3D Interior Model Extrusion from 2D Floor Plans

Matthew Chandler

Master of Computer Science Project
University of Alaska Fairbanks

Project Committee
Dr. Orion Lawlor -Chair
Dr. Jon Genetti
Dr Glenn Chappell

Abstract

It is often advantageous to have a 3D digital model of a building, whether for purposes of simulation, architectural design or whatever else. However, models are not commonly available. Much more common is the floor plan. Most of the information to build a 3D model is contained in the floor plan, so we built Imhotep: a tool to read in a floor plan and output a model of the building described therein. Imhotep reads from the .svg format and writes to Wavefront .obj files. It is able to robustly identify the various features in the floorplan, provided the input meets certain requirements. An interactive 3D render of the finished model is displayed so that the user is able to preview it.

Table of Contents

Introduction

Of the many tools used in architectural design, the floor plan is perhaps the most vital. floor plans describe buildings in two of three dimensions, in which most of the important design occurs. Rooms, walls, doors, windows, and stairs are shown, such that a viewer can visualize the completed building's layout. However, such a holistic view is not able to accurately convey the appearance of the building. People visualize objects more easily in 3D, as they would be experienced in person.

Because of this, it is often useful to create a 3D model of a building as it is being designed. This allows people to conceptualize the completed building far more readily than any combination of 2D projections. Unfortunately, public distribution of 3D models, when they exist, tends to be limited in comparison with floor plans. Often, one may find themselves with a floor plan, but no model, and thus no relatable visualization of the building. 3D modeling is beyond the abilities of most people.

For many simple buildings, the floor plan has all of the information needed for such a model. Such a building will have no interesting details in three dimensions that cannot be described in two, such as balconies or vaulted ceilings. The walls are vertically straight, and the floors and ceilings are at constant heights. If someone were to take one of these floor plans and extrude every line into 3D, the result would be a reasonable model of the building.

Where floor plans are available in a digital format, we can automate such a process with computer. To that end, we created Imhotep. Named after the ancient Egyptian architect and engineer[1], Imhotep is a program that, given a floor plan, will produce a model of the building it represents. The model is rendered and shown to the user, and we output a model file for usage elsewhere.

Unfortunately, floor plans, even when created on a computer, are made to be human-readable, quite often with no regard to how the data appears to the machine. To the computer, the floor plan is just lines; it has little to no information on what they represent. Several methods exist for interpreting line data and for extracting meaningful information from them, many of which we have used.

Imhotep leverages the fact that digital floor plans are often created with a CAD (Computer Aided Design) program that allows lines to be grouped together into layers. A layer may contain all of the lines defining the exterior wall, while another layer has the locations of the windows. Even when layers are not used, these different elements will quite often be colored differently from one another.

Imhotep uses this separation of elements as a first step in interpreting the floor plan. Such separation allows us to isolate components, which makes their identification easier. For example, if the wall data is considered separate from the rest of the floor plan, Imhotep can safely assume that all lines present are walls, without having to check to make sure that a given line is not part of a door, or something else.

The process is complicated by the fact that little to no standardization exists between separate floor plans, and sometimes not even across a single plan. There are generally agreed upon symbol sets, and in CAD created floor plans, templates exist for these, sets, however deviations from these templates are common. Some dimensions, such as the width of a door are commonly the same, but buildings may have doors of different sizes. We cannot assume that a single symbol, or even a set of symbols will be consistently used across every floor plan we encounter.

Our aim with Imhotep is to be general enough in its approach to support a wide variety of floor plans, if not all. It tries to be simple to use, and automates as much of the process as is feasible. The only exceptions to this are when data is not available in the floor plan (height of the ceiling, for example), or when we can utilize human visual recognition capabilities to make a much more accurate decision than one that could be done by a machine.

1 http://en.wikipedia.org/wiki/Imhotep

Rationale

- As stated previously, there are a number of uses for a 3D model of a building. Some possible cases are included here.

- When considering purchasing a home (or office, or any other building), it is generally desirable to view the house in person in order to see the layout, among other things. Even after looking at plans for the home, people seem to have trouble with forming a spatial concept of the layout. However, a visit in person is not always convenient, or even possible. The home may not have been built yet, or perhaps the would-be homeowner is unable to travel to the house. If they have access to floor plans, then Imhotep can allow them to virtually visit the building. if they then lose interest in the purchase upon seeing the structure in 3D, they can save themselves time and money.

- If the building is still in the design phase, it may be useful for the designer to periodically gain a 3D perspective on the plans being drafted. Imhotep provides a quick and simple method of doing so. These 3D models can also be shown as part of the design review and approval process, allowing a designer to produce models for several different designs without the need for 3D modeling abilities.

- Another aspect of building design is lighting. Ideally, the building should be evenly lit, and architects may desire to take advantage of natural lighting whenever possible. One way to determine where lights or windows are needed to achieve this ideal is to simulate the lighting in a computer. This obviously requires a computer model of the building, which we are able to provide with Imhotep. Such a simulation would allow the architect to see how much sunlight each room would receive for any time of day, for any day of the year. With that information, they could determine where artificial light is needed to supplement natural light, and where to place windows to maximize available light without causing glare.

- An interesting type of immersion occurs in video games (or simulations, CGI movies, etc.) when a real-world building is used as part of the scenery. Imhotep will allow modelers to quickly generate models of buildings, again with little to no modeling skills needed. Furthermore, it may be easier to do level design in 2D as a floor plan, and then use Imhotep to generate the 3D geometry.

- In the last few years, 3D printing has become increasingly popular and affordable. Imhotep's output could easily be imported into a 3D printer and a physical model of the building could be made. This in itself has a variety of uses. For example, a model could be printed without a roof attached. Models of furniture could also be made, and could be arranged in miniature before committing to move a several hundred pound table into a potentially undesirable location. If the models were made accurately enough, it could be determined if a sofa will even fit through the front door before spending hours trying in vain to move the real-world sofa into a real-world house.

- Undoubtedly, other uses could be found as well.

Prior Work

3D model generation of buildings is a topic that has seen a lot of work in both academic and commercial fields. Several methods and tools exist to automatically create these models aside from Imhotep.

One approach that seems fairly common is to have the user draw a floor plan directly into the tool. One example is Autodesk's *Homestyler*[2], but several other similar tools exist. These usually work by supplying the user with tools to create walls, add windows, and quite often allow the placement of furniture. The advantage to this approach is that the program doesn't have to do any object recognition because the objects are specified by the user as the plans are made.

A team at University of East Anglia[3] created a tool that generates a building model given a building's footprint. The focus of their research is roof generation, so the rest of the model created is fairly simple, merely extruding the footprint into walls, and the model lacks an interior entirely.

In addition to these projects, there has been considerable research into generating models from floor plans, much as we do with Imhotep. A comparison of these techniques was compiled at Arizona State University[4]. The techniques vary, but each of the methods considered took a drawing of a floor plan and made a 3D model. Still, the common technique used was to:

- Remove unneeded information
  - noise, text, labels...
- Recognize symbols
- Extrude 3D symbols
- Cleanup

A common theme is the tradeoff between automation and accuracy. Of the techniques listed, some would either robustly produce a model with considerable user interaction, while others were highly automated, but were not as accurate.

Comparisons to Prior Work

In building Imhotep, we followed the same basic outline as the examples listed in the Arizona State paper, though the methods we used to perform these steps differ in several key areas. Most notably is Imhotep's leverage of separation of information by layers/colors. This necessitates Imhotep's input to be made with CAD tools, and with considerable restrictions compared to these other methods. However, by using this information, symbol recognition becomes a much simpler, and therefore more accurate, process.

Imhotep differs from the University of East Anglia method significantly due to Imhotep's focus on interior model creation whereas the paper only creates an exterior model, even so, we can borrow elements from their method. Imhotep in its current form only models individual floors of a building,, and does not attempt roof generation. However, the method presented in this paper could be added as a module, which would allow for more accurate modeling of buildings with sloped roofs. As floor plans do not usually indicate any roof structure whatsoever, the roof structure might not accurately reflect the building's true roof structure, though in many applications such accuracy may not be necessary.

---

2 Autodesk;s *Homestyler* - http://www.homestyler.com/

3 Laycock R. G. “Automatically Generating Roof Models from Building Footprints.” http://wscg.zcu.cz/wscg2003/papers_2003/g67.pdf

4 Yin,Xuetao. Generating 3D Building Models from Architectural Drawings: A Survey. http://peterwonka.net/Publications/2009.CGA.Yin.floor planExtrusionSurvey.IEEEDigitalLibrary.pdf

Requirements

- Automatic generation of Building model from floor plan
- Read from .SVG format
- Identify and model:
    - Walls
    - Doors
    - Windows
    - Floor / Ceiling
- Render preview of building
- Output .obj/.mtl model file for use in other applications
- Automate as much as possible
- Easy to use UI
- Support some variety of floor plans

Process overview

The process of building a model from a floor plan can be described in four steps:

- Reading the data from the file
- Interpreting data
- Building the geometry
- Writing geometry to a file.

The most complicated portion of this process is the interpretation of the data. Imhotep uses a combination of human and algorithmic methods to accomplish this task. Human interaction is kept to a minimum, and is mostly used for collecting data not obtainable from floor plans (such as height and scale) and assisting in identification and separation of the layers of the floor plan.

In summary, the interpretation process follows these steps:

- The user selects layers to be identified with the exterior walls, interior walls, doors, and windows
- For each of those, individual elements are found algorithmically from line data.
- Interactions between elements, such as the intersection of a window and a wall, are determined

Once the data has been interpreted, geometry generation is comparatively straightforward. A number of techniques are used to generate the various types of objects. Walls are created using simple extrusion of line data, and holes are cut into these for doors and windows. Doors and windows are created using pre-made models that are dropped into place. Floors and ceilings are made with a tessellated polygon using the outline of the building as a perimeter. It then renderer sand displays the geometry to the user as well as write it to a file for future use.

We will describe the entire process in detail in the next section.

