

# FreeCAD World

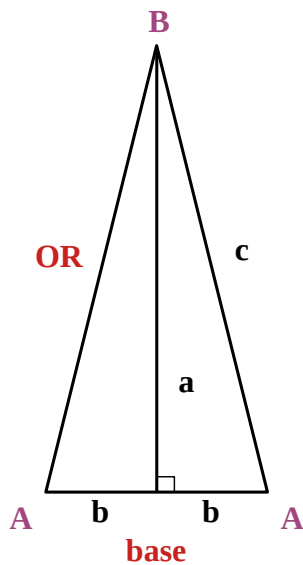
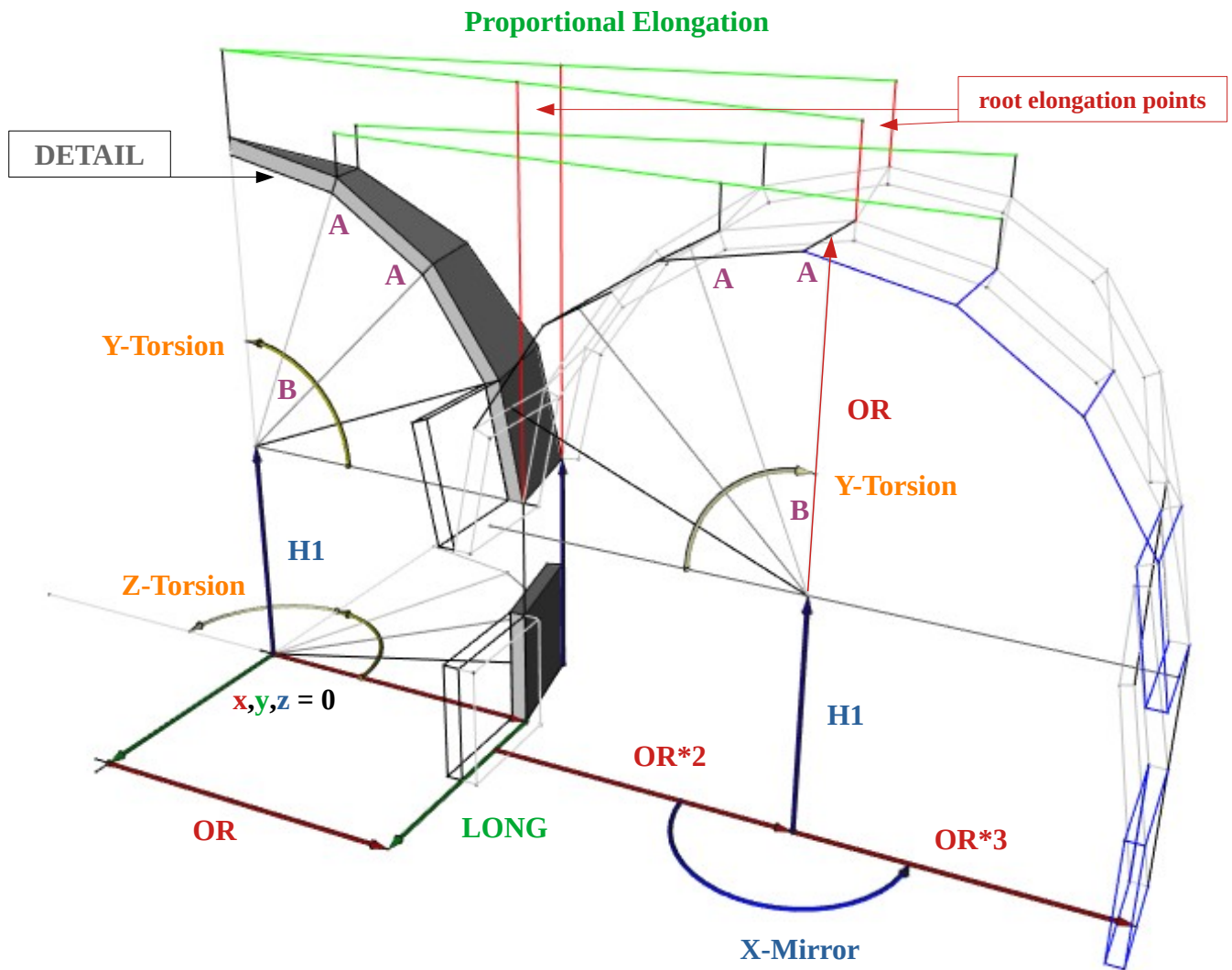
[ notes ]

