Efficient 3D Building Model Generation from 2D Floor Plans

by

Dmitry Kashlev

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2008

© Massachusetts Institute of Technology 2008. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
August 22, 2008

Certified by ....
Seth Teller
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Efficient 3D Building Model Generation from 2D Floor Plans

by

Dmitry Kashlev

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

3D building models are beneficial to architects, interior designers, and ordinary people in visualizing indoor space in three dimensions. 3D building models appear to be more aesthetic to ordinary people than architectural drawings. Architects can benefit from such models in detecting any inconsistencies in their designs. This thesis describes the design and implementation of an efficient 3D building model generator (3dGen) that can automatically create 3D building models from AutoCAD drawings. This thesis explains how 3dGen takes floor plan data in XML format (generated from AutoCAD drawings), extrudes the walls and vertical surfaces and adds additional 3D information to the existing floor plan. In doing so 3dGen aims to satisfy the complete watertight space and the manifold properties and attempts to minimize the amount of 3D data by eliminating redundant geometric primitives. This thesis explains the algorithms that were employed in order to generate correct surfaces with many types of portals in them and algorithms that detect inconsistencies in the 2D architectural drawings.

Thesis Supervisor: Seth Teller
Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to express gratitude to my advisor, Professor Seth Teller, for giving me the opportunity to work on this exciting and challenging project. Even more, I would like to thank Yoni Battat for guidance and help throughout the project and his willingness to lend me a hand and his expertise in rough times. I would also like to thank Emily Whiting and Yoni Battat for setting up BMG framework and providing me with XML data input that I used throughout the project. I would also like to thank Grayson Giovine, Yoni Battat, Seth Teller, and Emily Whiting for providing me with a great research environment during the early stages of the BMG project. I also would like to thank David Lambeth for his hand in understanding some of algorithms. Thanks also go to Ben Charrow for telling me about lab soccer matches. Lastly, I would like to thank my family and friends for their never-ending support and love throughout my educational career. I would most of all like to thank my mom for her devotion in raising me to become who I am today.

Contents

1 Introduction **17**

1.1 Motivation for 3D model generation from floor plans 18

1.2 Terminology Overview 20

1.2.1 Spaces and Portals: overview 20

1.2.2 Vertices and Edges: implementation 22

1.2.3 Surfaces/Faces and Walls 23

1.3 Manifold property 24

1.4 Watertight property 25

2 System Architecture **29**

2.1 System Overview 29

2.1.1 General Overview 29

2.1.2 Input Data: DXF and XML 30

2.1.3 3dGen: Building Model Generation 32

2.1.4 xml2ug: conversion of XML into unigrafix 34

2.1.5 Inventor: output data 34

2.2 3DGen in-depth overview 35

2.3 XML to Unigrafix to Inventor 38

3 Algorithms in 3DGen **41**

3.1 Data Structures 41

3.2 2D Vertex Order and Orientation 43

3.2.1 Manifold Property requirements revisited 43

3.2.2 Vertex Orientation Checker Algorithm 43
3.3 Portal Operations 45
3.3.1 Implicit Portal fixer 45
3.3.2 Multiple explicit portals on an edge 46
3.4 Implicit Surfaces 47
3.5 Generating 3D Geometry 49
3.5.1 Generating walls with no portals 49
3.5.2 Generating explicit portals 51
3.5.3 Generating implicit portals 54
3.5.4 Generating floors and ceilings 55
3.5.5 Watertight Property in Horizontal Portals 57
3.6 Multiple floors 58
3.6.1 Stacking floors on top of each other 58
3.6.2 Completing manifold in Vertical Portals 59
3.7 Limitations 60

4 Conclusions **61**

5 Future Work **63**

A Algorithms **65**
A.1 Algorithm: generation of explicit portal vertices 65
A.2 Algorithm: detection of overlapping portals and correction 68
A.3 Algorithm: generation of faces with multiple portals in them 71
A.4 Algorithm: detecting implicit surfaces 74
A.5 Algorithm: generation of horizontal portal surfaces 78
A.6 Algorithm: generation of vertical portal surfaces 80

B 3dGen Manual **81**
B.1 3dGen Package Overview 81
B.2 Where to obtain 3dGen 81
B.3 Running 3dGen 82

B.4 Parameters set in the code 83
B.5 Viewing geometric models in VRML 84

C 3D Geometric models 85

List of Figures

1-1 Illustration of 2D floor plan of a single floor (as defined in DXF.) . . 19
1-2 The 3D model of the floor generated by 3dGen. 19
1-3 Explicit and Implicit portals on a floorplan. 21
1-4 Explicit and Implicit portals, in 3D. 21
1-5 2D vertices, contour edges, and closed spaces. 22
1-6 2D edges from adjacent spaces. 23
1-7 Illustration of importance of properly-oriented vertices in drawing surfaces. 24
1-8 Illustration of manifold property in the 3D space. 25
1-9 Analogy of Watertight Property. 26

2-1 The BMG framework system overview. The yellow boxes are modules that fall into the scope of this paper. 30
2-2 XML output of DXF Parser and an input to 3DGen. 31
2-3 XML output of 3DGen (with 3D data included.) 33
2-4 Output of xml2ug after XML has been converted to unigrafix. 34
2-5 The 3D Generator system overview. 35
2-6 Illustration of vertex redundancy (when each wall has its own set of vertices). 36
2-7 Illustration of limited vertex redundancy at portals. 37
2-8 Visualization of the model building 10 on MIT campus, as seen in ivview. 39

3-1 The Data Structure of the Object Model of 3DGen. 42
3-2 Illustration of manifold property in the 3D space. 44

3-3 Illustration of vertex orientation checker. 44
3-4 Illustration of the sorted portals map. 46
3-5 Illustration of the portal overlap and subsequent fix. 47
3-6 Example of implicit surfaces. 48
3-7 Illustration of intersection of coincident contour edges. 49
3-8 Wall extrusion and 3D vertex generation. 50
3-9 Illustration of break-up of the wall into faces surrounding the explicit portal. 51
3-10 Adjacent semi-edges are directed in the opposite direction to form a complete shared edge. 52
3-11 Nooks and crannies of portal vertex generation. 53
3-12 Generation of faces for multiple explicit portals for an edge. 53
3-13 Illustration of break-up of the wall into faces surrounding the implicit portal. 54
3-14 Illustration of different positions of implicit portal in the wall. 55
3-15 Wall with portals of different types (implicit portals mixed with explicit portals.) 56
3-16 Shared edges between floor and the vertical wall: bad and good scenarios.) 56

A-1 Location of Portal Vertices. 67
A-2 Detecting overlapping portals. 69
A-3 Three cases of drawing portals. 71
A-4 Four cases for overlapping coincident edges. 76
A-5 Coincident Edges: Case 1. 76
A-6 Coincident Edges: Case 2. 76
A-7 Coincident Edges: Case 3. 77
A-8 Coincident Edges: Case 4. 77
A-9 Portal surfaces for explicit horizontal portals 79

B-1 The example sequence of commands to generate the geometric model of MIT building 10 83

C-1 Model 1 view 1: Looking into a neighboring space through the implicit portal 85

C-2 Model 1 view 2: An example of the doorframes at portals 86

C-3 Model 1 view 3: Hallway as seen inside the VRML viewer 86

List of Tables

Chapter 1

Introduction

Many people are accustomed to seeing floor plans of museums, airports, malls, and other public places. In home purchase, a real estate broker will provide future tenant with floor plans of his future home, along with pictures of the interior. Architects create elaborate architectural drawings that may not be readable by ordinary people. In all these scenarios architects, real estate brokers, and members of the public deal with two-dimensional plans of buildings. For some people 2D plans may be enough, but in many cases three-dimensional models are needed for better assessment of the building interior.

3D models would help architects discover inconsistencies in their initial design (such as leaky corners, improper location of doorways, amongst other things), something they may have missed on 2D drawings. In addition, architects may find it easier to communicate with their clients, who may not be well-versed in understanding multiple architectural symbols on their drawings [1]. Real estate brokers can provide potential buyers with 3D models to help buyers visualize the property better. An advanced version of the 3D building model with actual pictures of the interior would make a great virtual walkthrough for potential buyers and tenants. Most people prefer seeing 3D models instead of flat floor plans in order to feel the interior of any building. They would prefer to see doorways, peek into neighboring rooms as if they are inside the building itself. Virtual tours can be created to let members of the public tour buildings remotely, from their own computers.

The main purpose of this project is to generate a corpus of many buildings on MIT campus. The final model contains 3-dimensional representations of every floor, stacked on top of each other, forming each building as a whole.

This chapter introduces terminology that is used throughout this thesis, as well as certain properties that have to be satisfied. Chapter two describes the architecture of the BMG Framework and of 3D Building Model Generator. Chapter three describes the design of the 3D Building Model Generator as well as any algorithms that were used in generating the 3D corpus. Chapter four outlines the conclusion made from the previous three chapters. Chapter five discusses how ideas in this thesis can be extended to future projects. Appendix A gives technical details behind the algorithms used in this thesis, and appendix B describes how to set up and run 3dGen.

1.1 Motivation for 3D model generation from floor plans

There are a lot of tools available for graphics designers, architects, and interior designers to create 3D models by hand. In addition, anyone can use a popular Sketch-Up application provided by Google [2] to create three-dimensional building models. Game designers rely on the skill of professional interior designers to design interior of buildings in action games. Some even create whole graphics engines for generating visual representations of buildings. DOOM is an action game which was popular in the mid 1990's. It used a sophisticated graphic rendering engine that was supposedly easy to modify in order to create highly customizable floor plans. Around the same time, a group of MIT students from the Senior House dormitory hacked this engine and created a personalized virtual tour of their dorm [6]. Our project is different in the way that it seeks to generate 3D models directly from building floor plans automatically. Moreover, these floor plans are derived from original architectural drawings in AutoCAD format, not from some other floor plan editors. Architects and interior designers can use the 3D models of buildings to detect any inconsistencies that they

might have missed on the 2D floor plans. 3D model generation from 2D floor plans also helps architects communicate with clients who may not be well-versed enough to understand architectural symbols on AutoCAD drawings. For example, the symbols such as doors and dashed lines on the architectural floor plan may be better understood as openings in the walls in the 3D models. Figures 1-1 and 1-2 illustrate the input and output of 3dGen.

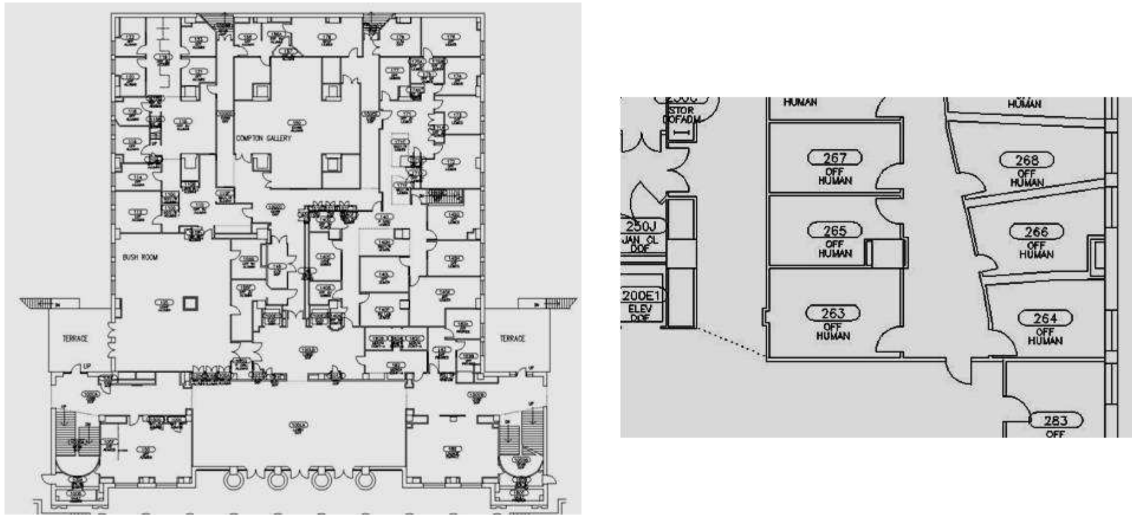


Figure 1-1: Illustration of 2D floor plan of a single floor (as defined in DXF.)

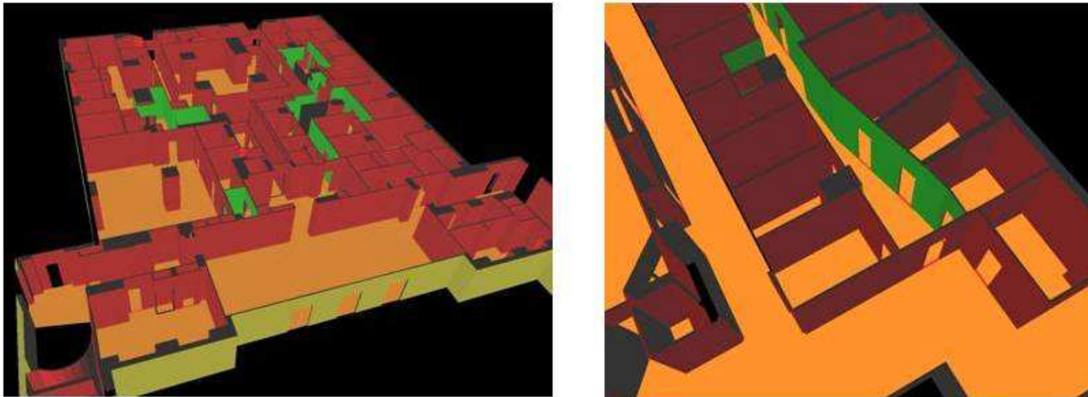


Figure 1-2: The 3D model of the floor generated by 3dGen.

1.2 Terminology Overview

This section explains each term that is used in this thesis in depth.

1.2.1 Spaces and Portals: overview

Before we dive deeper into the system design certain technical terms need to be clarified. A building consists of floors. Each floor can be broken down into chunks, called spaces (or room-sized regions). All spaces on the floor are interconnected to each other via portals (or openings). Portals can be classified according to two types: explicit portals and implicit portals. Explicit portals correspond to regular doorways and are usually defined as openings in the wall. Implicit portals, on the other hand, represent not doorways, but, rather, the functional interconnections between two open spaces. Usually implicit portals connect hallways with each other to allow for the impression that the hallway, such as the Infinite Corridor at MIT, spans multiple buildings, whereas the hallway is actually not a single space but a collection of multiple spaces fused together by implicit portals. Figure 1-3 illustrates this distinction. On the left is the section of a complete floor plan that includes both explicit and implicit portals. Implicit portals are defined by a dashed line. In the middle of the figure, only space contour edges are displayed (as seen from DXF file). These are 2D edges that define the contour of the space. Implicit portal is characterized by a single line between two spaces, 100CD and 100CC. In the picture on the right, explicit portals are shown. You can see that explicit portals occur only where the edges from adjacent spaces are parallel, not coincident, but close to each other. A wall layer (red) in DXF file specifies explicit portals. In the figure, there is one implicit portal between spaces 100CD and 100CC, and 5 explicit portals (4 that form 2 double doors, and one single door): one from space 100CC to 110, and four from space 150 to 100CC.

Figure 1-4 shows how different portal types appear in 3D.

Portals are also classified as horizontal and vertical portals. Horizontal portals are those that connect two spaces on the same floor. Vertical portals connect spaces that are on top of each other on different floors. Only horizontal portals can be implicit

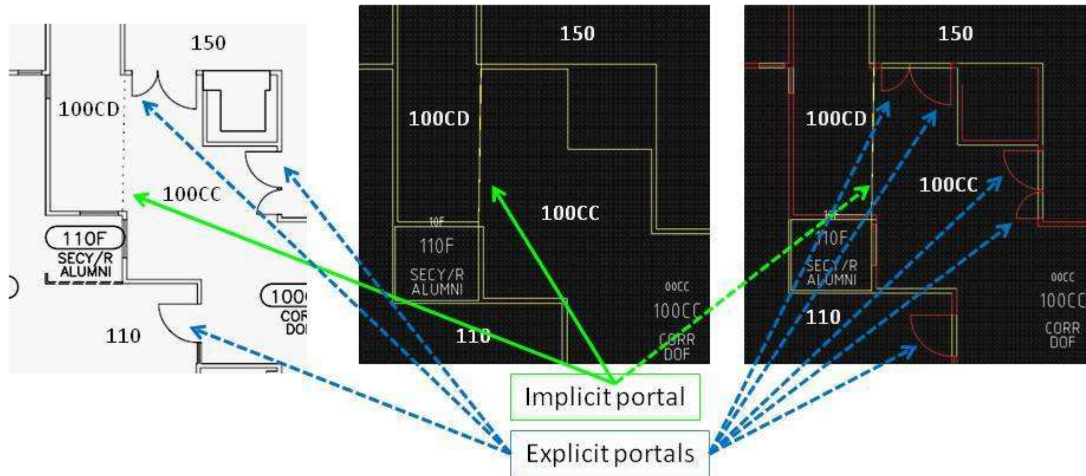


Figure 1-3: Explicit and Implicit portals on a floorplan.

