LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-487872

Benchmark Imagery FY11 Technical Report

R. Roberts, P. Pope

June 15, 2011

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Benchmark Imagery FY11 Technical Report

Randy Roberts[1] and Paul Pope[2]

[1]Lawrence Livermore National Laboratory
[2]Los Alamos National Laboratory

10 June 2011, v1

Preface

This report details the work performed in FY11 under project LL11-GS-PD06, "Benchmark Imagery for Assessing Geospatial Semantic Extraction Algorithms." The original LCP for the Benchmark Imagery project called for creating a set of benchmark imagery for verifying and validating algorithms that extract semantic content from imagery. More specifically, the first year was slated to deliver real imagery that had been annotated, the second year to deliver real imagery that had composited features, and the final year was to deliver synthetic imagery modeled after the real imagery.

Through discussions with the program manager, the FY11 project plan has been redirected into two tasks:

1. Create visualizations of synthetic urban scenes by parsing the output produced by ORNL's "SAM Ontology Demonstration for SNM" project (OR10 OntologyDemo-PD06)
2. Produce a set of annotated real-world imagery and demonstrate its use in algorithm V&V

The accompanying technical deliverables for Tasks 1 and 2 respectively are:

1. Q3 2011: Demonstrate software that produces: (1) a "mesh" which defines the 3D structure of all objects in a scene, and (2) visualization of same
2. Q4 2011: A set of annotated real-world imagery

The emphasis of FY11 is placed on the first task, with the second task executed to the greatest possible extent given funding resources. A draft addendum documenting these changes to the Benchmark Imagery LCP was sent to the program manager on 17 December 2010. This report describes the work performed under Task 1 of the revised project plan. The second task is described in another document.

Geometry Generation for Rendering Narratives and Modeling Radiation Transport

Randy Roberts[1] and Paul Pope[2]

[1]Lawrence Livermore National Laboratory
[2]Los Alamos National Laboratory

The objectives for this task in the Benchmark Imagery project were twofold: Create a geometric model to visualize scenarios produced by the Ontology Driven Scenario Generator (ODSG) and to enable input of the model into radiation transport codes. This task was part of a collaborative tri-lab project between Oak Ridge (ORNL), Los Alamos (LANL) and Lawrence Livermore (LLNL) National Laboratories. ORNL developed proliferation scenarios using the ODSG, and passed an Extensible Markup Language (XML) file to the LLNL/LANL team. That file described the objects of the scenario, such as buildings, rivers, roads, pedestrians, vehicles, etc., and their attributes. The XML file describing the scenario was parsed into two types of objects: simple geometric objects that need to be created, and complex objects for which we had pre-built models. (See Figure 1 for a block diagram of the processing.) Models for simple geometric objects, such as buildings, roads and rivers, were created on-the-fly, while models for complex objects, such as vehicles and pedestrians, were stored in a library. A script that described the objects to load and their position in the scene was created, and written to a file. That file was read in by scripts written at the Rochester Institute of Technology (RIT), which load and position the objects in Blender to create a scene model. Details of geometry generation are provided below.

