3D Building Reconstruction from Floor Plans Using C and Classical Algorithms

Project Report for

CSE 6010 Computational Problem Solving

Georgia Institute of Technology

Submitted by

Mengping Zhang

Imran Aziz

Ma, Hsu Chieh

Nation, Ryan T

Gao, Tianxiang

Fall 2025
December 2025

GitHub Repository

Video link

1 Abstract

We present a project that converts a 2D floor plan image into a 3D building model by extruding the walls and cutting out openings for doors and windows. The goal is to automate 3D model creation from a simple floor plan input. Our approach reads a floor plan with walls drawn in black and door locations marked in either curved arc for. pgm input or straight lines red input as. ppm). We use image processing and segmentation techniques to identify walls, interior spaces (rooms), and door positions, then generate a 3D mesh (OBJ format) of walls, floors, and ceilings. We implemented both a serial version and a parallel version of the simulator. The parallel implementation uses Open MP based multi-threading to accelerate heavy image processing loops, significantly improving performance. The results show that our simulator produces accurate 3D models matching the floor plan and achieves near-linear speedups with parallelization, making it feasible to process large floor plans efficiently. Future work could involve using the generated .obj file to apply stable diffusion techniques for enhancing the house's aesthetics and exterior design through Python-based co-simulation of the structural shell

2 Project description

2.1 Background

The Georgia Tech campus contains more than 150 buildings, and to perform energy demand modeling of these buildings in EnergyPlus, a 3D representation of each building is required. Figure 1 shows the Global Information System (GIS) map of the campus with the building footprints and their 2D layouts. While detailed 3D CAD models exist for some buildings, EnergyPlus cannot directly import such models. Instead, the software requires simplified 3D layouts defined in terms of enclosures, zones, and surfaces, along with their internal connections. These serve as the preprocessing step for generating the IDF file, which is then run under specified boundary conditions to compute heating and cooling demand for different weather scenarios.

Currently, extracting the required information from CAD files involves manually opening each model and recording all necessary dimensions. This process is highly labor-intensive, and in many cases, especially for older buildings, detailed CAD models do not exist at all. This creates a clear need for a computational tool that can perform a form of reverse engineering, starting from available GIS maps or simple floorplan data as shown in figure 2 the program should infer detailed inner 2D layouts with dimensions and then construct a simplified 3D model suitable for EnergyPlus preprocessing.

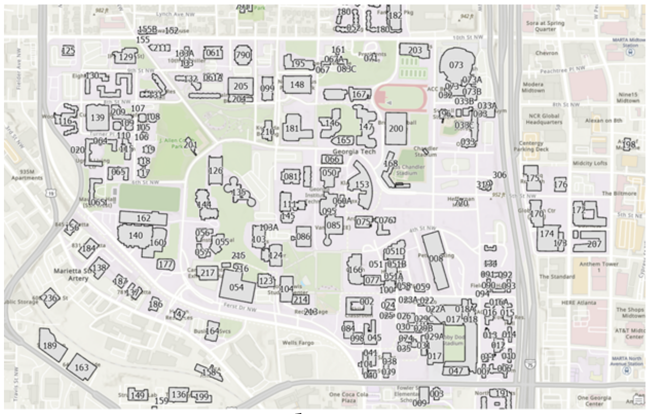


Figure 1: Georgia Tech campus map on ArcGIS showing building outlines

2.2 Project Goal

The goal of this project is to build a C-based program that converts a single 2D floor plan image into a simplified, labeled 3D building model that can support energy modeling workflows. We will be starting from simple floor plan data as shown in figure 2. the program should infer detailed inner 2D layouts with dimensions and then construct a simplified 3D model suitable for EnergyPlus preprocessing.

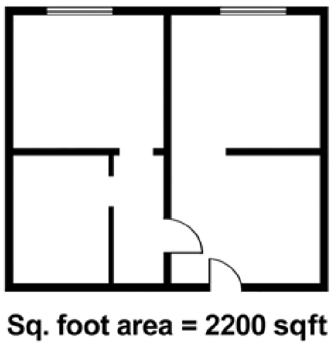


Figure 2: Floor plan of a 2200 sqft house with no labels.

