```python
def full_Model2POV(smallmodel,det):
    # Read a model without hole and convert it into a POV file

    """
    Comment:
        Main processing with c-code
    Input:
        smallmodel: Small model name (Path to file of small model)
    Output:
        aOutputname: Path of POV file of the model
    Used function:
        None
    Used in function:
        'GeoRayImg.Model2POV'
    """

    # Get model information
    dsm = xdibias.Image(smallmodel).readImageData()  # Create a numpy array with the image data of the model with XDibias

    # Split path of model file
    path1, bigmodelname = os.path.split(smallmodel)

    # Comment about the coordinate transformation between model and POV:
    # x, y and z refers the local defined POV coordinate system:
    #  x: Right
    #  y: Up
    #  z: Depth
    #  Origin: Lower left corner
    #  Pixel center: [0, h, 0]
    # Intensity I with indices [row, column] refers the coordinate system of the SAR image (and hence the model):
    #  Origin: Upper left corner
    #   Pixel center: [0, 0, h]
    # Transformation of the two coordinate systems:
    #  x = column
    #  y = dsm[row, column]
    #  z = dsm.shape[0] - 1 - row;
    #  row = dsm.shape[0] - 1 - z
    #  column = x
    #  I[row, column] = y

    # Number of rows and columns of the model
    rows = dsm.shape[0]  # Number of rows of the model
    cols = dsm.shape[1]  # Number of columns of the model

    #  -----------------------
    if len(det.path_to_adapted_pov_file) != 0:

        fin = open(det.path_to_adapted_pov_file, "r")  # Open POV file
        POVstr_full_scene = fin.read()  # Read POV file of full scene
        fin.close()  # Close POV file

    else:

        POVstr_full_scene = ' '

    #  -----------------------

    # Output name of POV file
    aOutputname = smallmodel + '/%s_full.pov' % bigmodelname

    # Give the definition of the needed matrix
    modelmarktext = " // %s" % smallmodel

    # Find the pixel that should be saved (at least two neighbour == 1)
    pixelnr_m = np.zeros((cols, rows))  # Initialisation
    pixelnr_m = pixelnr_m.astype(np.int)  # Convert datatype

    # Main processing with c-code
    code = '''
    # line 249 "dem2pov.py" // real line number + 1
```

```
    int x, z;
    int Pixelordernumber = 0; // the found pixel number
    double y;

    FILE * file = fopen(aOutputname.c_str() ,"w");
        if ( ! file) {
        fprintf(stderr,"Output file could not be opened.\\n");
        exit(0);
    }

    // 1.5. write the head part of this file
    fprintf(file,"mesh2 {\\n  vertex_vectors {%s\\n %d\\n",modelmarktext.c_str(),rows*cols); //# how many pixel saved

    // 2. loop runs on the coordinate system of the pov. left to right(east), unten to oben(north)
    for ( z=0; z<rows; z++)      // from lower to upper, (north)
    {
        for ( x=0; x<cols; x++ )     // from left to right, (east)
        {
            pixelnr_m(x,z) = Pixelordernumber;  // generate the pixel order number
            Pixelordernumber = Pixelordernumber +1;

            y = dsm(rows -1 -z, x);

            /*fprintf(file,"    <%d, %.3f, %d>,\\n",x,y,z );*/
            fprintf(file,"    <%d, %d, %d>,\\n",x,int(y),z );//<68, 5627, 115>,
            //y: .3f or d? or interger ???!!! should change the scale ?????????????????????????????????????
        }
    }

    //3. loop the triangle
    fprintf(file,"  }\\n  face_indices {\\n    %d\\n",(cols-1)*(rows-1)*2);

    for ( z=1; z<rows; z++) //from lower to upper, (north)
    {
        for ( x=0; x<cols-1; x++ )  //from left to right, (east)
        {
            // original not consider the height difference of the 4 points
            /*fprintf(file,"    <%d, %d, %d>,\\n", pixelnr_m(x,z),pixelnr_m(x+1,z-1),pixelnr_m(x,z-1));//<69, 1, 0>,*/
            /*fprintf(file,"    <%d, %d, %d>,\\n", pixelnr_m(x,z),pixelnr_m(x+1,z),pixelnr_m(x+1,z-1));//<69, 70, 1>,*/

            //if (fabs(Height(x,z)-Height(x+1,z-1)) <= fabs(Height(x+1,z)-Height(x,z-1)))
            if ( fabs(dsm(rows -1 -z, x)-dsm(rows  -z, x+1))  <= fabs( dsm(rows -1 -z, x+1) - dsm(rows -z, x)  ) )
            {
                fprintf(file,"    <%d, %d, %d>,\\n",pixelnr_m(x,z),pixelnr_m(x+1,z-1),pixelnr_m(x,z-1)); //<69, 70, 0>,write the pixel numbers
                fprintf(file,"    <%d, %d, %d>,\\n",pixelnr_m(x,z),pixelnr_m(x+1,z),pixelnr_m(x+1,z-1));
            }
            else
            {
                fprintf(file,"    <%d, %d, %d>,\\n", pixelnr_m(x,z),pixelnr_m(x+1,z),pixelnr_m(x,z-1));
                fprintf(file,"    <%d, %d, %d>,\\n", pixelnr_m(x+1,z),pixelnr_m(x+1,z-1),pixelnr_m(x,z-1));
            }
        }
    }

    //4, write end text of the file
    //fprintf(file,"  }\\n  texture {\\n    pigment { color rgb<1, 1, 1> }\\n finish { ambient rgb<0, 0, 0> diffuse 0.5 }\\n  }\\n}\\n");
    //fprintf(file,"  }\\n \\n}\\n //endmodel \\n \\n");
    fprintf(file,"  }\\n \\n}\\n \\n \\n");
```

```python

        // Add geometric information for full scene model (visibility check for depth level > 2)