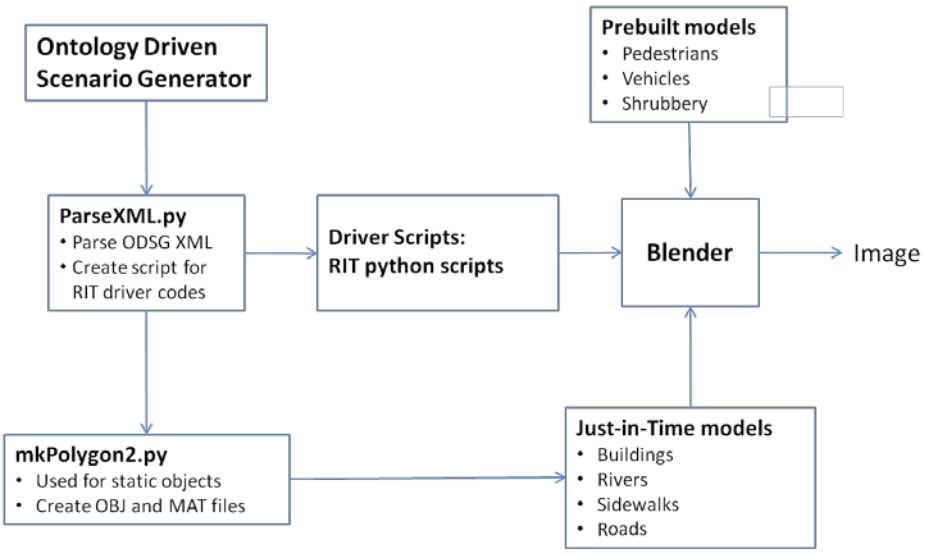


Figure 1: Schematic overview of processing for geometry generation for scenario visualization

1 Geometry generation for scenario visualization

Scenarios were provided by the ODSG as Extensible Markup Language (XML) files. XML is a foundational technology of the semantic web, and is intended to encode documents in machine interpretable form [1]. The project selected XML as the scenario description primarily for its flexibility. If additional detail is needed to describe a scenario, descriptive tags are defined and supporting data added to the file. An example of XML used to describe a scenario is shown in Figure 2. The opening line of the file <?xml version="1.0"?> indicates that this is an XML file. The scenario is surrounded by opening and closing tags: <scenario> and </scenario>, as is the case with all XML constructions. The "-" sign in front of a tag indicates that that block of XML has been expanded, whereas the "+" sign indicates that XML code between those tags is collapsed. The XML file shown in Figure 2 begins with a tag that describes the scenario, and a tag that defines the origin of the scenario in latitude and longitude. The particular coordinates in the file are for Nashville TN. These coordinates are used as offsets when the scene model is assembled in Blender. This particular scenario contains three buildings, three streets (denoted by the <pavement> tags), two sections of a river, three sidewalks and a searcher. The details of each of these constructions are described next.

```
<?xml version="1.0"?>
- <scenario>
    <description>test_SHP_to_NA22XML_v1.xml</description>
   -<origin>
      <LAT>36.15677</LAT>
      <LON>-86.776808</LON>
    </origin>
  +<building></building>
  +<building></building>
  +<building></building>
  +<pavement></pavement>
  +<pavement></pavement>
  +<pavement></pavement>
  +<river></river>
  +<river></river>
  +<sidewalk></sidewalk >
  +<sidewalk></sidewalk >
  +<sidewalk></sidewalk >
  +<searcher></searcher >
</scenario>
```

Figure 2: A small XML file describing a scene containing three buildings, three sections of street, two sections of river, four sections of sidewalk and a searcher. The latitude and longitude in this scenario are set for Nashville TN.

XML scenario files are processed by scripts written in Python, an object-oriented programming language that has found a large following in the scientific programming community [5]. Python is also the scripting language for Blender, and hence a preferred language for the processing. Python developers have

created an extensive library of modules that are used to extend the capabilities of the language. In particular, we use the ***ElementTree*** module to parse the XML file into a data tree that can be accessed as Python variables [6]. The Python file **ParseXML** is structured to process each type of object with sequential blocks of code. For example, all river segments are processed, followed by sidewalks, streets, building, pedestrians, etc. Adding new objects to the parser is straightforward; simply add a new block of code to process the new type of object.

As each object is processed by **ParseXML**, a statement describing the location of the geometry file for the object, its position and orientation in the scene, is written. The collection of all of these statements into a file, called the ABSOLUTE_SCRIPT, is used by the RIT scripts to create the scene model in Blender. Some objects, such as buildings, rivers, etc. have relatively simple geometry models. For these objects, another Python script, called **mkPolygonv2**, is used to create the 3D model on-the-fly. Other objects, such as vehicles and pedestrians, can be quite complex, and for these we use pre-built models. In the sections below, we describe the objects in the simulation and the XML codes that are used to describe them.