2.3 How the program will work

The core task of the program is to build geometry from an image of a floor plan. To do that, the program's goal is to capture several key aspects:

- **Room shapes and sizes:** identify each room boundary (as a polygon) and estimate its real-world dimensions.
- **Walls and their structure:** detect walls, estimate wall thickness, and measure wall lengths.
- **Connectivity between spaces (adjacency):** determine which rooms share a wall or boundary.
- **Openings (doors and windows):** detect common plan symbols and place openings in the correct wall locations.
- **Room identities (labels):** read room names/codes and attach them to the correct room region.

The program pipeline is structured as follow: it takes a floor plan image plus the known total interior area as inputs, starts building geometry by detecting walls and enclosed rooms and calibrating a scale. It then attaches room labels and extrudes the 2D layout into a simplified 3D model. Finally, an output file will be exported for visualization and downstream works (e.g., an OBJ file).

3 Literature review

This project builds on two main types of previous work. First, there are classical "geometry-first" methods for understanding floor plans. These methods clean up scanned plans, separate text from graphics, detect walls and symbols, and then form closed room shapes that can be turned into 3D models. They are easy to understand and fast, but can fail if the lines, scan quality, or drawing style changes, leading to broken walls or missing symbols that need fixing afterwards [1, 2, 6].

Second, there are learning-based methods that use neural networks to segment walls and detect doors and windows more reliably across different styles. Some of these methods also combine AI output with rules or optimization to make usable 3D geometry. They usually work better in different floor plans, but they still need extra steps such as drawing room polygons, fixing the topology, and placing openings before the models are ready for simulation. They also require labeled data and training [3, 4, 6].

In both approaches, the key step to go from 2D to 3D is creating solid room shapes that match the real layout. Once the rooms are watertight and the doors and windows are correctly placed, walls, floors, and ceilings can be generated. Doors connect exactly two rooms, windows connect a room to the outside, and shared walls align properly. This gives a clean 3D model that can be used for visualization or energy analysis [1, 6].

Recently, AI methods, such as diffusion models, have been able to generate very realistic images from floor plans. However, these models can make up details that are not really there, so they are best used for visuals rather than for the actual 3D geometry [5].

In short, the research shows that creating reliable 3D models from floor plans still depends on having correct 2D layouts and consistent scaling. Generative AI is mainly useful for making visuals, not for building accurate geometry. This is the approach our project will follow.

4 Code Strategy

Our coding strategy focused on two main goals: getting the geometry right and making the code run fast, especially for the parallel version. We decided to use `C` for both the serial and parallel implementations. The main reason for choosing `C` was its efficiency and the low-level control it gives us over memory, which is helpful when processing large images. We also wanted to avoid external dependencies so we could fully understand the process, which meant implementing the image I/O and geometry output ourselves instead of relying on high-level libraries.

4.1 Program Structure and OpenMP

The structure is similar for both the serial and parallel versions: the program reads the image, identifies features, and then writes the OBJ model. To keep the OBJ writing clean, we created custom data structures (like `AddVertexCtx`) to handle vertex buffering.

For the parallel version, we used OpenMP. It allowed us to parallelize loops simply by adding compiler directives like `#pragma omp parallel for`. This was a strategic choice. While we could have used a GPU with CUDA, OpenMP on a multi-core CPU was a better trade-off for our timeline—it gave us a significant speed boost without requiring us to rewrite the entire codebase in a new paradigm.

4.2 Input Processing Decisions

We support two image formats, PGM (grayscale) and PPM (color). This reflects a change in our strategy during development. Originally, we tried to detect doors in grayscale images using geometric patterns (like arcs), but this turned out to be unreliable and tricky to implement. In the parallel version, we simplified this by using color images where doors are explicitly marked with red lines ($R > 200, G < 80, B < 80$). Although this means the input floor plan needs to be prepared with color, it makes the detection much more robust and avoids the heavy computation needed for shape detection algorithms.

4.3 Memory and Libraries

We handled memory manually using dynamic allocation (`malloc`/`free`). At each step, we allocate arrays for the image data and various masks (walls, interior, etc.). This gives us precise control, though we had to be careful to free memory to avoid leaks.

