patch_size	int | None	None	Render patches of the DRR in series
bone_attenuation_multiplier	float	1.0	Contrast ratio of bone to soft tissue
renderer	str	siddon	Rendering backend, either “siddon” or “trilinear”
renderer_kwargs			

The forward pass of the `DRR` module generated DRRs from the input CT volume. The pose parameters (i.e., viewing angles) from which the DRRs are generated are passed to the forward call.

https://vivekg.dev/DiffDRR/api/drr.html

DRR.forward

```
DRR.forward (*args, parameterization:str=None, convention:str=None,
             bone_attenuation_multiplier:float=None, **kwargs)
```

Generate DRR with rotational and translational parameters.

DRR.perspective_projection

```
DRR.perspective_projection (pose:diffdrr.pose.RigidTransform,
                            pts:torch.Tensor)
```

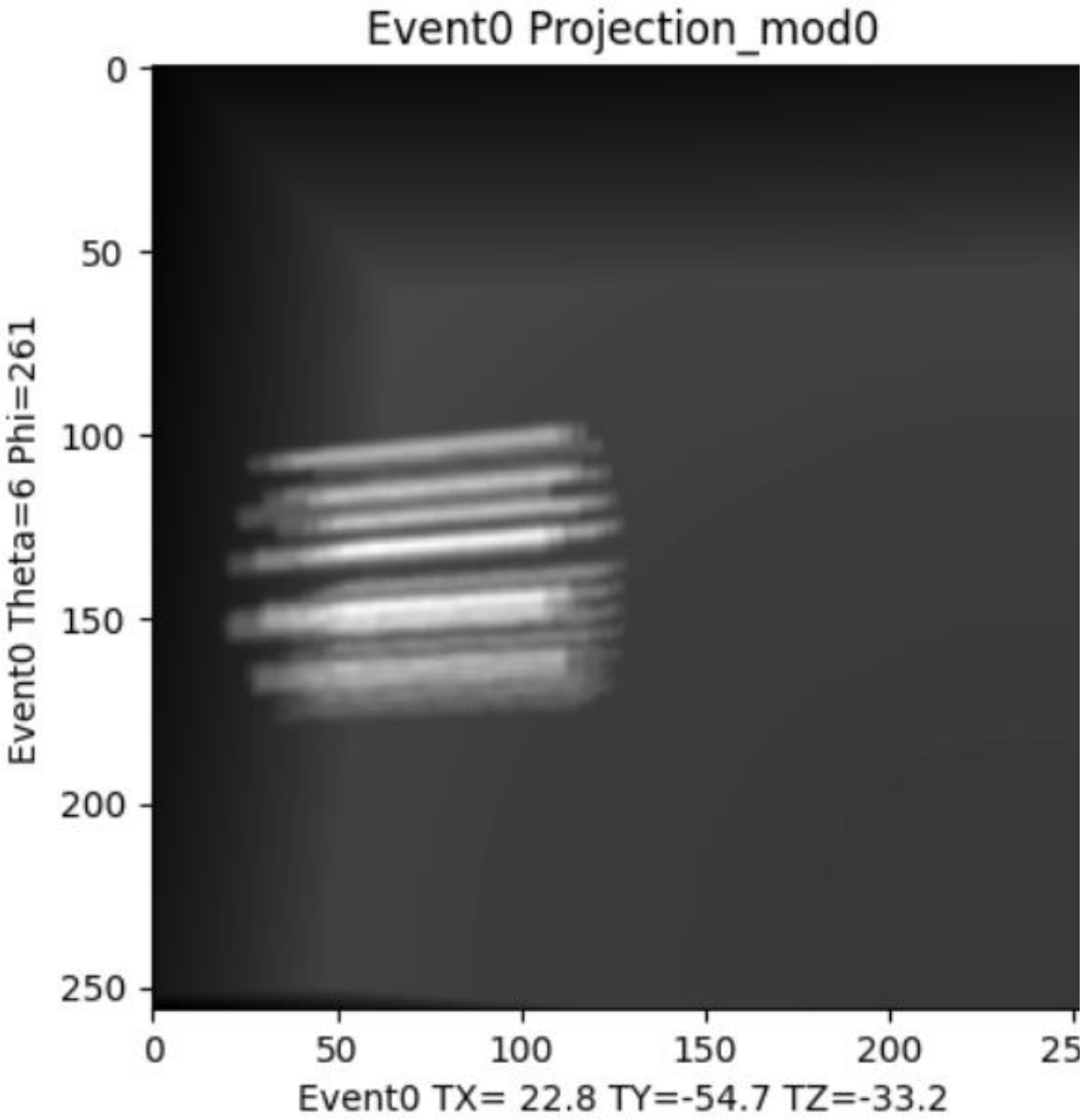
DRR.inverse_projection

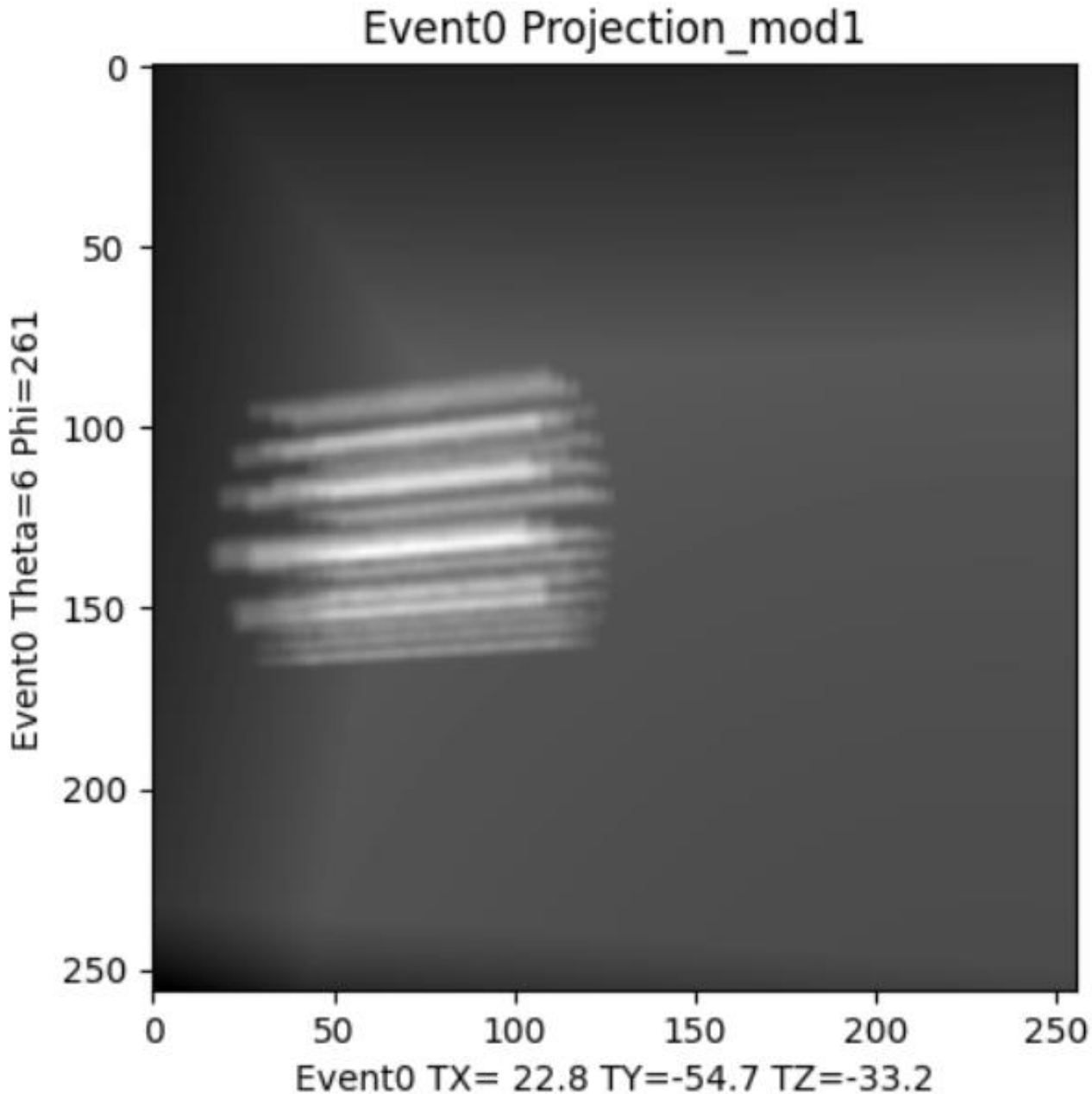
```
DRR.inverse_projection (pose:diffdrr.pose.RigidTransform,
                        pts:torch.Tensor)
```