        fprintf(file, "%s", POVstr_full_scene.c_str());

        fclose(file);
        '''
        err = weave.inline(code,['dsm', 'cols', 'rows', 'aOutputname', 'modelmarktext', 'pixelnr_m', 'POVstr_full_scene'], type_converters=converters.blitz)

        print "Model '%s' is converted into '%s' without hole." % (smallmodel, aOutputname)

        # Output
        return aOutputname



    def read_Metadata(metadatafile, bigmodel, smallmodel, det):
        # Read parameters from meta data file of image
        """
        Input:
                metadatafile: Orbit meta data file
                bigmodel: Big model name (Path to file of big model)
                smallmodel: Small model name (Path to file of small model)
                det: Definition of input parameters (Look at class 'InputPara' in 'Script.py')
        Output:
                sensor: Name of sensor
                filename:
                        SAR case: Filename of image
                        Optical case: Record time of image
                heading: Heading angle in [°]
                        SAR case: Heading angle of SAR sensor (clockwise) in [°] = Angle from UTM north direction Y to azimuth image axis of SAR image (clockwise) in [°] = Angle from UTM north direction Y to flight direction of SAR sensor (clockwise) in [°]
                        Optical case: Heading angle of optical sensor in [°]
                AngOfView:
                        SAR case: Angle from UTM north direction Y to line of sight of SAR sensor (clockwise) in [°] = Heading angle of SAR sensor + 90° (clockwise) in [°]
                        Optical case: Angle from UTM north direction Y to line of sight of optical sensor (clockwise) in [°]
                incidence:
                        SAR case: Interpolated incidence angle on ground of the image according to model center (center of model box) coordinates in [°]
                        Optical case: Incidence angle on ground in [°]
                meanHeight: Mean height (reference height) of image in [m]
                PS_columns: Spatial resolution along columns (in UTM east direction X) in [m] --> Is different for different sensors
                PS_rows: Spatial resolution along rows (in UTM north direction Y) in [m] --> Is different for different sensors
        Used function:
                'GeoRayImg.find_1Word'
                'GeoRayImg.find_2Word'
                'GeoRayImg.InterpIncidence'
                'GeoRayImg.RPC2Ang'
        Used in function:
                'GeoRayImg.gen_HeaderEndPOV'
                'GeoRayImg.gen_RaySAR_Para'
                'GeoRayImg.GeocodeSimuImg1'
                'GeoRayImg.GeocodeSimuImg2'
                'GeoRayImg.Model2GeoI'
                'GeoRayImg.select_ImgOption'
                'GeoRayImg.create_foldername'
                'Script.py'
        """

        # SAR image
        if det.ImgOption == 1:
            sensor = find_1Word(metadatafile, 'mission')  # Name of SAR sensor
```

```python
        filename = find_1Word(metadatafile, 'filename')[
            :-4]  # Filename of SAR image
        # Heading angle of SAR sensor in [°] = Angle from UTM north direction Y to flight direction of SAR sensor (clockwise) in [°] = Angle from UTM north direction Y to azimuth image axis of SAR image (clockwise) in [°]
        heading = float(find_1Word(metadatafile, 'headingAngle'))
        # Angle from UTM north direction Y to line of sight of SAR sensor (clockwise) in [°] = Heading angle of SAR sensor + 90° (clockwise) in [°]
        AngOfView = heading + 90
        # Interpolated incidence angle on ground of the SAR image according to model center (center of model box) coordinates in [°]
        incidence = InterpIncidence(metadatafile, bigmodel, det)
        # Mean height (reference height) of SAR image in [m]
        meanHeight = float(find_1Word(metadatafile, 'meanHeight'))
        # Spatial resolution along columns (in UTM east direction X) of SAR image in [m]
        PS_columns = float(find_2Word(metadatafile, 'pixelSpacing', 'easting'))
        # Spatial resolution along rows (in UTM north direction Y) of SAR image in [m]
        PS_rows = float(find_2Word(metadatafile, 'pixelSpacing', 'northing'))

    # Optical image
    else:  # det.ImgOption == 2:
        # Create an image object of the optical image meta data with XDibias
        img = xdibias.Image(metadatafile)

        sensor = img.Sensor  # Name of optical Sensor
        filename = img.RecordTime  # Record time of optical image
        heading = img.Azimuth  # Heading angle of optical sensor in [°]
        # Angle from UTM north direction Y to line of sight of optical sensor (clockwise) in [°] and incidence angle on ground of optical sensor in [°]

        # !!!!!!!!!!!!!

        # AngOfView = 0   # temporary for Data Fusion Contest 2019 (nadir view -> incidence angle = 0°)
        # incidence = 0   # temporary for Data Fusion Contest 2019 (nadir view -> incidence angle = 0°)