# Config types

[ all types and values ( except **str("eg. Value or valUe")** ) are **case sensitive** ]

- **int():** integers: (1, 2, 15, 144, 2048, etc);
- **float():** floating point numbers: (1.2, 2.3, 15.123, 144.9, 2048.1024,  $\frac{3}{4}$ ,  $\frac{1}{2}$ , etc);
- **bool():** boolean **True** or **False**, 1 or 0. (In real life is **On/Off**);
- **tuple():** in config used as **comma separated** tuple of integers or floats;
- **str():** string as any human readable words, eg **str("CORNER")**;
- **dict():** **comma separated** dictionary of options: str(key) = value, where value may be **one** of described above types;

# The Principle



$$B = 360^\circ / \text{DETAILS}$$

$$A + B/2 = 90^\circ$$

$$A = (180^\circ - B)/2$$

$$a = OR * \sin(A) = \text{cathetus}$$

$$b = OR * \cos(A) = \text{side } b$$

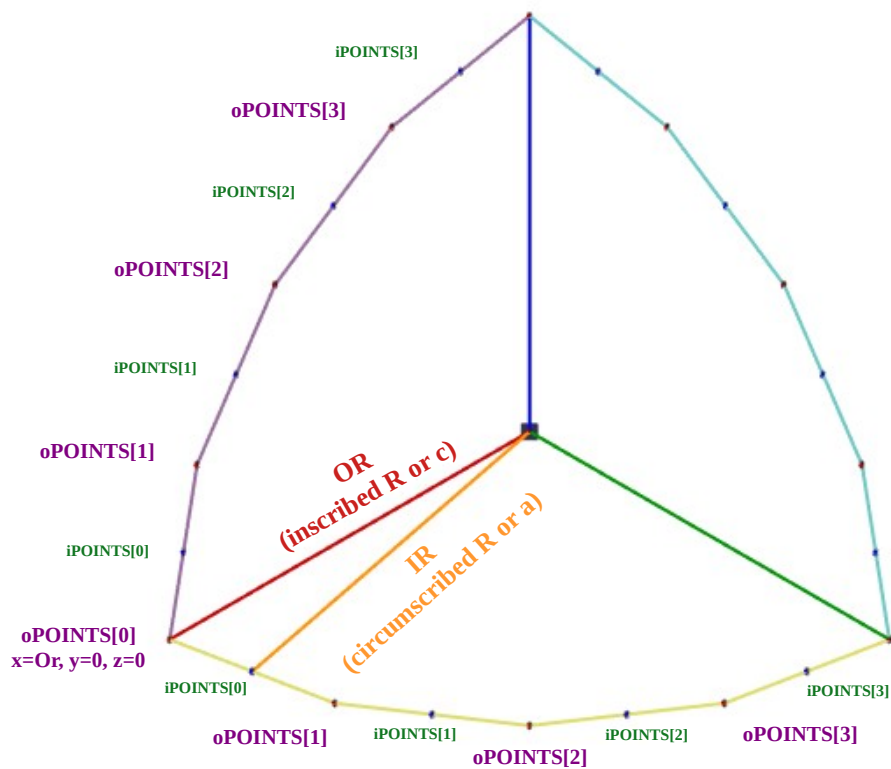
$$OR = a / \cos(B/2) = c = \text{hypotenuse}$$