DiffDRR / diffdrr / drr.py ↑ Top

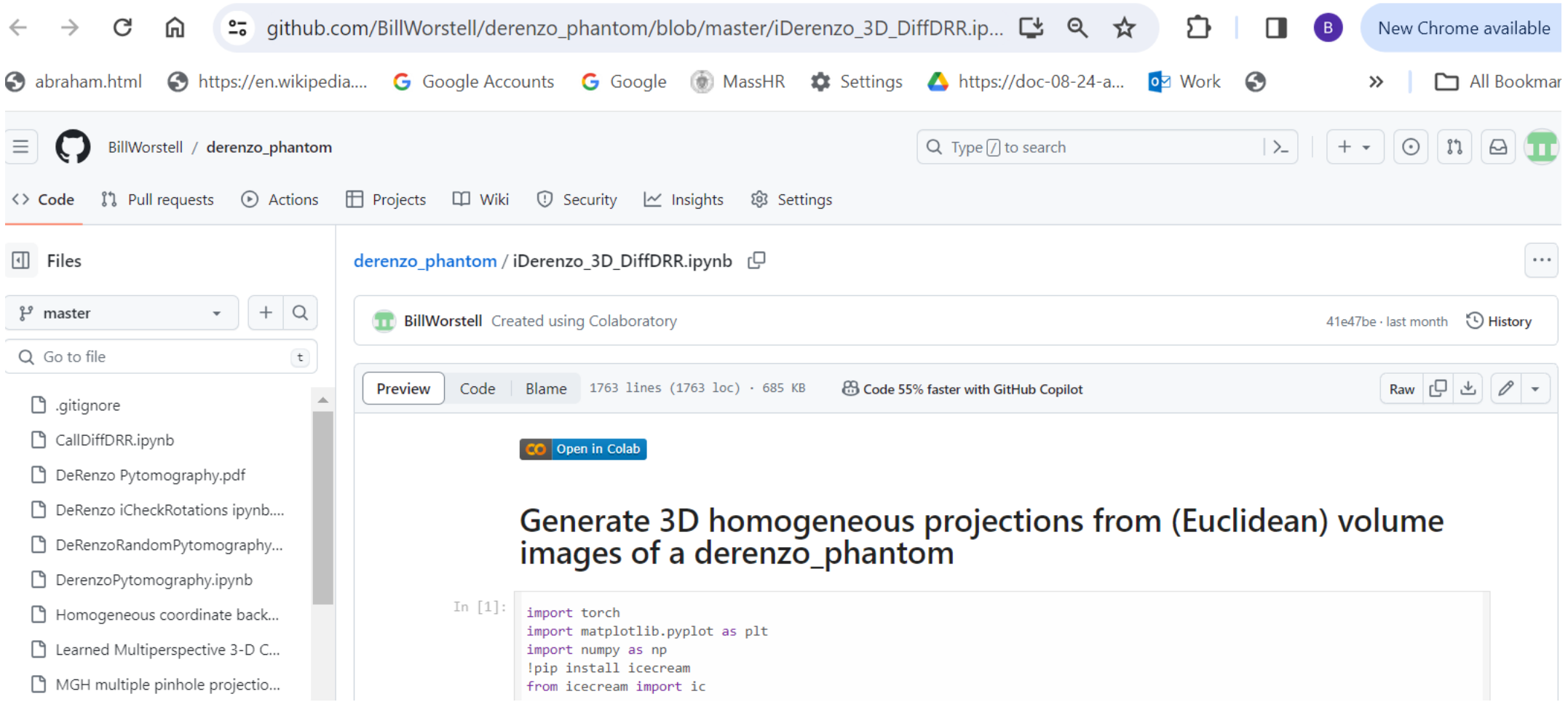
Code | Blame | 248 lines (218 loc) · 8.39 KB | Code 55% faster with Gi | Raw

```
171
172 # %% ../notebooks/api/00_drr.ipynb 13
173 from .pose import RigidTransform
174
175
176 @patch
177 def perspective_projection(
178     self: DRR,
179     pose: RigidTransform,
180     pts: torch.Tensor,
181 ):
182     extrinsic = (
183         pose.inverse().compose(self.detector.translate).compose(self.detector.flip_xz)
184     )
185     x = extrinsic(pts)
186     x = torch.einsum("ij, bnj -> bni", self.detector.intrinsic, x)
187     z = x[..., -1].unsqueeze(-1).clone()
188     x = x / z
189     return x[..., :2]
190
```