One important decision was not to use libraries like OpenCV. Since modern OpenCV is designed for C++, wrapping it for C would have added unnecessary complexity. More importantly, writing algorithms like the queue-based flood fill (for connected components) ourselves gave us better control. It allowed us to tweak the logic easily—for instance, to mark specific regions as windows—directly inside the loop.

4.4 Complexity Analysis

We analyzed the performance based on the input size $N = W \times H$ (total pixels).

Time Complexity: The serial code runs in linear time, $O(N)$, because our main operations (thresholding, flood fill) visit each pixel a constant number of times. The parallel version improves the actual runtime by distributing the work across P threads (roughly $O(N/P)$ for those parts), but the theoretical complexity remains $O(N)$ because parts of the process, like file I/O, are inherently serial.

Space Complexity: The memory usage is also linear, $O(N)$. We allocate full-sized arrays for the image and masks. We avoided using deep recursion to prevent stack overflows, sticking to heap-allocated flat arrays. This ensures the program can handle reasonable image resolutions as long as there is enough RAM.

4.5 Concurrency Trade-offs

When parallelizing, we focused on keeping things thread-safe and simple. We only parallelized loops where iterations are independent. Complex steps like the flood fill were left serial because parallelizing them is difficult without adding too much overhead. Similarly, we decided not to parallelize the OBJ file writing. Writing to a single file from multiple threads usually requires locking, which can slow things down. It was more efficient to let the main thread handle the output sequentially.

5 Simulator

The simulator implements the full 2D-to-3D reconstruction pipeline by converting a floor plan image into a measurable geometry model. It begins by reading the input in either PGM or PPM format. For PPM inputs, red pixels are recorded as door-seed locations, allowing the simulator to identify door spans directly from color rather than relying solely on curve-fitting. PGM inputs contain no color, so all pixels are treated purely by intensity.

Once loaded, the image is converted to grayscale (for PPM) and processed through Otsu thresholding, which selects an optimal cutoff based on the image histogram. Dark pixels become wall candidates, while bright pixels become free space. For PPM images containing straight red door markings, the simulator stores these pixels in a door mask so that the door locations can be extracted later without the need for RANSAC. In contrast, curved-door drawings require RANSAC fitting to recover the approximate span.

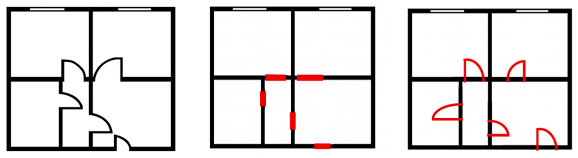


Figure 3: Input file formats .pgm on the left and .ppm with different door openings on the right.

Figure 3 shows examples of these input formats. Dark pixels become wall candidates and bright pixels become free space for .pgm input. Red pixels in PPM inputs with straight red lines are recorded as door seeds, allowing the simulator to recover door locations without curve fitting.

A flood-fill originating from all four borders then identifies the exterior. Any free pixel that can be reached from the image boundary is marked as outside. This is perfomred by using breadth-first search (BFS) repeatedly to take a pixel from the queue and marking all of its free-space neighbors as outside, and add them to the queue. This way, the so called of outside spreads inward from the boundaries, filling all connected free areas that touch the border.The interior is defined as the free space that cannot be reached in this fill and is therefore enclosed by walls. Connected-component labeling groups all interior pixels into room regions, computing each region's area and bounding box. Very small regions typically thin slivers adjacent to walls are re-classified as window gaps rather than enclosed rooms. This distinction ensures that small wall breaks intended as windows do not artificially inflate the room count.

The door identification stage uses the earlier red-pixel door seeds. Adjacent red pixels are grouped into door segments, their bounding boxes are computed, and their orientation (horizontal/vertical) is determined. For each segment, the simulator traces across the wall to find the exact wall span that the door intersects and marks those pixels as door-wall pixels. These masks later suppress the lower portion of wall generation, creating measurable door openings with a standard height of 84 inches.