$$\text{base} = b * 2 = OR * 2 * \cos(A)$$

**Isosceles Triangle  
as base of  
construction**

# Pointing

- **oPOINTS;**
- **iPOINTS;**



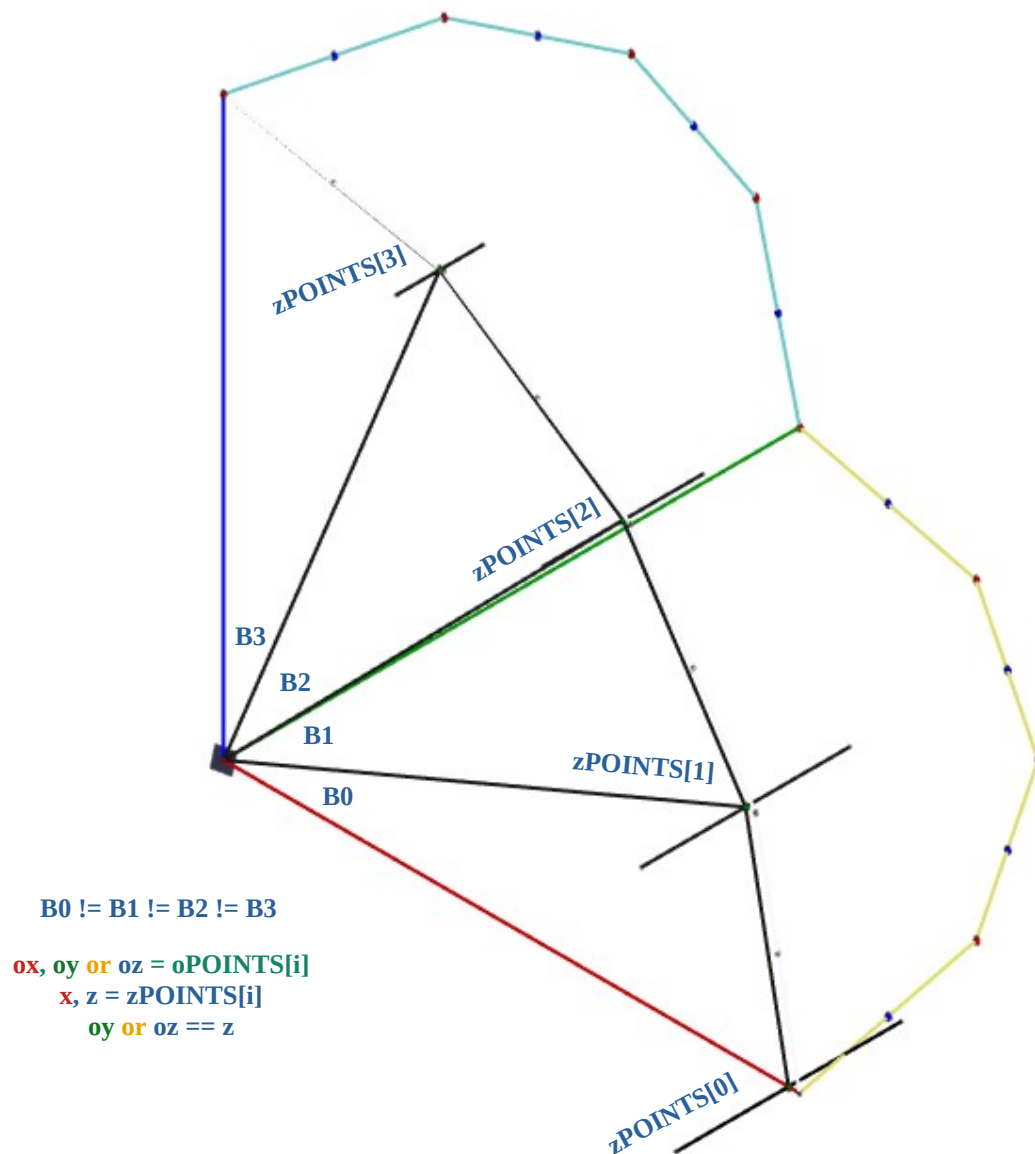
**See:**

- **oPOINTS** or **iPOINTS** = `PairOfCompasses.clockWiseArray(x, y);`
- `oi_pointing.FCStd;`

```
class PairOfCompasses(Trigon):
    _deviation_ = float(0.01)

    def clockWiseArray(self, x, y):
        """
            Clockwise bottom view
        """
        A = self.DETAIL_A
        B = self.DETAIL_B
        """
            Extra solution for thorus to find proportions
            on y axis of higher 'horison poly wireframe'.
            System-wide single deviation is:
            x == 0.01 mm if 0 on x & y axis
            in case of self.DETAILS % 4 == 0
        """
        if x < self.DEVIATION: x = self.DEVIATION
        base = self.isocelesBase(x)
        points = list( [[ x, y ]] )
        for i in range(self.POLY_QUARTER):
            x -= self.rightSideB_ByA(base, A)
            y += self.rightCathetusA_ByA(base, A)
            points.append([ x, y ])
            A -= B
        return points
```