Process Detail

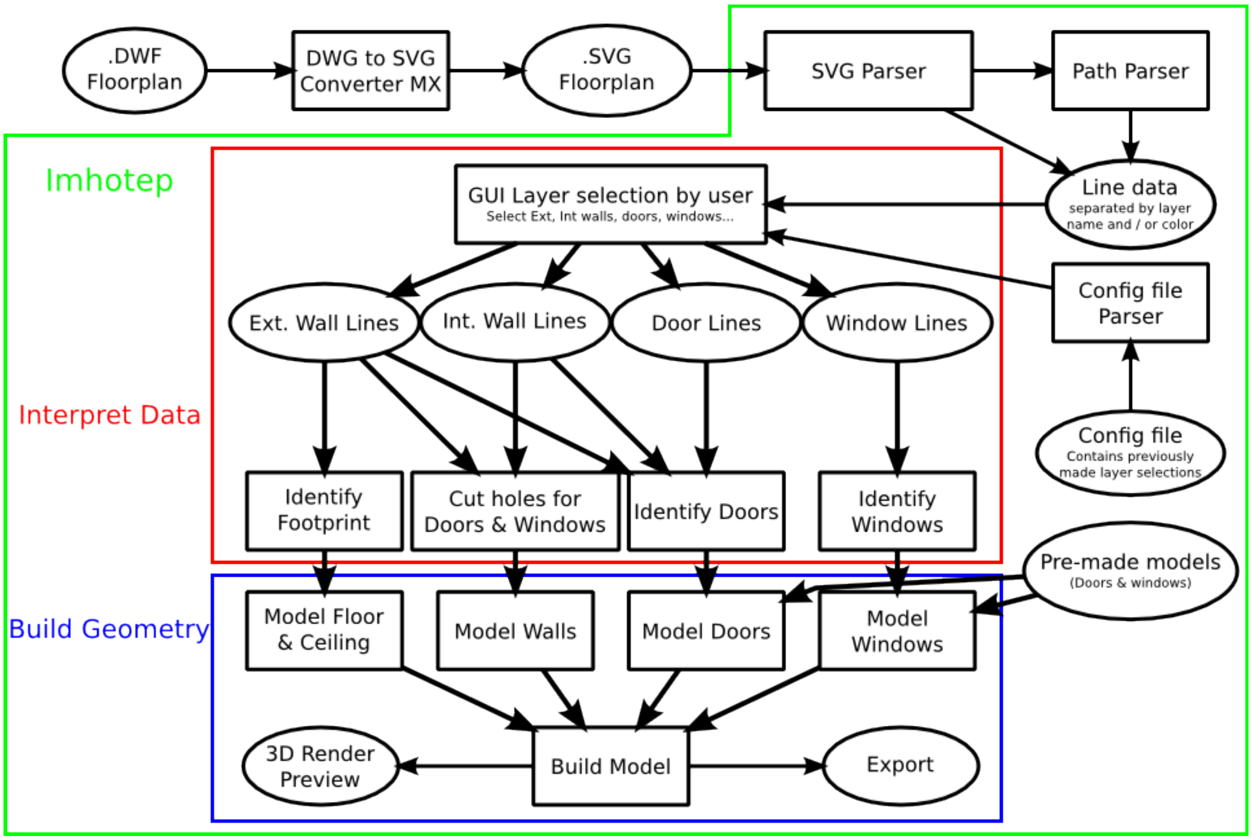

Figure 1: Process flowchart for Imhotep

Input expectations

Given the large variation present between floor plans, we need to place some restrictions on the input. Most notably, the data must be separated into layers, or at least be color coded. To be more specific, the exterior walls, interior walls, doors, and windows must be either in separate layers, or colored uniquely.

Another restriction is on the line data in each layer. For at least the previously mentioned elements, all line data must be complete, that is, there must not be any missing lines or holes. The lines must also never overlap. For any line that connects to another line, the endpoints at the connection must have the same coordinates. If the input does not meet these criteria, the detection routines will fail.

Walls

We use any and all lines appearing in the wall layers as walls. Superfluous data, such as labels or dimensions, should not be included in these layers.

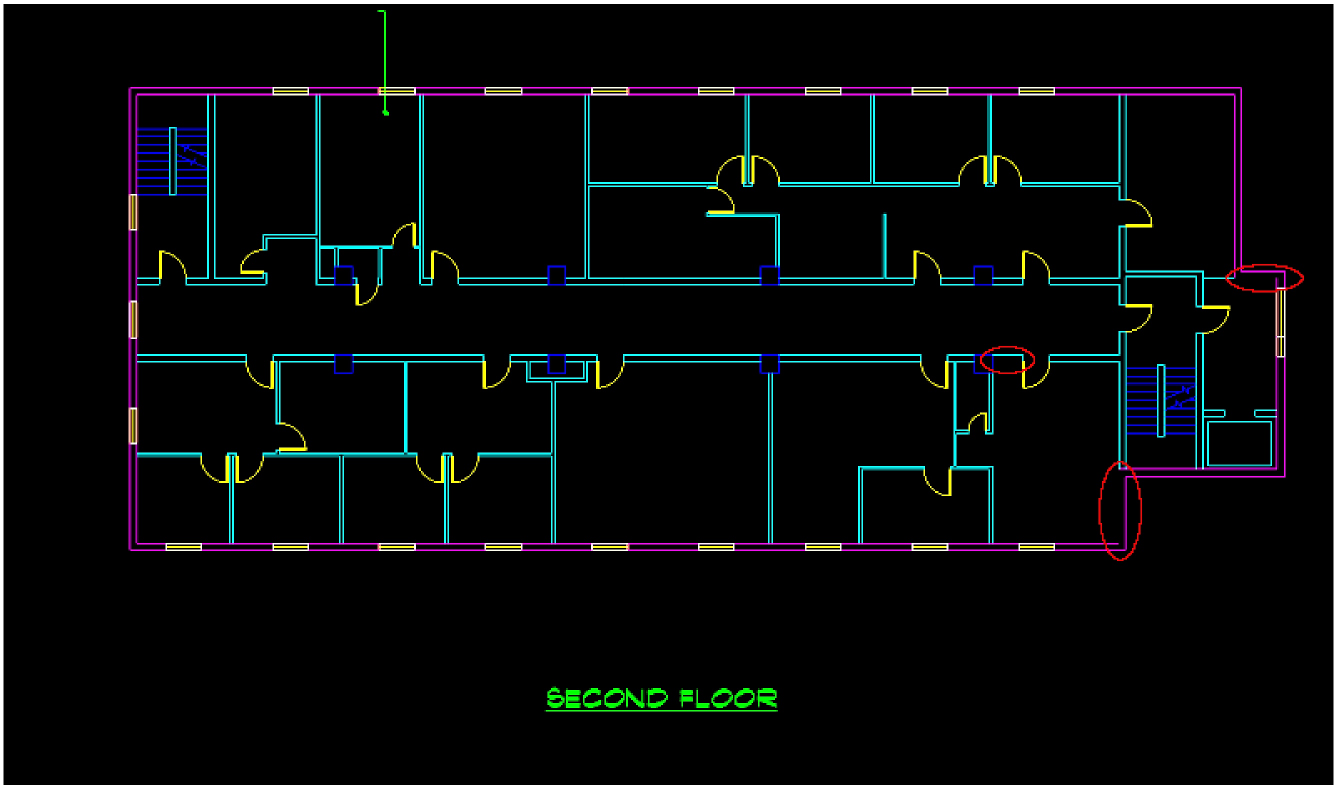

Figure 2: Invalid floor plan input - missing walls are circled

Imhotep can accept some variety in symbols for doors, windows, but they must meet the following criteria:

Doors

Doors must be represented as a set of lines that are connected together, that is, every line in the set must have at least one endpoint that joins the endpoint of another line in the set. Generally, the lines are drawn so that there is a rectangular portion symbolizing the door itself with a curve marking the path of the outer end of the door. As we do not support true curves in Imhotep, these curves must be composed of straight line segments. These lines must connect to endpoints of wall lines (either interior or exterior) in two places: the edges of the doorway.

If one of these points has two lines ,usually the corner of the door rectangle, it is assumed to be the hinge location. If we can't find such a point, the upper left-most point (point with the minimum distance to the drawing's origin) is assumed to be the hinge.

Double doors are a special case in which there is also an endpoint at the midpoint of the line between the two points touching the wall. Usually double doors are represented as two mirrored door symbols with the curves meeting at the middle. Both points that join the wall are considered to be hinges.

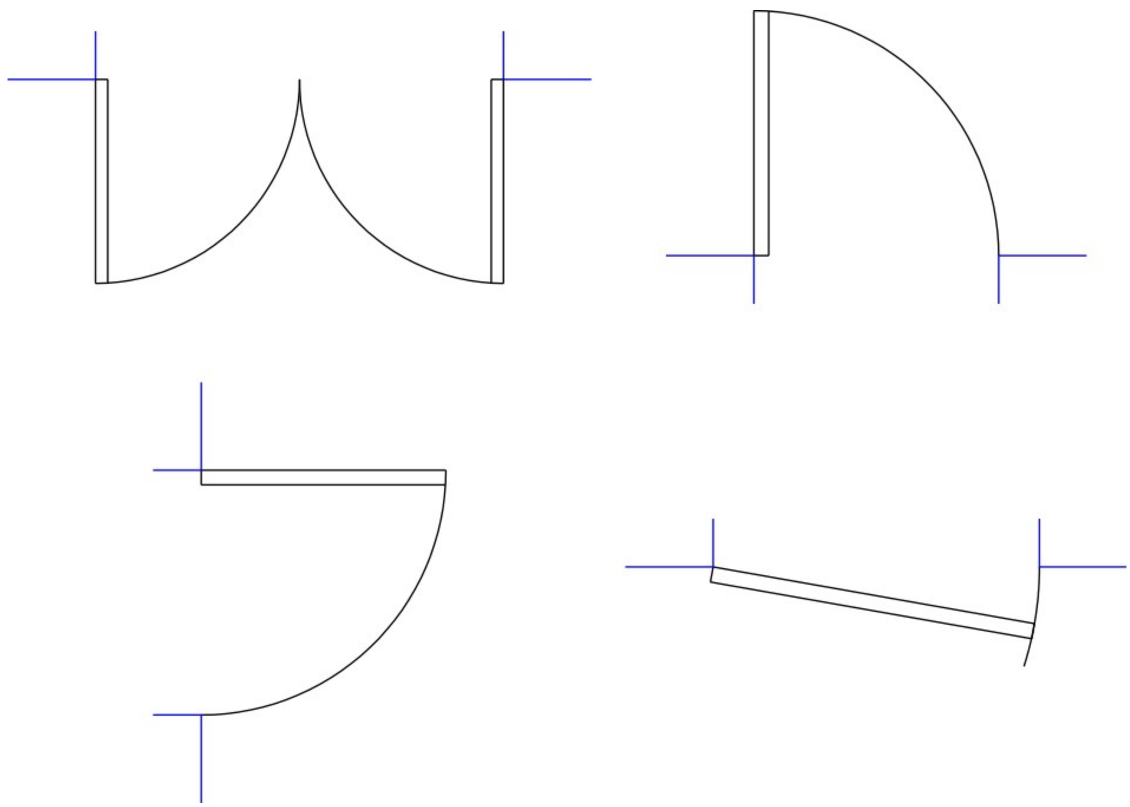

Figure 3: Valid door symbols. Walls are shown in blue.