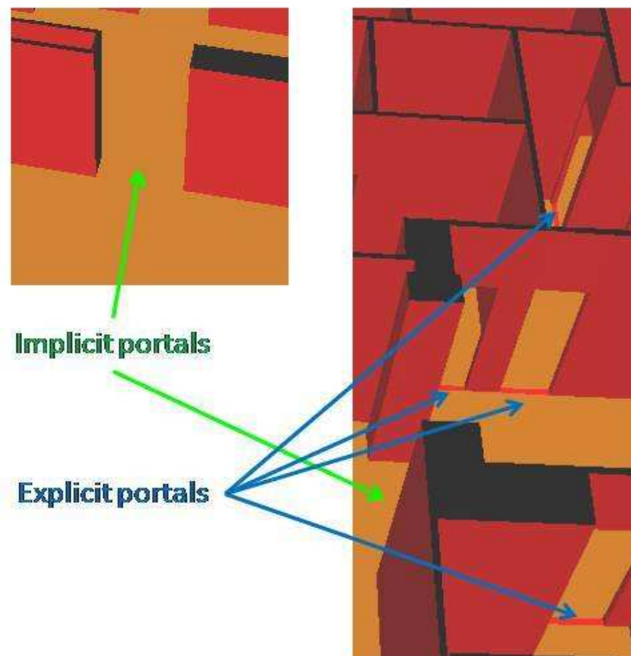


Figure 1-4: Explicit and Implicit portals, in 3D.

or explicit. Vertical portals by default span entire floor or the ceiling of the space that they occupy. Stairways, elevator shafts, rooms that span multiple floors are all examples where vertical portals are applicable.

1.2.2 Vertices and Edges: implementation

Each space is defined by 2D coordinates in the cartesian plane. Every space is defined as a list of vertices. Two vertices form an edge if they connect each other.

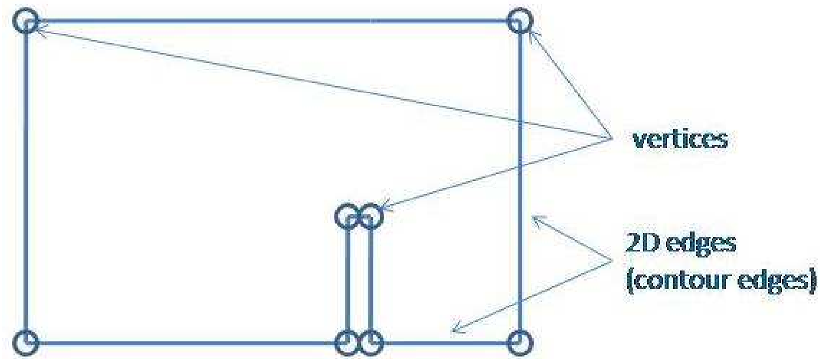


Figure 1-5: 2D vertices, contour edges, and closed spaces.

An edge in a geometric sense means a line segment that connects two vertices. An edge in our framework consists of two semi-edges, each pointing in the opposite direction. A semi-edge is a line segment connecting two vertices and pointed in one direction. Two semi-edges share the same two vertices, and therefore overlap. Direction is what makes them different. An edge pair in our framework consists of not one, but two contour edges, each pointing in the opposite direction. The reason behind this definition is that many spaces have walls (contour edges in 2D) that are adjacent to each other. These 2D edges are parallel, but point in opposite directions. Contour edge is a variation of the edge which is applicable only to 2D floor plans. Unlike a regular edge, contour edges are unidirectional. The distinction between contour edges and edges will become more apparent in the Surfaces/Faces and Walls subsection below.

An edge should not be confused with an edge pair. An edge is local to a space, whereas, an edge pair spans two spaces. Every contour edge in an edge pair belongs to one of the two adjacent spaces. Edge pairs ensure that every wall in a space has a neighboring wall in another space.

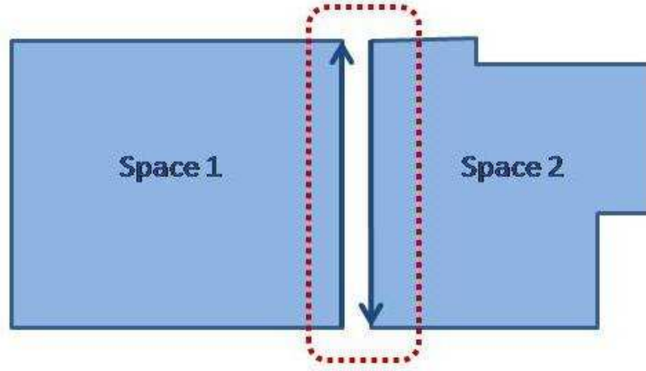


Figure 1-6: 2D edges from adjacent spaces.

1.2.3 Surfaces/Faces and Walls

A major part of BMG involves extruding surfaces from 2D edges to create walls for each space. The original floor plan is two-dimensional and does not contain any three-dimensional information such as vertices in 3D space that would define a face. A wall, or a surface, is defined by at least four vertices, two at the bottom, and two at the top. In which order the vertices are defined for the surface is important – it determines which side of the surface is the "front" side of the face [7]. If a face is defined by vertices appearing in a counterclockwise order, it is visible to the viewer. If a face is defined by vertices appearing in clockwise order, viewer observes the back face (face that is not visible). In order for all walls to be visible from inside the space, the vertices that define these walls must appear in the counterclockwise order. A wall is defined by two contour edges, each from one of the two adjacent spaces (these two contour edges form an edge pair), in 2D, or two faces in 3D that are visible in the opposite directions from each other. In fact, the surface normals of these faces point out of the wall in opposite directions. A face of the wall in one room is oriented in the opposite direction of the face of the same wall in adjacent room. This would result in this wall being visible from inside both rooms, but not from inside the wall (between the rooms). See Figure 1-7 for graphical illustration.

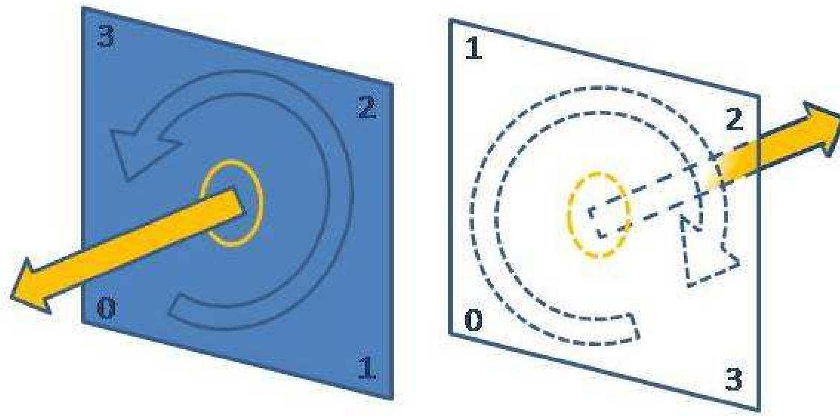


Figure 1-7: Illustration of importance of properly-oriented vertices in drawing surfaces.

1.3 Manifold property

Drawing programs render each polygon as a standalone entity without considering the fact that another polygon may share an edge with it. The same edge can be part of one polygon, and another edge – part of the other polygon, both in the same place. This leads to redundancy – why do we need to have two edges in the same location instead of one? It would have been better to have one edge that is shared between the two polygons. It would be more reasonable to put a single edge where two faces intersect instead of two edges, one for each face. The elimination of redundancy is good in cases when network bandwidth is limited and it would be best to send the least amount of data. The manifold property declares that an edge may only be used twice – since there are only two directions. A legitimate edge is defined as a combination of the two directions (e.g. up and down for vertical edge, left and right for horizontal edge). In this project we ensure that there can only be one edge at any location, and if there are two surfaces adjacent to each other at that edge, they have to share this single edge. Figure 1-8 illustrates the Manifold property. The two semi-edges that form the edge use the same two 3D vertices, but are drawn in the opposite directions.

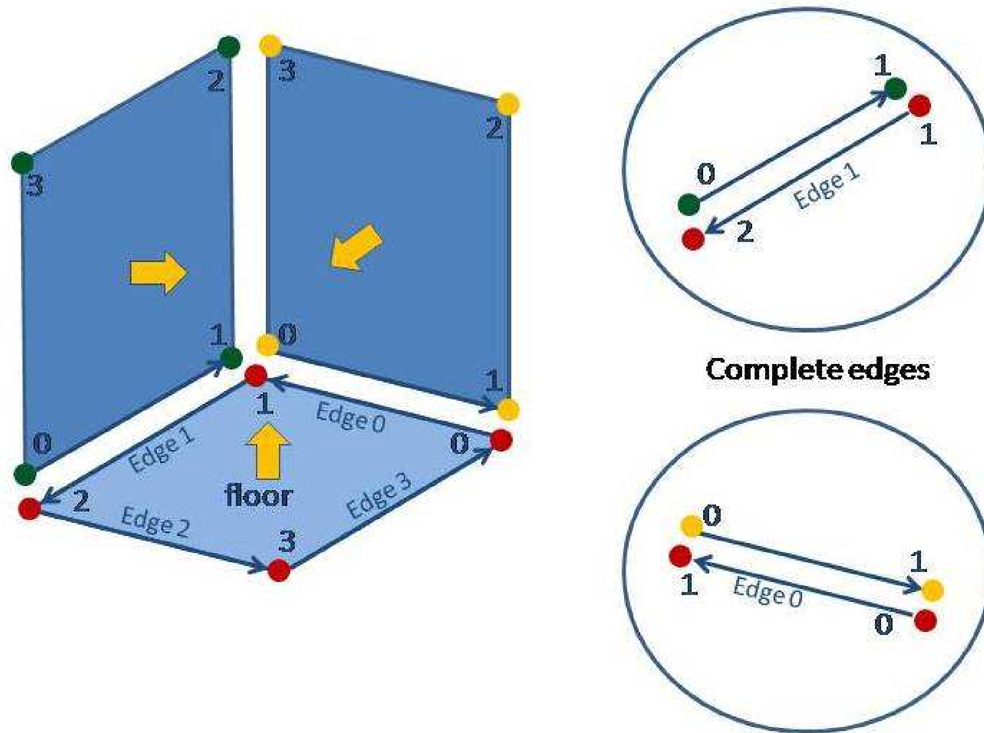


Figure 1-8: Illustration of manifold property in the 3D space.

1.4 Watertight property

The concept of a single shared edge not only removes redundancy, but also ensures that the space is "watertight", or, tightly sealed in a way if a water was poured into a geometric model, it would not be able to leak out of it. If two faces did not share an edge, and each had its own edge, then there would be a possibility of a leak if the two edges were not coincident.

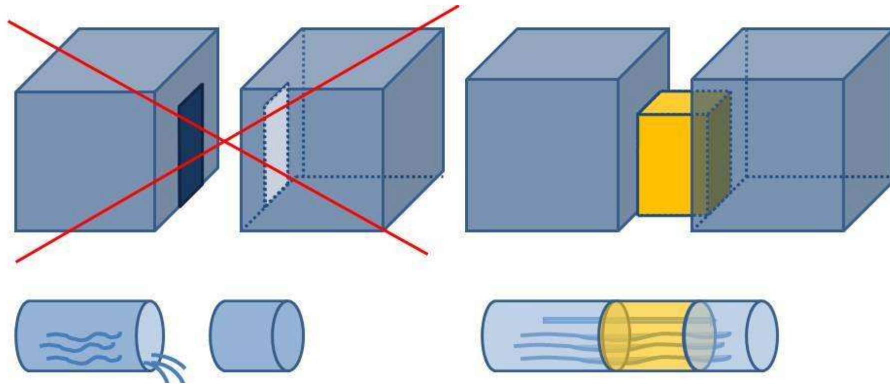


Figure 1-9: Analogy of Watertight Property.

In all cases spaces are connected to each other via portals. Just making an opening in the wall is not enough; the gaps between the walls at the portals also have to be sealed. As described earlier, a wall consists of two faces, each in its own space, facing into each of the respective spaces. There is empty space of some thickness between the faces of the wall. In order to ensure that the three-dimensional model of a floor is sealed and watertight, this gap between the faces of the wall has to be covered at the portal. This is like a doorstep and doorframe. To help you understand the watertight property, think of spaces as pipes, and portals as interconnections between the pipes. In order for pipes to be leak-proof, they have to be interconnected with each other. In order for the building model to be watertight, all spaces have to be connected; there cannot be a space with a portal that does not have another space on the other side of the portal. This also satisfies the manifold property by ensuring that every contour edge (edge in one direction) in edge pair in one space has another contour edge (in the opposite direction) in adjacent space. If one of the pipes in our analogy is unpaired, all water would leak out of the system. Figure 1-9 shows an example of spaces not being connected by interconnections.

3DGen takes care of the manifold property by ensuring that all surfaces are defined by vertices listed in a counterclockwise order (thus forming edges where any two surfaces intersect each other). The watertight property is implied by the manifold property and is satisfied partly by using edges that are shared between surfaces (walls,

faces), and partly by generating additional surfaces to connect the spaces at portals. Chapter 3 will do into more depth on the algorithms that ensure that these two properties are satisfied.

Chapter 2

System Architecture

This chapter describes the BMG framework and its modules, especially the 3dGen module.

2.1 System Overview

This section briefly describes each module of BMG framework.

2.1.1 General Overview

The BMG framework includes many separate steps for data processing. The floor plans are provided in the AutoCAD DXF format, then converted to XML format by the DXF Parser/Converter (dxf2xml). The XML format is used because it is easily readable by human eye, has a convenient tree structure and hence easier to parse with appropriate XML libraries. XML format may not be most compact format, but it is a very popular standard that is used across many industries and networks.

As you can see from Figure 2-1, the 3D Building Model Generator (3DGen) is only a subset of the framework. The principal component of the BMG framework is to provide spatial indoor information as a text stream in a human-readable format. This XML data of floor plans can be used by various applications. The XML stores the physical coordinates of each aspect of the space (room, hallway). The graphical

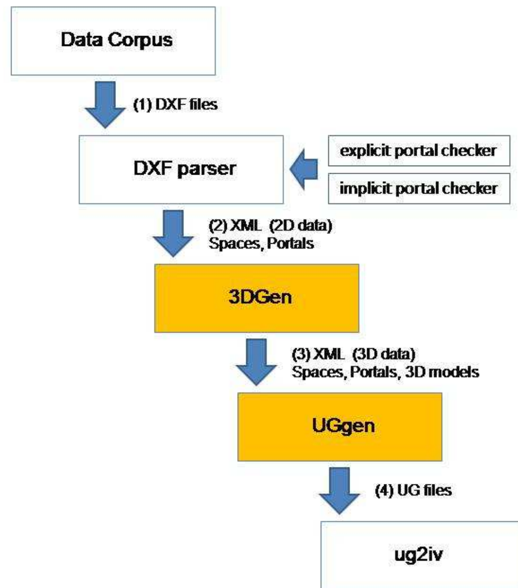


Figure 2-1: The BMG framework system overview. The yellow boxes are modules that fall into the scope of this paper.

rendering is left to outside applications, so there is a greater freedom in determining how this XML data is to be displayed.

2.1.2 Input Data: DXF and XML

According to figure 2-1, the DXF parser converts DXF from AutoCAD proprietary format to XML. Details of this parsing/conversions are described in Whiting et al. [9].

The XML output of the DXF parser contains only two-dimensional data such as the location of vertices that define a space, the location of the portals (both the edge on which portal lies and the position on that edge). An example XML space is shown in the figure below (Figure 2-2).

```xml
<MITquest>
  <space name="10-100" type="OFF">
    <contour>
      <centroid x="710109.204664" y="495618.803866"/>
      <extent maxx="710124.926644" maxy="495633.909708" minx="710094.974557" miny="495604.344736"/>
      <point x="710123.855353" y="495619.274498"/>
      <point x="710117.185703" y="495633.909708"/>
      <point x="710115.669101" y="495633.218553"/>
      <point x="710116.083794" y="495632.308591"/>
      <point x="710123.855353" y="495619.274498"/>
    </contour>
    <portal class="horizontal" source="false" target="10-100A" type="explicit">
      <edge index="11" param="0.517804"/></portal>
    <portal class="horizontal" source="true" target="10-100C" type="explicit">
      <edge index="3" param="0.574324"/></portal>
    <portal class="horizontal" source="true" target="10-100CA" type="explicit">
      <edge index="6" param="0.674419"/></portal>
    <portal class="horizontal" source="true" target="10-100D" type="explicit">
      <edge index="4" param="0.750000"/></portal>
    <portal class="horizontal" source="true" target="10-100D" type="explicit">
      <edge index="4" param="0.434211"/></portal>
    <portal class="horizontal" target="10-100F" type="implicit">
      <edge index="10" maxparam="1" minparam="0"/></portal>
  </space>
  <space name="10-100A" type="OFF_SV">
    <contour>
      <centroid x="710091.217248" y="495600.506820"/>
      <extent maxx="710098.368918" maxy="495609.170020" minx="710083.273802" miny="495595.236079"/>
      <point x="710094.850740" y="495609.170020"/>
      <point x="710094.893982" y="495607.266567"/>
      <point x="710094.523516" y="495605.266157"/>
      <point x="710093.531995" y="495603.623768"/>
      <point x="710094.850740" y="495609.170020"/>
    </contour>
    <portal class="horizontal" source="true" target="10-100" type="explicit">
      <edge index="13" param="0.461198"/></portal>
  </space>
</MITquest>
```


Figure 2-2: XML output of DXF Parser and an input to 3DGen.

Each space is defined by a Space Node in XML (`<space>`). The space contains a single contour (`<contour>`) and multiple portals (`<portal>`), each represented by its own node (or tag) in XML. A contour stores multiple point nodes (`<point>`), each of which has attributes that define x and y coordinates of the point. A portal tag has the following properties as attributes: classification (horizontal or vertical portal), source (whether the portal opens up to another space or another space opens up into the current space), target (the name of the space to which the portal leads), and type (explicit or implicit). Within the portal node there is an edge node (`<edge>`) which specifies the index of the edge the current portal lies on, and the parameter

(parametric location of portal relative to the endpoints of the edge between 0.0 and 1.0). Implicit and explicit portals have different edge nodes. In implicit portals, there are two parameters, instead of one – minparam, and maxparam attributes that define the starting and ending points on the edge. Recall, implicit portals have arbitrary width, and hence are not classified the same way as explicit portals (most doorways have similar width). Explicit portals store only one parameter attribute – the center of the portal along the edge. The width of the portal is determined by the default door width. The major difference between horizontal and vertical portal nodes is that there are no type and source attributes, and there is no internal edge node in vertical portals. Vertical portals have to specify vertical direction (UP or DOWN) that would specify whether the portal should be on the ceiling or the floor of the room.