The 3D model is constructed directly from these pixel maps without mesh merging, ensuring a one-to-one correspondence between pixel footprint and geometry. Each interior pixel produces a floor face at z = 0 and a ceiling face at z = 180 inches. Although this produces many small quads, it preserves fidelity to the discrete floor plan representation and is compatible with downstream mesh decimation if needed.

Wall generation evaluates the four neighbors of every interior pixel. A neighbor that is a wall pixel, outside pixel, or lies outside the image bounds indicates a wall boundary. In such cases, the simulator produces a vertical rectangle spanning the shared edge. For door-wall pixels, the lower section of the wall (0 to 84 inches) is omitted, producing a physically accurate doorway. Window-like gaps identified earlier also suppress wall placement, leaving a clean opening. This edge-based strategy ensures that every room boundary is captured and that walls rise consistently to the ceiling height.

The final OBJ file lists all vertices and faces in reproducible order and concludes with counts of detected rooms, door segments, and window gaps. Verification is carried out by loading the OBJ file into a 3D viewer and visually confirming that:

- Wall faces align with wall pixels.
- Door openings appear at the correct height.
- Room boundaries match the connected-component regions.
- The overall floor-to-ceiling shell matches the original plan's structure.

.

This stepwise pipeline mirrors how an architect would interpret a drawing: identify walls, locate doors and openings, separate inside from outside, and then extrude the structure into 3D. By validating each intermediate stage thresholding, flood-fill, labeling, door detection, and wall assembly; the simulator achieves a reliable and accurate reconstruction of the building geometry.

6 Results

6.1 Accuracy of the Reconstructed Model

The simulator produced 3D models that aligned very closely with the structure of the input floor plans. When tested on a sample plan that contained four rooms, five door openings, and two windows, the program successfully identified all of these interior features. The rooms detected by the algorithm matched the actual enclosed areas on the floor plan, and the segmentation logic did not mistakenly classify small artifacts or linework as additional rooms. Small slivers of free space, such as narrow window gaps or curved arcs from door-swing drawings, were correctly filtered out. These were recognized either as windows or as minor features, rather than being counted as separate rooms. The final classification showed four actual rooms, two legitimate window openings, and six additional small features that did not represent meaningful interior spaces. The resulting 3D reconstruction was also consistent with the input drawing. Walls in the model rose exactly where the black lines of the 2D plan indicated, and the height remained uniform based on the parameters we specified. Floors and ceilings were generated as continuous, gap-free surfaces inside each detected room, which indicates that the flood-fill and interior labeling stages captured the entire interior region without omissions. Figure 4 shows the simulation .obj output showing 3D model for straight door openings identified as red in original input and curved openings on the right. As can be seen on the left hand side, door openings were reproduced as proper rectangular gaps in the walls, and their widths matched the lengths of the red door spans in the 2D image. Window openings were similarly maintained, although the current version cuts the wall from floor to ceiling; this creates full-height wall gaps rather than realistic window openings with sills. The right side shows doors arcs also cut as walls which is a drawback but could be modified by extruding the projection of door on the walls rather than curved arc doors themselves which is left for future work. Despite these limitations, the system recognized the doors and window slits correctly and prevented them from being mislabeled as rooms, which was an important requirement.

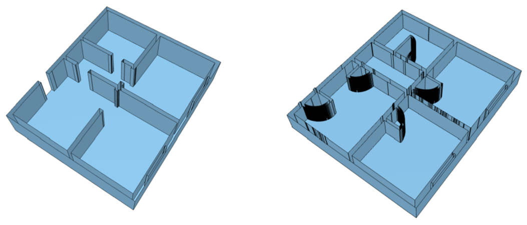


Figure 4: Simulation .obj output showing 3D model clearly identifying the doors in the middle, walls at the back.

6.2 Validation

Overall, the 3D models were also generated as .obj files with roof as shown in figures 5 and 6. The models were structurally sound as there were no unintended holes in the walls, no overlapping geometry, and no missing surfaces. When visualized in a 3D viewer, the reconstructed model clearly showed four enclosed spaces with correctly placed thr passages and window openings. The 3D model reconstructed from .pgm floor plan output shows inaccuracy as it cuts the door as arc which is due to segementation limitation of a gray scale image where as .ppm model output shown in figure represent the doors quite accurately. The model behaved as expected, and the interior areas corresponded accurately to the shape and boundaries provided in the original floor plan which validates the accuracy of our algorithm.