Windows

Windows must be represented as a rectangle, with all four lines connected at endpoints, and each endpoint at one corner of the window's frame. These points should lie on a wall line. Any additional, non-connected lines (such as a line across the middle) will be discarded. No additional lines should connect to the rectangle's corners. The longer of the two dimensions will be considered the width of the window.

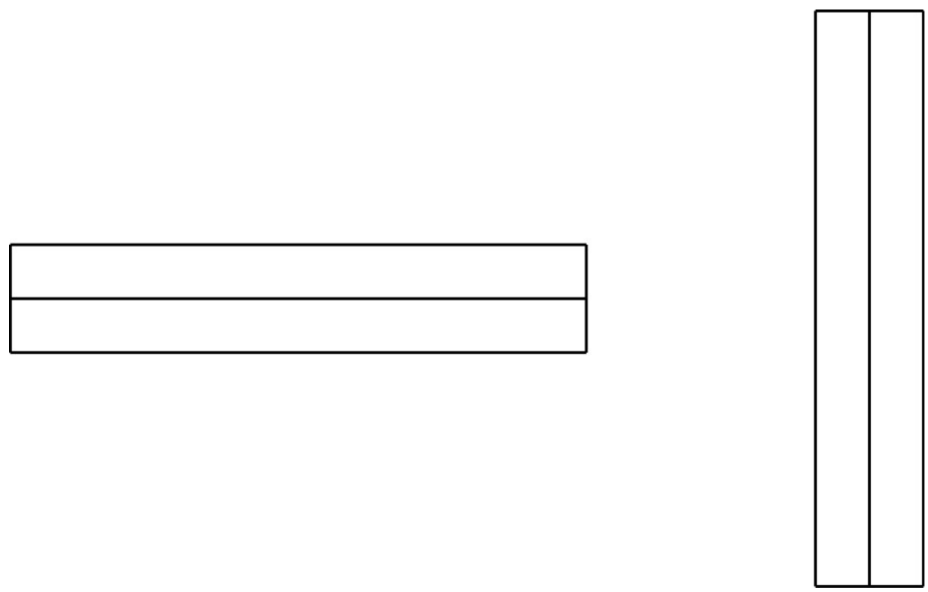

Figure 4: Example window symbols. The lines across the middle are ignored.

Layered / colored SVG input

AutoDesk's *AutoCAD* is one of the most common tools used for CAD design of buildings including the floor plans. *AutoCAD* supports several formats, one of which is DWF (Design Web Format).[5] DWF is meant to be a portable format for viewing, but not editing drawings made in *AutoCAD*. Specifically, it is intended to be used by an audience without access to *AutoCAD.* Autodesk distributes a free viewing tool,[6] and many other tools use the format as well.

We chose DWF as the starting format for Imhotep. Of the available formats for the floor plans we were interested in initially processing, DWF seemed to have the most usable information.

While the DWF format is open, and an API[7] is produced by Autodesk, both the format and the API are poorly documented and unwieldy. Because of this, the SVG format was chosen to be the input to Imhotep instead of DWF, adding a necessary conversion step. A number of tools exist to convert Autodesk formats to SVG. DWGTool Software's program, *DWG to SVG Converter MX*[8] was chosen, though other tools may work as well. [9]

The converted SVG floor plans retain the important data from the original. All of the line data is presented as SVG paths, which include coordinate data and colors, each with metadata, most notably the name of the layer to which each path belongs. These layers are the basis of our identification system for the various structures and elements as well as for discarding irrelevant data such as room labels. Additionally, it appears that most floor plans also separate these elements by coloring them differently, so we leverage that in conjunction with layers for an added measure of robustness.

Cleaning up input - UAF's Chapman Building

The floor plans tested during development are those for UAF's Computer Science and Math building: the Chapman Building. The flooplans were obtained from UAF's Facility Services website.[10]

In order to meet the previously defined input requirements, we had to make some modifications to the given files. Some walls and window were missing lines, and some walls had overlapping lines, all of which we mended. We found that the polygon tessellation tool we used (gluTess, described later) choked on walls where endpoints appeared in the middle of a window, so the offending endpoints were moved to the edge of the window.

The doors and windows all appeared in the same layer, with the same color (yellow), so we recolored the windows red. Additionally, the trial version of *DWG to SVG Converter MX* places a large watermark over the generated SVG, which we removed for convenience, though as it was in its own layer, it could be easily discarded by Imhotep's layer picker.

5 http://en.wikipedia.org/wiki/Design_Web_Format
6 http://usa.autodesk.com/design-review/
7 http://www.autodesk.com/dwftoolkit
8 http://www.dwgtool.com/index.htm
9 Theoretically, Imhotep should be able to extract at least the line data from any valid .SVG file without curves. However, we've seen variation in how path data is stored with various tools. For example, the popular SVG editor *Inkscape* stores color information in it's own tag, while *DWG to SVG Converter MX* places it in the path tag. Also, the layering scheme used by *DWG to SVG Converter MX* appears to be unique to this tool. If needed, Imhotep could be extended to support other conventions, possibly with little effort required.
10 http://facilities.alaska.edu/uaf/fsinfo/0record/home.html - Login Required

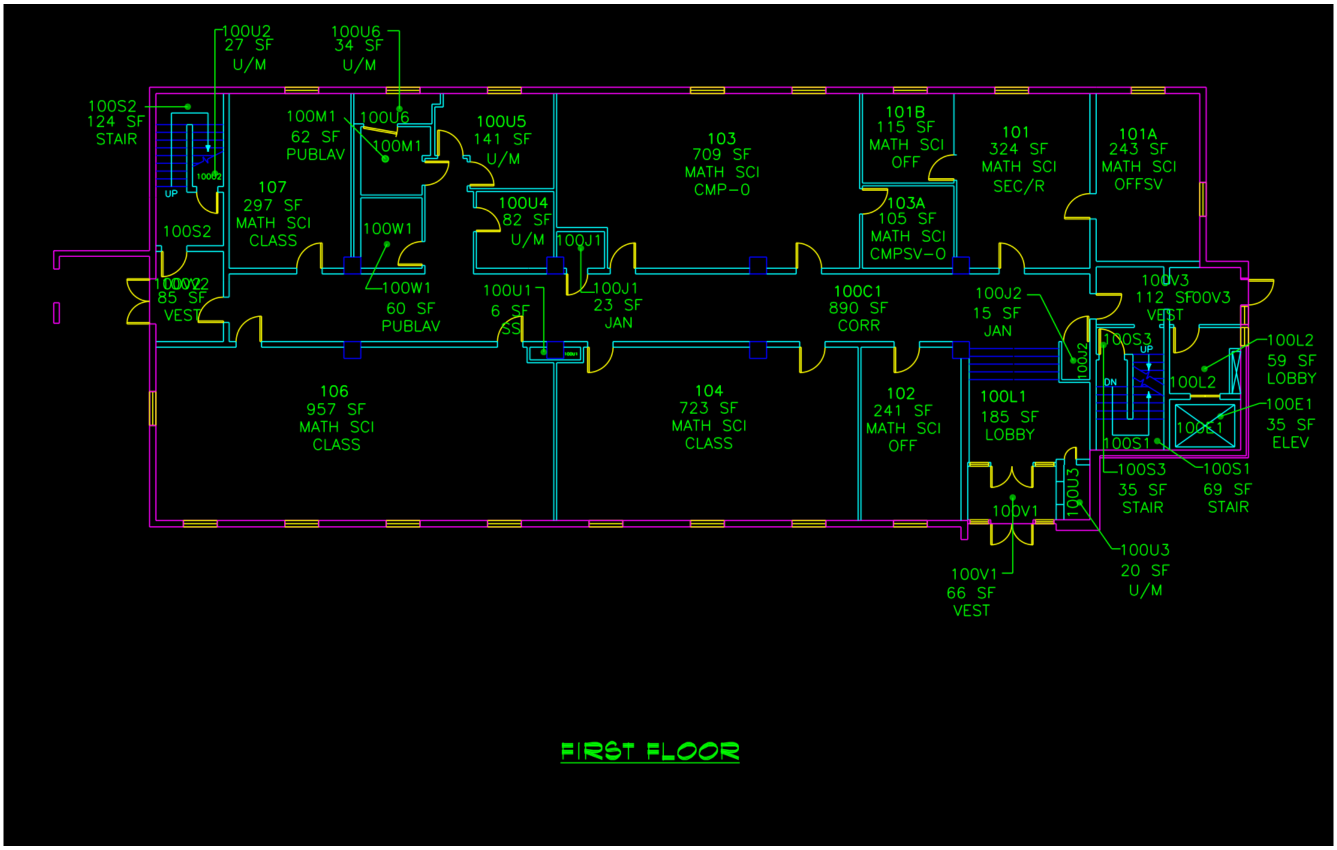

Figure 5: The floor plan for the 1st floor of UAF's Chapman Building

SVG Parser

The SVG format is XML based.[11] The format uses a number of XML tags to store data all enclosed inside a pair of <svg> tags. Of these tags, we are here interested in the path and color tags. Any data not enclosed in a tag specified in the SVG standard is to be ignored by standards-compliant readers, and *DWG to SVG Converter MX* leverages this to store the layer names, among other information as metadata.

```
<CADConverterDwgEntity>
<Type>AcDbPolyline</Type>
<Handle>96</Handle>
<Layer>FS516-1_A-WALL-EXT</Layer>
<Linetype>Continuous</Linetype>
</CADConverterDwgEntity>
<path d="M652.53 101.31L615.79 101.31"
       fill="none" stroke="fuchsia" stroke-width="0.1"/>
<EndCADConverterDwgEntity>
</EndCADConverterDwgEntity>
```

Figure 6: Example SVG element (exterior wall segment) as produced by DWG to SVG Converter MX

11 http://www.w3.org/Graphics/SVG/

A number of tools exist to read and parse XML; we chose to use libxml++ for Imhotep. libxml++ is a set of C++ bindings for the C library, libxml, created and maintained as part of the GNOME project.[12] libxml++ was chosen for ease of use, cross-platform availability, and maturity. We use the library's DOM parser, which allows a traversal of the XML hierarchy.

During the traversal, path data, layer names, fill color strings, and stroke color strings are collected. A hash key is created for each path found, where the key is composed of a combination of the layer name, stroke, and fill colors. For example, in the sample wall segment shown in figure 6, the layer name is "FS516-1_A-WALL-EXT", the stroke color is fuchsia, and the fill color is none, so the layer key for this path would be : " FS516-1_A-WALL-EXT-fuchsia-none".

We insert this key into a hash table along with its associated path data. The path data itself is obtained using a custom built parser, which we will describe in the next section.