2.1.3 3dGen: Building Model Generation

The 3D Generator computes additional vertices, generates appropriate surfaces in the 3D coordinate space and stores the newly generated 3D dataset inside the `ThreeD` node. This node stores a type of the space, multiple `<Vertex>` nodes, and multiple `Face` nodes. Each Vertex node stores x, y, and z coordinates, and each face node lists the IDs of the vertices in the appropriate order. The `ThreeD` node contains all the data needed to generate surfaces. 3D Geometry Generator alters the existing input XML to produce an updated version that includes the `ThreeD` tag inside each space. The illustration of XML output is below in Figure 2-3.

```
<MITquest>
  <space name="10-100" type="OFF">
    <contour>
      <centroid x="710109.204664" y="495618.803866"/>
      <extent maxx="710124.926644" maxy="495633.909708" minx="710094.974557" miny="495604.344736"/>
      <point x="710123.855353" y="495619.274498"/>
      <point x="710117.185703" y="495633.909708"/>
      <point x="710115.669101" y="495633.218553"/>
      <point x="710116.083794" y="495632.308591"/>
      <point x="710123.855353" y="495619.274498"/>
    </contour>
    <portal class="horizontal" source="false" target="10-100A" type="explicit">
      <edge index="11" param="0.517804"/></portal>
    <portal class="horizontal" source="true" target="10-100C" type="explicit">
      <edge index="3" param="0.574324"/></portal>
    <portal class="horizontal" source="true" target="10-100CA" type="explicit">
      <edge index="6" param="0.674419"/></portal>
    <portal class="horizontal" source="true" target="10-100D" type="explicit">
      <edge index="4" param="0.750000"/></portal>
    <portal class="horizontal" source="true" target="10-100D" type="explicit">
      <edge index="4" param="0.434211"/></portal>
    <portal class="horizontal" target="10-100F" type="implicit">
      <edge index="10" maxparam="1" minparam="0"/></portal>
    <ThreeD>
      <Type name="OFF"/>
      <Vertex edgeindex="0" name="0" x="710123.88" y="495619.28" z="0.00"/>
      <Vertex edgeindex="0" name="1" x="710123.88" y="495619.28" z="15.00"/>
      <Vertex edgeindex="1" name="2" x="710117.19" y="495633.91" z="0.00"/>
      <Vertex edgeindex="1" name="3" x="710117.19" y="495633.91" z="15.00"/>
      <Vertex edgeindex="2" name="4" x="710115.69" y="495633.22" z="0.00"/>
      <Vertex edgeindex="2" name="5" x="710115.69" y="495633.22" z="15.00"/>
      <Vertex edgeindex="3" name="6" x="710116.06" y="495632.31" z="0.00"/>
      <Vertex edgeindex="3" name="7" x="710116.06" y="495632.31" z="15.00"/>
      <Vertex edgeindex="3" name="8" x="710114.25" y="495631.47" z="0.00"/>
      <Vertex edgeindex="3" name="9" x="710114.25" y="495631.47" z="10.00"/>
      <Vertex edgeindex="3" name="10" x="710114.25" y="495631.47" z="15.00"/>
      <Vertex edgeindex="4" name="14" x="710110.50" y="495629.75" z="0.00"/>
      <Vertex edgeindex="4" name="15" x="710110.50" y="495629.75" z="15.00"/>
      <Face vertices="0,1,3,2,0"/>
      <Face vertices="2,3,5,4,2"/>
      <Face vertices="4,5,7,6,4"/>
      <Face vertices="6,7,10,9,8,6"/>
      <Face vertices="9,10,13,12,9"/>
      <Face vertices="11,12,13,15,14,11"/>
      <Face vertices="14,15,1,0,14"/>
      <Floor vertices="0,2,4,6,8,11,14,0"/>
    </ThreeD>
  </space>
</MITquest>
```


Figure 2-3: XML output of 3DGen (with 3D data included.)

2.1.4 xml2ug: conversion of XML into unigrafix

The next step is to convert the XML dataset into a format that is readable by graphical tools. For this project, Unigrafix [7] was used because it is easy to read to the human eye (and debug). Notice that the structure of the unigrafix format closely follows the structure of `ThreeD` node in XML. The following Figure 2-4 shows what UG output looks like.

```
c_rgb solelayer 1.000000 0.000000 0.000000 ; { color 1 }
c_rgb stairlayer 0.000000 1.000000 0.000000 ; { color 1 }
c_rgb floorlayer 204.000000 153.000000 0.000000 ; { color 1 }
c_rgb floorcontour 253.000000 245.000000 230.000000 ; { color 1 }

v 10-100_0 710123.88 495619.28 0.00 ;
v 10-100_1 710123.88 495619.28 15.00 ;
v 10-100_2 710117.19 495633.91 0.00 ;
v 10-100_3 710117.19 495633.91 15.00 ;
v 10-100_4 710115.69 495633.22 0.00 ;
v 10-100_5 710115.69 495633.22 15.00 ;
v 10-100_6 710116.06 495632.31 0.00 ;
v 10-100_7 710116.06 495632.31 15.00 ;
v 10-100_8 710114.25 495631.47 0.00 ;
v 10-100_9 710114.25 495631.47 10.00 ;
v 10-100_10 710114.25 495631.47 15.00 ;

f face41 ( 10-100_0 10-100_1 10-100_3 10-100_2 10-100_0 ) solelayer ;
f face42 ( 10-100_2 10-100_3 10-100_5 10-100_4 10-100_2 ) solelayer ;
f face43 ( 10-100_4 10-100_5 10-100_7 10-100_6 10-100_4 ) solelayer ;
f face44 ( 10-100_6 10-100_7 10-100_10 10-100_9 10-100_8 10-100_6 ) solelayer ;
```

Figure 2-4: Output of xml2ug after XML has been converted to unigrafix.

In unigrafix format only vertices and faces are defined. All faces and vertices have been generated by the prior step – 3DGen. Vertices are defined on lines starting with a v, while faces are defined on lines starting with an f.

2.1.5 Inventor: output data

The final step converts Unigrafix into Inventor format that is used by many graphical viewers. Inventor is the open source 3D vector graphics format to describe graphical objects in the 3-dimensional space. Inventor (IV) format is identical to VRML97 format, which is used to render graphical objects specifically for the World Wide Web environment. The decision to use IV format came from the fact that the 3D

Building Model Generator was originally conceived to generate building models for online virtual tours. The viewer from the Open Inventor Project [5], ivview, was used in displaying the generated 3D models.

2.2 3DGen in-depth overview

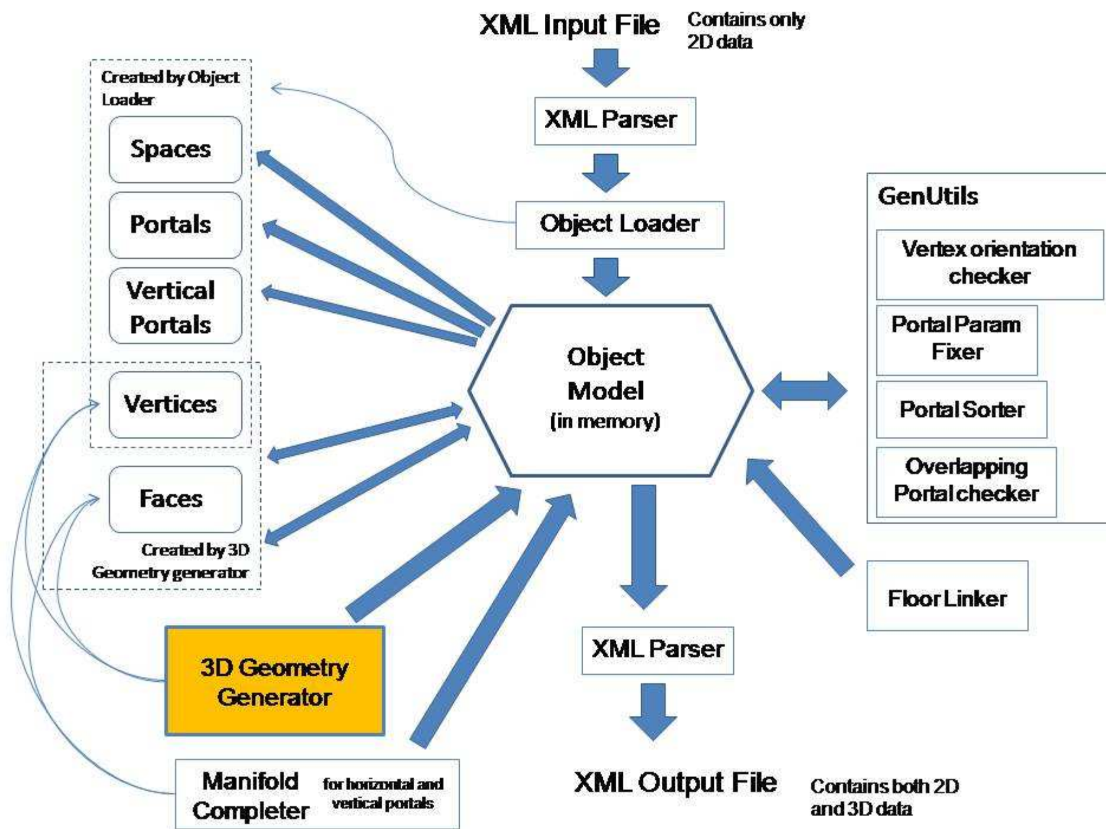


Figure 2-5: The 3D Generator system overview.

Figure 2-5 illustrates all important modules of 3DGen – a tool that generates three dimensional geospatial data for modeling of buildings. The XML input to 3DGen is the output of the DXF parser and can be seen in Figure 2-2. This XML data contains all 2D spatial data necessary for the building model generation, including spaces and portals, and their respective location information. Before any wall extrusion is done, all data is loaded into the object model which resides in the memory. The

reason for using object model instead of processing on the XML data directly is that all spaces are linked to each other, and in order for 3D data to be consistent and void of redundancy, 3DGen needs to look at the entire building in order to generate 3D information. Another advantage of object model is the speed of computation. Operations on memory are generally faster than operations disk file.

XML Parser reads the input XML file and passes this data into the Object Loader. Object Loader creates Space, Portal, and VPortal (Vertical Portal) objects, as well as Vertex objects. Vertices are stored locally with each space. Storing vertices with a space, as opposed to storing with each wall/face helps eliminate redundancy. There is only one vertex at each vertex location. If vertices were stored inside each wall, then there would be many cases where two walls in the same space intersect each other in the corner of the room; there would be two vertices at a single point location at the intersection of these two walls. See Figure 2-6 below.

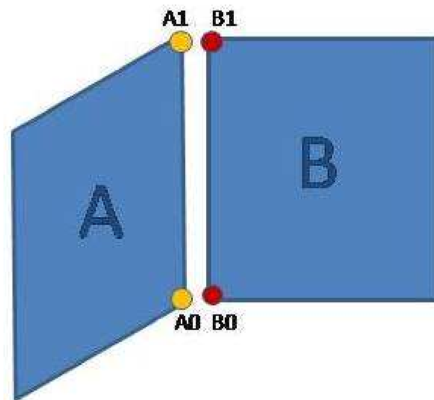


Figure 2-6: Illustration of vertex redundancy (when each wall has its own set of vertices).

A0 and A1 are in the same location, and so are B0 and B1, which is redundant. It makes much more sense if there was a single vertex A0 that would be shared by walls A and B. Same applies to vertices A1 and B1. This was the main reason vertices were incorporated into space objects rather than into wall objects. One would argue why not keep all vertices in a central location that would span across all spaces and floors of a building (as in UNIGRAFIX)? Doing so would make it very hard to

debug any potential issues. Storing vertices inside spaces offers a compromise between storing vertices for every graphical object and storing vertices in a central repository. The redundancy is somewhat broken when two spaces join each other at portals. In this case, where a doorstep meets the portal of some space, there are redundant vertices (each belonging to each of the two spaces). However, such occurrences are rare, relative to the total number of vertices in the floor. Figure 2-7 illustrates what happens when the portal doorstep meets the wall of the space.

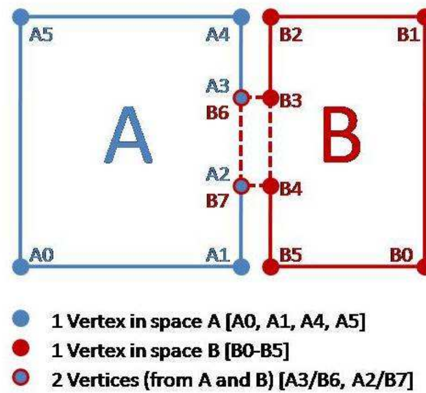


Figure 2-7: Illustration of limited vertex redundancy at portals.

At the portal, the doorstep extends from space B into space A. Since vertices are stored locally for each space, there is no easy way to share vertices across spaces. This is why redundancy is allowed here to a certain extent. There are more vertices that are not redundant in the object model than there are redundant vertices (for space A, it is 2 redundant vertices out of 8 vertices total for space A. For space B, 2 out of 6 vertices are redundant). The doorstep in this example belongs to space B. How the ownership of the doorstep was determined will be described later.

Once all spaces and portals have been initialized in the object model, several tools operate on the 2D data to fix any inconsistencies the data might have. Examples of inconsistencies are: (1) improper sequence (order) of vertices defining a space contour; (2) parameters in implicit portals that extend beyond the 0.0-1.0 range; (3) portals that are too close to each other. These tools are all part of genUtils package that was created for this project.

Once all these inconsistencies are corrected, the actual 3D Geometry generation begins. The 3D Geometry generator adds additional vertices to the object model and generates faces for each wall. It also takes horizontal portals (both implicit and explicit) into consideration in generating surfaces for the faces. The details of how such face generation occurs are outlined in chapter 3. In case of a model that spans multiple floors, the z-coordinates for each vertex are calculated based on the position of the floor relative to the bottom-most floor.

After all faces have been generated, the Manifold Completer is run to insert interconnections between spaces due to portals. This is necessary to ensure that the entire floor is watertight – that there would be no leak if water was poured into the model of the floor. These interconnections consist of four additional surfaces per portal to fill in gaps between the adjacent spaces inside each portal. This fix ensures that if an observer was put inside the portal, he would not be able to peek outside of the portal into the region between the walls of adjacent spaces.

Once the floor becomes watertight, the Floor Linker stacks floors on top of each other to produce a complete model of the building. It calls the Manifold Completer to create interconnections between floors at vertical portals.

Lastly, the XML parser is run on the entire object model and updates the input XML with a new `ThreeD` node for each space that contains all the geospatial information needed to generate three-dimensional surfaces, as seen in Figure 2-3. This node lists vertices and faces in the same way as unigrafix or inventor would. The output XML contains all data from the input XML plus the 3D data. It is later converted by xml2ug into Unigrafix format.

2.3 XML to Unigrafix to Inventor

The xml2ug tool reads in output XML from the 3DGen, parses it and uses only information from `ThreeD` in each space. It ignores everything else. Since all geospatial data has been generated by 3DGen, no additional processing is required from xml2ug. For each vertex and face, xml2ug creates a line in the unigrafix output file. It can

differentiate between a floor/ceiling and a face tag and assigns different colors to the floor and ceiling than for other faces (walls). Also, xml2ug can easily be modified to assign different colors to different space types (stair, halls, offices, etc.).

A tool created by BMG group [4] converts unigrafix to inventor format. The result inventor (iv) file can be loaded into the ivvew application to visualize the complete model of the building. See Figure 2-8 for a graphic visualization of the building model.

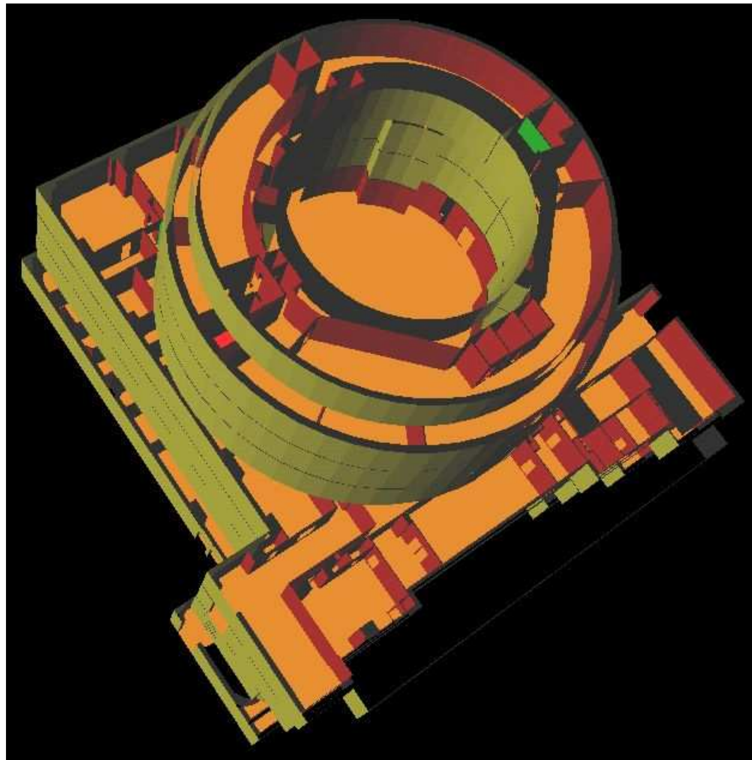


Figure 2-8: Visualization of the model building 10 on MIT campus, as seen in ivview.