1.1 Geometric Objects

1.1.1 Buildings

Buildings are described by a height and footprint (cf. Figure 3). The height is given in meters, and the footprint by latitude and longitude. The other tags surrounding the <coordinate> tags, (<Polygon>, <outerboundary> and <linearRing>) are constructions used in the Keyhole Markup Language (KML) [7]. The XML code enclosed by the <Polygon> tags is easily converted into a KML script that can be executed by GoogleEarth, thereby allowing visualization and validation of object footprints against overhead imagery in GoogleEarth. The tag <nom> denotes the name of the building, where we used randomly generated numbers. The color was also selected randomly to provide interest.

```
<building>
  <nom>93715</nom>
  <color>c3</color>
  <height>18.240000</height>
    <Polygon>
      <outerBoundaryIs>
        <LinearRing>
          <coordinates>
            -86.775379285390088, 36.161193427488286
            -86.775348029975191, 36.161145492873274
            -86.775390991559547, 36.161127072483779
            -86.775422246993415, 36.161175007087770
            -86.775379285390088, 36.161193427488286
          </coordinates>
        </LinearRing>
      </outerBoundaryIs>
    </Polygon>
</building>
```

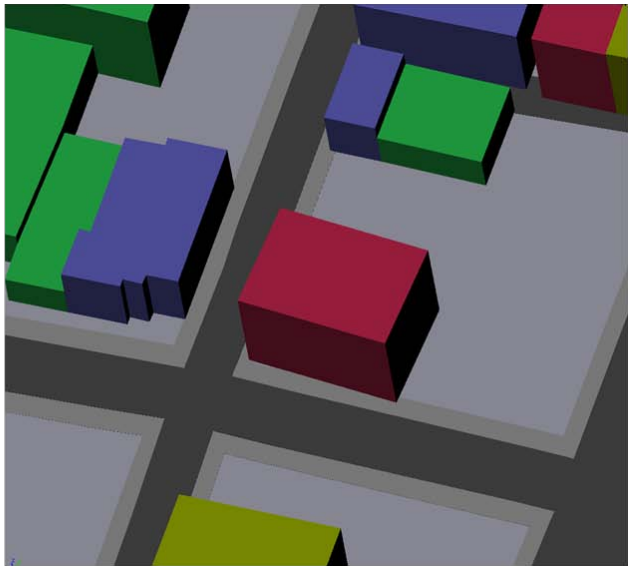


Figure 3: XML description of a building, and its realization in Blender

Footprints of buildings are parsed from the XML file, and the latitude/longitude coordinates are converted to UTM for ease of processing. The conversion is performed using the ***pyproj*** Python module. The coordinates of scene objects are centered using data defined by the <origin> tag. Together with height data, the coordinates are used by the **mkPolygonv2** script to create 3D right prisms for display in Blender. The format of the right prisms is OBJ, an ascii file that consists of a list of vertices, and sets of vertices that form object faces [2]. A material file, MAT, is also produced, and this file contains information on the color and reflective properties of the object. As previously mentioned, the colors in the visualization model are randomly defined.

1.1.2 Roads, sidewalks and rivers

Roads, sidewalks and rivers are described in a manner similar to buildings. An example of the XML description of a section of a river is shown in Figure 4, along with an illustration of the river, streets and sidewalks.

```
<river>
   <nom>58540</nom>
    <Polygon>
      <outerBoundaryIs>
        <LinearRing>
          <coordinates>
            -86.776358000000002,  36.168611999999996
            -86.776488999999998,  36.168554999999998
            -86.776881000000003,  36.168386999999996
            -86.777012999999997,  36.168330999999995
            -86.777158999999997,  36.168534000000001
            -86.777355000000000,  36.168742999999999
            -86.777446999999995,  36.168897999999999
            -86.777767999999995,  36.169435000000000
            -86.778126999999998,  36.169841999999996
            -86.778657999999993,  36.170671999999996
            -86.779044999999996,  36.171276999999996
            -86.778885000000002,  36.171340000000001
            -86.778737999999990,  36.171399999999998
          </coordinates>
        </LinearRing>
      </outerBoundaryIs>
    </Polygon>
</river>
```

