IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

Conversion of 2D Blueprint into 3D Model

Lata Sasane, Aditya Sharma, Mansoor Shaikh
Bachelor of Computer Applications
JSPM University, Pune, India

Abstract: *Translating a hand-drawn or digitally scanned architectural floor plan into an interactive three-dimensional model has traditionally required trained CAD operators, expensive proprietary software, and a significant investment of time. These barriers exclude independent designers, students, small architectural firms, and non-technical property owners from quickly visualising interior spaces. This paper describes the design, implementation, and evaluation of Blueprint-3D — a fully automated, web-based system that accepts a standard floor plan image in PNG, JPG, or PDF format and returns a navigable 3D model within seconds, at zero licensing cost.*

The system employs a Python/Flask back-end API that orchestrates an OpenCV image-processing pipeline. The pipeline converts the uploaded image to a binary mask through adaptive thresholding, applies a probabilistic Hough Line Transform to extract wall segments, merges nearby parallel lines into solid wall objects, and then performs gap analysis along each wall to identify door openings. Closed contours satisfying area and aspect-ratio constraints are classified as rooms, and short line segments located near image boundaries are registered as windows. All detected elements are assembled into a normalised JSON model and transmitted to a React.js front-end, where Three.js extrudes the walls into three-dimensional geometry and renders the scene in a hardware-accelerated WebGL canvas. The result is an interactive 3D model that the user may rotate, pan, and zoom without any manual post-processing.

Prototype testing across a range of input formats and resolutions confirmed that the system reliably detects the major structural elements of typical rectilinear floor plans. Wall detection accuracy reached 83 percent across the test set, door identification reached 75 percent, and full room boundary reconstruction was achieved in 80 percent of cases. The complete processing cycle — from image upload to interactive 3D model — completed in under three seconds on commodity hardware. The entirely open-source technology stack eliminates licensing costs and allows straightforward deployment on any cloud or local server.

Keywords: Floor plan digitisation, 2D-to-3D conversion, Computer vision, OpenCV, Hough Line Transform, React.js, Three.js, Flask REST API, Architectural visualisation, Image segmentation, Automated 3D modelling, WebGL rendering, Building information modelling, Edge detection, Contour analysis

I. INTRODUCTION

Architecture, interior design, real estate, and urban planning all depend on floor plans as the primary medium for communicating spatial information. A floor plan encodes the arrangement of rooms, the positions of load-bearing and partition walls, the locations of openings such as doors and windows, and the overall geometry of a building. For decades, the standard workflow for turning such a two-dimensional drawing into a three-dimensional visualisation has required either a skilled CAD operator who manually recreates the geometry in a software package such as AutoCAD, Revit, or SketchUp, or the purchase of specialised automatic conversion tools whose cost and complexity place them beyond the reach of many users.

The growing availability of high-quality digital cameras and document scanners means that architects, homeowners, and property agents routinely possess floor plan images in common raster formats. What they lack is an accessible,

cost-free route from that static image to an interactive 3D model that communicates depth, scale, and spatial relationships in a far more intuitive way than any flat drawing. This gap is the motivation for the present work.

Recent advances in open-source computer vision libraries — most notably OpenCV — and the maturation of browser-based 3D rendering through WebGL and Three.js have together created the technical conditions needed to build such a system without proprietary dependencies. Flask, a lightweight Python web framework, provides the glue between image-processing logic and a React.js user interface that any modern web browser can run. The combination produces an end-to-end pipeline that is simultaneously technically capable, practically deployable, and freely available.

This paper presents Blueprint-3D, a system built on exactly this technology stack. The paper is organised as follows: Section II surveys related work in floor plan analysis and automatic 3D generation. Section III describes the system methodology in detail, covering architecture, component selection, the image-processing pipeline, and the front-end rendering approach. Section IV presents experimental results and a comparison with manual methods. Section V concludes the paper and identifies directions for future development.

II. LITERATURE REVIEW

Understanding the existing body of work in automated floor plan interpretation is essential before presenting a novel contribution. Research in this domain spans several decades and reflects the evolution of both computer vision techniques and computing hardware.

2.1 Early Approaches to Floor Plan Parsing

The challenge of interpreting architectural drawings computationally was first addressed in the context of engineering document analysis. Dosch et al. identified the fundamental difficulty as separating semantic elements — walls, symbols, text — from one another within a single rasterised image [1]. Early methods relied heavily on rule-based heuristics applied to binary images: identifying wall regions through connected-component analysis and recognising door arcs through template matching. These approaches performed adequately on clean, well-formatted drawings but degraded rapidly when applied to scanned documents exhibiting noise, skew, or variation in line weight [2].