        # !!!!!!!!!!!!!

        AngOfView, incidence = RPC2Ang(smallmodel, metadatafile)   # Activate again after Data Fusion Contest 2019

        # Mean height (reference height) of optical image in [m]
        meanHeight = det.meanHeight
        # Spatial resolution along columns (in UTM east direction X) of optical image in [m]
        PS_columns = img.XCellRes
        # Spatial resolution along rows (in UTM north direction Y) of optical image in [m]
        PS_rows = img.YCellRes

    # Output
    return sensor, filename, heading, AngOfView, incidence, meanHeight, PS_columns, PS_rows


def gen_HeaderEndPOV(bigmodel, smallmodel, metadatafile, det):
    # Generate additional code parts (header and end part) for the POV file
    # 1. Generate header and end part for the POV file
    # 2. Open the POV file which is already in existence
    # 3. Add generated header and end part to POV file
    """
    Input:
            bigmodel: Big model name (Path to file of big model)
            smallmodel: Small model name (Path to file of small model)
            metadatafile: Orbit meta data file
            det: Definition of input parameters (Look at class 'InputPara' in 'Script.py')
    Output: Additional code parts for the POV file and Information needed for POV-Ray
            headtext: Header of POV file (Parameters for Ray Tracing)
            endtext: End part of POV file (Model information)
```

```
            columns_simu: Number of columns of simulated image
            rows_simu: Number of rows of simulated image
    Used function:
            'GeoRayImg.read_ModelPara'
            'GeoRayImg.read_Metadata'
            'GeoRayImg.UpRight'
            'GeoRayImg.pts'
            'GeoRayImg.cal_SunPosition'
    Used in function:
            'GeoRayImg.POVfile_addPara'
    """

    global sensor, filename, heading, AngOfView, incidence, meanHeight, PS_columns, PS_rows

    # Dummy distance in horizontal plane for calculating sensor position
    D = 500  # unit [m]

    # Split path of model file
    path1, bigmodelname = os.path.split(bigmodel)

    # Keep a list of lines that we want to save in POV file
    headtext = []  # Initialisation: Line to save in header of POV file
    endtext = []  # Initialisation: Line to save in end part of POV file

    # Read parameters of model
    model, H_max, H_min, H_mid, L, B, H = read_ModelPara(bigmodel, det)

    # Get information from orbit meta data (Read parameters from meta data file of image)
    sensor, filename, heading, AngOfView, incidence, meanHeight, PS_columns, PS_rows = read_Metadata(
        metadatafile, bigmodel, smallmodel, det)

    # Calculate parameters for Ray Tracing
    # Working with orthographic camera: Parallel projection

    # GIS data (CityGML)
    if det.ModelOption == 1:
        # Look at (Position where the sensor is looking at in POV coordinate system): Position of model center (center of model box)
        x_la = L / 2.0  # in [m]
        y_la = H_mid  # in [m]
        z_la = B / 2.0  # in [m]

        # Sensor location (Position of signal source in POV coordinate system)
        # --> Sensor location and Look at have the same POV right coordinate x and only differ in POV up coordinate y and POV depth coordinate z
        x_lo = x_la  # in [m]
        y_lo = y_la + D / math.tan(incidence / 180 * math.pi)  # in [m]
        z_lo = z_la + D  # in [m]

    # DSM/DTM/nDSM
    else:  # det.ModelOption == 2:
        # Look at (Position where the sensor is looking at in POV coordinate system): Position of model center (center of model box)
        x_la = (L - 1 * model.XCellRes) / 2.0  # in [m]
        y_la = H_mid  # in [m]
        z_la = (B - 1 * model.YCellRes) / 2.0  # in [m]

        # Sensor location (Position of signal source in POV coordinate system)
        # --> Sensor location and Look at have the same POV right coordinate x and only differ in POV up coordinate y and POV depth coordinate z

        if incidence == 0:     # special case: ortho image --> incidence angle = 0° (new for Data Fusion Contest 2019)
            x_lo = x_la  # in [m]
            y_lo = y_la + D  # in [m]
            z_lo = z_la
        else:
            x_lo = x_la  # in [m]
            y_lo = y_la + D / math.tan(incidence / 180 * math.pi)  # in [m]
```

```
                # in [m]; Sensor is positioned in positive POV depth direction z (Sensor looks downwards, top down, in negative POV depth direction z)
                z_lo = z_la + D
                # z_lo = z_la - D # in [m]; Sensor is positioned in negative POV depth direction z (Sensor looks upwards, from the bottom up, in positive POV depth direction z)

        # print "\nLook at (Position where the sensor is looking at in POV coordinate system): (x (right),y (up),z (depth)) = (%.2f,%.2f,%.2f) m"%(x_la,y_la,z_la)
        # print "Sensor location (Position of signal source in POV coordinate system): (x (right), y (up), z (depth)) = (%.2f,%.2f,%.2f) m"%(x_lo,y_lo,z_lo)
        # print "Horizontal distance between sensor location and look at: %.2f m"%D

        # SAR image
        if det.ImgOption == 1:
            # Rotation of model to get the right perspective of the defined sensor location in the POV coordinate system to the model
            # Model should be rotated around POV up direction y (in origin of POV coordinate system) by 'RotModel' degrees
            # If RotModel > 0°: Clockwise rotation
            # If RotModel < 0°: Anticlockwise rotation
            # Angle of view of SAR sensor can be calculated from 90° and heading angle, because SAR sensor is looking perpendicular to the flight direction
            # Rotation if sensor is positioned in positive POV depth direction z (Sensor looks southwards)
            RotModel = 90 - heading + 360
            # RotModel = 360 - heading - 90 # Rotation if sensor is positioned in negative POV depth direction z (Sensor looks northwards)

            # print "Rotation of model around POV up direction y (in origin of POV coordinate system) by %.2f°."%RotModel

            # The azimuth image axis is perpendicular to line of sight (to the left side)
            # --> The azimuth image axis equates to the flight direction, because SAR sensor is looking perpendicular to the flight direction (to the right side)