Figure 4: XML description of a section of river, and an illustration of the model including streets, sidewalks and river

1.1.3 Vehicles and pedestrians

Dynamic objects such as vehicles and pedestrians are represented in a static visualization by a coordinate and heading. Figure 5 shows the XML codes for several types of pedestrians (searcher, medical patient and regular pedestrian) and vehicles.

```
<stwagon>
 <heading>212.39228</heading>
 <coordinate>-86.775799, 36.164543</coordinate>
</stwagon>
<stwagon>
 <heading>212.39228</heading>
 <coordinate>-86.775842, 36.164619</coordinate>
</stwagon>
<medical_patient>
 <heading>392.60939</heading>
 <coordinate>-86.776126, 36.164860</coordinate>
</medical_patient >
<pedestrian>
 <heading>392.60939</heading>
 <coordinate>-86.776122, 36.164852</coordinate>
</pedestrian>
<searcher>
 <detector>detector_type</detector>
 <heading>392.60939</heading>
 <coordinate>-86.776122, 36.164852</coordinate>
</searcher>
```

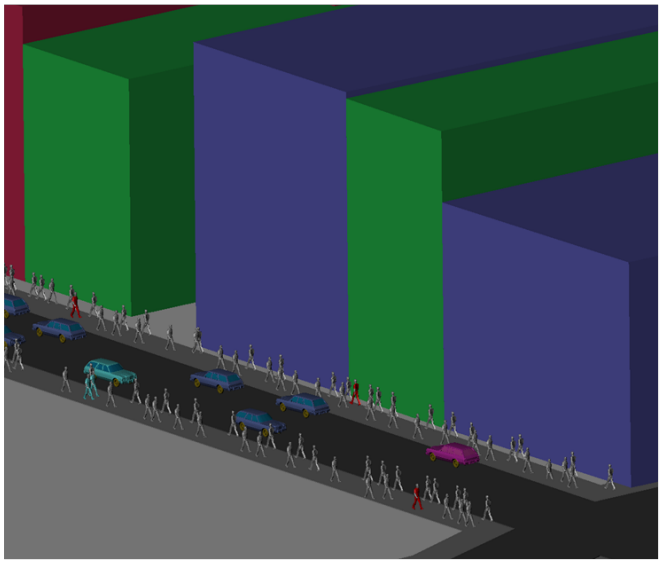


Figure 5: vehicles and pedestrians

Note that the different types of pedestrians are denoted by different colors: regular pedestrians are gray, searchers are blue and medical patients are red. Likewise, searcher vehicles are colored light blue while a threat vehicle is colored red.

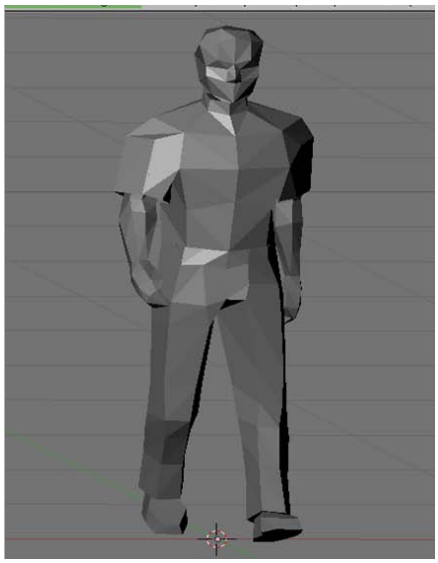


Figure 6: vehicle model and pedestrian, courtesy of RIT.