There can be issues with the light model inside ivview. The iv file that has been generated by ug2iv uses the default phong light model with a point light source. This results in some rooms seeming darker than others. A small fix was applied to the inventor file to make all walls ambient, or to make all walls show up, regardless of their position relative to the light source. The result is shown in Figure 2-8 above.

Chapter 3

Algorithms in 3DGen

3.1 Data Structures

According to Figure 3-1, there are 5 main data structures. Objects of type `Space` store all data about each room (space), including the list of 2D vertices that define the space contour, the list of portals, and the list of walls. 3D Generator creates more vertices and puts all vertices in the `vertices3D` list of the space. Objects of type `Vertex` store the x, y, and z coordinates of the points that they represent in the 3D space. Objects of type `Wall` store the list of indices of 3D vertices (the location of these vertices in the `Space`'s `vertices3D`. Objects of type `Portal` store name, type (Office, Hallway, Stair, etc...), name of the target space (space this portal connects current space to). Portals also store the index of the edge of the space contour on which they are located, and the parametric value that defines where on that edge the portal lies. For explicit portals only center parameter is stored (center parameter is a measure of the center of the portal relative to the endpoints of the edge, and can take values anywhere between 0.0 and 1.0 only. Portal's width is computed from default width). For implicit portals minParam and maxParam values are stored that define the edges of the portal (left and right side). These parameters define the width of the implicit portals. Often, an implicit portal would occupy the entire edge; in this case minParam would be 0.0 and maxParam would be 1.0. Portal objects also store the array of Vertex pointers that define the contours of the portal (normally 6 vertices

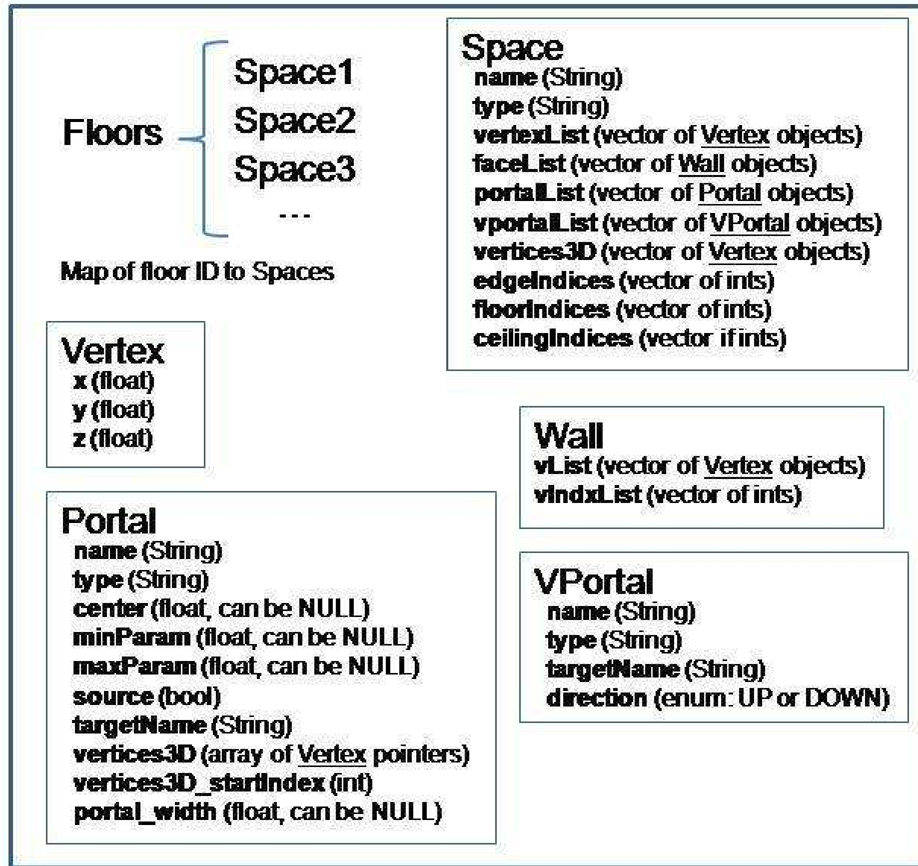


Figure 3-1: The Data Structure of the Object Model of 3DGen.

for explicit portal). `Portal` class is used to define only the horizontal portals; vertical portals have their own class, `VPortal`. Objects of this class (`VPortal`) store only name, type, name of the target space, and the direction (UP or DOWN, meaning connecting the space above or below). For every physical portal in the floor, there are two portal objects, one in each of the two spaces it is connected to.

Instead of replicating vertex objects for space, for walls, and for portals, vertex indices were used. These indices point to the location of the 3D vertex in the `vertices3D` list inside each `Space`. The reason is that we wanted to have as few data objects as possible and use pointers as often as possible since integers (the type of indices) are much smaller than `Vertex` data objects. This way, all 3D vertices are stored in only one place – inside the `Space` object for each space.

3.2 2D Vertex Order and Orientation

Before any 3D Geometry Generation occurs, certain irregularities have to be fixed. In Computer Graphics, the order in which vertices of the surface are defined is important. For instance, if 4 vertices defining the contour of a rectangle were listed in a counterclockwise order, then the surface drawn would have a normal pointing out of it, and the surface would be visible to the viewer. If the vertices, instead, were placed in the clockwise order, then the surface normal of the retangle would point into the surface, which means that the rectangle would be visible only from behind it, not from the front. See Figure 1-7. In this project the orientation of vertices for each wall depend on the orientation of 2D vertices of the space. This orientation closely follows the floor normal. The floor normal has to point up, and this means that for the bird-eye view of the floor, vertices defining the floor have to be oriented counterclockwise. The vertices defining the wall surface rely on the proper floor vertices (2D vertices) orientation.

3.2.1 Manifold Property requirements revisited

Figure 3-2 shows that for the manifold property to be satisfied, an edge needs to be bi-directional – meaning there have to be two segments, the so-called semi-edges, pointing in the opposite direction, one in each of the two adjacent spaces.

3.2.2 Vertex Orientation Checker Algorithm

Drawing a wall surface can be a challenge since the 2D vertices in the input XML file may not be listed in the correct order (counterclockwise). For each 2D edge, a wall at that edge can face in one of the two directions, into the wall, or out of the wall. Instead of trying to guess the proper orientation of the surface normal, we used the floor as a reference point. While wall normals can point in any direction, the normal of the floor in any space should always point up. It is much easier to check if the floor normal is pointing up or down (only two choices) as opposed to nearly infinite choices for the wall normal orientation. A special checker was implemented to verify that all

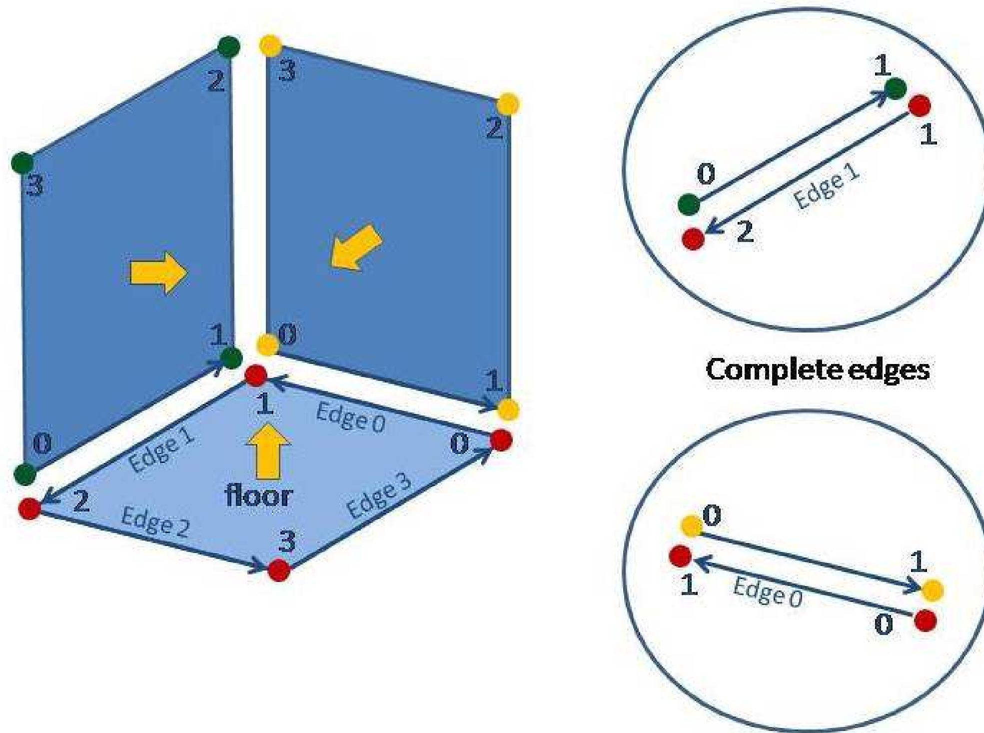


Figure 3-2: Illustration of manifold property in the 3D space.

floor vertices are defined in the counterclockwise order and that the floor normal is always pointing up. This would ensure that all walls in the space have normals that point toward the center of the space, as opposed to out of the space. This is why floor surface is generated before any other surfaces are generated.

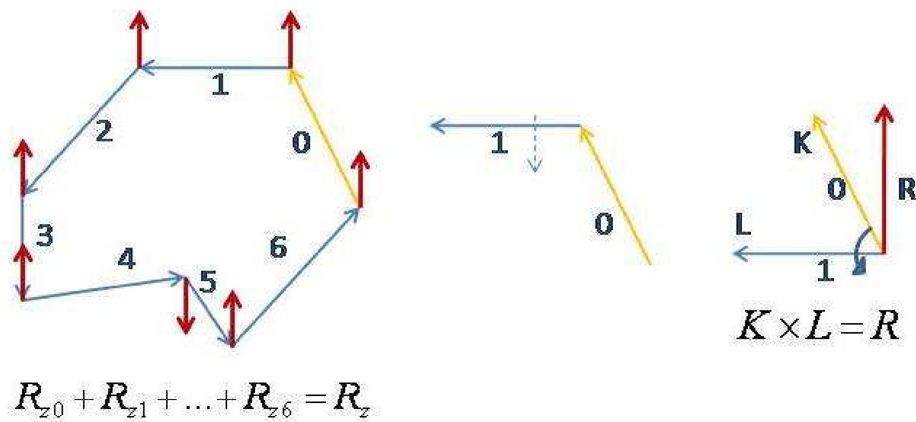


Figure 3-3: Illustration of vertex orientation checker.

Figure 3-3 illustrates the checking of the vertex orientation. The Cross Product was used to compute the curvature of the space contour. If the vertices were listed in counterclockwise order, then the cross product of two adjacent edges would be pointing up. The cross product algorithm was applied to every two adjacent edges formed by connecting 2D vertices (the last edge in the list was formed by connecting last vertex with the first vertex in the list). The result was normalized for every edge pair and the z-coordinates of all results were summed up. If the sum turns out to be a positive number, then the order of the vertices is the correct one – counterclockwise order. The floor would be pointing up. In this case, the ordering of the 2D vertices in the space is left intact. If z-sum is a negative number, then the floor would be pointing down and, and this would mean that 2D vertices are listed in the clockwise order. In this case, the order of 2D vertices for that space is reversed. In Figure 3-3 edges 0 and 1 together create a counterclockwise-oriented polygon. Thaking th ecross product of edges 0 and 1 results in the vector that points up.

Reversing the list of vertices is only one step. Another step involves changing the edge index in all portals that belong to the current space. Edge indices are set to $vertexList.size() - currentEdge - 1$. For example, if a space has 8 vertices, then it should have 8 edges. If the space's list of vertices was reversed, then the portal on original edge 3 gets 4 as the new edge index. The reversal of vertex order and portal edges is done before the geometry generation to avoid any complications in generating 3D geometry due to the improper vertex orientation (invisible floors, floor normals pointing down, wall surfaces pointing into the region between walls).

3.3 Portal Operations

3.3.1 Implicit Portal fixer

The input XML often has incorrect parameter values for implicit portals (`minParam` and `maxParam`. These two params could be outside the range they are allowed to occupy (0.0 to 1.0). If `minParam` is less than zero, it is clipped to zero, and if `maxparam`

is greater than one, it is clipped to 1.0. Also, sometimes, `minParam` could be greater than `maxParam`; in this case the values of `minParam` and `maxParam` are swapped.

3.3.2 Multiple explicit portals on an edge

How portals are represented on the wall is important. Since portal width for explicit portals is not captured by the DXF parser and hence is absent in the input XML, a default portal width is used. This presents a challenge of dealing with scenarios where two narrow portals are close to each other (as in double doors). In this case it is possible that the default width would be greater than the actual width of the portals. This would lead to two portals overlapping each other. This would complicate drawing walls with portals in them because in some cases there could be a chunk of the wall inside the portal. This is why we implemented the overlapping portal checker.

A special map data structure is created for each space. This map links edge index to the vector of portal pointers. This way, every edge that has multiple portals would contain an unordered list of portals. Before the overlapping portal checker is run, all portals are sorted for each edge based on the parameter as a key. If the portal is implicit, then an average parameter value between minParam and maxParam is computed. In the end, every edge has a sorted list of portals (Figure 3-4).

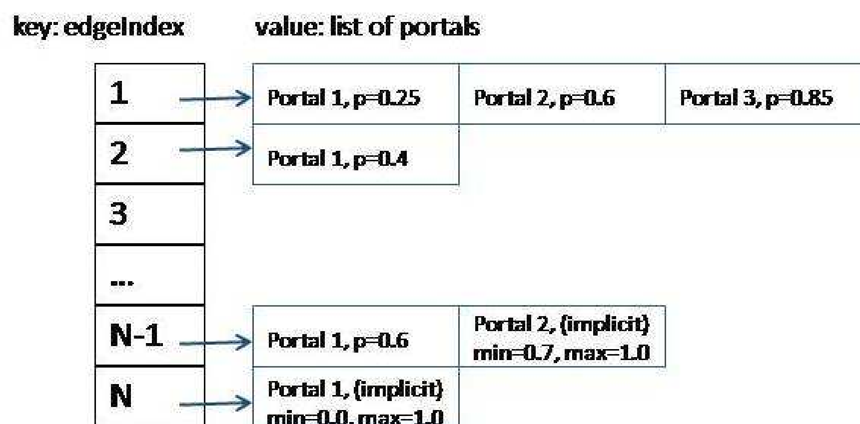


Figure 3-4: Illustration of the sorted portals map.

After all portals are sorted, the overlapping portal checker looks at every two

adjacent portals and determines whether they overlap. The formula used is:

$$\frac{portal1.portalWidth}{dist} + portal1.param < portal2.param \tag{3.1}$$

, where *dist* is length of the edge that the portal occupies. Notice that parameters, rather than actual distances are compared. If the equation 3.1 holds true then there is no portal overlap. Otherwise, the portals overlap and need to be corrected. In the portal correction phase, one of the portals is removed, and the other is modified. The modifications are made to center parameter and to portalWidth. The result portal is often wider than either of the original portals and spans the length of the edge occupied by both portals (Figure 3-5). Only explicit portals are checked for overlap. It is assumed that implicit portals do not overlap with other implicit or explicit portals, as such cases are extremely rare. If there is an overlap between implicit and explicit portal on an edge, then there is a bug in the input XML file. In this case the overlappting portal checker removes the explicit portal, leaving the implicit portal intact.

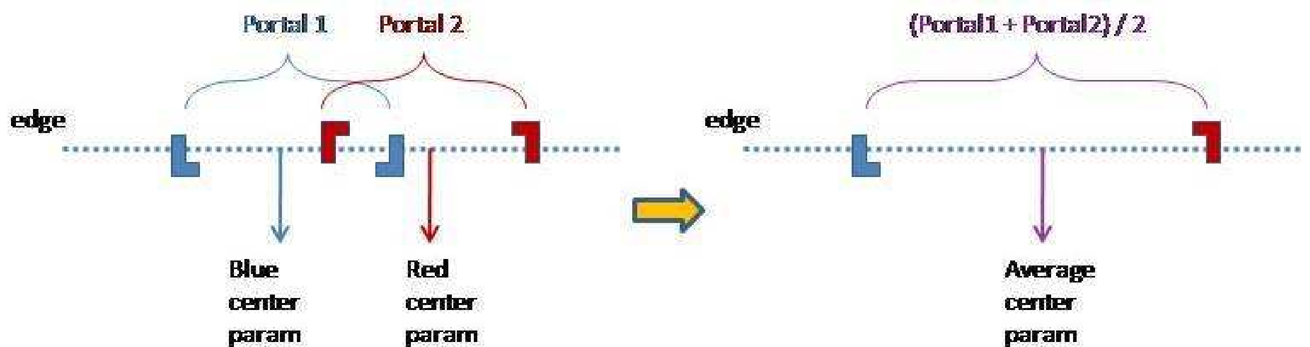


Figure 3-5: Illustration of the portal overlap and subsequent fix.

3.4 Implicit Surfaces

The input XML data is not without its faults. Very often inside the space contour there are lines that are coincident. For example, Figure 3-6 illustrates the case where

in order to draw a pillar in the middle of the room, the floor plan designers extend the space contour to the pillar by drawing the shortest line from the closest wall to the pillar. This is done to keep the space contour as a whole, in one piece without separating it into two contours. There can be only one contour for each space. This can result in a phenomenon called "Implicit Surfaces", which means that there are two surfaces at exactly the same location, with normals pointing in opposite directions. Inside inventor viewer, these would appear as flickering surfaces. This is itself an inconsistency in the 3D geometry and has to be eliminated.