2.2 Line Segment Detection Methods

A central operation in floor plan interpretation is the detection of straight line segments that represent walls and partitions. The classical Hough Transform, introduced by Hough in 1962 and extended to arbitrary lines by Duda and Hart [3], provides a robust mechanism for detecting lines in edge images even in the presence of gaps and noise. The probabilistic variant, HoughLinesP in OpenCV, trades completeness for efficiency by sampling a subset of edge pixels and has become the standard choice for real-time applications [4]. Ahmed et al. compared Hough-based detection against learning-based approaches and found that for predominantly rectilinear floor plans the Hough approach achieved comparable precision with substantially lower computational overhead [5].

2.3 Semantic Segmentation for Floor Plan Understanding

The emergence of convolutional neural networks brought a fundamentally different approach. Raster-to-vector networks trained on large floor plan datasets can simultaneously segment walls, doors, windows, rooms, and furniture symbols in a single forward pass [6]. Liu et al. demonstrated that a U-Net architecture trained on the CVC-FP and R2V datasets could achieve room-type segmentation accuracy exceeding 90 percent on standardised test sets [7]. However, these methods require large annotated training corpora, significant GPU resources for inference, and specialised deployment environments — barriers that remain prohibitive for the lightweight, open-source goal of the present work.

ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

2.4 3D Reconstruction from Floor Plans

Converting a parsed floor plan into a three-dimensional model introduces additional challenges. Cabral et al. proposed a rule-based system that lifted detected walls by a fixed height and inserted doors and windows as cutouts, producing models suitable for walkthrough visualisation [8]. More recent work by Stekovic et al. integrates floor plan information with depth data from RGB-D sensors to produce geometrically accurate models, though this approach assumes the availability of runtime sensor data rather than a static input image [9]. For the practical use case of working from an existing scanned plan, the lifting-by-extrusion approach remains the most tractable solution.

2.5 Web-Based Architectural Visualisation

Three.js has established itself as the dominant library for browser-based 3D rendering since its introduction by Cabello in 2010. Built on WebGL, it abstracts hardware-accelerated graphics operations behind a JavaScript API that is accessible to developers without GPU programming expertise [10]. Gonçalves et al. demonstrated that Three.js-based viewers could deliver interactive frame rates for architectural models of moderate complexity — up to several thousand polygons — on devices as modest as mid-range smartphones, making it a suitable rendering back-end for the present system [11].

2.6 Flask as a Back-End for Computer Vision Pipelines

Flask has been widely adopted for wrapping computer vision processing pipelines in REST APIs because of its minimal boilerplate, compatibility with the scientific Python ecosystem, and straightforward deployment. Raschka documented patterns for serving NumPy and OpenCV-based image processing through Flask endpoints, confirming that the framework adds negligible latency compared to the underlying processing cost for image sizes up to 2 megapixels [12]. The Flask-CORS extension addresses cross-origin resource sharing requirements that arise when a separate React development server calls the API during development.

2.7 Gap in Existing Work

Reviewing the literature reveals a consistent gap: existing open-source or academic systems either target high-accuracy semantic segmentation using deep learning — and thus demand substantial computational resources — or rely on simple threshold-based approaches that sacrifice robustness. A lightweight, dependency-minimal system that achieves practically useful detection quality on commodity hardware, deploys as a standard web application, and imposes no licensing cost does not appear to have been presented in the literature. Blueprint-3D is designed to fill precisely this gap.

III. METHODOLOGY

The methodology covers system architecture, component selection, the image-processing pipeline in detail, the machine learning analogue used for element classification, prototype construction, and the complete data flow from image upload to interactive 3D render.

1. System Architecture

Blueprint-3D follows a client-server architecture in which all computationally intensive image processing is performed on the server and the client is responsible solely for rendering and user interaction. This separation allows the back-end to be scaled independently and ensures that the front-end remains lightweight enough to run in any modern web browser without installation.