1.1.4 Paths

Animation was not part of the tasking, but it is desirable to include a notion of paths for searchers and vehicles engaged in a scenario. Paths were created using a timestamp and coordinates as shown in Figure 7.

```
<searcher>
   <detector>detector_type</detector>
   <path>
     <point>
       <timestamp>1298495343</timestamp>
       <coordinates>-86.777007,36.160969</coordinates>
     </point>
     <point>
       <timestamp>1298495345</timestamp>
       <coordinates>-86.776859, 36.161021</coordinates>
     </point>
     <point>
       <timestamp>1298495347</timestamp>
       <coordinates>-86.776747, 36.161071</coordinates>
     </point>
     <point>
       <timestamp>1298495347</timestamp>
       <coordinates>-86.776658, 36.161112</coordinates>
     </point>
</path>
  </searcher>
```

Figure 7: Path description

1.2 Scene assembly using Blender

The models of buildings, streets, sidewalks, rivers, pedestrians and vehicles were assembled into a scene using Blender [3]. As previously mentioned, Blender is an open source 3D content creation suite that is used for a wide range of modeling, image synthesis and animation. An illustration of the final downtown Nashville model is shown in Figures 8 and 9, where for comparison an overhead image is provided. The model was assembled in Blender using the ABSOLUTE_SCRIPT produced by **ParseXML** and Python scripts developed by the Digital Image and Remote Sensing Laboratory at RIT. These scripts, executed within Blender, load objects, and place and rotate the objects into the scene.

Model creation is fast. Parsing a 100,000+ line XML file, and creation of approximately 500 .obj files for buildings, etc. representing downtown Nashville, took approximately 5 minutes on an inexpensive desktop system. Model assembly took approximately 10 minutes using Blender on the same system.