Event0 Projection_mod0
0
50
100
150
200
250
Event0 Theta=6 Phi=261
0
50
100
150
200
25
Event0 TX= 22.8 TY=-54.7 TZ=-33.2


Event0 Projection_mod1
0
50
100
150
200
250
Event0 Theta=6 Phi=261
0
50
100
150
200
250
Event0 TX= 22.8 TY=-54.7 TZ=-33.2

https://colab.research.google.com/github/BillWorstell/derenzo_phantom/blob/master/iDerenzo_3D_DiffDRR.ipynb



https://github.com/eigenvivek/DiffDRR/blob/main/diffdrr/drr.py#L18

```
# %% ../notebooks/api/00_drr.ipynb 7
class DRR(nn.Module):
    """PyTorch module that computes differentiable digitally reconstructed radiographs."""

    def __init__(
        self,
        volume: np.ndarray,  # CT volume
        spacing: np.ndarray,  # Dimensions of voxels in the CT volume
        sdr: float,  # Source-to-detector radius for the C-arm (half of the source-to-detector distance)
        height: int,  # Height of the rendered DRR
        delx: float,  # X-axis pixel size
        width: int | None = None,  # Width of the rendered DRR (default to `height`)
        dely: float | None = None,  # Y-axis pixel size (if not provided, set to `delx`)
        x0: float = 0.0,  # Principal point X-offset
        y0: float = 0.0,  # Principal point Y-offset
        p_subsample: float | None = None,  # Proportion of pixels to randomly subsample
        reshape: bool = True,  # Return DRR with shape (b, 1, h, w)
        reverse_x_axis: bool = False,  # If pose includes reflection (in E(3) not SE(3)), reverse x-axis
        patch_size: int | None = None,  # Render patches of the DRR in series
        bone_attenuation_multiplier: float = 1.0,  # Contrast ratio of bone to soft tissue
        renderer: str = "siddon",  # Rendering backend, either "siddon" or "trilinear"
        **renderer_kwargs,  # Kwargs for the renderer
    ):
        super().__init__()

        # Initialize the X-ray detector
        width = height if width is None else width
        dely = delx if dely is None else dely
        n_subsample = (
            int(height * width * p_subsample) if p_subsample is not None else None
        )
```

```
        # Initialize the volume
        self.register_buffer("spacing", torch.tensor(spacing))
        self.register_buffer("volume", torch.tensor(volume).flip([0]))
        self.reshape = reshape
        self.patch_size = patch_size
        if self.patch_size is not None:
            self.n_patches = (height * width) // (self.patch_size**2)

        # Parameters for segmenting the CT volume and reweighting voxels
        self.air = torch.where(self.volume <= -800)
        self.soft_tissue = torch.where((-800 < self.volume) & (self.volume <= 350))
        self.bone = torch.where(350 < self.volume)
        self.bone_attenuation_multiplier = bone_attenuation_multiplier

        # Initialize the renderer
        if renderer == "siddon":
            self.renderer = Siddon(**renderer_kwargs)
        elif renderer == "trilinear":
            self.renderer = Trilinear(**renderer_kwargs)
        else:
            raise ValueError(f"renderer must be 'siddon', not {renderer}")
```

```
    def reshape_transform(self, img, batch_size):
        if self.reshape:
            if self.detector.n_subsample is None:
                img = img.view(-1, 1, self.detector.height, self.detector.width)
            else:
                img = reshape_subsampled_drr(img, self.detector, batch_size)
        return img
```

%% ../notebooks/api/00_drr.ipynb 7

DiffDRR / notebooks / api / 00_drr.ipynb

eigenvivek Add projection code ✓

Preview | Code | Blame | 458 lines (458 loc) · 15.7 KB | Code 55% faster with

In [ ]:
```
#| default_exp drr
```

In [ ]:
```
#| hide
from nbdev.showdoc import *
```

In [ ]:
```
#| export
from __future__ import annotations

import numpy as np
import torch
import torch.nn as nn
from fastcore.basics import patch

from diffdrr.detector import Detector
```