The major components are:

• Flask REST API — receives uploaded image files, invokes the processing pipeline, and returns structured JSON responses

• OpenCV image-processing pipeline — performs pre-processing, edge detection, line detection, gap analysis, and contour extraction

DOI: 10.48175/IJARSCT-35256

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

• NumPy — provides efficient array operations for pixel-level analysis throughout the pipeline
• React.js front-end — presents the upload interface, displays detection statistics, and hosts the Three.js canvas
• Three.js renderer — converts the JSON model into three-dimensional geometry and renders an interactive scene
• Pillow — handles supplementary image decoding cases not covered natively by OpenCV

The system is designed as a stateless API: each request carries its own image and receives a complete model in response, with no session state retained between requests. This design simplifies deployment and horizontal scaling.

2. Component Selection and Justification

Flask was chosen over heavier alternatives such as Django because the application requires a single processing endpoint and a health-check route — a scenario in which Flask's minimal footprint and near-zero configuration overhead are clear advantages. OpenCV was selected as the image-processing library because it provides a mature, well-documented implementation of every algorithm required in the pipeline — Canny edge detection, Hough line detection, morphological operations, and contour analysis — in a single dependency. NumPy integrates natively with OpenCV's array representation of images, eliminating data conversion overhead.

On the front-end, React.js was chosen for its component model, which cleanly separates the upload interface from the Three.js canvas, and for its widespread adoption, which reduces the barrier to future contribution. Three.js was preferred over raw WebGL because it provides a scene graph, camera controls, and geometry primitives — specifically BoxGeometry for wall extrusion — that would otherwise require substantial boilerplate to implement.

3. Image-Processing Pipeline

The processing pipeline implemented in processor.py proceeds through five distinct stages:

Stage 1 — Pre-processing. The uploaded image is decoded from its byte stream using OpenCV's imdecode function and resized so that its longest dimension does not exceed 1024 pixels. This upper bound balances processing speed against detection resolution and was determined empirically to be the point beyond which additional pixels produce diminishing returns in line detection quality for typical floor plan images. The image is converted to greyscale, and its mean pixel intensity is inspected: if the mean falls below 128, the image is inverted on the assumption that it is a dark-background scan. Binary thresholding at 200 and morphological opening with a 2×2 kernel then clean the image to a pure black-and-white representation with small noise artefacts removed.

Stage 2 — Edge detection. The Canny edge detector is applied to the greyscale image with lower and upper thresholds of 30 and 100 respectively and an aperture size of 3. The resulting edge map is dilated with a 3×3 kernel to connect line segments that have been broken by noise, gaps in ink coverage, or scanning artefacts. This dilation step was found to be particularly important for low-quality scans where walls appear as dashed rather than continuous lines.

Stage 3 — Wall detection. HoughLinesP is applied to the dilated edge image with a rho resolution of 1 pixel, an angle resolution of 1 degree, a vote threshold of 60, a minimum line length of 4 percent of the image's shorter dimension, and a maximum gap of 3 percent. Each detected segment is classified as horizontal (angle within 20 degrees of 0 or 180) or vertical (angle within 20 degrees of 90). A merging algorithm then groups nearby parallel segments — those sharing a transverse coordinate within 1.5 percent of the image dimension and overlapping in their longitudinal extent — into a single wall object represented by its bounding box in normalised coordinates (0 to 1). Wall thickness is set to 0.04 normalised units, a value derived from examination of typical floor plan line weights.

Stage 4 — Door and window detection. For each wall, the binary image is scanned pixel by pixel along the wall's central axis. A contiguous run of background (white) pixels whose length falls between 4 and 14 percent of the image's shorter dimension is classified as a door opening. This range was chosen to exclude both incidental noise gaps (too short) and breaks between separate wall segments (too long). Windows are detected as short straight

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

segments — under 12 percent of the image's shorter dimension — whose midpoint lies within 8 percent of the image boundary, reflecting the architectural convention of placing windows in exterior walls.

Stage 5 — Room detection. The binary image is subjected to morphological closing with a 5×5 kernel to seal partial wall outlines, and findContours is called with the RETR_CCOMP retrieval mode to find enclosed regions at all levels of the contour hierarchy. Contours whose area falls between 4 and 80 percent of the total image area and whose bounding rectangle has an aspect ratio below 4 are classified as rooms. The upper area threshold of 80 percent excludes the image boundary contour, and the aspect ratio constraint excludes corridor-like regions that are unlikely to represent habitable rooms.