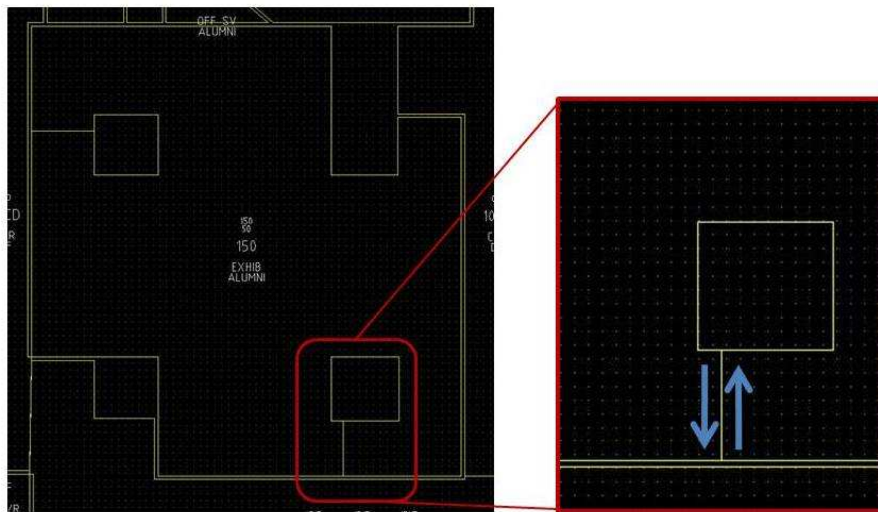


Figure 3-6: Example of implicit surfaces.

The detection of implicit surfaces is based on parametric algebra. Once two contour edges are determined to be collinear, another check is made to verify that they overlap. If the overlap is detected, an implicit portal from the space to itself is created with parameters being the endpoints of the overlap region between these two contour edges. Figure 3-7 shows that the red line is an intersection of two coincident edges, a and b. An implicit portal is created that would span this red line, thus giving an illusion that there is an opening between two regions of the space (as it should have been in real life). The space itself remains intact and is not split into two spaces.

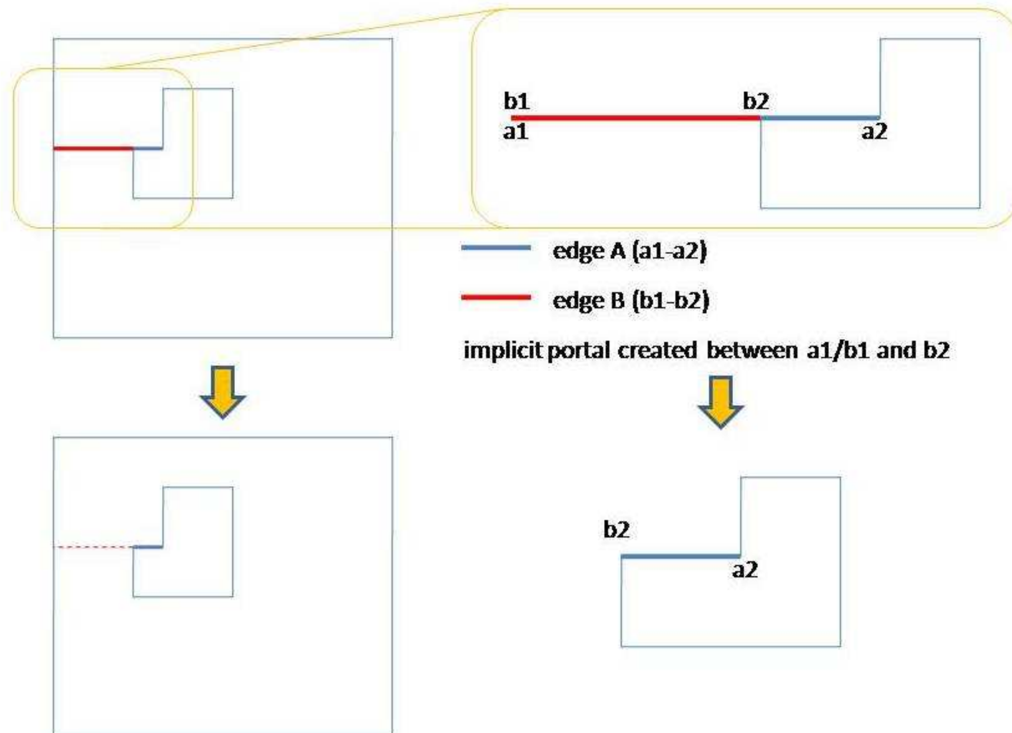


Figure 3-7: Illustration of intersection of coincident contour edges.

3.5 Generating 3D Geometry

This section describes how walls are extruded and covers cases of walls with no portals, walls with implicit portals, walls with explicit portals, and walls that have both implicit and explicit portals.

3.5.1 Generating walls with no portals

The ordering of 3D wall vertices that define the wall surface relies on the proper orientation of the 2D vertices of the space. All walls are drawn by (1) generating two extra vertices above the edge, one for each endpoint of the edge, (2) adding these two vertices, as well as two original floor vertices to the `vertices3D` list, and (3) creating a Wall object that would contain indices of these four vertices (2 new, 2 original) in the proper order. This order is determined by the direction of the bottom edge of the wall (clockwise or counterclockwise relative to the space contour). As illustrated

in Figure 3-2, the bottom edge of the wall is always pointed in the opposite direction than that of the adjacent floor edge. This satisfies the manifold property. Figure 3-8 illustrates how walls are generated.

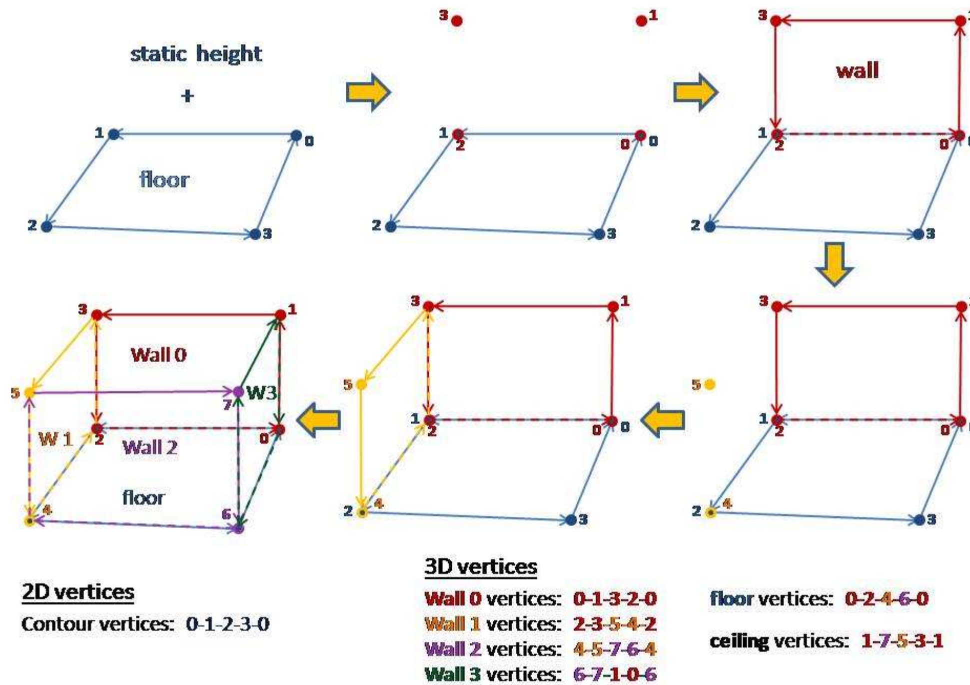


Figure 3-8: Wall extrusion and 3D vertex generation.

For each edge, two 3D vertices are generated, one from 2D vertex of the 2nd endpoint of the the edge (such as vertex 4 of wall 1 from 2D floor vertex 2 in figure 3-8), and another at the same 2D location, but with a z-coordinate set to that of the fixed floor height defined before the 3D geometry generation. If the 2D vertex is at (5,5,0), the two new 3D vertices would be (5,5,0) and (5,5,15), assuming that the floor height is 15. Then the 3D Geometry generator lists these new 3D vertices, along with the previous two vertices in the appropriate order, based on the order of floor vertices. There is an exception in drawing first and last walls for the space. In generating the first wall, four 3D vertices are generated, instead of the usual two. The other two vertices have indices 0 and 1. In generating the last wall, no new 3D vertices are generated, rather, the last two vertices that define that wall have indices

0 and 1 (back at the start, and the loop is therefore complete).

The height of the wall is not specified on the initial floor plan, or in the DXF input, and hence, not available in the XML input. For the purposes of this project, the wall height was set to a fixed number (the default is 15 pixels in the geospatial space), for every floor, every building, before the 3D Geometry Generator kicks in.

3.5.2 Generating explicit portals

There are two cases for generating explicit portals: (1) when there is only one explicit portal on an edge, and (2) when there are multiple explicit portals on an edge. These two cases are treated separately. For the first case, when there is only one portal in the wall, the wall is broken into three quadrilaterals (faces). Each face is treated as a wall object. The three faces are (1) right-of-portal (RoP) face, (2) above-of-portal (AoP) face, and (3) left-of-portal (LoP) face, as seen in Figure 3-9.

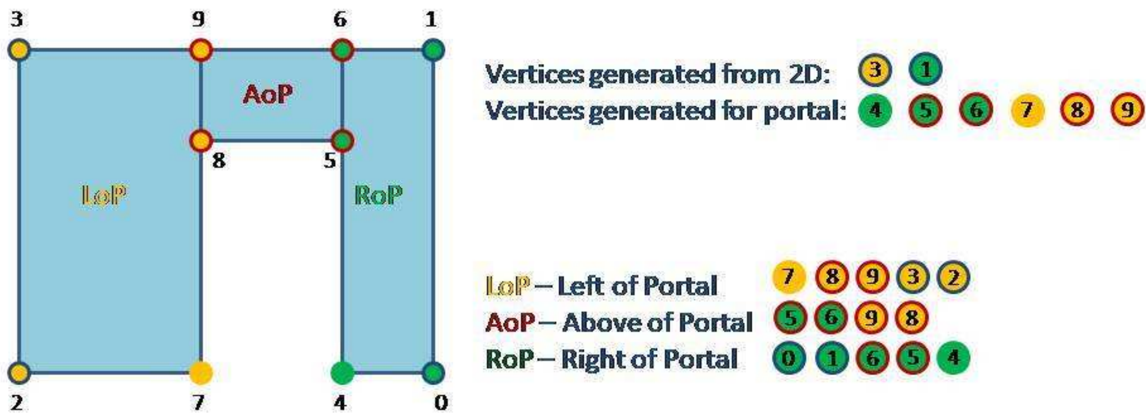


Figure 3-9: Illustration of break-up of the wall into faces surrounding the explicit portal.

In Figure 3-9 the indices are listed in the order they are generated. (1) First, the wall indices are generated as if there is no portal in the wall [indices 0,1,2,3]. (2) Portal vertices are generated based on edge, portal center, and portal height (specified at the start of 3D Geometry Generation. See Appendix B.4 on where the portal height is defined). (3) Then the wall is broken up into three faces, and each face is drawn

separately. The Manifold Property still holds because the two adjacent faces are both defined by vertices in the counterclockwise order and the adjacent edges are pointed in the opposite direction as in Figure 3-10 below.

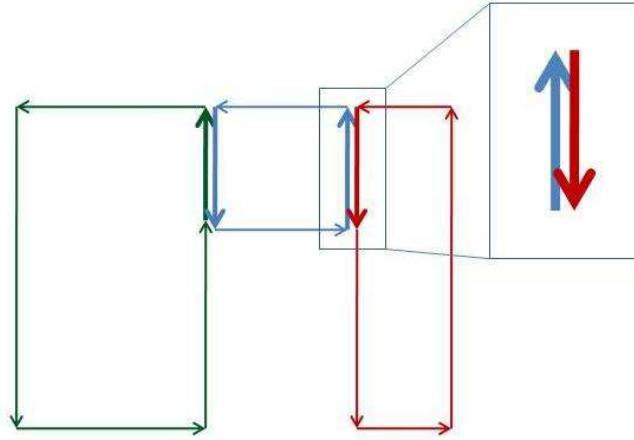


Figure 3-10: Adjacent semi-edges are directed in the opposite direction to form a complete shared edge.

The Portal Vertices Generator follows the algorithm outlined in Appendix A.1. Portal width is determined by a fixed portalWidth parameter, and any explicit portal should have a width equal to portalWidth unless it overlaps another explicit portal, or is partially outside the 2D contour edge. The coordinates for portal vertices are computed based on the center of the explicit portal, and half of portal width on either side of the center. There are only two locations of all portal vertices in the 2D contour space (for each portal), and three 3D vertices at each of these two locations. For each location, one vertex is on the floor level, another at the portal height distance along the z-axis away from the floor, and third one at the ceiling level. See Figure 3-11.

Sometimes, portals with default width may extend beyond the endpoint of the 2D edge; in this case, the portal is clipped and narrowed up to the endpoint of the 2D edge. This is checked for by Portal Vertices Generator.

According to Figure 3-9, the faces surrounding the portal are sometimes defined by more than four vertices. If RoP (Right of Portal) face were to be defined by four corner vertices instead (0-1-6-4-0), as opposed to five vertices (0-1-7-6-5-0), then the semi-edge that AoP (Above of Portal) face 'shares' with RoP face would not be

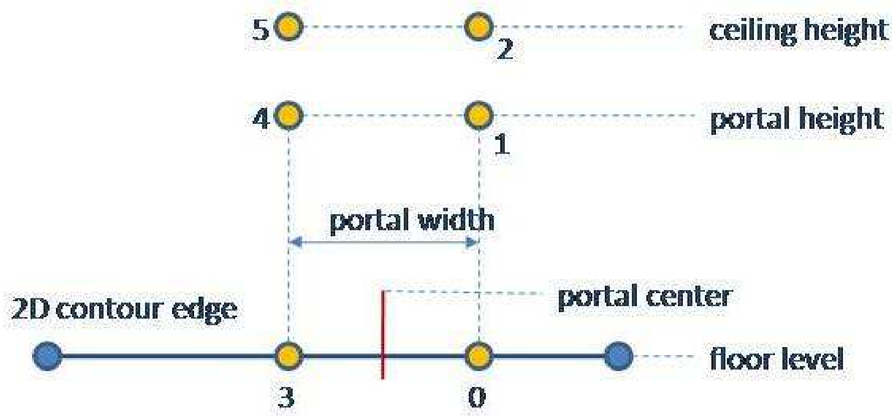


Figure 3-11: Nooks and crannies of portal vertex generation.

considered a complete shared edge. A complete shared edge is one that has two semi-edges that share the whole segment of the edge, and the two endpoints. Therefore, RoP and LoP (Left of Portal) faces are defined by 5 vertices instead of four.

There are some exceptions to the above guideline – when a space contour edge contains more than one portal. In this case, faces are drawn differently, especially faces between any two adjacent portals. Such faces are defined by six 3D vertices, three vertices for each portal on each side of the face. The algorithm for generating walls with portals in them is described in the appendix A.3. Figure 3-12 shows how a wall is broken down in case of multiple explicit portals on an edge.

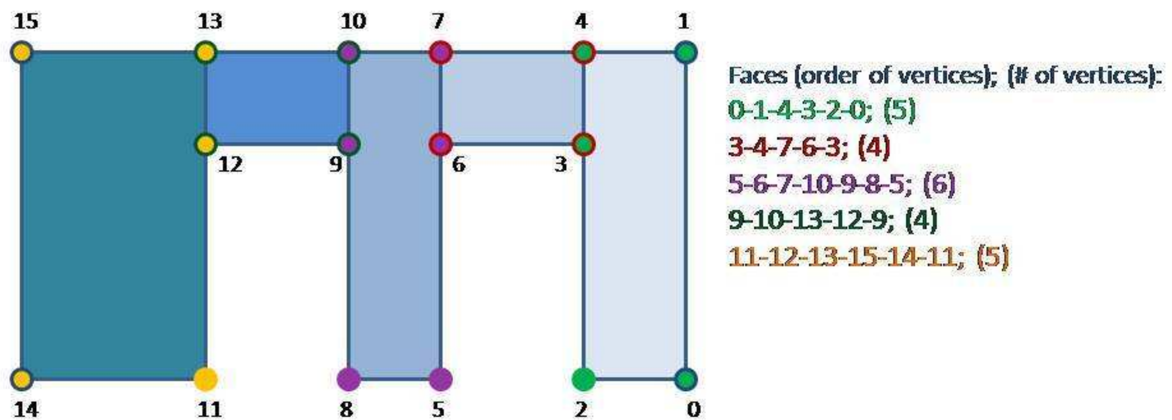


Figure 3-12: Generation of faces for multiple explicit portals for an edge.

3.5.3 Generating implicit portals

When there is an implicit portal in the wall, there are two possibilities: (1) The implicit portal covers the entire wall (portal with `minParam=0.0` and `maxParam=1.0`, or (2) the implicit portal covers part of the wall. In the first case, no wall is generated for a 2D edge, and in 2nd case, only LoP (Left of Portal) face, RoP (Right of Portal) face or both faces are generated.

Implicit portals are different from explicit portals in the way that they do not look like physical portals. Instead, they are simply continuations of one space into another. There is no doorway visible at implicit portal. This is why no AoP (Above of Portal) face is generated for implicit portals. Figure 3-13 gives a graphical illustration of the implicit portal generation.

