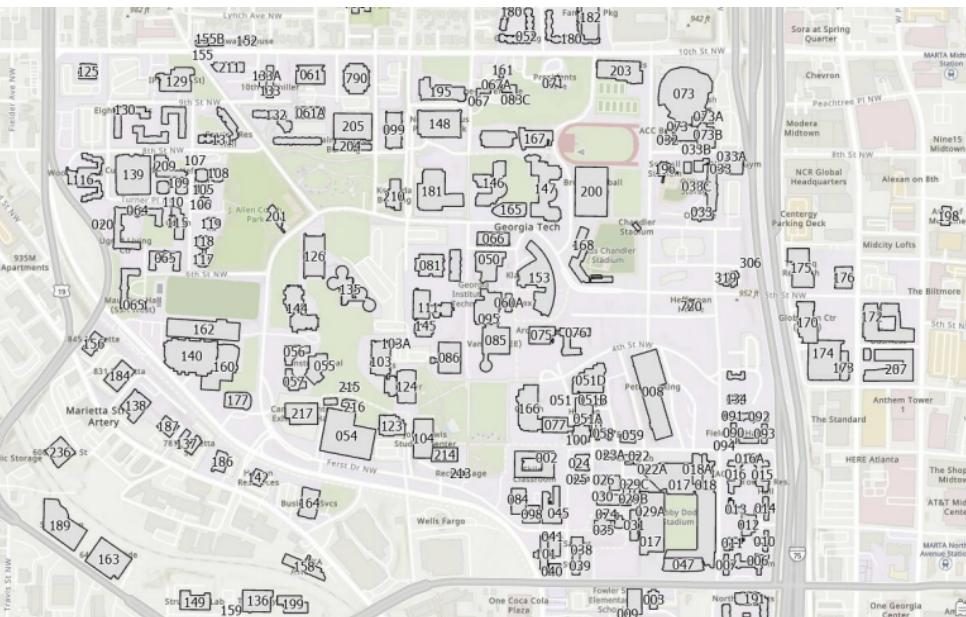


3D Building Reconstruction from Floor Plans Using C and Classical Algorithms

Hsu Chieh Ma, Imran Aziz, Mengping Zhang, Ryan Travis Nation, Tianxiang Gao

Why Constructing 3D Geometry from Floor Plans?

- Campus energy analysis needs building geometry to run simulations.
 - Tools like EnergyPlus require simplified 3D models of rooms as inputs.
 - Today, this geometry is often built manually: rooms, walls, connections.
 - We intend to automate the process by using parallel processing to scale up the floor plan for a 3D surface model for easy preprocessing input for energy simulation tools



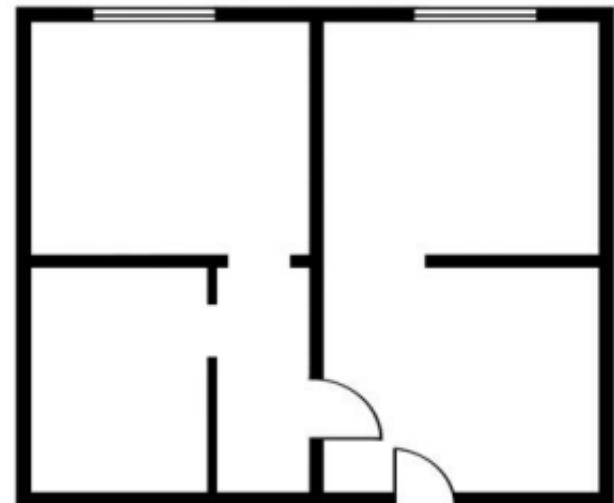
What the Program needs to “Understand” from Floor Plans?

Project Goal:

- Build a C-based program that converts 2D floor plan images into 3D models.
- Make use of classical data structures and algorithms.

The Program Should Detect:

- Room shapes and sizes
- Walls and their structure
- Connectivity and windows
- Openings such as doors



Sq. foot area = 2200 sqft

Literature Foundations: Two Frameworks

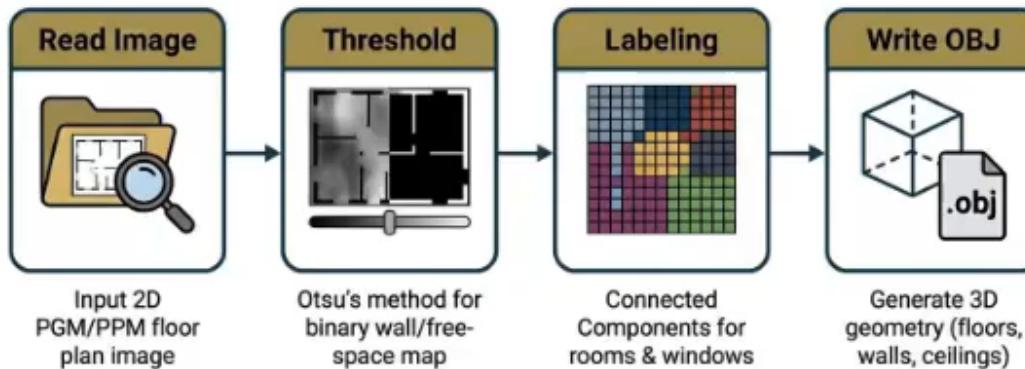
Geometry-first Methods:

- Determinant rules to detect walls and openings from shapes.
- Pros: fast, interpretable.
- Cons: sensitive to input quality, needs manual fixes.

Learning-based Methods:

- Trained Neural Networks, e.g. Apple RoomPlan.
- Pros: robust across different floor plan formats.
- Cons: needs labeled training data.