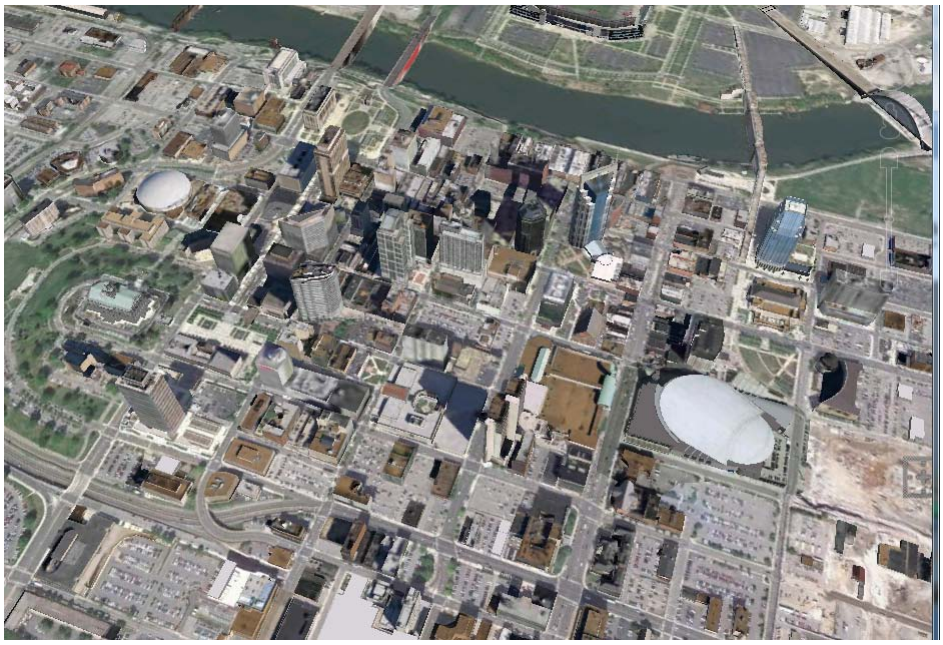


Figure 8: Image of downtown Nashville

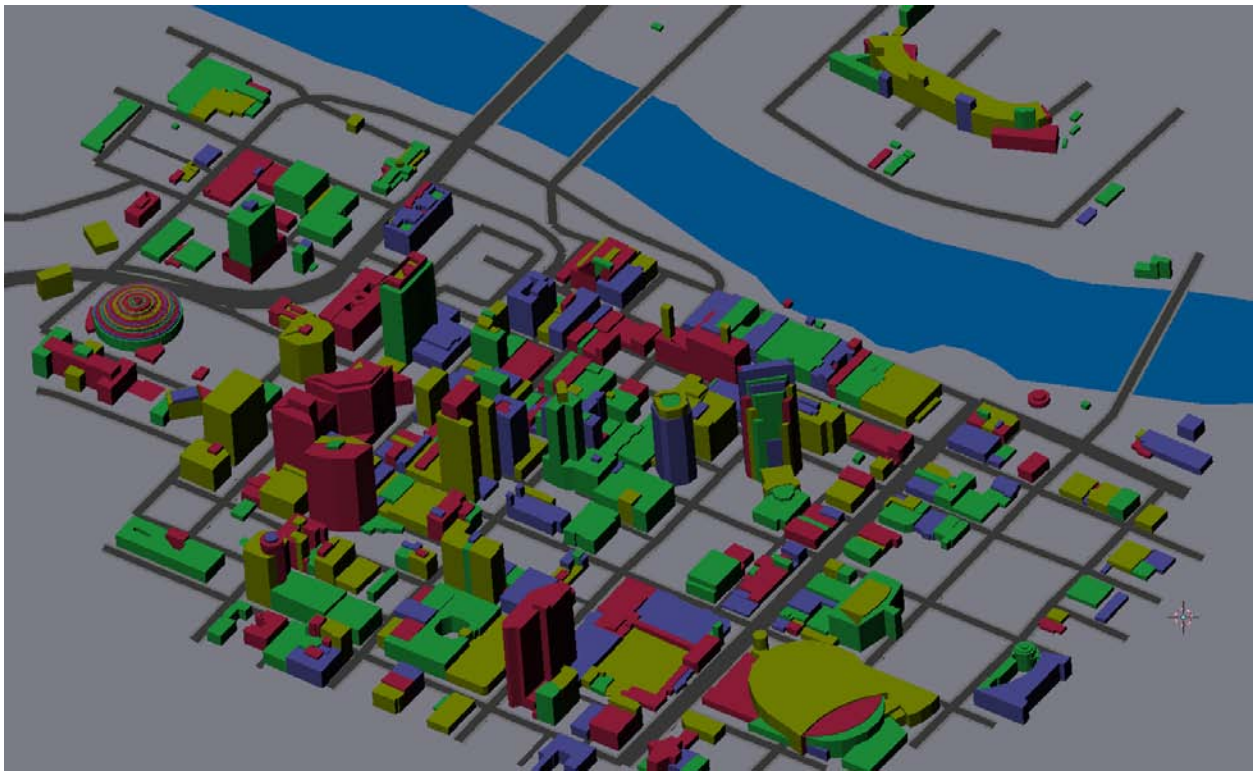
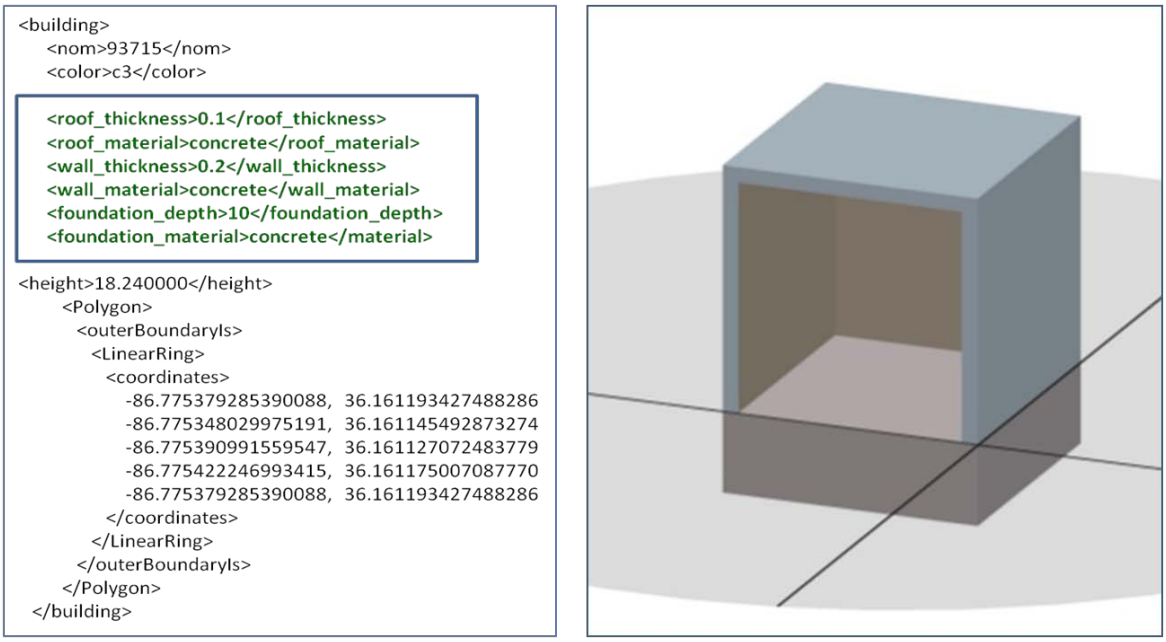