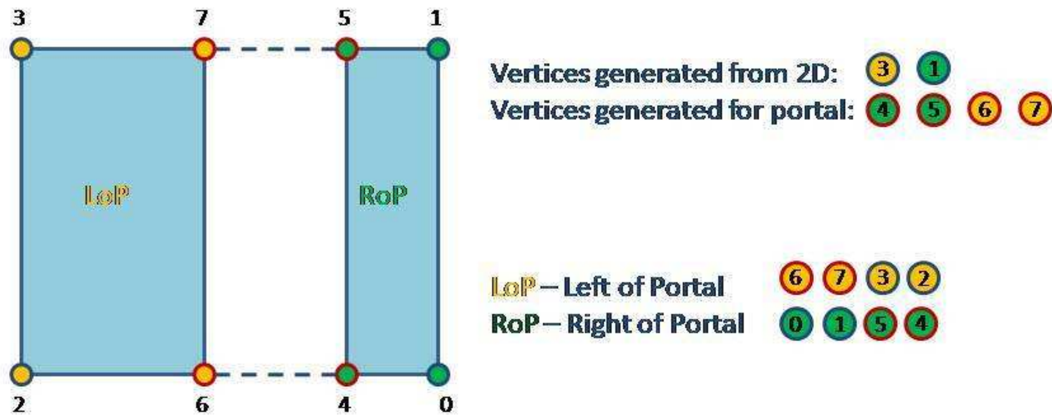


Figure 3-13: Illustration of break-up of the wall into faces surrounding the implicit portal.

It is very possible that either `minParam` or `maxParam` of the implicit portal are equal to 0.0 or 1.0, respectively. In this case, caution is exercised in detecting such boundaries. If `minParam` is set to 0.0, then no vertices 4-5 are generated and no RoP face is generated (compare Figures 3-13 and 3-14). Likewise, no vertices 5-6 and no LoP face are generated if `maxParam` of the implicit portal is at 1.0.

For implicit portals, the same Portal Vertices Generator was used as for explicit portal generation, except, in the implicit portal case, only four out of 6 vertices are

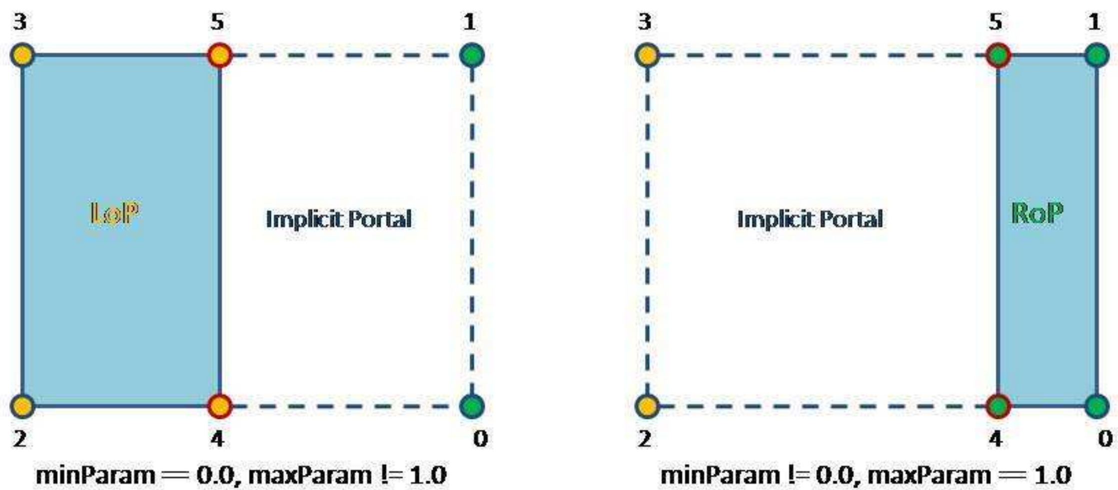


Figure 3-14: Illustration of different positions of implicit portal in the wall.

used. If Portal Vertices Generator generates 6 new 3D vertices, for the implicit portal that is not anywhere near the endpoints of the 2D contour edge, only vertices 0,2,3,5 are used, and vertices 1 and 4 (middle vertices) are ignored (for reference, look at Figure 3-11).

For 2D contour edges that have both implicit and explicit portals, the algorithm is more complicated. It is described more in detail in Appendix A.3. Some of the faces may have 4, 5, or 6 vertices. In such case, it is appropriate to remember the previous portal type and generate the face based on the current portal type and the previous portal type. See Figure 3-15.

3.5.4 Generating floors and ceilings

Floor and ceilings in absence of vertical portals are very easy to generate. Each space has one face that is considered a floor, and one face that is considered a ceiling. Whenever portal vertices are generated, their indices have to be added to the list of vertex indices that define the floor or the ceiling for the space. This way, anywhere where a face of the wall is adjacent to the floor or the ceiling, they share a complete edge. The scenario on the right in Figure 3-16 is desirable, while the scenario on the left is not. The scenario on the left breaks the manifold property because the red

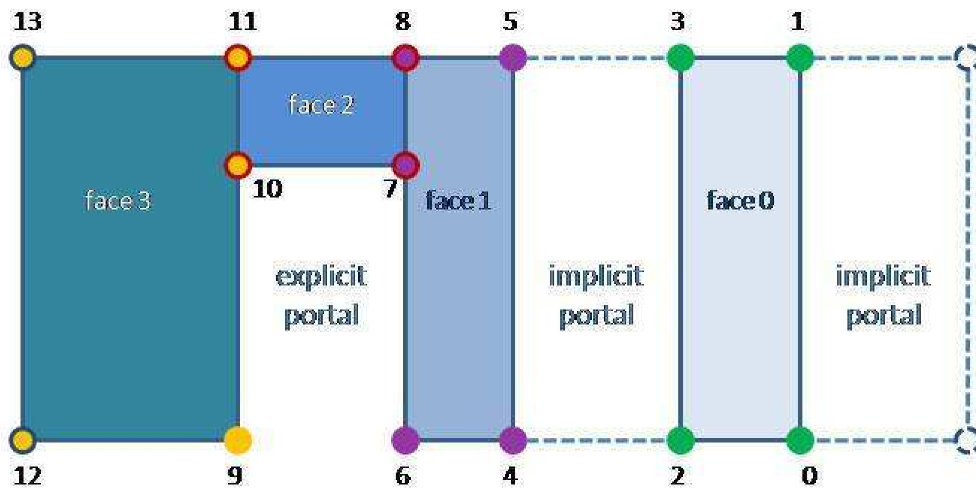


Figure 3-15: Wall with portals of different types (implicit portals mixed with explicit portals.)

semi-edge of the wall shares only portion of the blue semi-edge of the floor.

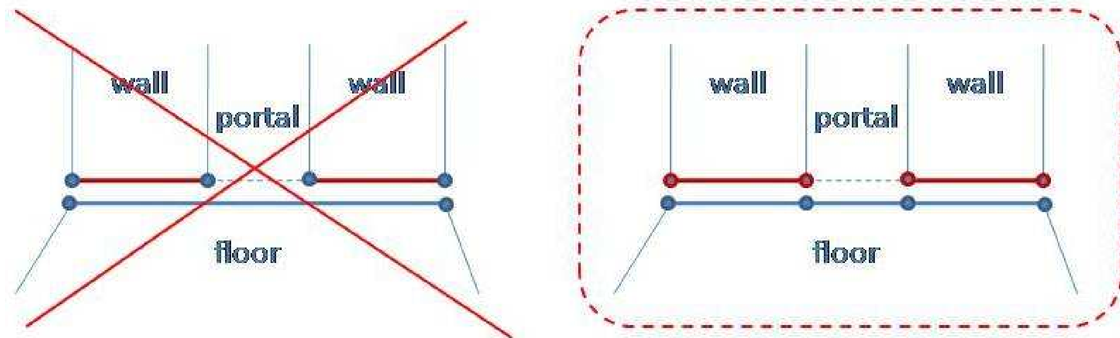


Figure 3-16: Shared edges between floor and the vertical wall: bad and good scenarios.)

Floor vertices are always listed in the counterclockwise order, so the surface generated for the floor is visible from inside the space by looking down. For ceiling, the vertices are listed in the counterclockwise order, so the ceiling surface is visible from inside the space by looking up.

3.5.5 Watertight Property in Horizontal Portals

In order for the finished model to be watertight, all spaces have to be watertight and have to be connected to each other via sealed portals. The watertight property of spaces is ensured by the manifold property (every semi-edge of any surface has its opposite semi-edge of another surface. These two semi-edges form a complete edge, sealing the two adjacent surfaces to each other. For portals to be sealed, however, more surfaces have to be generated that would wrap around the portal and fill out the region between spaces at the portal. This is analogous to the doorframe – every door has a doorframe, so it is not possible to peek into the area between the opposite walls.

There is a challenge, however. All vertices are stored locally within the space. If surfaces have to be generated between spaces, the question arises – where should these new surfaces be stored? Which space should they belong to? It was decided to place such surfaces into one of the spaces. Which space stores these surfaces depends on whether the source flag inside the portal is set to true or false. If the portal object has a source flag that is set to True, the space to which this portal object belongs to claims the ownership of the surfaces connecting this space to its neighbor. That is, for two portals, both belonging to adjacent spaces, and representing one physical portal, only one of these two portals has the source flag set to True. The other portal must have the source flag set to false.

Another issue arises from the fact that since 3D vertices are stored locally within each space, the vertices that connect the portal surfaces with the space that they do not belong to, cannot be shared with vertices of that space. In Figure 2-7, vertices $A3$ and $B6$ belong to two separate spaces, despite the fact that they have the same geospatial location. This is the only case when redundancy is allowed, due to the design of 3DGen. Vertices $B6$ and $B7$ are needed for portal surfaces generation, while vertices $A3$ and $A2$ are needed to ensure that the manifold property is satisfied. The two semi-edges formed by $B6 - B7$ and by $A3 - A2$ create a complete edge.

The portal surfaces are generated after all space surfaces have been generated in all

spaces in the floor because, in order for the portal surface generation to work, portal vertices should exist in both spaces. Then, every portal in every space is examined and additional inter-space faces are generated. For explicit portals, four faces are generated, one for each side of the "doorframe". For implicit portals, no faces are generated since it is assumed that All implicit portals have no gaps between spaces. For every explicit portal, portal vertices are obtained from each of the two semi-portals of adjacent spaces, and faces are generated based on these vertices. The algorithm is written out in more detail in Appendix A.5.

3.6 Multiple floors

All material above applies to 3D geometry generation for a single floor. The whole model is not complete until after all floors are bound together to create the whole building.

3.6.1 Stacking floors on top of each other

The input XML file contains data about a single floor only, so, in order to get data about all the floors (entire building), multiple XML files need to be parsed. Originally 3DGen was intended to be applied on single floors; and the best way to extend it to multiple floors is to run the 3D Geometry Generator on each floor separately. The command line input of 3DGen was altered to allow inputting multiple filenames as arguments. For each floor XML file, a special internal floor ID is passed to the 3D Geometry Generator to specify the floor of the building. The ground level of every floor is computed based on this floor ID. The floor level, the portal height level and the ceiling level are all determined by this floor ID. Suppose, 3DGen is run with the following files: 10-1.xml, 10-2.xml, 10-3.xml, in that order, each signifying the first three floors of MIT Building 10. XML file 10-1.xml gets a floor ID of 1, since it is the first file in the list. 10-2.xml gets floor ID of 2, and so on. Floor ID determines the reference ground for each floor from which to generate surfaces. In our case, the first floor model would lie on the z=0 plane. The floor model of floors above would lie at

$z = floorID * (floorHeight + 0.4)$ plane. This means that the third floor would lie at $z = 3 * (floorHeight + 0.4)$ plane.

3.6.2 Completing manifold in Vertical Portals

Floors are connected to each other via vertical portals. These are stairs, elevator shafts, and open multi-floor spaces. Whenever a space has a vertical portal, there is an opening either in the floor or in the ceiling, depending on whether the vertical portal has UP or DOWN direction. There can be only one portal in the UP direction for each space. Likewise, there can be only one portal in the DOWN direction for each space.

Floor models are not placed right on top of each other – rather they are separated by 0.4 units (in geospatial coordinates). This introduces gaps between the floors at vertical portals. Stairs, elevator shafts, or multi-floor spaces all have these gaps that have to be patched if the entire building model is to remain watertight. One idea to connect floors included removing the whole floor from each space with the "DOWN" vertical portal and removing the whole ceiling from each space with the "UP" vertical portal. However, this does not work in practice, since the two spaces on different floors that are connected to each other via the vertical portal may not have the same space contour. This means that it is very possible for the above space to have fewer or more floor vertices than the ceiling vertices in the space below. This complicates the process of connecting the two spaces on top of each other. This is why it is necessary to create rectangular openings in the floor instead of dropping the whole floor. This would ensure that both the opening in the floor of the above space and the opening in the ceiling of the below space have the same number of portal vertices (4 in case of a rectangular opening).

For vertical portals, portal surfaces are stored in the space of the floor that is on the top of the portal (the space with the "DOWN" portal). The reason for that is that both spaces need to be generated first, before they are connected by the portal surfaces. The floor model generation happens from botom up, so the bottom space needs to be generated first, followed by the top space, and then by the vertical portal

surface generation.

The algorithm for generating vertical portal surfaces is more complex than the one for generating horizontal portal surfaces and is outlined in Appendix A.6

3.7 Limitations

There are several limitations to the 3D building model generation. One is 3DGen is heavily reliant on the input XML. If this input XML is incorrect, then the resulting model will also be incorrect. Currently there is no checking for cases where implicit portals overlap with explicit portals or other implicit portals, since these cases are rare and would require more code. Only the overlap of explicit portals is checked for. Another limitation is that all implicit portals have to be accounted for in the floor plans. If there is no implicit portal where there should be one, an implicit surface could spring up and result in a flickering model. Implicit portals occur where the contour edges from the adjacent spaces are coincident with each other, as opposed to the edges from the adjacent spaces being not coincident but parallel and very close to each other. Implicit surfaces are detected only inside the space realm, and not in the inter-space realm, since that is the responsibility of the DXF to XML converter.

Also, windows are not present in the end building model because the input XML does not contain any data pertaining to the windows. The generation of the windows should be parallel to the generation of portals and will be covered in the Future Work section below.

Chapter 4

Conclusions

This thesis culminates a 2-year long project in 3D Building Model Generation from 2D floor plans, where the input is the XML file. There was a similar effort prior to my arrival to the group, by Sean Markan [4]. He generated 3D models right from AutoCAD, while this project generates 3D models from XML data, after some pre-processing. The advantage of having XML as the middleman between the 3DGen and the DXF Parser lies in the fact that additional preprocessing can be done on XML data, such as creation of implicit portals and the detection of inconsistencies in the AutoCAD floor plans.

The major goal of this project is not to just generate 3D building models from floor plans, but also to do so as efficiently as possible. This means generating the fewer number of 3D vertices as possible and sharing as many of these vertices and edges as possible. 3DGen creates almost no redundant vertices (the only exception is where portal surfaces meet one of the spaces). This is by no means an easy endeavor since all shared vertices need to be tracked continuously. This is accomplished by storing the 3D vertex data locally in each space and generating faces/walls for each space independently. During the generation of vertices and surfaces, 3DGen carefully makes sure that the manifold property is satisfied (every edge should have two co-incident semi-edges pointing in the opposite directions) and that everything in the building model is watertight. The Manifold Property helps maintain the watertight nature of the building models.

Chapter 5

Future Work

All work up to now concentrated on generating geometric models of whole buildings; of the floors stacked up on each other. The end building models look monotonous in color and texture. Although there are openings between rooms at portals, the rooms still feel like jail cells, mainly because of the absence of the windows. Adding windows should be fairly straightforward and similar to the algorithm for generating portals (in Appendix A.3). In addition to AoP (Above of Portal face, a BoP (Below of Portal) face could be generated to form the window (assuming that windows have similar attributes as portals, such as the default width, height and center parameter).

Another good feature to add to the geometric model is the texture for stairs and ramps. Currently stairwells are openings in the floor and ceilings (vertical portals), and there are no steps generated. An observer would not be able to detemine whether a room with vertical portals is an elevator shaft or a stairwell (although most elevator shafts are square). Adding steps would make the model more aesthetically accurrate.

Lastly, 3D models that have walls of the same color are relatively boring to view. It would be nice to add texture to the walls, perhaps photographic images of the same walls in the real life, and slap them into the virtual walls. These pictures would have to be warped to fit the walls. They would have to be re-warped every time the viewer changes his/her viewpoint. Warping itself is a very expensive operation and complicated, hence would be best implemented as part of a separate major project. There is a similar project in the area of mapping real-life images to the 3-dimensional

models. This project is called PhotoSynth and it is spearheaded by Microsoft Labs. [3], [8]

The original goal for this project was to create a virtual tour web service that would let anyone tour buildings remotely. However, due to the time constraints and many bugs, this ambitious idea had to be scaled down to just the 3D model generation. This project could be extended in the future, along with real-life images, to create 3-dimensional virtual tour guides that would be generated from 2D floor plans automatically, instead of being sketched by hand by special designers.

Appendix A

Algorithms

A.1 Algorithm: generation of explicit portal vertices

Every floor contains an unique floor ID. This floor ID is used in determining the ground level, the portal and ceiling levels along the z-coordinate. For every portal, six 3D vertices are generated, 3 for each vertical edge of the portal. Each vertex in such triplets has same x and y coordinates in 2D plane, but a different z-coordinate. Figure 3-11 shows this in greater detail.

For every floor static `portalHeight`, `height`, and `portalWidth` are set to their default values (in our case it was 10, 15, and 3, respectively.