SVG path parser

Core to the SVG format is the path element. Paths store line and curve data, which for floor plans will be walls, windows, doors, etc... For the sake of simplicity, curves are not currently supported in the parser, although support could be easily added in later. The other main element is text, although *DWG to SVG Converter MX* draws text out in lines rather than using SVG's text capabilities, so we have made no effort to utilize them so far.

Path data is encoded using a sub-language which encodes locations of end points for lines and type of line. The grammar for the language is reproduced, sans curves, below.

12 http://libxmlplusplus.sourceforge.net/

```
svg-path:
    wsp* moveto-drawto-command-groups? wsp*
moveto-drawto-command-groups:
    moveto-drawto-command-group
    | moveto-drawto-command-group wsp* moveto-drawto-command-groups
moveto-drawto-command-group:
    moveto wsp* drawto-commands?
drawto-commands:
    drawto-command
    | drawto-command wsp* drawto-commands
drawto-command:
    closepath
    | lineto
    | horizontal-lineto
    | vertical-lineto
moveto:
    ( "M" | "m" ) wsp* moveto-argument-sequence
moveto-argument-sequence:
    coordinate-pair
    | coordinate-pair comma-wsp? lineto-argument-sequence
closepath:
    ("Z" | "z")
lineto:
    ( "L" | "l" ) wsp* lineto-argument-sequence
lineto-argument-sequence:
    coordinate-pair
    | coordinate-pair comma-wsp? lineto-argument-sequence
horizontal-lineto:
    ( "H" | "h" ) wsp* horizontal-lineto-argument-sequence
horizontal-lineto-argument-sequence:
    coordinate
    | coordinate comma-wsp? horizontal-lineto-argument-sequence
vertical-lineto:
    ( "V" | "v" ) wsp* vertical-lineto-argument-sequence
vertical-lineto-argument-sequence:
    coordinate
    | coordinate comma-wsp? vertical-lineto-argument-sequence
coordinate-pair:
    coordinate comma-wsp? coordinate
coordinate:
    number
```

Figure 7: SVG path BNF grammar. Curved paths are not currently supported by Imhotep, and have been removed. Rules for matching numbers, whitespace, and commas have also been removed for brevity. Full grammar is available at http://www.w3.org/TR/SVG/paths.html

Using Bison[13] and Flex[14], we built a parser to read this language and obtain the line data. We needed to make some modifications to the standard SVG grammar to better meet our needs. As mentioned above, curves are not supported, so we removed the relevant grammar rules. The other changes we made were slight; some reorganizations of rules needed for the grammar to work better with Bison, none of which altered the language itself.

As libxml++ is parsing the SVG data, whenever a path string is found, we pass it to the path parser. The parser reads the svg path string, parses it, and extracts the path's line data from it. We return the path as arrays of lines, each of which contain a pair of 2D vectors representing the endpoints of the line. We then store the path in the hash table described previously.

13 GNU parser generator. Reads a BNF-like grammar with associated actions and outputs a parser in C for that grammar. http://www.gnu.org/software/bison/

14 Fast lexical analyzer. Often used with Bison for the lexical analysis stage of parsing. http://flex.sourceforge.net/

Layer Picker

Potentially, we could identify layers by the layer names extracted from the SVG, since many layers are given somewhat meaningful names. However, there is no guarantee that every floor plan will have usable layer names. Several have differing colors for the various elements, but there doesn't seem to be any sort of standardization in the colors. Another method that is used is to do symbol recognition on the entire dataset. This is a computational complex operation, as much of the data (such as labels and arrows) is not useful. The excess data would require that we check every set of lines with a battery of very specific tests to determine what the lines are meant to represent, and even then there is a high possibility for error.

Instead, we ask the user to identify the layers found while parsing the input with architectural elements they represent. A window is shown to the user, with a column of check boxes along the right side. A check box is shown for each layer / color combination found during the parsing stage. When a box is checked, each line in that layer is shown on the screen.

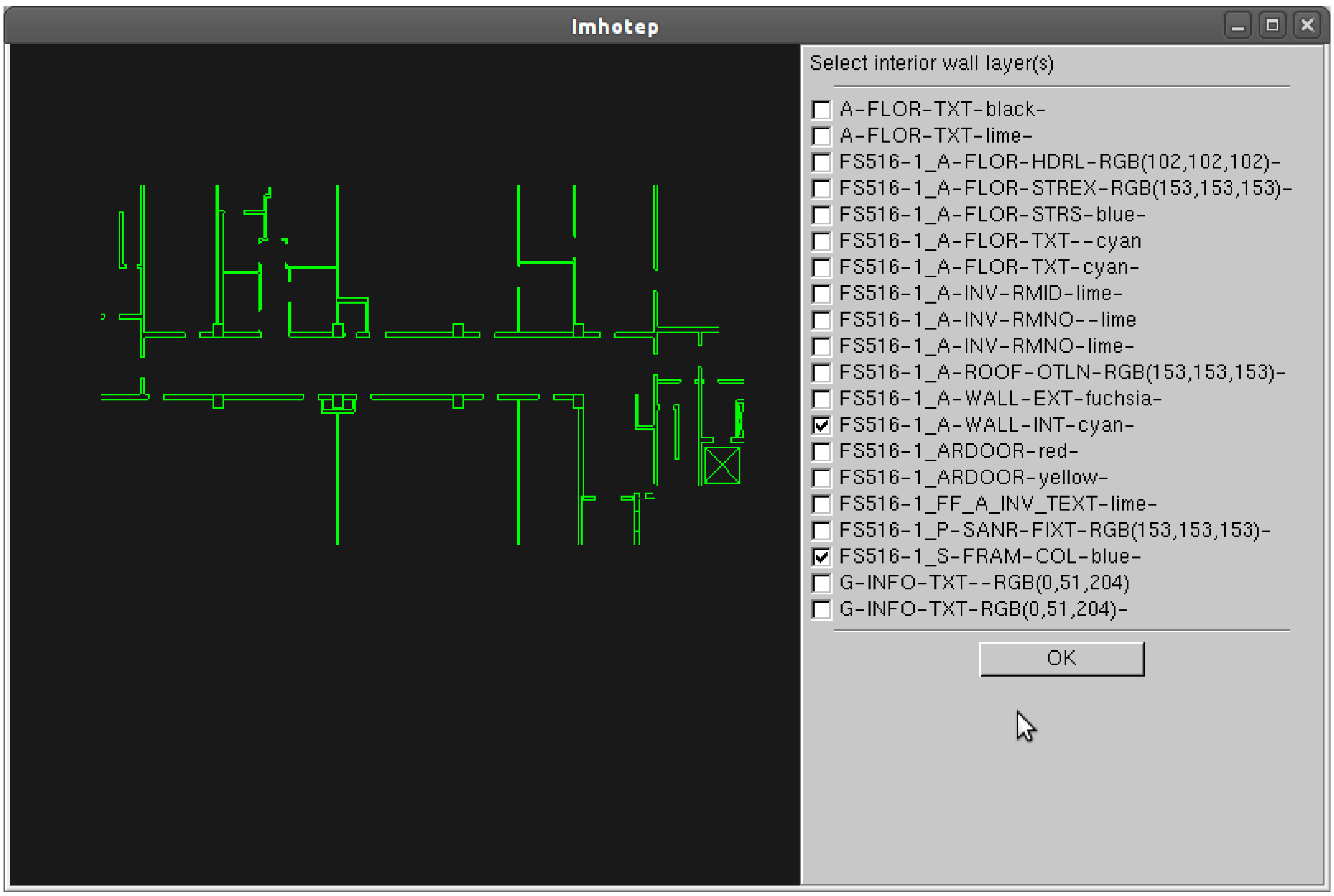

Figure 8: The layer selection interface for Imhotep

The user is prompted to check the boxes corresponding to the layer, or layers for the exterior wall. Once the selection is made, the user is to press an "OK" button, whereupon the selection clears. The user is then prompted to use the same interface to choose the interior walls, doors, and windows. The final screen in this series prompts the user to enter the scale factor and to enter the ceiling height.

There is no guarantee that the floor plan's author gave meaningful names to the layer or to the colors used, but as long as the necessary layers are separated according to the input requirements above, the user will be able to find the correct layers even if they have to click through each one to do so. The user ought to be able to determine the layer's identity visually by looking at the displayed lines.

After the user has made these selections, we clear all unused layers from memory. The data that remains is properly categorized so that when doing symbol recognition on these categories, we can assume that only elements of that category are present. For example, the wall data will not have windows that we may accidentally interpret as walls. This lets us make certain assumptions about the data which greatly simplifies the process of identification.

The entire process shouldn't take a user more than a minute to complete. In exchange for this, we have a guarantee that the selections made are accurate as far as the user can determine. No such guarantee can be made using an automated approach.

As a matter of convenience, we store the layer selections made for every floor plan processed to a configuration file. Every subsequent run for that floor plan file will have the selections made previously loaded for each element. The user is still able to alter the selection, if necessary, but is also able to quickly click through the selection process if no changes are needed.

Element Identification

Walls

Walls are easily identified. We simply assume that every line in the selected wall layers (both interior and exterior) is a wall.

Doors

To identify doors, we need to find every group of connected lines in the selected door layer, and test them to see if they match the criteria defined previously for door symbols.

We do this by first creating a hash table from the door layer's line data, with endpoint coordinates as the keys. Each line has two entries in the table, one for each endpoint, and each point that joins to another line will have each line associated with it. To avoid comparing floating-point numbers and the problems associated with doing so, we use a rounded fixed point representation of the endpoint coordinates as our hash table key. This rounding allows us to do 'fuzzy' matching: endpoints that are close enough to each other that they appear to be connected, but have slightly different coordinates, will be stored with the same hash key as long as they round to the same value.