            # --> Angle from UTM north direction Y to azimuth image axis of SAR image (clockwise) in [°] = Heading angle of SAR sensor (clockwise) in [°] = Angle from UTM north direction Y to flight direction of SAR sensor (clockwise) in [°]
            AngImgAxis = heading

            # print "Angle from UTM north direction Y to azimuth image axis of SAR image (clockwise): %.2f°"%AngImgAxis

        # Optical image
        else:  # det.ImgOption == 2:
            # Rotation of model to get the right perspective of the defined sensor location in the POV coordinate system to the model
            # Model should be rotated around POV up direction y (in origin of POV coordinate system) by 'RotModel' degrees
            # If RotModel > 0°: Clockwise rotation
            # If RotModel < 0°: Anticlockwise rotation
            # Angle of view of optical sensor must be calculated in 'RPC2Ang'
            # Rotation if sensor is positioned in positive POV depth direction z (Sensor looks southwards)
            RotModel = - AngOfView
            # RotModel = 180 - AngOfView # Rotation if sensor is positioned in negative POV depth direction z (Sensor looks northwards)

            # print "Rotation of model around POV up direction y (in origin of POV coordinate system) by %.2f°."%RotModel

            # The easting image axis is perpendicular to line of sight (to the left side)
            # --> The easting image axis do not equate to the flight direction, because optical sensor is looking in arbitrary direction (independent of the flight direction and hence the heading angle)
            # --> Angle from UTM north direction Y to easting image axis of optical image (clockwise) in [°] = Angle from UTM north direction Y to line of sight of optical sensor + 90° (clockwise) in [°]
            AngImgAxis = AngOfView + 90

            # print "Angle from UTM north direction Y to easting image axis of optical
```

```
        image (clockwise): %.2f°"%AngImgAxis

    # Find the size of the projection image = cover area of sensor
    # (Columns * Rows; in POV software: Rendering with Az X Rg)

    # SAR case:
    #   up: Height of image plane of simulated SAR image in [m] --> Along range image axis
    #   right: Width of image plane of simulated SAR image in [m] --> Along azimuth image axis
    # Optical case:
    #   up: Height of image plane of simulated optical image in [m] --> Along northing image axis
    #   right: Width of image plane of simulated optical image in [m] --> Along easting image axis

    if incidence != 0:  # if no ortho image as special case (new for data fusion contest 2019)
        up, right = UpRight(AngImgAxis, incidence, L, B, H)
    else:
        up = L
        right = B

    # SAR image
    if det.ImgOption == 1:
        # Size of the simulated SAR image: Number of rows of simulated SAR image
        rows_simu = round(up / PS_rows)
    # Optical image
    else:  # det.ImgOption == 2:
        # Size of the simulated optical image: Number of rows of simulated optical image
        rows_simu = round(
            up / (PS_rows * math.cos(incidence / 180.0 * math.pi)))

    # Size of the simulated SAR image and optical image: Number of columns of simulated SAR image and optical image
    columns_simu = round(right / PS_columns)

    print "Size of the simulated image:"
    # print "Height of image plane of simulated image: %.2f m"%up
    # print "Width of image plane of simulated image: %.2f m"%right
    # print "Number of rows of the simulated image: %.2f"%rows_simu
    # print "Number of columns of the simulated image: %.2f"%columns_simu

    # Generate header of POV file and add it to POV file

    pt = '#include "colors.inc"\n#include "finish.inc"\n\n// File produced by AccuTrans 3D\n\n#version 3.5;\n'
    pts(pt, headtext)  # Save print output in log file

    # SAR image
    if det.ImgOption == 1:
        # Contribution file should be created
        pt = "global_settings {max_trace_level %s SAR_Output_Data 1 SAR_Intersection 0}\n" % det.max_bounce
        pts(pt, headtext)  # Save print output in log file
    # Optical image
    else:  # det.ImgOption == 2:
        # Contribution file should not be created
        pt = "global_settings {max_trace_level %s SAR_Output_Data 0 SAR_Intersection 0}\n" % det.max_bounce
        pts(pt, headtext)  # Save print output in log file

    # Sensor information
    pt = "#declare Cam = camera {"
    pts(pt, headtext)  # Save print output in log file
    # Type of sensor
    pt = "orthographic"
    pts(pt, headtext)  # Save print output in log file
    # Position where the sensor is looking at
    pt = "look_at <" + str(x_la) + "," + str(y_la) + "," + str(z_la) + ">"
    pts(pt, headtext)  # Save print output in log file
```

```
        # Sensor location (Position of signal source)
        pt = "location <" + str(x_lo) + "," + str(y_lo) + "," + str(z_lo) + ">"
        pts(pt, headtext)  # Save print output in log file
        pt = "right " + str(right) + ' *x'
        pts(pt, headtext)  # Save print output in log file
        pt = "up " + str(up) + ' *y'
        pts(pt, headtext)  # Save print output in log file
        pt = "}\n\ncamera{Cam}\n"
        pts(pt, headtext)  # Save print output in log file

        # Light source
        pt = "light_source {"
        pts(pt, headtext)  # Save print output in log file
        pt = "0*x"
        pts(pt, headtext)  # Save print output in log file
        # Color of light source: white
        pt = "color rgb <1,1,1>"
        pts(pt, headtext)  # Save print output in log file
        # Type of light source
        pt = "parallel"
        pts(pt, headtext)  # Save print output in log file

        # SAR image
        # Light source is at the same position as the sensor (signal source) --> Sun position = sensor position
        if det.ImgOption == 1:
            # Translate light source to position of sensor (signal source)
            pt = "translate <" + str(x_lo) + "," + \
                str(y_lo) + "," + str(z_lo) + ">"
            pts(pt, headtext)  # Save print output in log file
        # Optical image
        else:  # det.ImgOption == 2:
            # Light source is at the same position as the sensor (signal source)
            if det.light == 1:  # (sun position = sensor position)
                print "Light source is at the same position as the sensor (signal source)."