In order to generate x, and y coordinates for the portal vertices, two vertices of edge the portal is on are obtained, along with the portal's center parameter.

$$delta = \sqrt{(x2 - x1)^2 + (y2 - y1)^2} \tag{A.1}$$

This is the total length of the edge on which the portal is on. Then the angle of the edge is computed.

$$angle = \arctan\left(\frac{y1 - y2}{x2 - x1}\right) \tag{A.2}$$

Then $delta_{p1}$ and $delta_{p2}$ are computed – these are the parametric distances from

the first endpoint of the edge to the first and second portal points, respectively.

$$delta_{p1} = delta * center - \frac{portalWidth}{2} \quad (A.3)$$

$$delta_{p2} = delta * center + \frac{portalWidth}{2} \quad (A.4)$$

The following conditions ensure that the portal does not extend beyond the edge:

`If` $delta_{p1}$ `< 0, set` $delta_{p1}$ `= 0`

`If` $delta_{p2}$ `> delta, set` $delta_{p2}$ `= delta`

$x_1 = delta_{p1} * (\cos angle)$

$y_1 = delta_{p1} * (\sin angle)$

$x_2 = delta_{p2} * (\cos angle)$

$y_2 = delta_{p2} * (\sin angle)$

Then we have to make sure that if deltas are negative, we have to subtract instead of adding.

if $(x2 - x1) < 0$ **then**
$x_1 \leftarrow x_1 * -1$
$x_2 \leftarrow x_2 * -1$
end if
if $(y2 - y1) < 0$ **then**
$y_1 \leftarrow y_1 * -1$
$y_2 \leftarrow y_2 * -1$
end if

Then we compute z coordinates for all three levels (floor, portal, and ceiling).

if $floorID == 0$ **then**
$z_1 \leftarrow 0.000$
$z_2 \leftarrow portalHeight$ {portalHeight is a static variable defined in genUtils header file}

$z_3 \leftarrow height$ {height is the ceiling level, also defined in a static variable as above}

else

$z_1 \leftarrow (height + gapBetweenFloors) * floorID$

$z_2 \leftarrow (height + gapBetweenFloors) * floorID + portalHeight$

$z_3 \leftarrow (height + gapBetweenFloors) * floorID + height$

end if

Finally, 6 vertices are generated and returned as portal vertices, as in figure A-1 below. (x_1, y_1, z_1), (x_1, y_1, z_2), (x_1, y_1, z_3), (x_2, y_2, z_1), (x_2, y_2, z_2), (x_2, y_2, z_3),

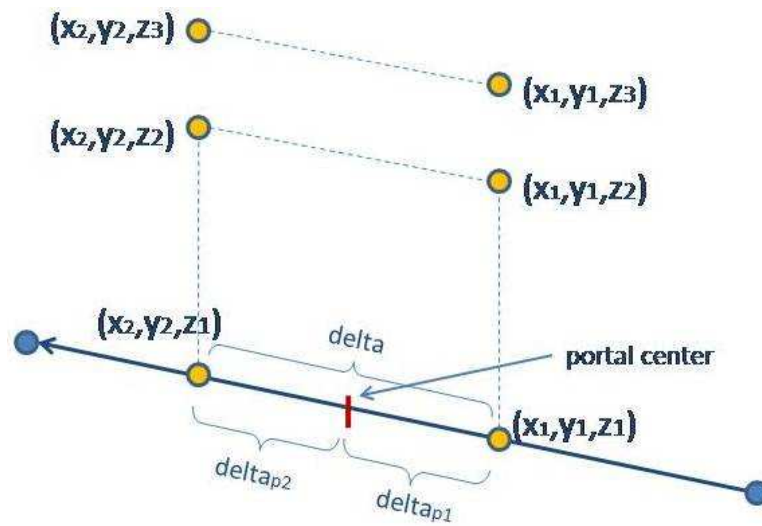


Figure A-1: Location of Portal Vertices.

A.2 Algorithm: detection of overlapping portals and correction

In order to detect whether portals overlap, portals have to be ordered for each edge, with param as the key. This way it is possible to look at adjacent portals on an edge, instead of looking for any portals that may be adjacent. The algorithm is as follows:

1. Create a map data structure that maps edge index to the vector (list) of portals: `map<edgeIndex, vector<Portal*>>`

2. For every edge e, get a list of portals, and populate the vector of portals. Put this new vector into the map at key e.

3. For every edge, sort the vector of portals (insertion sort was used for this purpose). If a portal is an implicit portal, take the average of its `minParam` and `maxParam` as its center param.

4. **for** every edge e **do**

   for every ith portal in the vector for that edge e **do**

   Obtain $v1$ and $v2$ - endpoint vertices for the edge e, and compute distance $dist$ between them.

   Focus on portals at i and $i+1$. If i is the last portal in the list, then wrap around to the first portal, and focus on ith portal and portal at index 0.

   Make sure that both portals are explicit portals.

   if $(\frac{portalWidth}{dist} + (\text{param of portal } i)) \geq (\text{param of portal } i+1)$ **then**

   $newCenter \leftarrow \frac{(\text{param of portal i})+(\text{param of portal i+1})}{2}$

   Compute the actual distance between centers of two portals

   $m \leftarrow (\text{param of portal i+1 } - \text{param of portal i }) * dist$

   {See Figure A-2 below}

   if m == portalWidth **then**

{Two portals are directly adjacent to each other.}

$newWidth \leftarrow 2 * portalWidth$

else if m < portalWidth **then**

{Two portals are overlapping}

$newWidth \leftarrow portalWidth + m$

else

{Two portals are not overlapping but are very close to each other}

$newWidth \leftarrow portalWidth + m$

end if

modify portal i with values of $newWidth$ and $newCenter$.

Remove portal $i + 1$.

end if

end for

end for

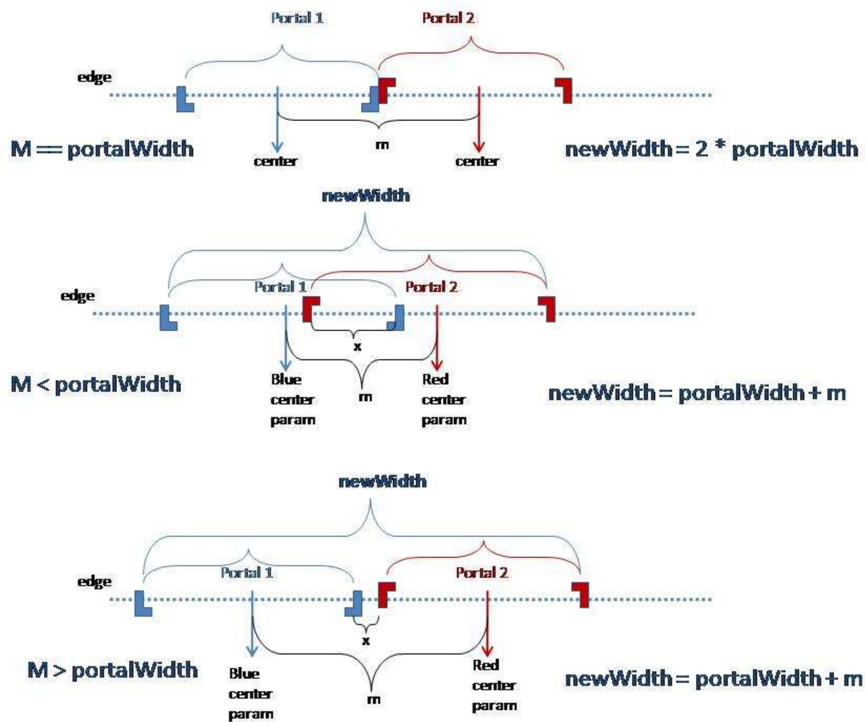


Figure A-2: Detecting overlapping portals.

newWidth is computed for each of the new portals based on the following equations (use Figure A-2 for reference):

for $m < portalWidth$

$$m = \frac{1}{2}p + \frac{1}{2}p - x \tag{A.5}$$

$$x = p - m \tag{A.6}$$

$$newWidth = p + p - x = 2p - (p - m) = p + m \tag{A.7}$$

$$\tag{A.8}$$

for $m > portalWidth$

$$newWidth = \frac{1}{2}p + \frac{1}{2}p + m = p + m \tag{A.9}$$

A.3 Algorithm: generation of faces with multiple portals in them

The algorithm for 2D contour edges that contain multiple portals involves a lot of conditional statements. There are three main cases: (1) First portal on edge; (2) the portal in the middle, flanked by other portals on both sides; and (3) the last portal on edge. All three cases will be covered below separately. The indices used are derived from the Figure A-3 below. Each case has its own subfigure.

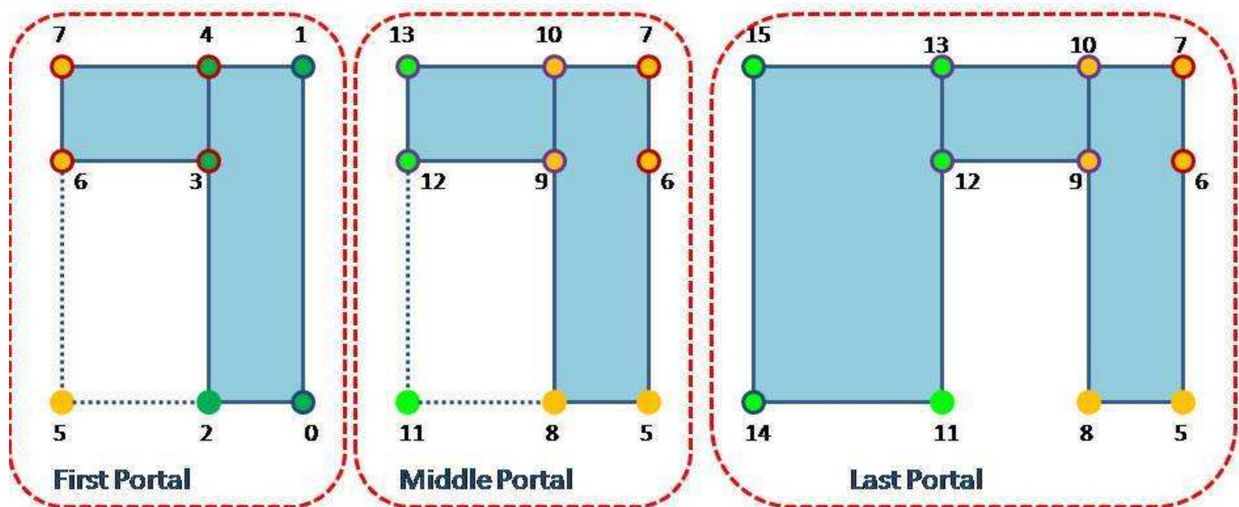


Figure A-3: Three cases of drawing portals.

```
CASE 1:  First Portal
```

if portal is explicit **then**

draw 5 vertices for face Left of Portal: (0,1,4,3,2,0)

draw 4 vertices for face Above of Portal: (3,4,7,6,3)

else if portal is implicit **then**

if minParam == 0 **then**

do not draw face Left of Portal

do not draw face Above of Portal

else if minParam != 0 **then**

draw 4 vertices for face Left of Portal: (0,1,4,3,0)

do not draw face Above of Portal

end if

end if

```
CASE 2:  Mid Portal
```

if portal is explicit **then**

if previous portal is explicit **then**

draw 6 vertices for face Left of Portal: (5,6,7,10,9,8,5)

draw 4 vertices for face Above of Portal: (9,10,13,12,9)

else if previous portal is implicit **then**

draw 5 vertices for face Left of Portal: (5,7,10,9,8,5)

draw 4 vertices for face Above of Portal: (9,10,13,12,9)

end if

else if portal is implicit **then**

if previous portal is explicit **then**

draw 5 vertices for face Left of Portal: (5,6,7,10,8,5)

do not draw face Above of Portal

else if previous portal is implicit **then**

draw 4 vertices for face Left of Portal: (5,7,10,8,5)

do not draw face Above of Portal

end if

end if

```
CASE 3:  Last Portal
```

if portal is explicit **then**

if previous portal is explicit **then**

draw 6 vertices for face Left of Portal: (5,6,7,10,9,8,5)

draw 4 vertices for face Above of Portal: (9,10,13,12,9)

draw 5 vertices for face Right of Portal: (11,12,13,15,14,131)

else if previous portal is implicit **then**

draw 5 vertices for face Left of Portal: (5,7,10,9,8,5)

draw 4 vertices for face Above of Portal: (9,10,13,12,9)

draw 5 vertices for face Right of Portal: (11,12,13,15,14,11)

end if

else if portal is implicit **then**

if previous portal is explicit **then**

draw 5 vertices for face Left of Portal: (5,6,7,10,8,5)

do not draw face Above of Portal

if maxParam == 1.0 **then**

do not draw face Right of Portal

else if maxparam != 1.0 **then**

draw 4 vertices for face Right of Portal: (11,13,15,14,11)

end if

else if previous portal is implicit **then**

draw 4 vertices for face Left of Portal: (5,7,10,8,5)

do not draw face Above of Portal

if maxParam == 1.0 **then**

do not draw face for Right of Portal

else if maxparam != 1.0 **then**

draw 4 vertices for face Right of Portal: (11,13,15,14,11)

end if

end if

end if

A.4 Algorithm: detecting implicit surfaces

There are two steps to detecting implicit surfaces: (1) detect collinear edges, and (2) determine whether these collinear edges are coincident.

Assume we have 2 segments, A and B. Each segment has two endpoints, (x_1, y_1) and (x_2, y_2). Any point along the segment can be expressed in parametric terms. There are two equations, one for each of x and y coordinate of any point that lies along the segment.

$$x = x_1 - (x_1 - x_2)t_x \tag{A.10}$$

$$y = y_1 - (y_1 - y_2)t_y \tag{A.11}$$

Note that x, and y have to be computed for a specific segment, either segment A or segment B. Any single point that lies on the segment should satisfy the following equation: $t_x = t_y$. If this is not the case, then this point does not lie on the line.

We then rewrite equations A.10 and A.11 to get t_x and t_y.

$$t_x = \frac{x - x_1}{x_2 - x_1} \tag{A.12}$$

$$t_y = \frac{y - y_1}{y_2 - y_1} \tag{A.13}$$

Any 2D points can be plugged in place of x and y in the equations A.12 and A.13.

Now, to determine if the two segments, A and B are collinear, we need to put coordinates of endpoints of segment B into the segment A's parametric equations in place of x and y. Let's assume that segment A contains endpoints a_1 and a_2, and segment B contains endpoints b_1 and b_2. We need to determine whether both b_1 and b_2 lie on the line that goes through segment A (segment A is defined by endpoints a_1 and a_2).

For every two 2D contour edges (not just two adjacent contour edges) inside the space, four parameters are computed (t_{a1}, t_{a2}, t_{b1}, t_{b2}). Assume any two contour edges in the space, A, and B. t_{a1} and t_{a2} are parameters of the endpoints of edge A relative

to endpoints of edge B. t_{b1} and t_{b2} are parameters of the endpoints of edge B relative to the endpoints of edge A.

The following notation: $t_{x,b1,A}$ means t_x where $x = x_{b1}$ (x-coordinate of endpoint b_1) on segment A.

To see if segment B is collinear with segment A, we need the following two conditions to hold:

$$t_{x,b1,A} = t_{y,b1,A} \tag{A.14}$$

$$t_{x,b2,A} = t_{y,b2,A} \tag{A.15}$$

Actually, we do not need to compute $t_{x,b1,A}$, $t_{y,b1,A}$, $t_{x,b2,A}$, or $t_{y,b2,A}$. Combining equations A.12, A.13, A.14 and A.15 and eliminating all the t's, we get the following two new conditions to watch for:

$$(x_{b1} - x_{a1})(y_{a2} - y_{a1}) = (y_{b1} - y_{a1})(x_{a2} - x_{a1}) \tag{A.16}$$

$$(x_{b2} - x_{a1})(y_{a2} - y_{a1}) = (y_{b2} - y_{a1})(x_{a2} - x_{a1}) \tag{A.17}$$

Segments A and B are collinear only and only if both equations A.16 and A.17 hold true.

After two segments are deemed to be collinear, we can proceed with the next step – which is determining whether these two segments are coincident. In order to do that, t's have to be computed. Assume that we want to see if segment B lies on or overlaps with segment A. In this case, we would want to compute $t_{x,b1,A}$, $t_{y,b1,A}$, $t_{x,b2,A}$, and $t_{y,b2,A}$. For now, we can assume that we can pick either t_x or t_y, but which one will be important in a special case, when segments are horizontal or vertical. This case will be explained a little bit later. If we consider the scenario above (finding the location of points b_1 and b_2), then we can (for now) focus on $t_{x,b1,A}$ and $t_{x,b2,A}$, the t's based on x-coordinate.

The conditions below are illustrated in Figure A-4.

Now we will explore all four cases in greater detail.

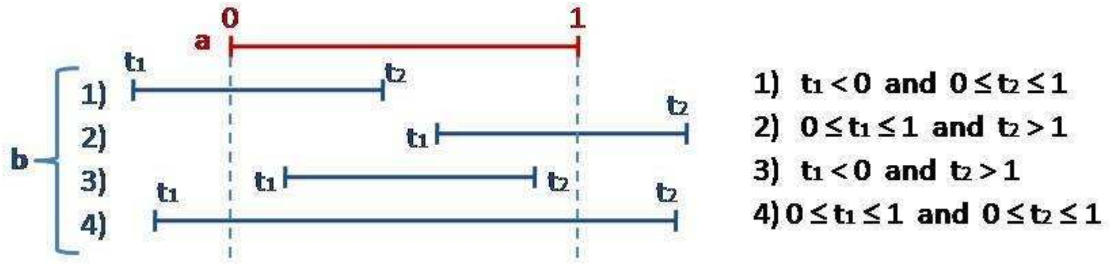


Figure A-4: Four cases for overlapping coincident edges.