Figure 9: Screen shot from Blender of downtown Nashville model

2 Geometry Generation for Radiation Transport Modeling

The geometry generation process for visualization is not adequate for radiation modeling, but simple enhancements can be used to make it so. The main problem with the previous geometry generation is that buildings are modeled as solid objects, which is inadequate for radiation transport modeling. Rather, buildings need to have interior structures for appropriate modeling. Note the concept of a box for the radiation transport codes, and how we can make boxes from intersecting planes if a box is not inherent in the code.

A first order building representation is shown in Figure 10 along with the XML code to support the enhancements. The building is represented by a simple shell consisting of a foundation, roof and four walls. The XML needed to describe this enhancement includes tags for roof thickness and material, wall thickness and material, and foundation depth and material.



```
<building>
   <nom>93715</nom>
   <color>c3</color>
   <roof_thickness>0.1</roof_thickness>
   <roof_material>concrete</roof_material>
   <wall_thickness>0.2</wall_thickness>
   <wall_material>concrete</wall_material>
   <foundation_depth>10</foundation_depth>
   <foundation_material>concrete</material>
<height>18.240000</height>
     <Polygon>
       <outerBoundaryIs>
         <LinearRing>
           <coordinates>
              -86.775379285390088,  36.161193427488286
              -86.775348029975191,  36.161145492873274
              -86.775390991559547,  36.161127072483779
              -86.775422246993415,  36.161175007087770
              -86.775379285390088,  36.161193427488286
           </coordinates>
         </LinearRing>
       </outerBoundaryIs>
     </Polygon>
 </building>
```

Figure 10: Simple model for a building

Radiation transport codes, such as **Mercury** [4], typically have geometric constructions such as *boxes*. While Mercury has direct geometry called a *box*, other codes can build boxes by intersecting planes. There are seven boxes in the simple building model: one foundation, four walls, one roof and an air void that represents the interior of the building. Hence, it is relatively straight-forward to create geometries that are suitable for radiation transport codes: create boxes as illustrated in Figure 10, and ascribe materials to these boxes such as concrete and air.

A second order building representation would include floors as shown in Figure 11a. New XML tags that give the number of floors, the floor thickness and materials would need to be added to the description. (Note the simplification that the floors are equally spaced, equally thick, and are built from the same materials.) Additional complexity is also possible, as seen in Figure 11b, requiring additional XML tags to describe the model.