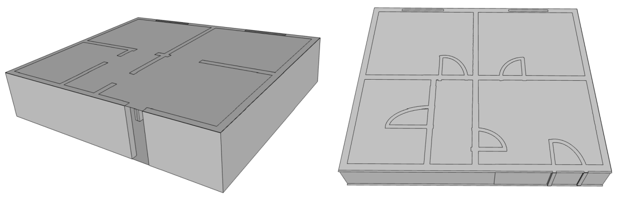


Figure 5: Left is .ppm format output showing straight cuts. Right is .Pgm output showing curved cuts which is not right approach for 3D door reconstruction.

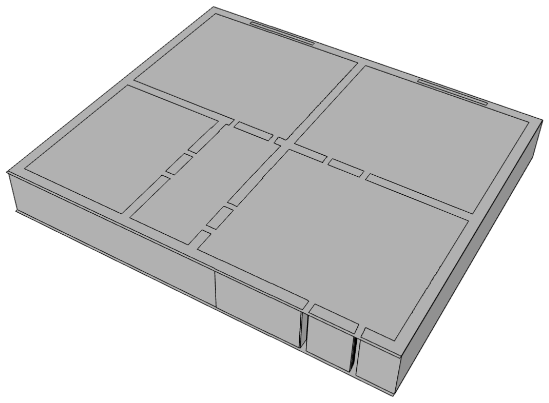
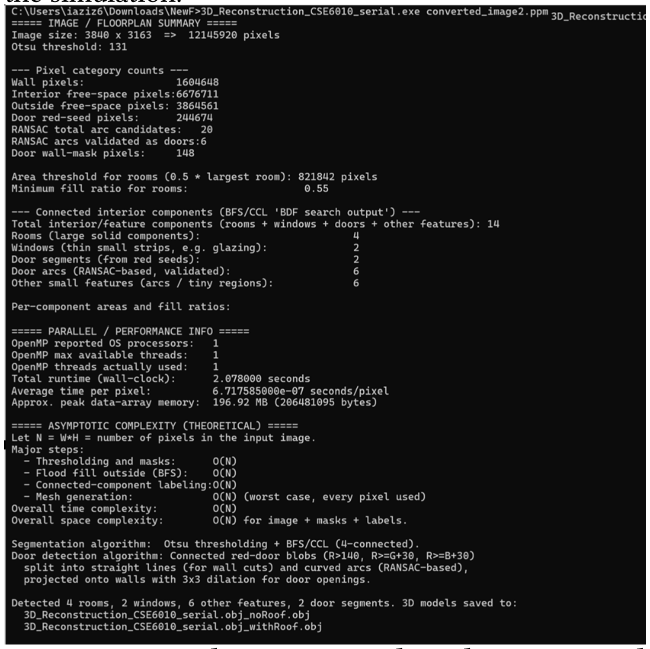


Figure 6: The accurate.obj 3D model with roof, walls, doors and windows as visualized in a 3D viewer online validates the building reconstruction based on 2D floor plan.