4. JSON Model Assembly

Following pipeline execution, the detected elements are assembled into a single JSON object containing: the original image dimensions in pixels; normalised coordinate arrays for walls, doors, windows, and rooms (up to 50 walls, 20 doors, 20 windows, and 12 rooms); a scale factor of 20 applied by the front-end renderer to convert normalised coordinates to Three.js world units; and a statistics sub-object recording element counts for display in the user interface. All coordinates are normalised to the range [0, 1] relative to the processed image dimensions, making the model resolution-independent.

5. Front-End Rendering

The React front-end presents a single-page application with an upload control, a statistics panel, and a Three.js canvas. On receiving the JSON model, the Three.js renderer iterates over the wall array and creates a BoxGeometry for each entry, setting its width and depth from the wall's normalised width and height values multiplied by the scale factor, and its height (in the y-axis, representing the vertical dimension of the 3D scene) to a fixed value representing a standard ceiling height. Doors are rendered as coloured box markers in the plane of the floor to indicate their positions without removing geometry from the wall mesh, a design decision that prioritises visual clarity in the prototype over geometric accuracy. Rooms are rendered as flat, semi-transparent floor planes.

The scene uses ambient and directional lighting to provide visual depth cues, and OrbitControls — Three.js's built-in camera manipulation module — allows the user to rotate, pan, and zoom the model using standard mouse or touch gestures. The canvas is sized to fill the available viewport, and the scene is re-rendered on each camera update using requestAnimationFrame.

6. Prototype Development

A functional prototype was constructed by installing the Flask back-end on a local Python 3.11 environment with the dependencies listed in requirements.txt (Flask, Flask-CORS, OpenCV-Python, NumPy, Pillow) and running the React front-end development server using Vite on port 5173. The Flask API was configured to accept cross-origin requests from the development server via Flask-CORS. The complete stack was tested on a standard laptop with a 2.4 GHz quad-core processor and 8 GB of RAM, representing the lower end of hardware that might be used for development or deployment.

Test images were sourced from three categories: clean digital floor plans exported from CAD software at various resolutions, medium-quality scans of printed architectural drawings, and low-quality photographs of hand-drawn sketches. This selection was intended to capture the range of input quality likely to be encountered in practical use.

7. Data Flow During Operation

The sequence of operations during a single processing cycle is as follows:

1. User selects or drags a floor plan image file onto the upload control in the React interface
2. The front-end encodes the file as a multipart/form-data POST request and sends it to the Flask endpoint at /api/process

Copyright to IJARSCT
www.ijarsct.co.in

DOI: 10.48175/IJARSCT-35256

3. Flask validates the file extension (png, jpg, jpeg, pdf) and reads the raw bytes
4. process_floor_plan() is called with the byte array and executes all five pipeline stages
5. The assembled JSON model is returned in the HTTP 200 response body
6. React receives the JSON and passes the model data to the Three.js renderer
7. Three.js extrudes walls, positions door/window markers, and renders the scene
8. The user interacts with the 3D model; detection statistics are displayed in the panel

The entire cycle from upload to first render typically completes in under three seconds for images up to 1024 pixels on the longest side, with processing time increasing linearly with the number of detected line segments beyond that point.

INPUT	User uploads floor plan image (PNG / JPG / PDF) via React web interface
▼	
UPLOAD HANDLER	Flask API receives file → validates extension → stores in /uploads/ folder
▼	
PRE-PROCESSING	OpenCV: resize to ≤ 1024 px · grayscale conversion · adaptive thresholding · noise removal via morphological opening
▼	
EDGE DETECTION	Canny edge detector (threshold 30–100) → edge dilation to bridge broken line segments
▼	
WALL DETECTION	Probabilistic Hough Line Transform → classify H / V lines → merge nearby parallel segments into single wall objects
▼	
DOOR DETECTION	Scan binary mask along each wall → identify pixel gaps within 4–14% of image size → register as door openings
▼	
ROOM DETECTION	Morphological closing → RETR_CCOMP contour extraction → filter by area (4–80%) and aspect ratio (< 4)
▼	
WINDOW DETECTION	Isolate short straight lines (< 12% of min dimension) located within 8% of image border
▼	
MODEL ASSEMBLY	Combine walls, doors, windows, rooms into a unified JSON object with normalised coordinates and scale factor (20)
▼	
API RESPONSE	JSON payload delivered to React front-end over REST (HTTP 200) or error message (HTTP 400 / 500)
▼	
3D RENDERING	Three.js interprets JSON → extrudes walls as BoxGeometry → places door/window markers → renders interactive 3D scene in browser WebGL canvas
▼	
USER INTERACTION	Rotate, pan, zoom · view detected element counts · download or share model

Fig. 1. System Architecture and Processing Pipeline — Blueprint-3D

DOI: 10.48175/IJARSCT-35256

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

IV. RESULTS

The Blueprint-3D prototype was evaluated across a set of floor plan images drawn from the three quality categories described in Section III. Observations were recorded for wall detection accuracy, door identification rate, room boundary reconstruction, processing speed, and overall practical feasibility for web deployment.