DRR

`DRR` is a PyTorch module that compues differentiable digitally reconstructed radiographs. The viewing angle for the DRR (known generally in computer graphics as the *camera pose*) is parameterized by the following parameters:

- SDR : Source-to-Detector radius (half of the source-to-detector distance)
- $\mathbf{R} \in \mathrm{SO}(3)$: a rotation
- $\mathbf{t} \in \mathbb{R}^3$: a translation

If using Euler angles, the parameters are

- `alpha` : Azimuthal angle
- `beta` : Polar angle
- `gamma` : Plane rotation angle
- `bx` : X-dir translation
- `by` : Y-dir translation
- `bz` : Z-dir translation
- `convention` : Order of angles (e.g., `ZYX`)

`(bx, by, bz)` are translational parameters and `(alpha, beta, gamma)` are rotational parameters. The rotational parameters are detailed in Spherical Coordiantes Tutorial.

In [ ]:
```
#| export
class DRR(nn.Module):
    """PyTorch module that computes differentiable digitally reconstructed radiographs."""

    def __init__(
        self,
        volume: np.ndarray,  # CT volume
        spacing: np.ndarray,  # Dimensions of voxels in the CT volume
        sdr: float,  # Source-to-detector radius for the C-arm (half of the source-to-detector
        height: int,  # Height of the rendered DRR
        delx: float,  # X-axis pixel size
        width: int | None = None,  # Width of the rendered DRR (default to `height`)
        dely: float | None = None,  # Y-axis pixel size (if not provided, set to `delx`)
        x0: float = 0.0,  # Principal point X-offset
        y0: float = 0.0,  # Principal point Y-offset
        p_subsample: float | None = None,  # Proportion of pixels to randomly subsample
        reshape: bool = True,  # Return DRR with shape (b, 1, h, w)
        reverse_x_axis: bool = False,  # If pose includes reflection (in E(3) not SE(3)), rever
        patch_size: int | None = None,  # Render patches of the DRR in series
        bone_attenuation_multiplier: float = 1.0,  # Contrast ratio of bone to soft tissue
        renderer: str = "siddon",  # Rendering backend, either "siddon" or "trilinear"
        **renderer_kwargs,  # Kwargs for the renderer
    ):
        super().__init__()
```

```
# Initialize the X-ray detector
width = height if width is None else width
dely = delx if dely is None else dely
n_subsample = (
    int(height * width * p_subsample) if p_subsample is not None else None
)
self.detector = Detector(
    sdr,
    height,
    width,
    delx,
    dely,
    x0,
    y0,
    n_subsample=n_subsample,
    reverse_x_axis=reverse_x_axis,
)

# Initialize the volume
self.register_buffer("spacing", torch.tensor(spacing))
self.register_buffer("volume", torch.tensor(volume).flip([0]))
self.reshape = reshape
self.patch_size = patch_size
if self.patch_size is not None:
    self.n_patches = (height * width) // (self.patch_size**2)

# Parameters for segmenting the CT volume and reweighting voxels
self.air = torch.where(self.volume <= -800)
self.soft_tissue = torch.where((-800 < self.volume) & (self.volume <= 350))
self.bone = torch.where(350 < self.volume)
self.bone_attenuation_multiplier = bone_attenuation_multiplier
```

```
# Initialize the renderer
if renderer == "siddon":
    self.renderer = Siddon(**renderer_kwargs)
elif renderer == "trilinear":
    self.renderer = Trilinear(**renderer_kwargs)
else:
    raise ValueError(f"renderer must be 'siddon', not {renderer}")

def reshape_transform(self, img, batch_size):
    if self.reshape:
        if self.detector.n_subsample is None:
            img = img.view(-1, 1, self.detector.height, self.detector.width)
        else:
            img = reshape_subsampled_drr(img, self.detector, batch_size)
    return img
```

https://github.com/eigenvivek/DiffDRR/blob/main/diffdrr/renderers.py

https://colab.research.google.com/github/BillWorstell/derenzo_phantom/blob/master/CollectiveDiffDRR.ipynb