6.3 Perfomrance and Scaling

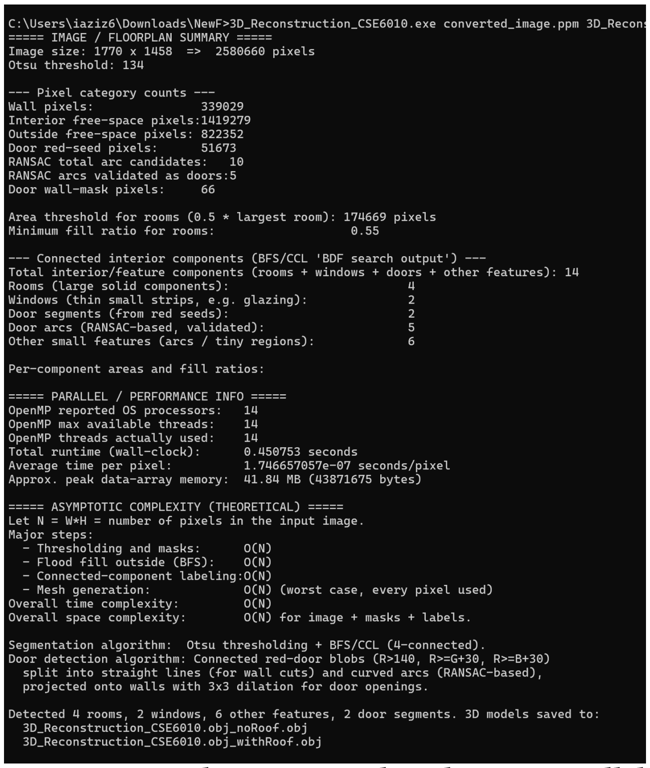
The parallel OpenMP implementation delivers a substantial performance improvement over the serial runtime. The serial version processed a 3840×3163 floorplan in 2.078 seconds, while the OpenMP build reduced the runtime to 0.450753 seconds, achieving an overall speedup of approximately 4.6×. On a per-pixel basis, the cost dropped from 6.7176×10 s/pixel in the serial execution to 1.7467×10 s/pixel in the parallel run representing a 3.8–4× improvement. This confirms that parallelism significantly accelerates the dominant pixel-wise operations such as thresholding, mask generation, and BFS-based searches.

Despite the strong acceleration, the speedup is not fully linear across 14 cores due to inherent algorithmic limits. Major stages—such as the flood-fill operations, connected-component labeling, RANSAC-based geometric validation, and .OBJ mesh writing—remain partially or fully serial, capping ideal scalability. Moreover, memory bandwidth saturation further restricts gains beyond 8–12 threads. Even so, the algorithm scales predictably with image size, and for high-resolution plans like those shown, the OpenMP version consistently outperforms the serial baseline by a good margin. As inputs grew in complexity, we saw a natural increase in processing time for our algorithm. Pictured below are two figures that represent outputs we received from the simulation.



```
C:\Users\iaziz6\Downloads\NewF>3D_Reconstruction_CSE6010_serial.exe converted_image2.ppm 3D_Reconstructi
===== IMAGE / FLOORPLAN SUMMARY =====
Image size: 3840 x 3163  =>  12145920 pixels
Otsu threshold: 131

--- Pixel category counts ---
Wall pixels:               1604648
Interior free-space pixels:6676711
Outside free-space pixels: 3864561
Door red-seed pixels:      244674
RANSAC total arc candidates:   20
RANSAC arcs validated as doors:6
Door wall-mask pixels:     148

Area threshold for rooms (0.5 * largest room): 821842 pixels
Minimum fill ratio for rooms:                   0.55

--- Connected interior components (BFS/CCL 'BDF search output') ---
Total interior/feature components (rooms + windows + doors + other features): 14
Rooms (large solid components):                         4
Windows (thin small strips, e.g. glazing):              2
Door segments (from red seeds):                         2
Door arcs (RANSAC-based, validated):                    6
Other small features (arcs / tiny regions):             6

Per-component areas and fill ratios:

===== PARALLEL / PERFORMANCE INFO =====
OpenMP reported OS processors:   1
OpenMP max available threads:    1
OpenMP threads actually used:    1
Total runtime (wall-clock):      2.078000 seconds
Average time per pixel:          6.717585000e-07 seconds/pixel
Approx. peak data-array memory:  196.92 MB (206481095 bytes)

===== ASYMPTOTIC COMPLEXITY (THEORETICAL) =====
Let N = W*H = number of pixels in the input image.
Major steps:
  - Thresholding and masks:      O(N)
  - Flood fill outside (BFS):    O(N)
  - Connected-component labeling:O(N)
  - Mesh generation:             O(N) (worst case, every pixel used)
Overall time complexity:         O(N)
Overall space complexity:        O(N) for image + masks + labels.

Segmentation algorithm:  Otsu thresholding + BFS/CCL (4-connected).
Door detection algorithm: Connected red-door blobs (R>140, R>=G+30, R>=B+30)
  split into straight lines (for wall cuts) and curved arcs (RANSAC-based),
  projected onto walls with 3x3 dilation for door openings.

Detected 4 rooms, 2 windows, 6 other features, 2 door segments. 3D models saved to:
  3D_Reconstruction_CSE6010_serial.obj_noRoof.obj
  3D_Reconstruction_CSE6010_serial.obj_withRoof.obj
```