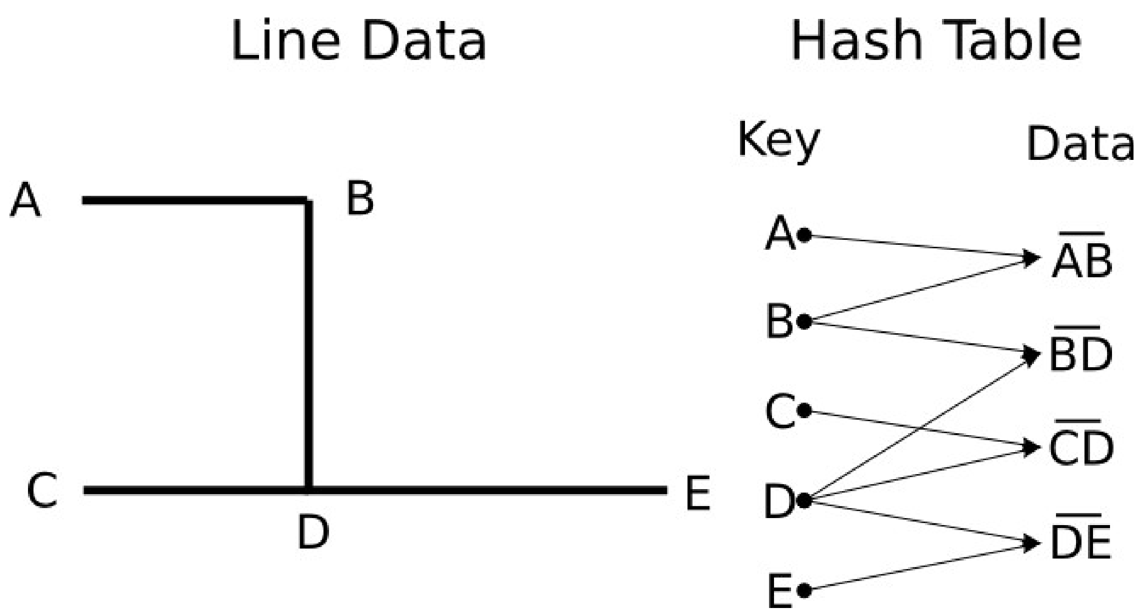

Figure 9: Illustration of how endpoint data is stored in a hash table

This hash table allows us to take a given point and quickly find all the lines that connect to that point. For example, in the illustration above, if query the table for point D, we find that D joins 3 lines, BD, CD, and DE.

We also build a second hash table, using this same scheme, for wall data, containing both interior and exterior walls with no distinction between the two. This will be used to determine if a point is touching a wall and retrieve the wall's lines at this point.

We choose the first entry in the door table as an arbitrary starting point and find the number of unvisited lines connected to this point. If there is more than one line, the point is added to a set of available points to be visited later. Otherwise, we have considered every line at this point, so it is removed from the set of available points, if present.

One of the unvisited lines is chosen arbitrarily, and this line is added to the set of visited lines. The opposite endpoint of the line is chosen as the next point to consider, and the algorithm will thus travel along connected lines. If a point is found that has no unvisited lines, we take the next available as the next point to consider, and the algorithm resumes from that point.

We add every point considered in this traversal to a searchable list to be used later. We also query the wall hash table for the current point during each iteration, and if the point is found in the wall data, we add it to a separate list.

This process continues until the current point has no unvisited lines, and the set of available points is also empty.

If exactly two points are found corresponding to points in the wall, we have detected a door. Otherwise, we delete the visited points and move on.

To get the geometry of the door frame, we find all of the lines in the wall data meeting at the two matched points. We choose the line at each of these points that is perpendicular to the line between the matched points, and retrieve the opposing endpoints. These two new points, along with the original matched points form a rectangle that will define the door frame. The four points are sorted so that the first is the upper left, and the points are ordered clockwise. The long sides of the rectangle are determined in order to find the door's orientation.

Next, we check to see if the door is a double door. This is done by finding the midpoint between the two points on the wall. If this point appears in the list of points comprising the door built in the traversal, then a flag is set marking the door as a double door.

If the door in consideration is a double door, we split the geometry in half, and mark the two original matched corners to be hinges. These two doors are then added to a list.

For a single door, the hinge is determined by finding which of the matched points has two lines connected to it in the door hash table. If no such point is found, than the upper left corner of the door is assumed to be the hinge. The finished door is added to the list.

Once the door(s) have been added to the list, we delete all of the points we visited. The whole process repeats until all the points in the door hash table have been deleted, at which point we return the finished set.

```
enum Door_orient_t {DOOR_HORIZ, DOOR_VERT};
struct Door
{
    Door(const std::vector<vec2> & Verts = std::vector<vec2>(),
        const Door_orient_t & Orient = DOOR_HORIZ, const size_t & Hinge = 0)
        :verts(Verts), orient(Orient), hinge_vert_i(Hinge)
    {}
    std::vector<vec2> verts;  //corners, starting with upper left, continuing CW
    Door_orient_t orient;
    size_t hinge_vert_i;
};
```

Figure 10: Door structure used in the door detection algorithm

```
Function get_doors:
Input: List of door Lines door_lines, list of exterior wall Lines ext_wall,
    list of interior wall Lines int_wall
Returns: list of Door objects
if door_lines is empty
    return empty list
ends = hash table of endpoints and lines from door_lines
wall_ends = hash table of endpoints and lines from ext_wall and int_wall

doors = empty list of Door objects to return

while ends is not empty
    visited_lines = empty set of lines
    available_points = empty set of coordinates // previously visited, but still join unvisited lines
    points = empty list of points // all points for this set of lines
    matched = empty list of points // points that touch the wall (should only be 2)
    curr_pt = arbitrary point in ends
    while true
        eq = list of lines at curr_pt
        if eq is empty
            break
        if curr_pt in wall_ends and not in matched
            append current_pt to matched
        if curr_pt not in points
            append curr_pt to ends
        available = number of lines not in visited_lines in eq
        if available > 0
            line = arbitrary unvisited line from eq
            if available > 1
                add curr_pt to available_points
            else
                remove curr_pt from available_points (if present)
            add line to visited_lines
            curr_pt = other endpoint of line
        else
            remove current_pt from available_points (if present)
            if available_points is empty
                break
            curr_pt = arbitrary point in availible_points
    //we have now visited every connected point
    if length of matched is 2 // we found a valid door - build the object
        door_line = line between matched[0] and matched[1]
        door_frame = empty list of points
        for each point in matched
            if there is a  line from wall_ends at this point that is perpendicular to door_line
                add other endpoint to door_frame
        add points from matched to door_frame
        sort door_frame so that door_frame[0] is closest to origin, and points are ordered clockwise

        door = Door object with door_frame as corners.
        if door is longer horizontally
            door.orient = horizontal
        else
            door.orient = vertical
        //detect double door
        if midpoint of door_line is in points
            split door in half into door1 and door2
            door1.hinge = point from door1's corners that is in matched
            door2.hinge = point from door2's corners that is in matched
            add door1 and door2 to doors
        else
            door.hinge = point form door's corners that has >= 2 lines in ends
            add door to doors

    remove all points in points from ends
return doors
```

Figure 11: Pseuodocode for door detection algorithm

Windows

We identify windows by finding rectangles in the window layer. We do so by using a method similar to that used for the doors, though somewhat less complex due to the comparative simplicity of the window symbol we are looking for.

As before, we build a hash table of endpoints and their lines from the window layer to be used in finding connected lines. We choose an arbitrary point to start (again, the first entry in the window table). Using this point, we query the hash table for all lines connecting to that point. We select the first of these found that has not been visited previously, and its other endpoint is chosen as the next point to consider. We add the previous point to a list, and remove it from the hash table.

We follow the above procedure until either no connecting lines are found, or we arrive back at the starting point.

If the search arrived back at the starting point and four points are found, then we consider the lines to be a window. Next, we reorganize the points so that the point closest to the origin is first, and so they continue in a clockwise direction. Additionally, we determine the longest side of the rectangle, which corresponds with the direction the window is facing. The completed window is then added to a list.

We continue this process of window identification until the hash table is empty and all points have been considered. The finished list of windows is returned.

```
enum Window_orient_t {WINDOW_HORIZ, WINDOW_VERT};
struct Window
{
    Window(const std::vector<vec2> & Verts = std::vector<vec2>(),
        const Window_orient_t & Orient = WINDOW_HORIZ)
        :verts(Verts), orient(Orient)
    {}
    std::vector<vec2> verts;  //corners, starting with upper left, continuing CW
    Window_orient_t orient;
};
```

Figure 12: Window type used in the window detection algorithm

```
Function get_windows:
Input: list of window Lines window_lines
Returns: list of Windows
if window_lines is empty
    return empty list
ends = hash table of endpoints and lines from window_lines

window = empty list of Window objects to return

while ends is not empty
    first_pt = arbitrary point in ends
    curr_pt = first_pt
    next_pt = (0,0)
    matched = empty list of points
    add curr_pt to matched
    prevline = null Line

    while true
        eq = all lines from ends at curr_pt
        line = arbitrary line from eq not equal to prevline
        if none available
            break
        next_pt = other endpoint of line
        if next_point == first_point
            break
        append curr_pt to matched
        curr_pt = next_pt
        prevline = line
        erase all points in eq from ends
    //did we find a loop of 4 lines (a valid window)
    if length(matched) is 4 and next_pt == first_point
        win = empty Window
        sort matched so that matched[0] is closest to origin and points are ordered clockwise
        win.verts = matched
       if window is longer horizontally
            win.orient  = horizontal
        else
            win.orient = vertical
        add win to windows

return windows
```

Figure 13: Pseudocode for window detection algorithm

Exterior Wall / Building Footprint

It is necessary to identify the exterior walls of the building in order to build the building's floor and ceiling.

Again, a hash table is created from the line data with endpoints as keys. As we construct the hash tablet we also determine the point closest to the origin. This will be the starting point of a traversal of the exterior, as it is guaranteed to be on the outside of the wall.

Starting with this point, the hash table is queried for all lines with it as an endpoint. The angle between each line and the horizontal is found, ranging from 0 to 360 degrees counterclockwise. The line with the angle equal to the previous line is chosen, so that the traversal will prefer to not turn. If we cannot find such a line (or there is no previous line to be found), the line with the greatest angle is chosen, which will cause the traversal to turn left at corners. Next we set the current point to be the other endpoint of this new line. We add the current line to the floor plan perimeter and the algorithm finds all the lines connecting to the new point. We continue until either the starting point is reached, or a dead-end is encountered (a point that doesn't connect to any other line).

A problem may occur during our traversal in which a loop is encountered. To avoid loops, we keep a set of visited points. If our traversal arrives at a previously visited point, we remove lines from the floor perimeter until we encounter the line containing the previously visited point, and the corresponding points are removed from the visited set. We find the lines from this point as before, but this time the line that led to the loop is excluded from this list, forcing the traversal to go into another direction.

```
Function get_footprint:
Input: list of exterior wall Lines wall_lines
Returns: list of Lines forming a polygon of the floor
if wall_lines is empty
    return empty list
ret = empty list of Line objects to return
ends = hash table of endpoints and lines from wall_lines
min = point in ends closest to origin
visited = empty set of visited points
curr_pt = min
prev_line = null Line
prev_angle = -1
do
    //loop advoidance
    loop_pt = null point
    if curr_pt in visited
        pop lines from ret until last's start point == curr_pt
            remove start point of line being popped from visited set
        loop_pt = ret's last lines's end point
        remove curr_pt from visited
        pop last line from ret
        prev_line = last line from ret
        prev_angle = angle of prevline

    max_angle = -1
    max_angle_line = null Line
    count = 0
    eq = list of lines in ends at curr_pt
    for i in eq
        if i == prev_line
            continue
        if either endpoint of i == loop_pt
            continue
        ++ count
        angle = angle of i
        if angle = prev_angle // try to contiue going straight
            max_angle = angle
            max_angle_line = i
            break
        if angle > max_angle // if we have to turn, favor the largest angle
            max_angle = angle
            max_angle_line = i

    add curr_pt to visited
    if count == 0
        break
    swap endpoints of max_angle_line if needed so that max_angle_line's start point is the endpoint
        of the last line in ret
    add max_angle_line to ret
    curr_pt = end point of max_angle_line
    prev_line =max_angle_line
    prev_angle = max_angle
while curr_pt != min

return ret
```

Figure 14: Pseudocode for footprint traversal algorithm

Geometry Generation

Walls

Wall generation is fairly straightforward compared to other elements. We simply extrude the line data extracted from the floor plan into three dimensions, forming quadrilaterals. The height of these quadrilaterals is given by the user, and defaults to seven feet. We perform this procedure for both interior and exterior walls.