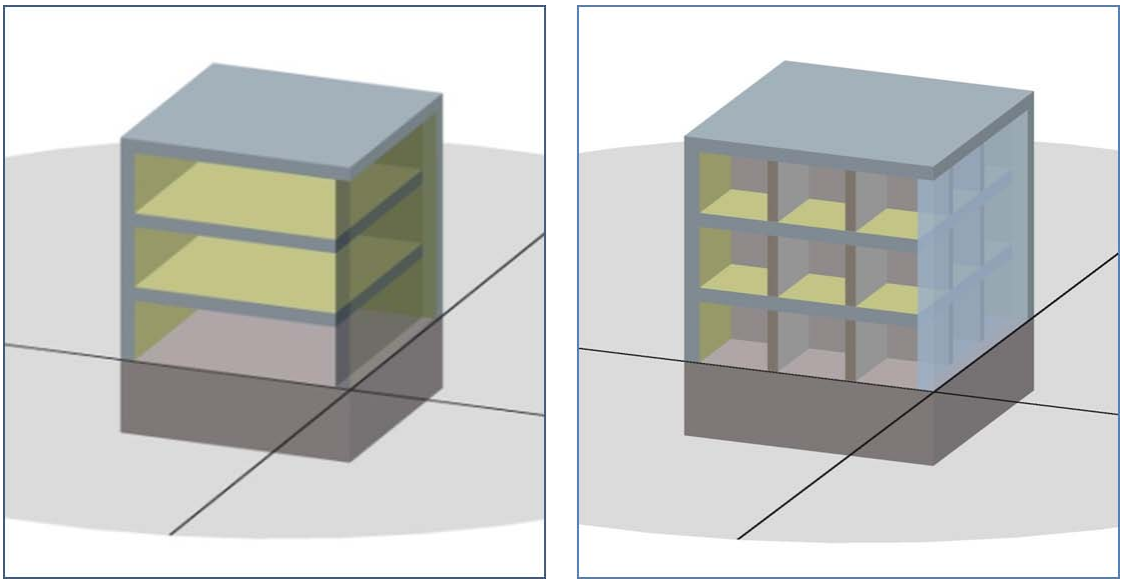


Figure 11: More complex building models. Left hand side (Figure 11a) is a simple extension of the model shown in Figure 10. The right hand side (Figure 11b) is a simple extension of the model shown in Figure 11a.

As shown above, the complexity of building interiors can be increased in several ways. Before developing more complex building interiors, it is necessary to determine the purpose of the simulation, and consult with the end-users and radiation transport modelers to determine the cost-benefits of various modeling simplifications.

3 Summary

An approach to generating geometries for scenario visualization and radiation transport modeling has been described. The approach is based on an XML description of the geometries in the scenario, such as buildings, roads, rivers, pedestrians and vehicles. The XML scenario is parsed using specialized Python scripts, and objects in the scenario are created. The scene is assembled in Blender, an open-source 3D content creation suite using Python scripts developed by the Rochester Institute of Technology. From here, the scenario can be visualized in a number of ways, using the capabilities of Blender. Although the models build for visualization in blender are not suitable for radiation transport modeling, they are easily modified to a form that is.

4 Acknowledgements

The authors would like to thank and acknowledge the support of **Dr. Alexander Slepoy**, Program Manager, Simulations, Algorithms, and Modeling; Office of Nonproliferation Research & Development, National Nuclear Security Administration. They would also like to thank Paul Peaslee, LLNL, for creating the building models in Figures 10 and 11. Finally, thanks to Rolando Raqueno, Andrew Scott and Niek

Sanders, Chester F. Carlson Center for Imaging Science, Rochester Institute of Technology, for providing assistance with Blender, providing scripts to drive the assemble of scenes using Blender, and for models of vehicles, pedestrians and vegetation.

5 References

[1] "XML," http://en.wikipedia.org/wiki/XML 1 June 2011 [1 June2011].

[2] "Wavefront .obj file," http://en.wikipedia.org/wiki/Wavefront_.obj_file 13 April 2011, [1 June 2011].

[3] "Blender," http://www.blender.org/ [1 June 2011].

[4] Richard Procassini, et al., **"**Update on the Development and Validation of MERCURY: A Modern, Monte Carlo Particle Transport Code," *Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications,* Palais des Papes, Avignon, France, September 12-15, 2005, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2005).

[5] "Python Programming Language – Official Website," http://www.python.org/ [1 June 2011].

[6] "ElementTree Overview," http://effbot.org/zone/element-index.htm [1 June 2011].

[7] "Keyhole Markup Language," http://en.wikipedia.org/wiki/Keyhole_Markup_Language [1 June 2011].