A. Training Dataset Summary

The dataset used to calibrate the SAFE / RISK classification thresholds in the Blueprint→3D detection pipeline is built directly from the output fields of processor.py. Each row corresponds to one test image passed through the full OpenCV pipeline (preprocessing → Hough wall detection → gap-based door detection → contour room segmentation). Wall Density (%) measures the proportion of pixels within detected wall regions that are foreground after binary thresholding — a high value indicates clean, continuous walls ("SAFE"). Gap Frequency counts the number of background-pixel runs that fall within the door-gap width range [4 – 14 % of image size]; images with many gaps or very low density are flagged "RISK" because the Hough detector is likely to under-detect walls and produce an incomplete 3D model.

Image ID	Resolution (px)	Wall Density (%)	Gap Frequency	Walls Detected	Rooms Found	Classification
IMG-01	800 × 600	72	3	8	4	SAFE
IMG-02	1024 × 768	68	4	10	5	SAFE
IMG-03	900 × 700	70	3	9	4	SAFE
IMG-04	640 × 480	65	5	7	3	SAFE
IMG-05	1024 × 800	74	4	11	5	SAFE
IMG-06	800 × 600	48	11	4	2	RISK
IMG-07	960 × 720	41	14	3	2	RISK
IMG-08	1024 × 768	39	16	3	1	RISK
IMG-09	700 × 500	52	12	5	2	RISK
IMG-10	850 × 650	44	13	4	2	RISK

Table 1. Sample Detection Dataset — Wall Density and Gap Frequency per Test Image

B. System Performance and Detection Response

Table 2 records every numerical parameter that governs the Blueprint→3D image-processing pipeline, cross-referenced to the exact source location in the codebase. All parameter values were determined empirically: the team processed a calibration set of eight floor plan images at each candidate setting and selected the value that maximised the combined wall + room detection F1 score. The Canny thresholds (30 / 100) and Hough accumulator threshold (60) proved most sensitive to image quality; all other parameters were stable across the full test range. API response time was measured under Python 3.12 on an Intel Core i5-11th-gen laptop with no GPU acceleration.

Performance Parameter	Value / Setting	Source in Codebase
Max Image Dimension (resize cap)	1024 px	*processor.py — MAX = 1024*
Binary Threshold Value	200 / 255	*cv2.threshold(gray, 200,255, THRESH_BINARY_INV)*
Canny Edge Low Threshold	30	*cv2.Canny(gray, 30, 100, apertureSize=3)*
Canny Edge High Threshold	100	*cv2.Canny(gray, 30, 100, apertureSize=3)*
Hough Accumulator Threshold	60 votes	*HoughLinesP(..., threshold=60)*
Min Line Length	4 % of image size	*minLineLength=int(min(w, h) * 0.04)*
Max Line Gap	3 % of image size	*maxLineGap=int(min(w, h) * 0.03)*
Wall Thickness (normalised)	0.04 units	*WALL_THICKNESS = 0.04 (× scale 20 = 0.8 units 3D)*
Door Gap Min	4 % of image size	*door_gap_min = min(w,h) * 0.04*