                # Translate light source to position of sensor (signal source)
                pt = "translate <" + str(x_lo) + "," + \
                    str(y_lo) + "," + str(z_lo) + ">"
                pts(pt, headtext)  # Save print output in log file
            # Light source is at that position at which it was during image acquisition
            else:  # det.light == 2:  (sun position = real sun position)
                print "Light source corresponds to sun position."
                # Calculate position of sun during image acquisition
                x_sun, y_sun, z_sun = cal_SunPosition(
                    RotModel, metadatafile, det, x_la, y_la, z_la)
                # Translate light source to that position at which it was during image acquisition
                pt = "translate <" + str(x_sun) + "," + \
                    str(y_sun) + "," + str(z_sun) + ">"
                pts(pt, headtext)  # Save print output in log file

        # Light source point at the position of model center (center of model box)
        pt = "point_at <" + str(x_la) + "," + str(y_la) + "," + str(z_la) + ">"
        pts(pt, headtext)  # Save print output in log file
        pt = "}\n"
        pts(pt, headtext)  # Save print output in log file

        # Add dummy command for geometry part of the model
        if bigmodelname == "box_full_scene.txt":
            pt = "#declare Building = union{\n"
            pts(pt, headtext)  # Save print output in log file

        # DSM/DTM/nDSM
        if det.ModelOption == 2 and det.object_flag == 1:
            pt = "#declare %s = " % bigmodelname
            headtext.append(pt)

        # Define surface reflection parameters of model

        # SAR image
        if det.ImgOption == 1:
```

```
            if det.refl_comb == 1:
                refl_params = 'reflection {0.5} ambient 0 diffuse 0.3 specular 0.5 roughness 0.0033'
            elif det.refl_comb == 2:
                refl_params = 'reflection {0.5} ambient 0 diffuse 0.7 specular 0.5 roughness 0.0033'
            elif det.refl_comb == 3:
                refl_params = 'reflection {0.5} ambient 0 diffuse 1.0 specular 0.5 roughness 0.0033'
            # No option of reflection parameters is choosen
            else:  # det.refl_comb != 1 and det.refl_comb != 2 and det.refl_comb != 3:
                print "\nWarning: The parameter 'refl_comb = %d' in function 'gen_HeaderEndPOV' is not part of [1,2,3]!" % det.refl_comb
        # Optical image
        else:  # det.ImgOption == 2:
            if det.refl_comb == 1:
                refl_params = 'reflection {0} ambient 0 diffuse 0.3 specular 0'
            elif det.refl_comb == 2:
                refl_params = 'reflection {0} ambient 0 diffuse 0.7 specular 0'
            elif det.refl_comb == 3:
                refl_params = 'reflection {0} ambient 0 diffuse 1.0 specular 0'
            # No option of reflection parameters is choosen
            else:  # det.refl_comb != 1 and det.refl_comb != 2 and det.refl_comb != 3:
                print "\nWarning: The parameter 'refl_comb = %d' in function 'gen_HeaderEndPOV' is not part of [1,2,3]!" % det.refl_comb

        print "The surface reflection parameters of the model are '%s'." % refl_params

        # Generate end part of POV file and add it to POV file

        # GIS data (CityGML)
        if det.ModelOption == 1:
            # Add brace for model
            if bigmodelname == "box_full_scene.txt":
                pt = "\n}"
                pts(pt, endtext)  # Save print output in log file

            # Model information
            pt = "\n\nobject {Building"
            pts(pt, endtext)  # Save print output in log file
            pt = "rotate <0," + str(RotModel) + ",0>"
            pts(pt, endtext)  # Save print output in log file
            pt = "translate <" + str(x_la) + "," + \
                str(y_la) + "," + str(z_la) + ">"
            pts(pt, endtext)  # Save print output in log file

        # DSM/DTM/nDSM
        else:  # det.ModelOption == 2:
            # Model information
            # pt = "\n}\n\nobject {%s"%bigmodelname; pts(pt,endtext) # Save print output in log file
            pt = "object {%s" % bigmodelname
            pts(pt, endtext)  # Save print output in log file
            # Scale model coordinates --> Model coordinates in POV file have unit [m] (The unit of the spatial resolution of the model)
            pt = "scale <" + str(model.XCellRes) + "," + \
                str(model.ZCellRes) + "," + str(model.YCellRes) + ">"
            pts(pt, endtext)  # Save print output in log file
            # Translate model to origin of POV coordinate system
            pt = "translate <" + str(-x_la) + "," + \
                str(-y_la) + "," + str(-z_la) + ">"
            pts(pt, endtext)  # Save print output in log file
            # Rotate model in its center around POV up direction y
            pt = "rotate <0," + str(RotModel) + ",0>"
            pts(pt, endtext)  # Save print output in log file
            # Translate model to original location in POV coordinate system
            pt = "translate <" + str(x_la) + "," + \
                str(y_la) + "," + str(z_la) + ">"
            pts(pt, endtext)  # Save print output in log file
            # Define texture of model with color pattern (pigment) and surface reflection features (finish)
            pt = "texture {"
```

```python
        pts(pt, endtext)  # Save print output in log file