1. $t_1 < 0$ and $0 \leq t_2 \leq 1$:

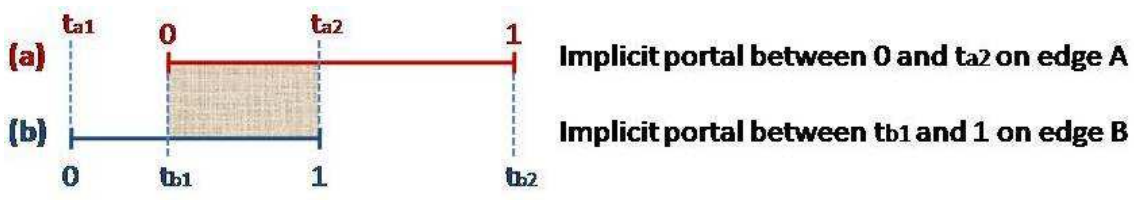


Figure A-5: Coincident Edges: Case 1.

2. $0 \leq t_1 \leq 1$ and $t_2 > 1$:

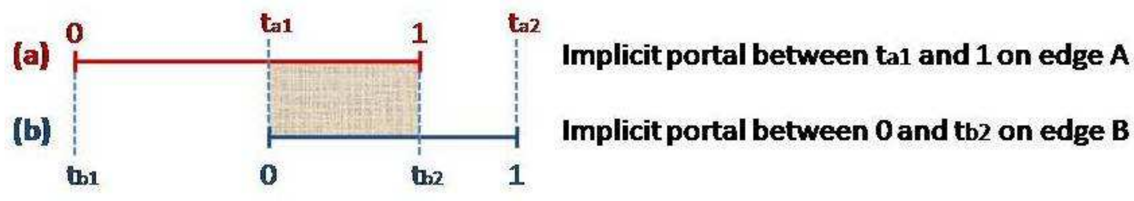


Figure A-6: Coincident Edges: Case 2.

3. $t_1 < 0$ and $t_2 > 1$:

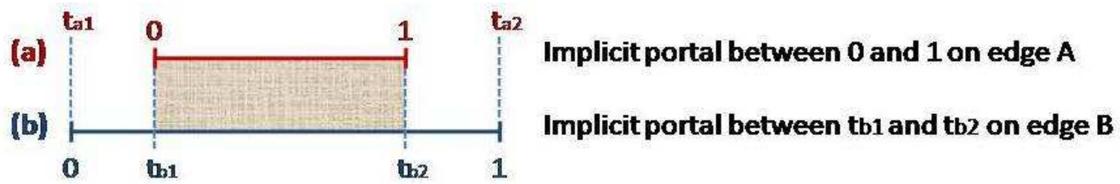


Figure A-7: Coincident Edges: Case 3.

4. $0 \leq t_1 \leq 1$ and $0 < t_2 \leq 1$:

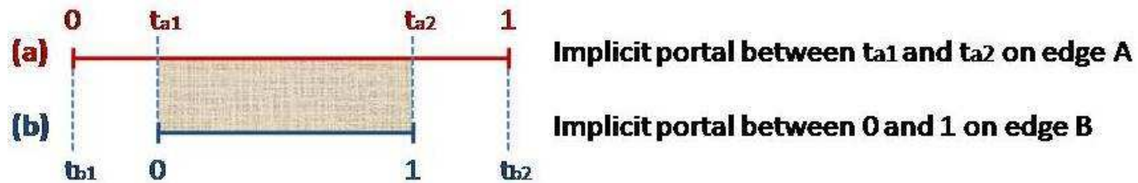


Figure A-8: Coincident Edges: Case 4.

There are two exceptions, however. It is possible that one of the two or both edges are perfectly horizontal or vertical. In this case, there would be a DivideByZero error in computing either t_x or t_y. In case of horizontal edge, t_y would have the undefined value since $y_2 - y_1 = 0$ in the denominator of equation A.13. Likewise, with t_x in case of vertical edge.

These two exceptions are handled by using t_x in all cases except vertical lines. In case of vertical lines, t_y is used in place of t_x.

A.5 Algorithm: generation of horizontal portal surfaces

- For any two adjacent spaces on the floor, look for "semi-portals" from both spaces that may connect to each other (this is deduced from the `target` attribute of the portal and the name of the space the portal belongs to). Let's call them *portalA* and *portalB*. It is quite possible that there are multiple portals connecting the two spaces, therefore, it is not a good idea to rely on the name of the space and the target of the portal to determine if the "semi-portals" connect to each other. It is also important to observe the location of the two "semi-portals" before declaring them to be part of a complete portal.

- Obtain floor-level vertices from the two "semi-portals" and compute the squared distance *dist* between them.

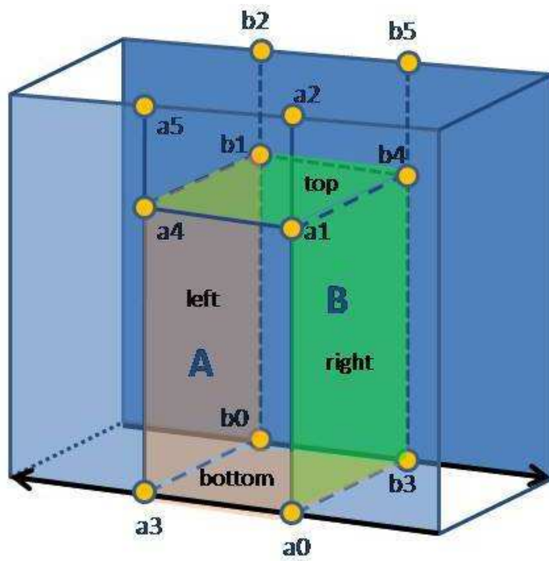
- **if** $dist < 0.49$ **then**

  {Check if the portal is implicit or explicit}

  copy all 3D portal vertices from portal that has source set to false to portal that has source set to true. Be mindful of the fact that the portal may be implicit or explicit. Copy vertices only from explicit portals (implicit portals are assumed to have no gaps) See figure A-9 for illustration.

  end if

- Create four new Wall (face) objects for portals that are explicit (left, right, top, and bottom). Implicit portals need not have portal surfaces since they are assumed to have no gaps.



Vertices a0-a5 belong to semi-portal A
Vertices b0-b5 belong to semi-portal B

Generate four portal surfaces:

Right:	a0-a1-b4-b3-a0
Top:	a1-a4-b1-b4-a1
Left:	a3-b0-b1-a4-a3
Bottom:	a0-b3-b0-a3-a0

Figure A-9: Portal surfaces for explicit horizontal portals

A.6 Algorithm: generation of vertical portal surfaces

Not yet implemented. This is work in progress and may not even be able to be finished in time for thesis deadline. I apologize for the inconvenience.

Appendix B

3dGen Manual

B.1 3dGen Package Overview

The code for 3DGen is inside the 3dGen package. Below is the list of files in that package: genUtils.h, genUtils.cpp, Space.h, Space.cpp, Portal.h, Portal.cpp, VPortal.h, VPortal.cpp, Vertex.h, Vertex.cpp, Wall.h, Wall.cpp, congif.h, tools.h, tools.cpp, XMLtoUG.cpp, and main.cc.

Code in main.cc is responsible for reading XML files, and making the appropriate calls to GenUtils (which contains almost all algorithms described in this thesis). No new XML files are created – only existing XML files are modified. This is why it is a good idea to create backup versions of the input XML files. In order to read XML files, you need to have the xerces-c library installed on the machine you are compiling the code on. Currently, the code is compiled and run on `tritium.csail.mit.edu` machine.

Code in XMLtoUG is responsible for converting XML into UG (Unigrafix format) and is saved in the output.ug file.

B.2 Where to obtain 3dGen

3dGen has been checked into the RVSN repository. To obtain the code, perform a SVN checkout as follows:

```
svn co svn+ssh://user@login.csail.mit.edu/afs/csail/group/rvsn/Repository/3dGen
```

The 3dGen package has the following structure:

```
3dGen
 |- src
 |- scripts
 |- input
 |- output
```

The `src` directory contains all the code and outside libraries necessary for the code to run (such as gpc library for geometry operations). Xerces-c library is not included since it has to be compiled for each platform separately. `tritium.csail.mit.edu` has xerces-c library installed, so it should be used to run 3dGen for now.

The `scripts` directory contains all scripts necessary to run the 3dGen as a whole package. The details of such scripts are listed below. In order to run the whole 3dGen framework on building 10, run the `start` script by typing `source start`. This script should generate the `10-all.iv` file that can be viewed by the iv viewer.

The `input` directory contains input XML files. The `output` directory contains both XML files generated by gen3d and Inventor files generated by uggen and ug2iv.

B.3 Running 3dGen

Here is how I compile and run 3dGen. The sequence of commands bellow is taken from the `start` script inside the `scripts` directory.

```
svn update   %this is needed to update any files in the SVN repository
%make sure that you are currently inside the scripts directory.
%assume that backed-up files lie in the input directory
cp ../input/10-1.xml../ output/10-1.xml
cp ../input/10-2.xml ../output/10-2.xml   % do this for as many files as you need.
...
cp ../input/10-8.xml ../output/10-8.xml

%compile 3dGen
g++ -l xerces-c -o app ../src/Portal.cpp ../src/Vertex.cpp ../src/Space.cpp ../src/Wall.cpp
../src/VPortal.cpp ../src/tools.cpp ../src/main.cpp ../src/genUtils.cpp
%compile XMLtoUG
g++ -l xerces-c -o uggen ../src/XMLtoUG.cpp ../src/tools.cpp
%run 3dGen
%to omit the ceilings in the geometric models, put the -noceil argument after gen3d
./gen3d ../output/10-1.xml ../output/10-2.xml output/10-3.xml
%run XMLtoUG and generate faces
%(to generate wire mesh, use w flag instead of f after uggen command)
./uggen f output/10-1.xml output/10-2.xml output/10-3.xml
%convert ug to iv (only works on tritium)
./ug2iv output.ug > output.iv
%fix lighting irregularities
./ivfix output.iv output/10-all.iv
```

Figure B-1: The example sequence of commands to generate the geometric model of MIT building 10

It is important to note that the files have to be listed from the lower floor to the higher floor (the order of floors does matter) in the input to both app and uggen scripts.

The iv files can be viewed by the inventor viewer, ivview.

B.4 Parameters set in the code

The three static parameters `height`, `portal_height` and `portal_width` are defined in the genUtils.h file. These define the height of the floor, the height of every explicit portal (implicit portals have height of the floor), and width of explicit portals, respectively.

B.5 Viewing geometric models in VRML

The Inventor viewer (ivview) does not provide the user with the ability to navigate the hallways of the geometric models. Another viewer is necessary in order to view geometric models. Cortona Player has been verified to work on our models.

The following steps only work on Windows machines. You should download Open Inventor Tools from the web at http://merlin.fit.vutbr.cz/ivTools/ivTools-bin-2.0.zip. The tools package contains the `ivvrml` executable file. Running it as follows will convert the output file from inventor format to VRML2 format that can later be opened by VRML viewers.

```
ivvrml -2 10-all.iv 10-all.wrl
```

The only VRML viewer that has been verified to work with out output files is the Cortona VRML viewer. It can be obtained from

http://www.parallelgraphics.com/products/cortona/. The installer for Cortona Player installs a browser plugin. There are issues with running Cortona plugin on Firefox 3, however, The Cortona Player has been verified to work in Internet Explorer 7. For now, use Internet Explorer 7 to view the geometric models.

Once the wrl file has been loaded into the Cortona player, the geometric model will not appear on the screen automatically. This is because teh building models are not drawn anywhere near the origin of the coordinate system. Clicking on the `Fit` button on the bottom right of the viewer window will focus and center the viewer on the geometric model. Please read the User Guide before attempting to navigate the building model. The User Guide can be accessed by right-clicking on the window in which the model is visible and going to Help− >User Guide in the context menu that pops up.

Appendix C

3D Geometric models

This appendix shows the result geometric models of various floors.

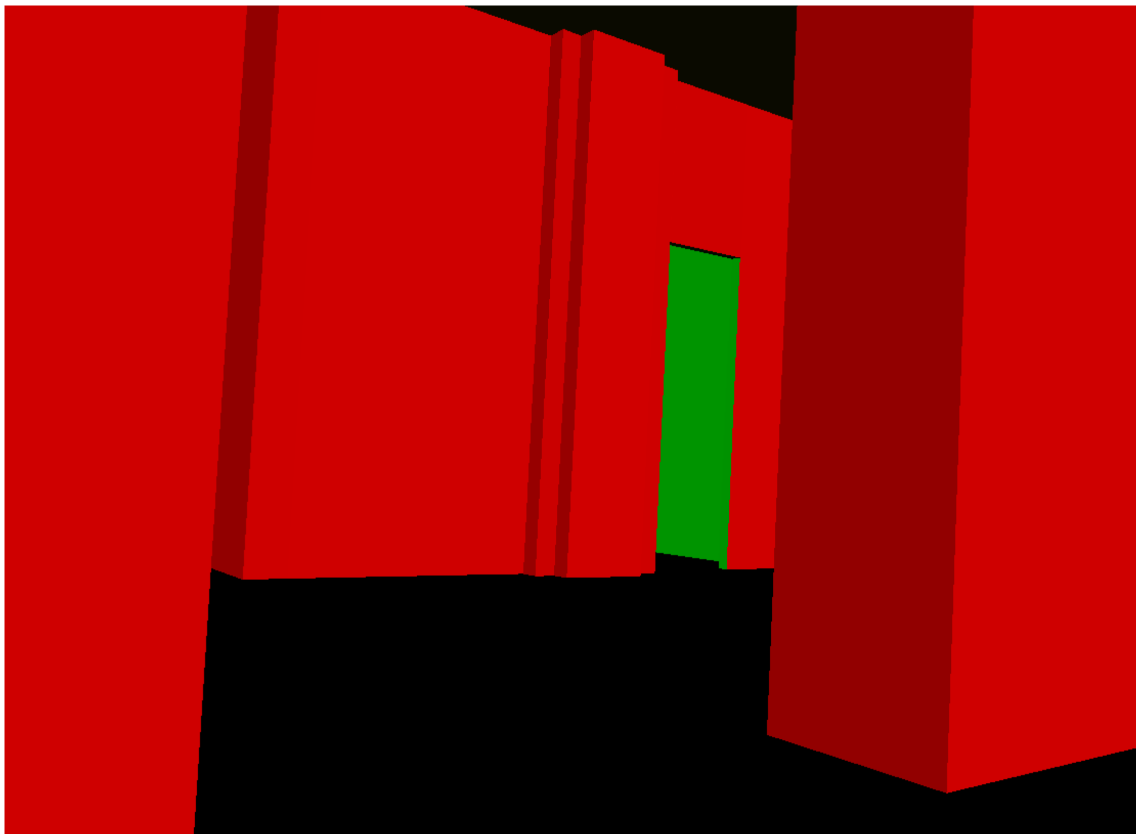


Figure C-1: Model 1 view 1: Looking into a neighboring space through the implicit portal

Figure C-2: Model 1 view 2: An example of the doorframes at portals

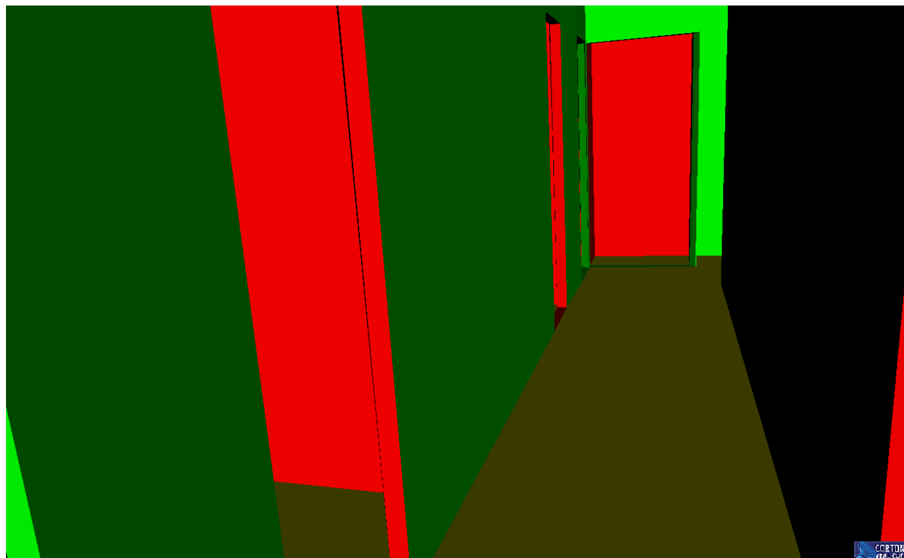


Figure C-3: Model 1 view 3: Hallway as seen inside the VRML viewer

Bibliography

[1] Thomas Funkhouser, Seth Teller, Carlo Sequin, and Delnaz Khorramabadi. The UC berkeley system for interactive visualization of large architectural models. *Presence*, 5(1), January 1996.

[2] Google sketchup. http://sketchup.google.com/product/gsu.html.

[3] Microsoft Labs. Photosynth. http://labs.live.com/photosynth/.

[4] Sean Markan. BMG notes as of october 2002. http://city.csail.mit.edu/bmg/bmgnotes.html.

[5] Open inventor project. http://oss.sgi.com/projects/inventor/.

[6] Senior house interactive environment. http://web.mit.edu/21w785/sh/www/wad/intro.html.

[7] C. H. Sequin and P. S. Strauss. UNIGRAFIX. In *Proceedings of the 20th conference on Design automation*, pages 374–381. Annual ACM IEEE Design Automation Conference, 1983.

[8] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3D. In *ACM Transactions on Graphics (SIGGRAPH Proceedings)*, volume 25(3), pages 835–846, 2006.

[9] Emily Whiting, Jonathan Battat, and Seth Teller. Topology of urban environments. In *CAAD Futures 2007*, Sydney, July 2007. MIT.