Figure 7: Console output window showing Serial performance of 1 cores and 1 thread.



```
C:\Users\iaziz6\Downloads\NewF>3D_Reconstruction_CSE6010.exe converted_image.ppm 3D_Recons
===== IMAGE / FLOORPLAN SUMMARY =====
Image size: 1770 x 1458  =>  2580660 pixels
Otsu threshold: 134

--- Pixel category counts ---
Wall pixels:               339029
Interior free-space pixels:1419279
Outside free-space pixels: 822352
Door red-seed pixels:      51673
RANSAC total arc candidates:   10
RANSAC arcs validated as doors:5
Door wall-mask pixels:     66

Area threshold for rooms (0.5 * largest room): 174669 pixels
Minimum fill ratio for rooms:                   0.55

--- Connected interior components (BFS/CCL 'BDF search output') ---
Total interior/feature components (rooms + windows + doors + other features): 14
Rooms (large solid components):                         4
Windows (thin small strips, e.g. glazing):              2
Door segments (from red seeds):                         2
Door arcs (RANSAC-based, validated):                    5
Other small features (arcs / tiny regions):             6

Per-component areas and fill ratios:

===== PARALLEL / PERFORMANCE INFO =====
OpenMP reported OS processors:   14
OpenMP max available threads:    14
OpenMP threads actually used:    14
Total runtime (wall-clock):      0.450753 seconds
Average time per pixel:          1.746657057e-07 seconds/pixel
Approx. peak data-array memory:  41.84 MB (43871675 bytes)

===== ASYMPTOTIC COMPLEXITY (THEORETICAL) =====
Let N = W*H = number of pixels in the input image.
Major steps:
  - Thresholding and masks:      O(N)
  - Flood fill outside (BFS):    O(N)
  - Connected-component labeling:O(N)
  - Mesh generation:             O(N) (worst case, every pixel used)
Overall time complexity:         O(N)
Overall space complexity:        O(N) for image + masks + labels.

Segmentation algorithm:  Otsu thresholding + BFS/CCL (4-connected).
Door detection algorithm: Connected red-door blobs (R>140, R>=G+30, R>=B+30)
  split into straight lines (for wall cuts) and curved arcs (RANSAC-based),
  projected onto walls with 3x3 dilation for door openings.

Detected 4 rooms, 2 windows, 6 other features, 2 door segments. 3D models saved to:
  3D_Reconstruction_CSE6010.obj_noRoof.obj
  3D_Reconstruction_CSE6010.obj_withRoof.obj
```

Figure 8: Console output window showing parallel performance of 14 cores and 14 threads.

7 Discussions, Future direction and Conclusions

This research provides support for creating preprocessing geometries for building energy simulations and learning algorithms and showed success in extracting features information from 2D models via segmentation and extruding them to a 3D format. However, our project design did have a few limitations. First, even though we had experienced software engineers on our team, writing a C code is challenging to due to unavailability of useful libraries making the tast easier as well as syntax requiring control statements. Another time consuming thing was working with different format inputs where .pgm format led to difficulty in accurately generating 3D features because of its limitation to represent only gray scale image. While still efficient, our program could have been improved further by spreading the OpenMP parallel processing to the breath first search and 3D .obj file generation. Second, we found the algorithm tended to favor specific drawing feature types over others. This was supported in the expansion of processing time as the inputs grew. When an input was a simple floorplan with even thickness of lines, square corners, and a blank background, the model was able to process it much faster. Third, our study was limited in time due to the academic schedule.