        pt = "         pigment {color rgb <1.0,1.0,1.0>}"
        pts(pt, endtext)  # Save print output in log file
        pt = "         finish {%s}" % refl_params
        pts(pt, endtext)  # Save print output in log file
        pt = "         }"
        pts(pt, endtext)  # Save print output in log file
        pt = "}"
        pts(pt, endtext)  # Save print output in log file

    # Output
    return headtext, endtext, columns_simu, rows_simu


def POVfile_addPara(aPovfile, smallmodel, bigmodel, metadatafile, det):
    # Take POV file and add header and end part of the file as required

    """"
    Input:
        aPovfile: Name of POV file
        smallmodel: Small model name (Path to file of small model)
        bigmodel: Big model name (Path to file of big model)
        metadatafile: Orbit meta data file
        det: Definition of input parameters (Look at class 'InputPara' in 'Script.py')
    Output:
        POV file with updated header and ending, after translation
        columns_simu: Number of columns of simulated image
        rows_simu: Number of rows of simulated image
    Used function:
        'GeoRayImg.gen_HeaderEndPOV'
        'GeoRayImg.TransBigSmallModel'
    Used in function:
        'GeoRayImg.Model2GeoI'
    """

    # Split path of model file
    path1, smallmodelname = os.path.split(smallmodel)

    # Generate additional code parts (header and end part) for the POV file
    headtext, endtext, columns_simu, rows_simu = gen_HeaderEndPOV(bigmodel, smallmodel, metadatafile, det)

    # Translation if 'bigmodel' != 'smallmodel'
    # --> If so: The simulated image will have the size of the model extent

    # GIS data (CityGML)
    if det.ModelOption == 1:
        # Add translation between big model and small model to POV file
        xtranslate, ytranslate, ztranslate = TransBigSmallModel(bigmodel, smallmodel, det)

        # Add translation between big model and small model to POV file
        pt = "translate <" + str(xtranslate) + "," + str(ytranslate) + "," + str(ztranslate) + "> // Difference between big model '%s' and small model '%s'\n" % (bigmodel,smallmodel)
        str_endtext = "".join(endtext)
        index_translate = str_endtext.find('translate <')
        str_endtext_new = str_endtext[:index_translate] + pt + str_endtext[index_translate:]

        fin = open(aPovfile, "r")  # Open POV file
        POVstr = fin.read()  # Read POV file
        fin.close()  # Close POV file

        # Update filename for output
        dot_index = smallmodelname.index('.')
        modelname_pure = smallmodelname[0:dot_index]
        aPovfile2 = path1 + '/' + modelname_pure + ".pov"

        fout = open(aPovfile2, "w")  # Open POV file
        fout.write("".join(headtext) + POVstr + "\n\n" + str_endtext_new)  # Write in POV file
```

```
            fout.close()  # Close POV file

        # DSM/DTM/nDSM
        else: # det.ModelOption == 2:
            # Add translation between big model and small model to POV file
            xtranslate, ytranslate, ztranslate = TransBigSmallModel(bigmodel, smallmodel, det)

            # Add translation between big model and small model to POV file
            pt = "translate <" + str(xtranslate) + "," + str(ytranslate) + "," + str(ztranslate) + "> // Difference between big model '%s' and small model '%s'\n" % (bigmodel,smallmodel)
            str_endtext = "".join(endtext)
            index_translate = str_endtext.find('translate <')
            str_endtext_new = str_endtext[:index_translate] + pt + str_endtext[index_translate:]

            fin = open(aPovfile, "r")  # Open POV file
            POVstr = fin.read()  # Read POV file

            fin.close()  # Close POV file

            if det.object_flag == 1 and det.plane_flag == 0:
                a1 = POVstr.find('mesh2')
                # a2 = POVstr.find('//endmodel')

            if det.object_flag == 0 and det.plane_flag == 1:
                a1 = POVstr.find('plane')
                # a2 = POVstr.find('//endplane')

            if det.object_flag == 1 and det.plane_flag == 1:
                a1 = POVstr.find('mesh2')
                # a2 = POVstr.find('//endplane')

            if det.depth == 4:
                a1 = POVstr.find('sphere')
                # a2 = POVstr.find('//endmodel')

            # Substitute translate and rotate routine for full scene model
            # (required only for visibility check in processing depth level 2)
            if len(det.path_to_adapted_pov_file) != 0 and det.object_flag == 1:

                # Find key positions for inserting new code in the POV-Ray file
                index_translate_full_model = POVstr.find('translate')
                index_texture_full_model = POVstr.find('texture')

                # Extract strings of full scene model
                POVstr_part_1 = POVstr[0:index_translate_full_model]
                POVstr_part_2 = POVstr[index_texture_full_model:]

                bigmodel_temp = path1 + "/preDSM"  # name of full scene model (needed for visibility check)

                # Calculate necessary shifts along POV-Ray axes
                xtranslate_temp, ytranslate_temp, ztranslate_temp = TransBigSmallModel(bigmodel_temp, smallmodel, det)

                # Generate string for unifying the small and big model in the same POV-Ray coordinate system

                # Difference between big model and small model (context: visibility check)