Next, we must cut holes in the walls for windows and doors to sit in. This is done by using gluTess. GluTess[15] is a tool from the Open GL Utility library (GLU) that is able to tessellate complex polygons (concave or self-intersecting polygons, or polygons with holes, such as our walls here) into simple triangles. GluTess takes a set of contours as input, and has various settings to describe what should happen when contours overlap. We used the wall quadrilaterals as one contour, and the window and door outlines as others. We configured gluTess so that when the door / window contours overlap the walls, the resulting polygon will be made with a hole in the overlapping locations. For doors, we made the hole stretch from floor to ceiling, because many times a doorway will not have any wall lines drawn across it in the floor plan. For those that do, we clear the wall away entirely to be consistent. For windows, we make the hole (for our default seven feet high ceilings) two feet from the ground to 1 foot from the ceiling. Throughout this process, we scale each point from the floor plan with the scale factor entered by the user.

GluTess is primarily made to render the polygon right away, but it allows us to register a series of callback functions that we use to collect the triangle information as gluTess generates it. This gives us the vertex information. To find the surface normal, we simply take the cross product of 2 of the wall edges. This does not guarantee that the normal will be pointing out of the wall, but it will at least be perpendicular to it, and using double-sided lighting that's good enough. If the user needs normals that do point out of the wall, most good 3D modeling software is able to recalculate normals fairly easy, so the user can do so on the exported model. To find texture coordinates, we use the plane equation for the wall to determine each vertex's position in texture space and assign texture coordinates for a tiled texture accordingly,

Floor and ceiling

Once we have identified the external wall, the floor and ceiling can be generated. We do this by creating a polygon with the lines forming the exterior as edges, again by using gluTess, with the floor plan's perimeter as the only contour.

We duplicate this geometry and use it as both the floor and ceiling.

Doors

We build the doors by dropping in a premade door model. We modeled a generic door in *Blender*[16], with a segment of wall above the top of the door frame, since, as stated previously, our doorways will not have any previously existing walls present. We then scaled the model so that it fit into a unit cube centered on the origin. This initially looks strange, but allows us to easily scale it to the dimensions of whatever doorway we need to fill. Once finished, we exported the model as Wavefront

15 See the OpenGL Programming Guide, chapter 11. Online version: http://www.glprogramming.com/red/chapter11.html
16 http://www.blender.org/

.obj /.mtl files[17].

We wrote a simple .obj import function that is able to read the model file and build a list of triangles. We are only interested in obtaining vertex location, texture coordinates, normals, and texture names, so we ignore all other information.

Once we have read the triangle geometry from the file, we iterate over the list of doors obtained earlier. For each door in this list, we generate a copy of the imported model, move it to the door's location, scale it to fill the doorway, and rotate or mirror it so that the door opens on the side of the doorway we detected the hinge to be earlier.

Windows

We create windows using almost exactly the same method as we do with the doors. Again, we made a model window, scaled it to a cube, and exported it. We use he same .obj / .mtl importer used for the doors to read in this model, and place it using the same methods also.

The only notable difference in our technique, beside using a different model, is that the window rotation is simpler, since we do not have to match a hinge location.

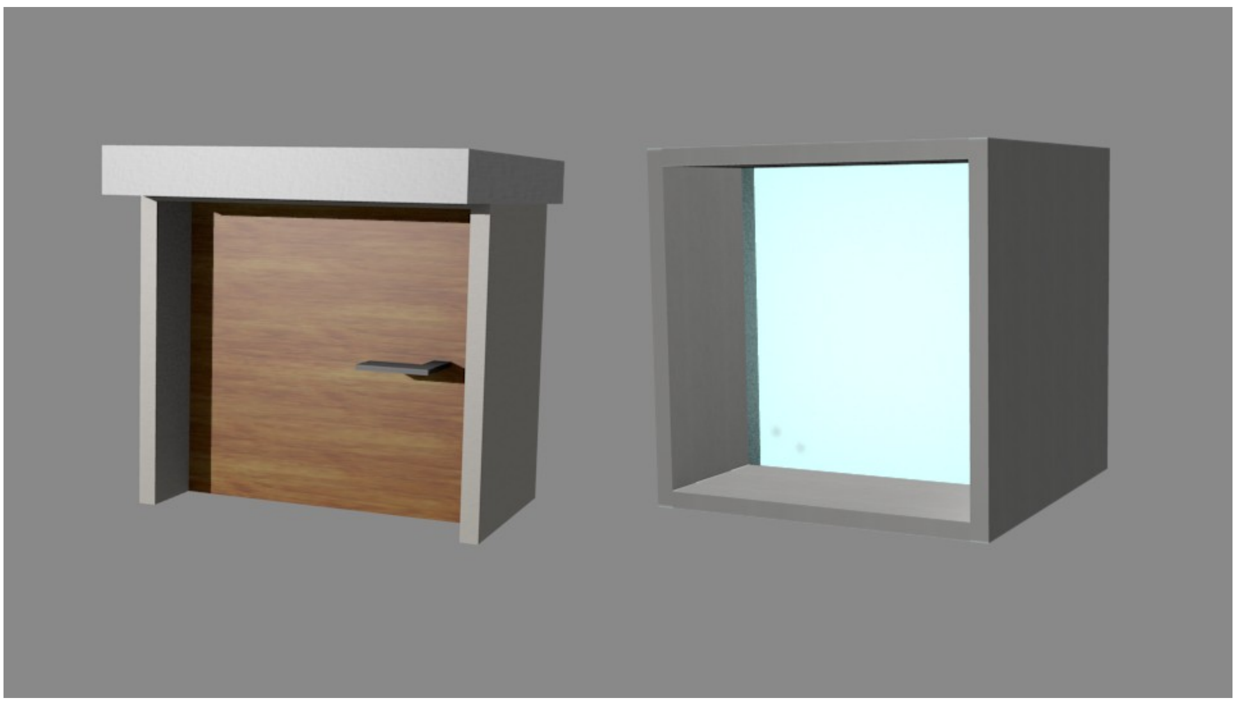

Figure 15: Scaled down models for doors and windows as rendered in Blender

17 http://en.wikipedia.org/wiki/Wavefront_.obj_file

Preview Render

Now that we have all of the geometry, we need to display it. We set up a simple OpenGL scene to display the model in with a dark gray background. We set up a camera system so that the user will be able to move around and view the model from any point. Lateral movement is controlled with the W, A, S, and D keys, or with the arrow keys. Vertical movement is achieved with Q and Z keys. Finally, the user can pan and tilt the display by clicking and dragging with the mouse.

Next, we draw the geometry. We do so in two parts, owing to the two methods used to create the geometry: gluTess for the walls, floor and ceiling; and premade models for the doors and windows. We will describe the methods used for each below. Currently, we do very little in the way of graphical effects. This allows Imhotep to be used on even low end hardware without having performance issues. We simply display the geometry, and apply appropriate textures.

The user is now able to move around and preview the model. This allows the user to inspect the model for defects, such as missing walls or misidentified objects. When finished, the user may press the escape key or close the window, at which point the program will end.

Walls, Floor, and Ceiling

As previously mentioned, gluTess is primarily designed to be used to render geometry, and we had to take extra steps to extract the geometry it created instead of drawing it. However, we were able to leverage its drawing ability without recreating the geometry every frame by having it draw to an OpenGL display list.[18]

When we need to draw a frame, we simply call the appropriate display list for the walls, floor, and ceiling, and bind the proper textures to each. OpenGL will then replay the drawing calculations done while tessellating, resulting in the geometry being rendered to the screen.

Doors and Windows

For efficiency, we use display lists to render the doors and windows as well. After importing the models, we render them to display lists.

Each time we need to draw a door, we simply call the door's list at the door frame's location. We rotate it so that it is facing the proper direction and so the door's handle is on the opposite side of the hinge we detected earlier. Next we scale the model to fit the door frame. If the scale factor entered by the user is accurate, the resulting door model will now appear to be normally proportioned.

Windows are done exactly the same way, except we don't need to worry about rotating to line up a hinge.