- **zPOINTS;**

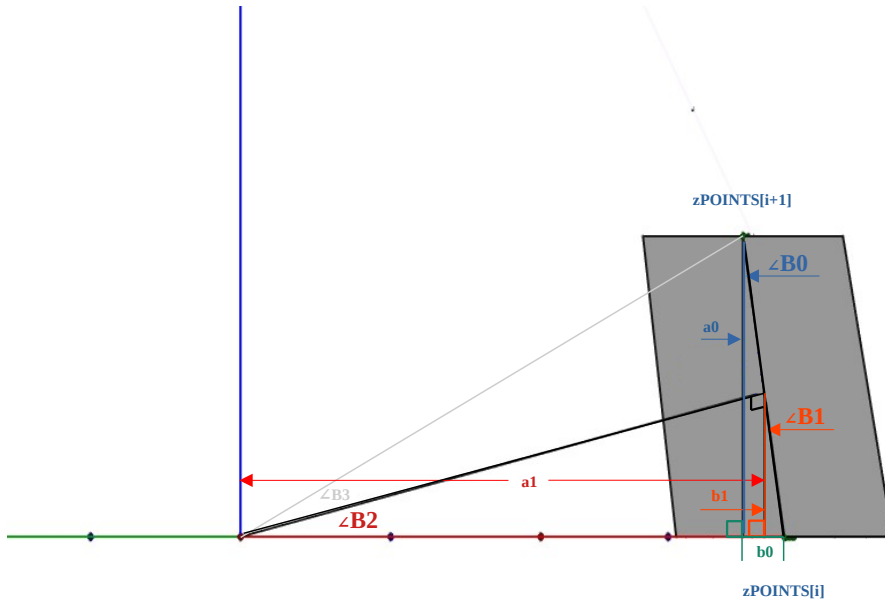


See:

- **TwoPoints.zPolyPoints(Or);**
- **z\_pointing.FCStd;**

```
def zPolyPoints(self, Or):
    """
    Receives: OR of inscribed into circle polygon;
    Returns: sequence of mid x, z points of base
    of isosceles triangle on y=0 and ZERO_Z=0;
    """
    oPOINTS = self.oPolyPoints(Or)
    points = list()
    for x, z in oPOINTS:
        x, y = self.firstSectionToCircumscribed(x, z)
        x, y = self.circumscribedCounterClockWiseOnce(x, y)
        points.append([ x, z ])
    return points
```