                shift_small_vs_big = "translate <" + str(xtranslate_temp*(-1)) + "," + str(ytranslate_temp*(-1)) + "," + str(ztranslate_temp*(-1)) + ">"

                # Difference between big model and small model (e.g. shift between small and extended model)
                shift_extended_model = "translate <" + str(xtranslate) + "," + str(ytranslate) + "," + str(ztranslate) + ">"

                # Build string for geometric object manipulation
                POVstr = POVstr_part_1 + shift_small_vs_big + "\n" + shift_extended_model + "\n" + endtext[4] + endtext[5] + endtext[6] + endtext[7] + endtext[8] +
```

```
            endtext[7] + POVstr_part_2

        fout = open(aPovfile, "w")  # Open POV filev

        if det.object_flag == 1:
            fout.write("".join(headtext) + POVstr[a1:] + str_endtext_new)  # Write in POV file
        else:
            fout.write("".join(headtext) + POVstr)  # Write header and plane information to file

        fout.close()  # Close POV file

    # Output
    return columns_simu, rows_simu


def POVRay_POV2Img(aPovfile, columns_simu, rows_simu, sensor, det):
    # SAR case: Run POV-Ray to render optical image based on POV file and to generate signal contribution file based on POV file
    # Optical case: Run POV-Ray to render optical image based on POV file
    """
    Input:
            aPovfile: Name of POV file
            columns_simu: Number of columns of simulated image
            rows_simu: Number of rows of simulated image
            sensor: Name of sensor
            det: Definition of input parameters (Look at class 'InputPara' in 'Script.py')
    Output:
            SAR case: Rendered optical image and contribution file which contains signal contributions with coordinates and signal strengths and is necessary for Rendering of SAR image in RaySAR
            Optical case: Rendered optical image
    Used function:
            None
    Used in function:
            'GeoRayImg.Model2GeoI'
    """

    # Simulate the optical image

    # Split path of POV file
    path1, povfile = os.path.split(aPovfile)

    # Using POV-Ray for doing Ray Tracing
    os.system(r'cd %s && povray %s +W%d +H%d Output_File_Name=%s -D' %
              (path1, aPovfile, columns_simu, rows_simu, path1 + '/'))
    # os.system(r'povray %s +W%d +H%d Output_File_Name=%s +D'%(aPovfile,columns_simu,rows_simu,path1 + '/')) # POV-Ray should render optical image and display it

    # SAR image
    if det.ImgOption == 1:
        # Change contribution filename
        contribution_filename = aPovfile[:-4] + '_Contributions.txt'
        # Change name of contribution
        check_call('cd %s && mv Contributions.txt %s' %
                   (path1, contribution_filename), shell=True)

        # print "\nRendering of optical image and simulation of contribution file by Ray Tracing using POV-Ray:"
        # print "The rendered optical image is '%s'"%(aPovfile[:-4] + '.png')
        # print "The contribution file is '%s'"%contribution_filename

        # Output
        return contribution_filename

    # Optical image
    else:  # det.ImgOption == 2:
        # print "\nSimulation of %s image by Ray Tracing using POV-Ray:"%sensor
        # print "Rendered optical image (%s): %s"%(sensor,aPovfile[:-4] + '.png')
```

```python

            # Output
            return 1


    def gen_RaySAR_Para(bigmodel, smallmodel, metadatafile, RaySARparafile, det):
        # Generate SAR image parameters for the simulation in RaySAR (MATLAB part)
        """
        Input:
                bigmodel: Big model name (Path to file of big model)
                smallmodel: Small model name (Path to file of small model)
                metadatafile: Orbit meta data file of SAR image
                RaySARparafile: Generated output file for SAR image parameters for RaySAR
                det: Definition of input parameters (Look at class 'InputPara' in 'Script.py')
        Output:
                Parameter file with SAR image parameters for RaySAR
        Used function:
                'GeoRayImg.read_ModelPara'
                'GeoRayImg.read_Metadata'
                'GeoRayImg.UpRight'
                'GeoRayImg.pts'
        Used in function:
                'GeoRayImg.Model2GeoI'
        """

        global sensor, filename, heading, AngOfView, incidence, meanHeight, PS_columns, PS_rows

        # Dummy distance in horizontal plane for calculating sensor position
        D = 500  # unit [m]

        # Split path of model file
        path1, bigmodelname = os.path.split(bigmodel)

        # Keep a list of lines that we want to save in POV file
        lts = []  # Initialisation: Line to save in POV file

        # Read parameters of model
        model, H_max, H_min, H_mid, L, B, H = read_ModelPara(bigmodel, det)

        # Get information from orbit meta data (Read parameters from meta data file of image)
        #sensor,filename,heading,AngOfView,incidence,meanHeight,PS_columns,PS_rows = read_Metadata(metadatafile,bigmodel,smallmodel,det)

        # Find the size of the projection image = cover area of sensor
        # up: Perspective height of simulated optical image
        # right: Azimuth of simulated optical image = Azimuth of simulated SAR image
        up, right = UpRight(heading + 180, 90 - incidence, L, B, H)

        # Size of simulated images (Columns * Rows; in POV software: Rendering with Az X Rg)
        # Number of columns of simulated optical image and of simulated SAR image