18 OpenGL Programming Guide, chapter 7: http://www.glprogramming.com/red/chapter07.html

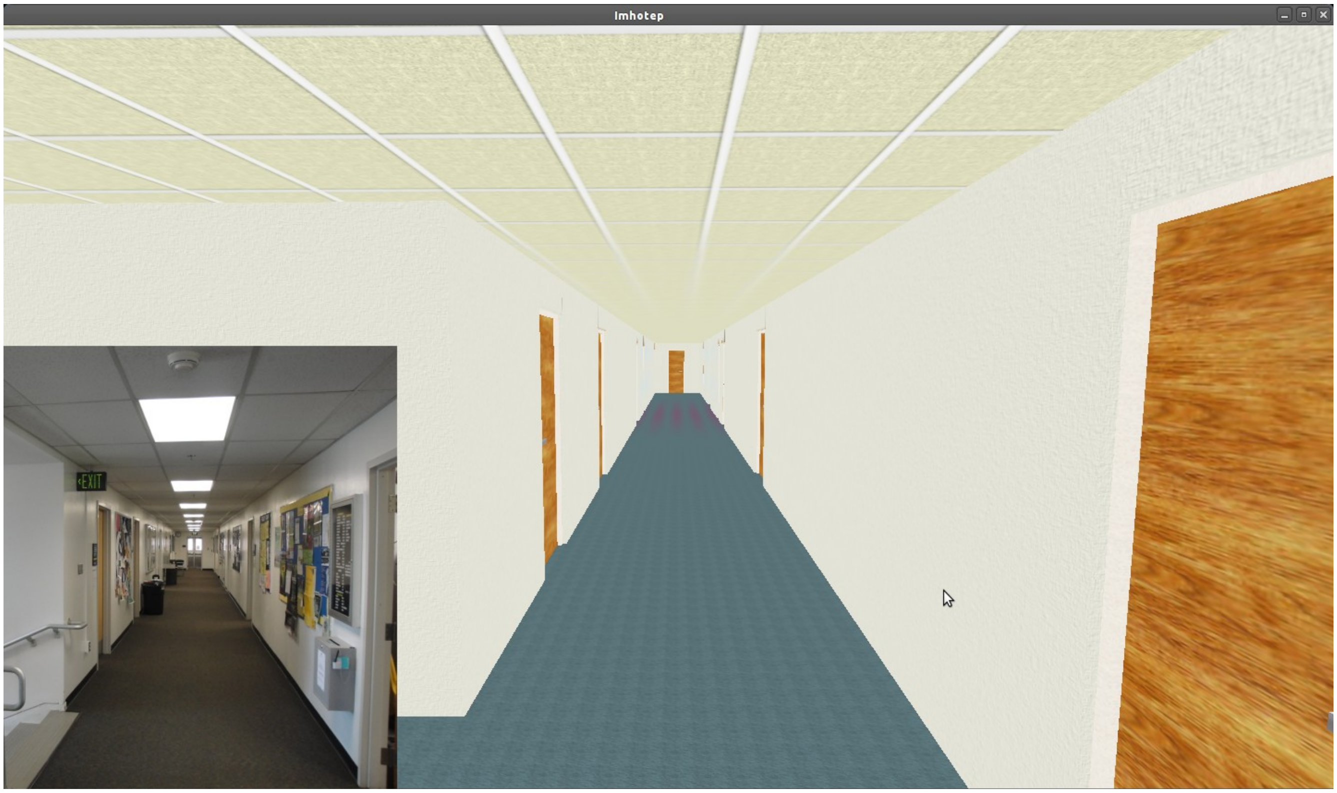

Figure 16: Render of a hallway in Chapman, with photo inset

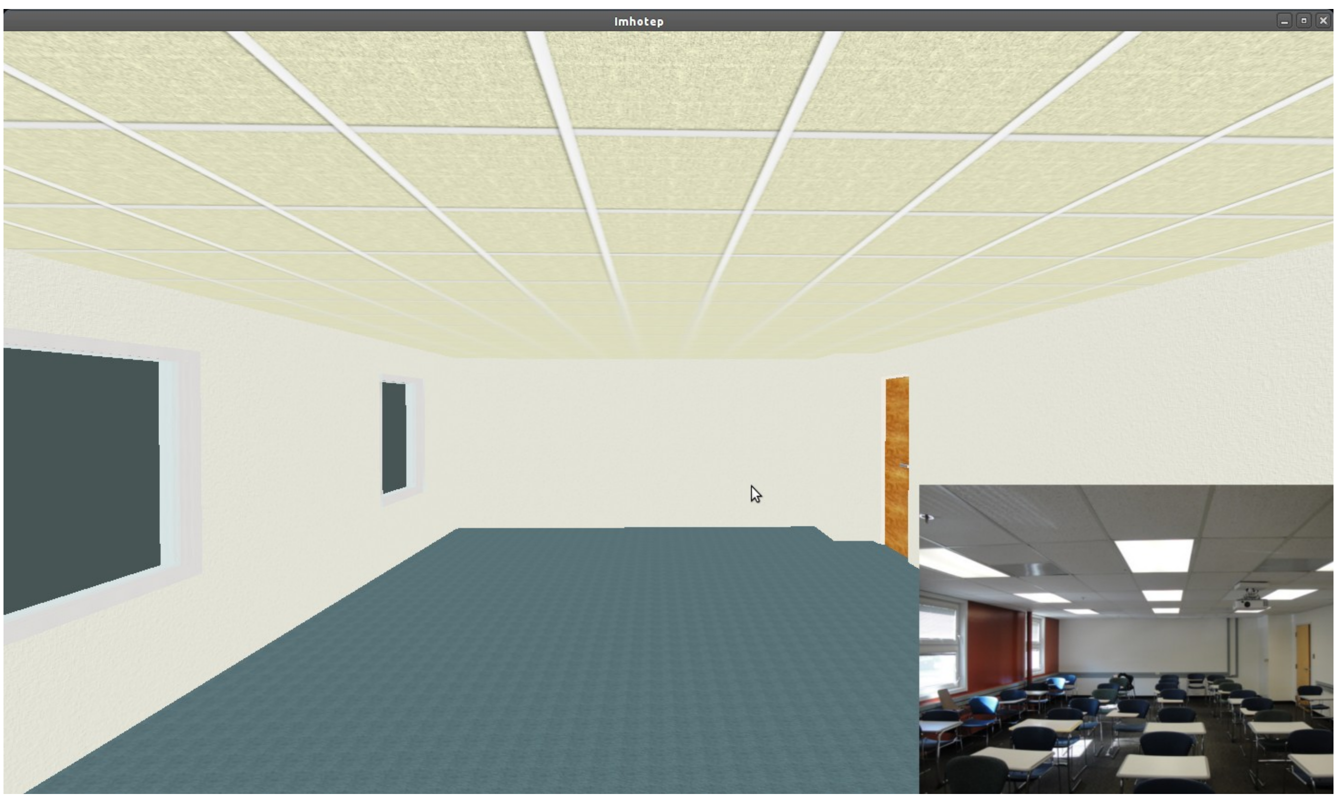

Figure 17: A classroom in Chapman

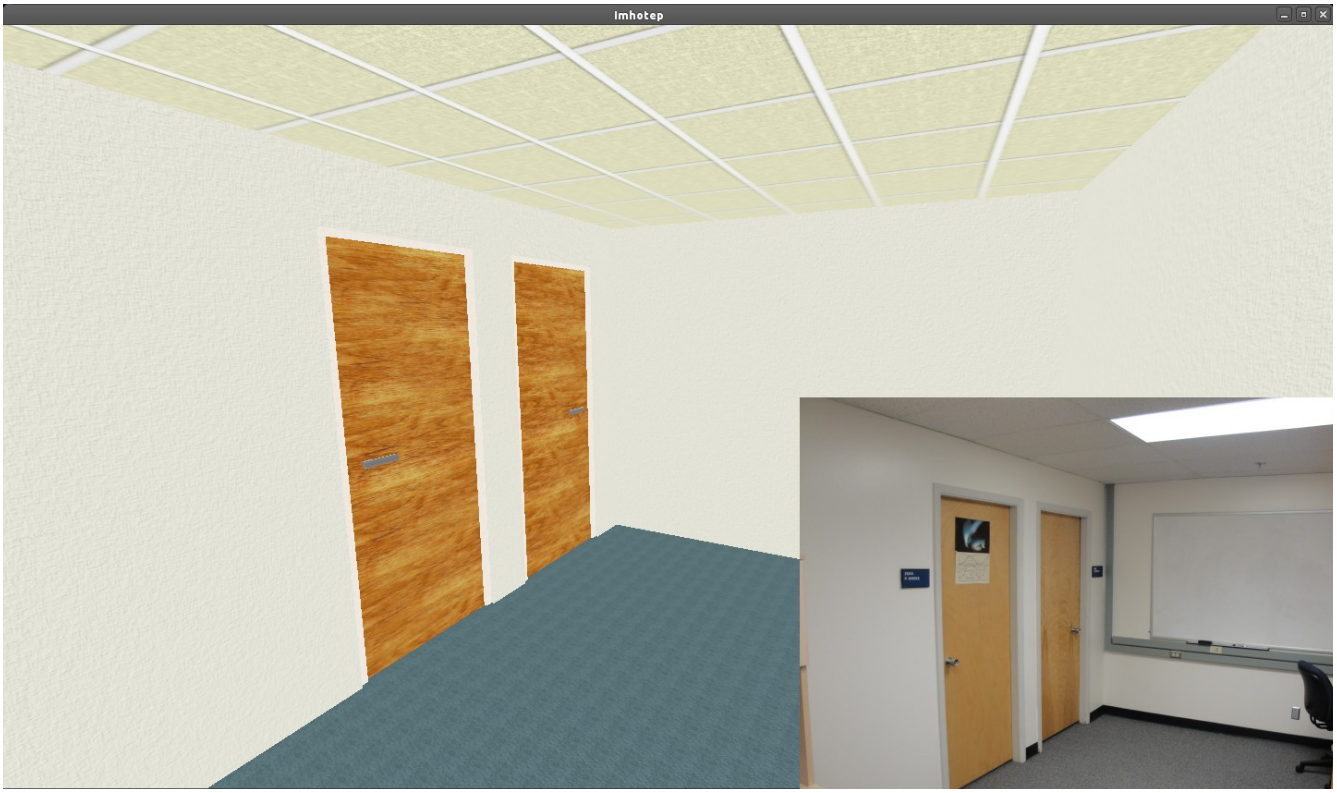

Figure 18: Outside offices in Chapman

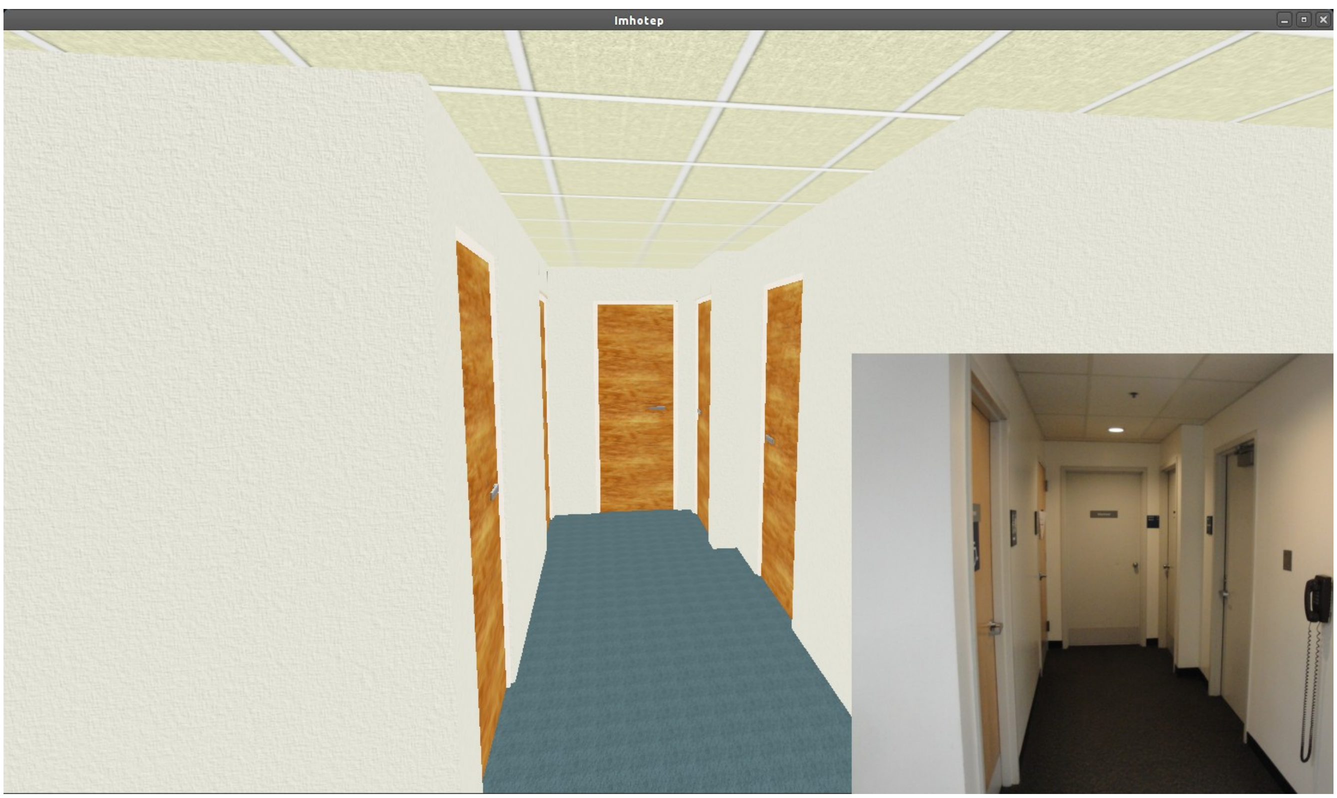

Figure 19: Small alcove in Chapman.

Export

After we finish generating the geometry, we export it to a .obj file. This is a fairly straightforward task, as we designed the internal representation of the geometry with this goal in mind. Basically, we just need to dump the each set of mesh data to a file with .obj formatting.

The .obj format does have one caveat that prevents us from simply writing every single triangle as stored internally. For each mesh, it stores a list of vertexes, a list of texture coordinates, and a list of normals, none of which ought to have duplicate values for the sake of file size. Following this is the actual triangle geometry, which specifies each vertex with indexes into these lists. We store each face with full vertex information, which leads to a lot of duplicate data, as it is very uncommon for a vertex to not be shared by several faces. While we could ignore the convention used by the .obj format and create our lists so that each vertex of each face has a unique entry, this make the output file much larger than it needs to be.

Instead, we remove duplicate vertex information by building hash tables of unique vertexes, texture coordinates, and normals, with indexes into the lists we write to the output file. For each face in the mesh, we query these tables and write the index stored for each vertex.

We repeat this process for every object generated, and write the .obj file to disk. The filename will be the .svg input file's filename with ".obj" appended to the end. (thus, for fp01.svg we write to fp01.svg.obj)

Next we need to export the material library. As we ignore everything specified in this library except the texure filename, we simply need to write the material name, some default color values (we arbitrarily use the same defaults used in *Blender*'s .obj exporter) and the texture filename for each material used to a .mtl file. The naming of this file matches that of the .obj file, substituting the .obj extension for .mtl.

The completed model is now ready for whatever use the user may have for it. If any modifications are required, such as adding features not found in the floor plan, cleaning up the geometry, or combining multiple stories of the same building, they ought to be able to easily do so with 3D modeling software. If nothing else, the exported model should give the user a substantial head start over creating the model from scratch.

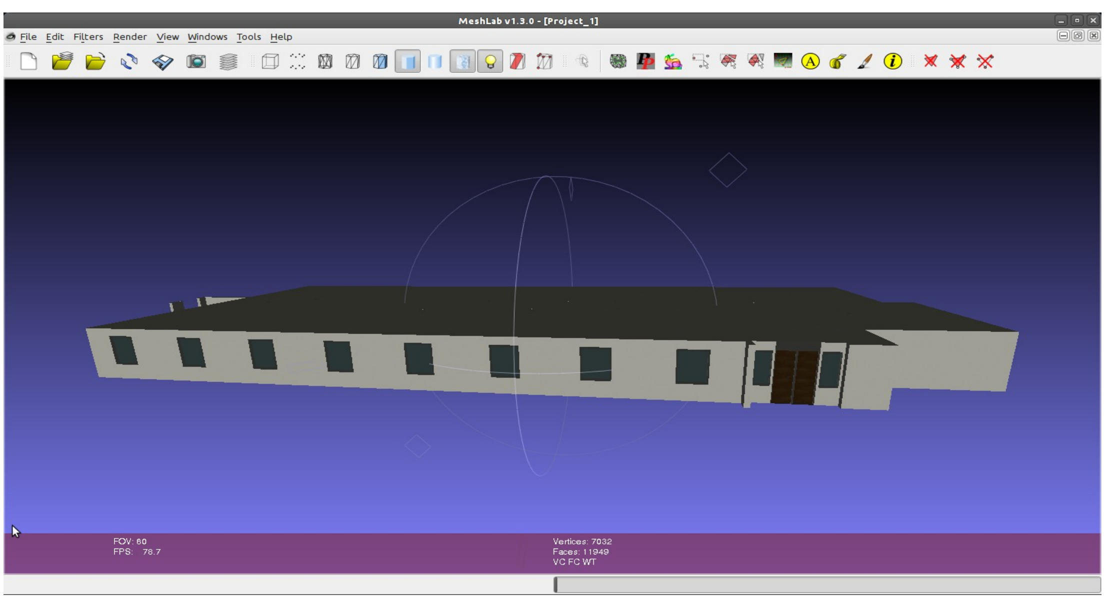

Figure 20: Imhotep's output in MeshLab

Building and Running

Imhotep is written almost entirely in C++. The only exception is the flex and bison code, which themselves contain C++ code.
Imhotep depends on the following 3rd party tools and libraries.

- Flex
- Bison
- libxml++
- libconfig++[19]
- SOIL[20]
- GLUI[21]
- freeglut[22]
- OpenGL[23]
- Orion Lawlor's vector and matrix libraries[24]

All of these libraries are also available in the software repositories of Ubuntu, and probably many other Linux distributions as well, with the exception of Lawlor's vector and matrix libraries. These are included with Imhotep's source for convenience.

Imhotep also makes frequent and heavy usage of the new C++11 standard.[25] As the standard is still very new at the time of this writing, compiler support varies. Imhotep should build on with GCC 4.6 and higher with the -std=c++0x flag. Clang currently supports all the features used as of version 3.1 (which is not yet released as of this writing), though we have not tested Imhotep with Clang. Microsoft Visual Studio still lacks support for for some features, so Imhotep will not compile in Visual Studio without major overhauls. Windows users are recommended to build with MinGW, which we have also not yet tested.[26]

A simple makefile is included with Imhotep's source to automate the build process on POSIX platforms. The only atypical step in Imhotep's compilation is that we first run Flex and Bison, and then rename their outputs to be more consistent with Imhotep's naming conventions.

Imhotep is run with one parameter: the .svg input. If the parameter is omitted, or the file cannot be opened, or if it is not a valid SVG file, Imhotep shows an error message and terminates immediately. Imhotep looks for textures and door / window models in the ./materials directory. If it is unable to load any of these files it will terminate with an error.