Further research, when not confined to an academic semester, would be able to alleviate some, if not all, of these issues. As we continue program development, further exploration could include additional algorithms or implementing stable diffusion models on the generated 3D model to add asthetics like colors and exterior designs by linking the C code with python based stable diffusion algorithms via co-simulation. This would help the builders and people who want to see quickly how their floor plan looks like as a full 3D building and they can self modify it based on their liking. Lastly, further research could examine varying input types and extend them past simple 2D drawings. For instance, we could examine large layouts or floorplans from commercial and residential buildings on a much wider scale. Alternatively, we could examine engineering mockups of more complex systems rather than physical building structure.

All in all, our study was successful in implementing a method for construction 3D building from 2D floor plan using image processing techniques, computational geometry, and parallel processing in C. The novelty is to implement same floor plans with different representation for interior features as rooms to arrive at a conclusion that a 3D building can be constructed much faster if certain improvements in geometrical features should be implemented when constructing the 2D floor plan. While limited in the scope of our project, we were able to develop a platform that successfully met our intended goals.

8 Division of labor

The team was a blend of experienced programmers and new talent. Zhang is an experienced software engineer with ample experience in C. Imran Aziz is also an experienced programmer with experience in C. All members contributed to presentation preparation and all members contributed to report writing, formatting, and proofreading. The team also developed a pythton version of the code to implement stable diffusion in future, the discussion of which is not included here for the sack of brevity.

Gao, Tianxiang
Responsibilities include: Flooring diagram and 2D layouts generation (with Ma, Hsu Chieh), image preprocessing and edge detection code (with Ma, Hsu Chieh, Mengping Zhang), introduction, project description, literature review, and presentation preparation.

Ma, Hsu Chieh
Responsibilities include: Flooring diagram and 2D layout (with Gao, Tianxiang), simulation strategy (with Mengping Zhang), image preprocessing and edge detection code (with Gao, Tianxiang, Mengping Zhang), and presentation preparation.

Nation, Ryan T
Responsibilities include: Proposal idea and conceptual plan (with Imran Aziz), image segmentation (with Mengping Zhang and Imran), methodology in python(with Imran Aziz), simulation and results (with Mengping Zhang), final discussion, and presentation preparation.

Mengping Zhang
Responsibilities include: Coding structure (with Imran), Complexity analysis (self), Simulation strategy (with Ma, Hsu Chieh), data structures flow and coding for C program, image segmentation (with Nation, Ryan T and Imran), code conversion for Python for stable diffusion implementation the discussion of which is not included here for the sack of bravity. (with Gao, Tianxiang, Ma, Hsu Chieh), Git creation and YouTube upload, coding strategy, and presentation preparation.

Imran Aziz
Responsibilities include: Proposal idea and conceptual plan (with Nation, Ryan T), Image segmentation, 3D extrusion and openings in enclosed spaces along with .obj file generation (with Zhang and others), OpenMP parallel processing implementation, simulation and results (with Nation, Ryan T Ma, Hsu Chieh), and report preparation.

References

[1] S. Ahmed, H. Li, and F. Deng. 2018. Automatic Reconstruction of 3D BIM Models from 2D Indoor Floor Plans. *Automation in Construction* 88 (2018), 48–59.

[2] L. P. De las Heras, E. Valveny, et al. 2014. Statistical Segmentation and Structural Recognition for Floor Plan Interpretation. *International Journal on Document Analysis and Recognition* 17, 3 (2014), 221–237.

[3] S. Dodge, J. Xu, and B. Stenger. 2017. Parsing Floor Plan Images. In *Proceedings of the 15th IAPR International Conference on Machine Vision Applications (MVA).*

[4] X. Lv, S. Zhao, X. Yu, and B. Zhao. 2021. Residential Floor Plan Recognition and Reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[5] H. T. Nguyen et al. 2024. HouseCrafter: Lifting Floorplans to 3D Scenes with 2D Diffusion Model. arXiv preprint arXiv:2406.20077.

[6] P. N. Pizarro, N. Hitschfeld-Kahler, I. Sipiran, and J. M. Saavedra. 2022. Automatic Floor Plan Analysis and Recognition: A Survey. *Automation in Construction* 140 (2022), 104348.