Core Strategy & Architecture



Objectives:

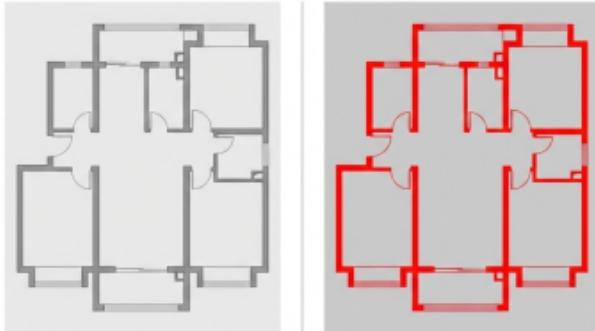
- Geometric Correctness (Output matches floor plan).
- Computational Performance (Speed focus).

Technology Stack:

- **Language:** C (Low-level memory control, High efficiency).
- **No External Libraries:** Custom implementation of Image I/O and Geometry writing.
- **Structure:** Modular design (Reader -> Processor -> Writer).

Parallelization & Key Decisions

```
#pragma omp parallel for
```



Parallelism (OpenMP):

- Shared-memory parallelism on CPU.
- Focus on independent pixel loops (e.g., Thresholding).
- *Why not GPU?* Minimal code changes required vs. complete rewrite.

Input Strategy Evolution:

- **Initial:** PGM (Grayscale) -> Hard to detect door arcs.
- **Final:** PPM (Color) -> Explicit Red Lines ($R>200$) for doors.
- **Result:** Improved robustness and removed complex shape detection.

Complexity & Trade-offs

Complexity Analysis:

- **Time:** $O(N)$ -> Each pixel visited constant times.
- **Space:** $O(N)$ -> Full-resolution arrays for masks.

Parallel vs. Serial Trade-offs:

- **Parallelized:** Independent calculations (Thresholding, Classification).
- **Kept Serial:**
 - a. Flood Fill (Complex to split).
 - b. Flood Fill (Complex to split).
- **Outcome:** Code is thread-safe and logically simple.

Simulation: Pipeline

Image Processing & Structure Extraction

1. Read PGM/PPM; detect red pixels as door seeds
2. Threshold (Otsu) → binary wall map + free-space map
3. Flood-fill from borders → outside; remaining free space → interior
4. Connected components → rooms; tiny regions → window gaps

Door & Boundary Logic

1. Group door pixels into segments
2. Determine horizontal/vertical orientation
3. Scan to wall edges → mark full door span
4. `door_wall_mask` removes wall up to door height

Simulation: 3D Geometry Generation

Floors & Ceilings

1. For each interior pixel: create 1×1 floor quad at $z=0$
2. Mirror as ceiling at full height (reverse winding for normals)

Walls

1. For each interior pixel, check 4-neighbor boundary
2. Neighbor = wall/outside → extrude vertical wall face
3. If door span → start wall at 84"; else from floor
4. Skip tiny regions → window openings

Output & Validation

1. OBJ file with floors, ceilings, walls, door gaps
2. Printed counts of rooms, windows, door segments
3. Visual check confirms alignment and opening placement

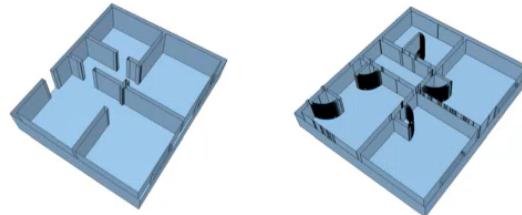
Results

Experimental design

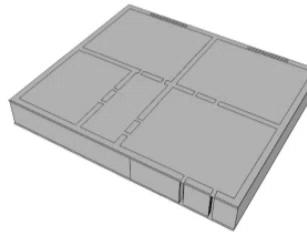
1. Pre-selected input
2. Image processing was timed
3. CPU and memory usage monitored
4. Qualitative observations were recorded on 3D image quality

Accuracy and Validation:

- Simulator accurately identified rooms, doors, windows, and small non-room features, matching the input floor plan without false room detections.
- Accurate segmentation and classification produced correct rooms, door openings, and window slits, filtering out minor artifacts and preserving true geometry in the 3D output.
- Final .obj models showed clean, consistent reconstruction with correct openings and no geometry errors, validating the algorithm's reliability.



*Figure : 3D Extrusion with Door
Doors, windows and walls at
correct location*



*Figure : The .obj 3D models with roof as
visualized in a 3D viewer online.*



Nation, Ryan T

Results: Performance

Parallel vs. Serial Runtime: Serial took around 1.8–2.0 s, while OpenMP reduced runtime to 0.476 s, giving about 3.8× overall speedup.

Per-Pixel Improvement: Cost dropped from around 9.3×10^{-7} s/pixel (serial) to 1.84×10^{-7} s/pixel (parallel), a 4 to 5 times gain.

Limits to Speedup: Flood-fill, Connected component labeling, and .OBJ 3D output writing remained serial, preventing linear scaling across 14 cores.

Scalability: Near-linear gains up to 8-12 threads; performance levels off from memory bandwidth limits but scales predictably with image size.

Figure: Console output window showing parallel

```
C:\Users\iaziz6\Downloads\NewF>3D_Reconstruction_CSE6010.exe converted_image.ppm 3D
===== 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
```

Discussion

- **Research provided support for energy simulations and learning algorithms**
- **Limitations:**
 - Challenging to work with C due to unavailability of libraries.
 - algorithm tended to favor some designs over the other
 - time constraints of academic semester
- **Future Research:**
 - inclusion of additional algorithms or models
 - additional time would allow for further algorithm improvement.
 - investigate inputs from other design softwares
 - Comparison with Python based 3D generation.