Imhotep will output 2 files upon successfully running: <*input filename*>.obj and <*input filename*>.mtl.

---

19 http://www.hyperrealm.com/libconfig/
20 http://www.lonesock.net/soil.html
21 http://www.cs.unc.edu/~rademach/glui/
22 http://freeglut.sourceforge.net/ - We do not use any freeglut specific features, so any other GLUT should work as well.
23 http://www.opengl.org/
24 http://www.cs.uaf.edu/~olawlor/ref/osl/index.html These libraries are included with the Imhotep's source
25 Specifically, we use Initializer lists, auto-typed variables, Right angle brackets, Range-based for-loops and Extended integer types.
26 An up-to-date matrix of C++11 compiler support can be found at http://wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport

Future Work

- Currently, Imhotep is rather limited in what it will accept as valid input. Some of the restrictions we enforce are fairly arbitrary. It would be fairly easy for us to extend the SVG path parser to accept curved paths, and to look for layer identification information in more locations than we do now, such as those used by *Inkscape*.
- Along with this, our element identification algorithms can be extended and improved upon to more robustly identify objects, and to be able to identify more variety in symbols. The current scheme will only work on a small subset of floor plans, and this change will be necessary for us to expand Imhotep to work on a wider set.
- Imhotep could potentially benefit from using fixed-point numbers internally. Both the .svg input and .obj outputs store numbers as text, which is fixed-point, but we currently translate these to and from floating-point, add accumulates roundoff error while processing. The main reasoning we use floating-point is ease of use, as we can use C++ native operations, and because OpenGL is floating-point based. Still, we could fairly easily add support for fixed-point everywhere, and translate to floating-point when making OpenGL calls. This would remove much of the need for our fuzzy matching techniques (though they would still be useful when points from the input are not exactly algined).
- Imhotep currently operates only on a single floor at a time. We could theoretically add the ability for Imhotep to take a set of floor plans, one for each floor of a building, and output a full multi-story building, possibly generating a roof as well. There are some rather difficult problems to overcome before this could be feasible. We found the floor plans we were testing on to not have a common coordinate system. Origins can differ, and the scaling may differ as well. Some method would be needed to align the floors.
  In addition, stairs would be difficult to properly model. This is mostly due to complications due to inherently inaccurate two-dimensional representations of these three-dimensional objects. floor plans tend to use overly simplistic representations for stairs, so extracting the true geometry is challenging. This is further complicated as stair symbols are commonly broken up to represent staircases above one another.
- The built in rendering engine is extremely simplistic. We consider this renderer to be essential, despite the fact that Imhotep's primary goal is to produce a model file to be used by other programs, since the preview allows the user to do a walkthrough of the model to look for potential problems, such as a door that had failed to be detected. Still, this is no reason for Imhotep to not have a more featured renderer. Particularly useful would be more realistic lighting. We do, however, desire Imhotep to be a fairly lean program, and to be capable of running on as many platforms as possible, so care must be taken to avoid introducing unnecessary features.

Conclusion

Imhotep has shown itself to successfully produce good quality models when the input meets its requirements. Even when cleanup is necessary to meet these, there is still a substantial time savings when compared to creating a model from scratch. The entire process can be done in seconds, and the overwhelming majority of the time is spent waiting on user input; the actual automated portions happen imperceptibly fast.

Imhotep ought to be a useful tool in many situations, such as architecture, simulation, game design, etc. While there is still much work to be done on expanding the feature set and enabling support of a wider variety of floorplan symbols, Imhotep's current form is an excellent start in that direction.