        columns_simu = round(right / PS_columns)
        # Number of rows of simulated optical image
        rows_simu = round(up / (PS_rows * math.sin(incidence / 180 * math.pi)))

        # Calculate the min and max values in range direction for the following SAR simulation in MATLAB (RaySAR)
        # Slant distance between model center (center of model box) and SAR sensor
        Sslant = D / math.sin(incidence / 180 * math.pi)

        # Generate SAR image Parameters required for RaySAR simulation in MATLAB

        pt = "PixelSpacing_columns = " + str(PS_columns)
        pts(pt, lts)  # Save print output in log file
        pt = "PixelSpacing_rows = " + str(PS_rows)
        pts(pt, lts)  # Save print output in log file
        pt = "Azimuth_Min = " + str(-columns_simu * PS_columns / 2.0)
        pts(pt, lts)  # Save print output in log file
        pt = "Azimuth_Max = " + str(columns_simu * PS_columns / 2.0)
```

```python
    pts(pt, lts)  # Save print output in log file
    pt = "Range_Min = %.2f" % (
        (Sslant - up / 2.0) / math.sin(incidence / 180 * math.pi))
    pts(pt, lts)  # Save print output in log file
    pt = "Range_Max = %.2f" % (
        (Sslant - up / 2.0) / math.sin(incidence / 180 * math.pi) + rows_simu * PS_rows)
    pts(pt, lts)  # Save print output in log file
    pt = "Bounce_level = " + str(det.max_bounce)
    pts(pt, lts)  # Save print output in log file
    pt = "dB_range_Min = -25"
    pts(pt, lts)  # Save print output in log file
    pt = "dB_range_Max = Max"
    pts(pt, lts)  # Save print output in log file
    pt = "RangeCoord_topdown = 1"
    pts(pt, lts)  # Save print output in log file
    pt = "Binomial_filt = 1"
    pts(pt, lts)  # Save print output in log file
    pt = "GroundRangeGeometry = 1"
    pts(pt, lts)  # Save print output in log file
    pt = "Incidence = " + str(incidence)
    pts(pt, lts)  # Save print output in log file

    # Additional parameters to identify the parameter file
    pt = "modelfile = %s" % (os.path.join(path1, bigmodelname))
    pts(pt, lts)  # Save print output in log file
    pt = "%s_image = %s" % (sensor, filename)
    pts(pt, lts)  # Save print output in log file
    pt = "heading_Az = " + str(heading)
    pts(pt, lts)  # Save print output in log file

    # print "\nInput for calculation of %s simulation parameters for RaySAR:"%sensor
    # print "%s image: '%s'"%(sensor,filename)
    # print "Model: '%s'"%(path1 + '/' + bigmodelname)
    print "\nOutput of %s simulation parameters for RaySAR is stored in '%s'\n" % (sensor, RaySARparafile)

    fileobj = open(RaySARparafile, "w")  # Open parameter file
    fileobj.write("".join(lts))  # Write in parameter file
    fileobj.close()  # Close parameter file

    # Output
    return 1


def RaySARSimu(contribution_filename, Output_path, RaySARparafile):
    # Run RaySAR of linux version and do SAR simulation
    """
    Comment:
            Contribution file not allowed to be bigger than 50 MB
    Input:
            contribution_filename: Contribution file which contains signal contributions with coordinates and signal strengths (Output of 'POVRay_POV2Img')
            Output_path: Path where the output folder should be saved
            RaySARparafile: Parameter file with SAR image parameters for RaySAR (Output of 'gen_RaySAR_Para')
    Output:
            Directory 'Maps' in 'Output_path'
            To see: 'Maps/Frames/Ref_Maps/... .tif'
    Used function:
            None
    Used in function:
            'GeoRayImg.Model2GeoI'
    """

    # Start RaySAR with MATLAB
    #os.system(r' ../RaySARLinux/matlab_batcher_raysar.sh raysar_linux_function "' + "'%s'"%contribution_filename + '" "' + "'%s'"%Output_path + '" "' + "'%s'"%RaySARparafile + '"')

    import Gen_Refl_Map
```

```
        # Initiate SAR simulation
        Gen_Refl_Map.raysar_linux_function(
            contribution_filename, Output_path, RaySARparafile)

        print "\nSAR simulation done."
        print "\nThe SAR simulation results are stored in '%s/Maps/'" % Output_path

        # Change name of simulated SAR images in folder 'Maps'
        listing = os.listdir(os.path.join(
            Output_path, 'Maps', 'Frames', 'Ref_Maps'))
        for infile in listing:
            texta, textb = os.path.splitext(infile)
            bounce = texta
            if bounce == 'All Reflections_Fr':
                simpsimun = 's0'
                bounce = 'All\ Reflections_Fr'
            elif bounce == 'Single Bounce_Fr':
                simpsimun = 's1'
                bounce = 'Single\ Bounce_Fr'
            elif bounce == 'Double Bounce_Fr':
                simpsimun = 's2'
                bounce = 'Double\ Bounce_Fr'
            elif bounce == 'Triple Bounce_Fr':
                simpsimun = 's3'
                bounce = 'Triple\ Bounce_Fr'
            else:  # There is no simulated SAR image available
                print "No suitable simulated SAR image available for geocoding."
                simpsimun = 'NOFILE'
            if simpsimun != 'NOFILE':  # Change name
                # Change name of folder with simulated SAR images in folder 'Maps'
                check_call('mv %s/Maps/Frames/Ref_Maps/%s.tif' % (Output_path, bounce) +
                           ' %s/Maps/Frames/Ref_Maps/%s.tif' % (Output_path, simpsimun),
                           shell=True)
                print "Image '%s%s' represents '%s'" % (simpsimun, textb, texta[:-3])
```