- **fPOINTS;**



$$\angle B0 == \angle B1 == \angle B2$$

$$\angle B3 \neq \angle B2$$

**a0 / 2 != b1**

```
a1, b1 = fPOINTS[i]
```

**See:**

- $(\angle B2 + \angle B3), x, a0, b0 = \text{GraduatedArc.sequenceByGraduatedArc}(z\text{POINTS});$
- $a1, b1$  or  $f\text{POINTS} = \text{MonoGraduatedArc.monoSurfaceMidPoints}(\text{Or}, \text{manipulator});$
- $f\_pointing.FCStd;$

```
class GraduatedArc(TwoPoints):
    def sequenceByGraduatedArc(self, sequence):
        """
        To work with surface zPOINTS;
        """
        assert isinstance(sequence, list), "TypeError: sequence must be type of list"
        realB = 0
        points = list()
        for x, y in sequence[:-1]:
            next_i = sequence.index([x, y])+1
            next_x, next_y = sequence[next_i]
            next_c = self.rightHypothenuse_ByAB(next_x, next_y)
            B = (self.angleB_ByAB(next_x, next_y) - realB)
            a = self.rightSideB_ByA(next_c, 90-B)
            x = self.rightHypothenuse_ByAB(x, y)
            b = x - self.rightCathetusA_ByBA(a, 90-B)
            realB += B
            points.append([ B, x, a, b ])
        return points
```

```

class MonoGradedArc(Mono3DPoints):

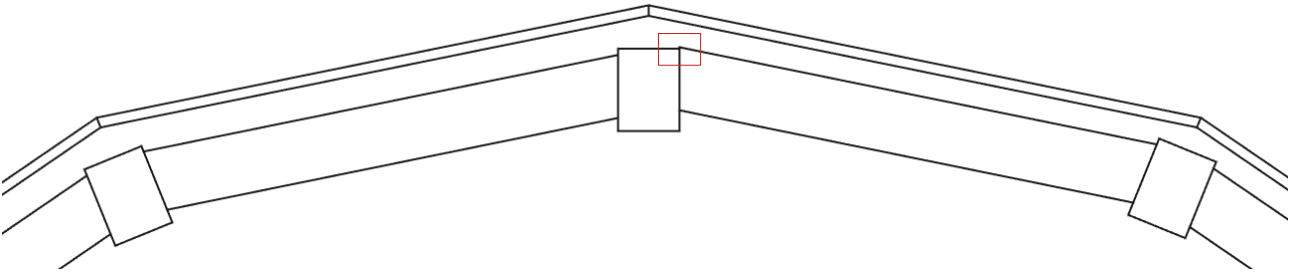
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.fPOINTS = list()

    def monoSurfaceMidPoints(self, Or, manipulator=0):
        """
        Receives: Or and manipulator;
        Operates with: GraduatedArc.sequenceByGraduatedArc(list);
        Returns: surface mid points, where cathetus lies
        on face at an angle of 90 degrees;
        All _bottom wires_ of faces lies on z=y=0;
        """
        Or += manipulator
        zPOINTS = self.zPolyPoints(Or)
        arc = self.sequenceByGraduatedArc(zPOINTS)
        points = list()
        for B, x, a, b in arc:
            B = self.angleB_ByAB(a, b)
            c = self.rightCathetusA_ByA(x, 90-B)
            a = self.rightCathetusA_ByA(c, 90-B)
            b = self.rightSideB_ByCA(c, a)
            points.append([a, b])
        return points

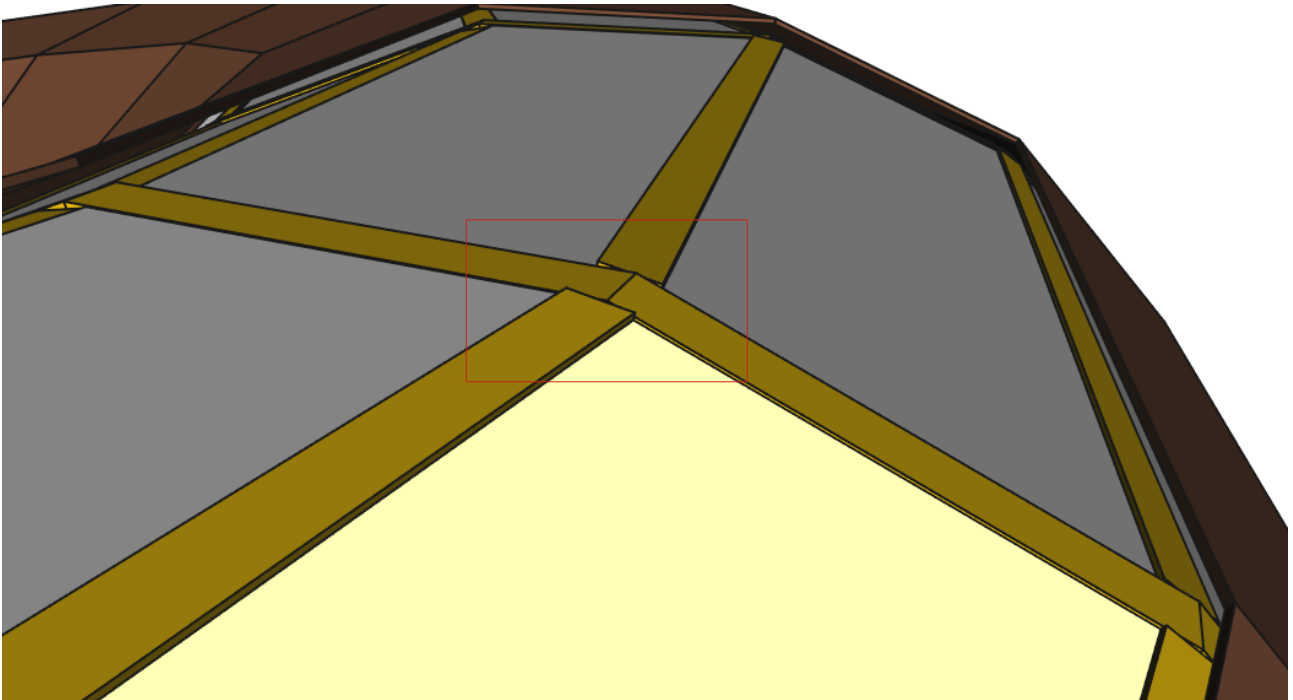
```

[illegible]

# Manipulator issue in torus mode



Manipulates differences while proportional elongation:



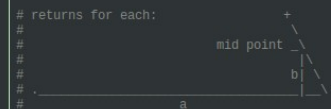
See:

```
class Mono3DPoints(ModelRoot):
    def mono3DPolyPoints(self, Or, z=0, manipulator=0):
        Or += manipulator
        OPOINTS = self.oPolyPoints(Or)
        points = list([[x, y, z] for x, y in OPOINTS ])
        for i in range(1, len(OPOINTS)):
            x, y, z = *OPOINTS[i], z
            outer = self._arrange3DPoints(x, y, z)
            points.append(outer)
        return points

class MonoGraduatedArc(Mono3DPoints):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.fPOINTS = list()

    def monoSurfaceMidPoints(self, Or, manipulator=0):
        """
        Receives: Or and manipulator;
        Operates with: GraduatedArc.sequenceByGraduatedArc(list);
        Returns: surface mid points, where cathetus lies
        on face at an angle of 90 degrees;
        All _bottom wires_ of faces lies on z=y=0;
        """
        Or += manipulator
        zPOINTS = self.zPolyPoints(Or)
        arc = self.sequenceByGraduatedArc(zPOINTS)
        points = list()
        for B, x, a, b in arc:
            B = self.angleB.ByAB(a, b)
            c = self.rightCathetusA.ByA(x, 90-B)
            a = self.rightCathetusA.ByA(c, 90-B)
            b = self.rightSideB.ByCA(c, a)
            points.append([a, b])
        return points
```

# returns for each:



```

class Frame3DPoints(ModelRoot):
    def frame3DConverter(self, x, y, z, **kwargs):
        """
        To align well reduced root 3D points of dome on x,y,z axis
        """
        assert kwargs['reductor'] > 0, "ValueError: zero length of reductor"
        x, y, z = self.first_3D_SectionToCircumscribed(x,y,z, **kwargs)
        x, y, z = self.first_3D_SectionToInscribed(x, y, z)
        points = self.clockWiseArray(x, y)
        return [ [x, y, z] for x, y in points ]

    def frame3DPolyPoints(self, Or, z=0, manipulator=0, **kwargs):
        """
        Receives:
            Or of inscribed polygon;
            global z (or ZERO_Z) of construction;
            z_correct as an additional move in/out frame
            reductor as a material (frame) height;
        Returns:
            list of all 3D points of quarter of dome;
        """
        Or += manipulator
        OPOINTS = self.oPolyPoints(Or)
        IPOINTS = self.iPolyPoints(Or, **kwargs)
        outer = [ [x, y, z] for x, y in OPOINTS ]
        REDUCED = self.toInscribedConverter(*IPOINTS[0])
        inner = [ [x, y, z] for x, y in REDUCED ]
        points = list([ [outer, inner] ])
        for i in range(1, len(OPOINTS)):
            x, y, z = OPOINTS[i], z
            outer = self._arrange3DPoints(x, y, z)
            x, y, z3d = outer[0]
            inner = self.frame3DConverter(x, y, z3d, z0=z, **kwargs)
            points.append([ outer, inner ])
        return points

```

Definitions in:

- **FrameRoot.wireFrame;**
- **InsulantRoot.wireFrame;**

```

class FrameRoot(FrameModelLayer):
    """Creator of bar root objects"""
    _mtrl_ = FrameWireFrame._mtrl_
    _botm_ = WireFrame._btm_
    _hiba_ = WireFrame._hi_
    _hdrp_ = str('hidrops')
    _vplb_ = WireFrame._vpl_ + _mtrl_
    _hplb_ = FrameWireFrame._hpl_
    _lng_ = FrameWireFrame._rof_
    _shte_ = FrameWireFrame._sht_

    def __init__(self, tp, matW, matH, *args, **kwargs):
        self._setMaterial(width=matW, height=matH, RGB=(1.0,0.8,0.0), **kwargs)
        super().__init__(tp, *args, **kwargs)

    def wireFrame(self, ny, nz, marking=True, **kwargs):
        super().wireFrame(ny, nz, **kwargs)
        mnp = (self.MTL.W/2)/self.POLY_QUARTER
        mnp = mnp/self.DETAILS if self.dometric else -mnp
        return self._createFrameWireFrame(
            ny, self.MTL, manipulator=mnp, marking=marking
        )

```

```

class InsulantRoot(MonoRoot, InsulantModelLayer):
    _mtrl_ = InsulantWireFrame._mtrl_

    def __init__(self, tp, thickn, *args, # absolute self __init__
                 frame_w=None, RGB=(1.0,1.0,1.0),
                 **kwargs):
        assert thickn, "No Insulant thickness defined"
        assert frame_w, "No Frame width defined"
        self._setMaterial(thickn=thickn, RGB=RGB, frame_w=frame_w, **kwargs)
        MonoGradedArc.__init__(self, tp, *args, **kwargs)

    def wireFrame(self, *args, **kwargs):
        mnp = -(self.MTL.FRAME_W/(self.DETAILS/2)) if self.corn else 0 # additional move just for corner
        super().wireFrame(*args, marking=True, manipulator=mnp, **kwargs)

```