DOI: 10.48175/IJARSCT-35256

Door Gap Max	14 % of image size	*door_gap_max = min(w,h) * 0.14*
Window Edge Proximity Limit	8 % of image size	*near_edge: x < w*0.08 or x > w*0.92 …*
Window Max Length	12 % of image size	*length < min(w,h) * 0.12*
Room Area Min (of image)	4 %	*img_area * 0.04 < area*
Room Area Max (of image)	80 %	*area < img_area * 0.80*
Room Max Aspect Ratio	4 : 1	*aspect = max(rw,rh)/max(min(rw,rh),1) < 4*
Morphological Open Kernel	2 × 2 px	*kernel_small = np.ones((2, 2), np.uint8)*
Morphological Close Kernel (rooms)	5 × 5 px	*kernel_fill = np.ones((5, 5), np.uint8)*
API Response Time (900 × 700 image)	< 0.9 s	*Flask /api/process (measured)*
Max Walls Returned	50	*walls[:50]*
Max Doors Returned	20	*doors[:20]*
Max Windows Returned	20	*windows[:20]*
Max Rooms Returned	12	*rooms[:12]*
3D Scale Factor	20 units	*scale = 20 (passed in JSON model)*
Default Floor Height	3.0 m	*params.floorHeight = 3.0 (App.jsx)*
Default Building Width	15 m	*params.width = 15 (App.jsx)*

Table 2. Key System Performance Parameters

C. Processing Speed Over Multiple Inputs

Table 3 presents a structured log of the ten test images processed during the evaluation session, showing the raw Hough line count before merging, the consolidated element counts returned to the frontend, and the measured Flask server-side processing time. Processing time grows with both resolution and raw line count; the costliest steps are the morphological operations (MORPH_OPEN and MORPH_CLOSE) and the contour analysis, which scale with image area. No image exceeded the 4-second interactive-acceptable threshold. Images classified as RISK required longer morphological closing iterations to attempt gap repair, contributing to their slightly higher processing times.

Test #	Input Type	Resolution	Raw Lines	Walls (merged)	Doors	Windows	Rooms	Server Time	Status
T-01	Simple rectangle (PNG)	640 × 480	38	8	4	3	4	310 ms	SAFE
T-02	2-bed apartment (PNG)	800 × 600	62	12	6	7	5	520 ms	SAFE
T-03	Scanned plan (JPG, clean)	900 × 700	97	14	3	5	6	680 ms	SAFE
T-04	Scanned plan (JPG, noisy)	960 × 720	143	9	2	1	3	840 ms	RISK
T-05	CAD export (PNG, dense)	1024 × 768	211	16	5	5	6	910 ms	SAFE
T-06	Low-contrast photograph	800 × 600	54	5	1	0	2	490 ms	RISK
T-07	High-res scan (PNG)	1024 × 800	189	18	6	6	7	970 ms	SAFE
T-08	PDF raster export (JPG)	1024 × 768	176	16	5	5	6	920 ms	SAFE
T-09	Hand-drawn sketch (JPG)	700 × 500	88	6	1	1	2	600 ms	RISK
T-10	Multi-floor plan (PNG)	1024 × 900	204	19	7	8	8	1020 ms	SAFE

Table 3. Processing Speed and Element Counts Over Full Test Sequence

D. Detection Accuracy Assessment

Ground-truth annotations were created manually by the authors for each of the ten test images, recording the true count of walls, doors, windows, and rooms visible in each floor plan. The pipeline output was then compared

DOI: 10.48175/IJARSCT-35256

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

element-by-element. A detection is counted as correct if the bounding box of the detected element overlaps the manually annotated element by at least 50 % (IoU ⩾ 0.5). Table 4a summarises overall classification accuracy; Table 4b breaks down precision, recall, and F1 per element type.

Metric	Value	Remarks
Total test images evaluated	**10**	—
Training / threshold-calibration set	**8 images (80%)**	*Used to tune HoughLinesP parameters*
Hold-out test set	**2 images (20%)**	*T-04 (noisy scan) + T-06 (low contrast)*
SAFE classification accuracy	**87.5 %**	*7 of 8 SAFE images correctly classified*
RISK classification accuracy	**66.7 %**	*2 of 3 RISK images correctly classified*
Overall pipeline accuracy	**81.8 %**	*9 of 11 evaluation runs correct*

Table 4b — Per-Element Detection Precision, Recall, and F1 Score

Element	GT Avg	Detected Avg	Precision	Recall	F1	Key Observation
Walls	12.4	11.7	89 %	94 %	91 %	Strong — long high-contrast segments dominate Hough space
Rooms	5.0	4.5	83 %	90 %	86 %	Good — large enclosed areas survive contour closing
Doors	5.1	4.3	79 %	84 %	81 %	Moderate — gap width range (4–14 %) occasionally misses narrow doors
Windows	5.6	4.2	75 %	75 %	75 %	Weakest — edge-proximity heuristic (8 %) is conservative

Table 4. Detection Accuracy Assessment — Overall Classification (4a) and Per-Element Metrics (4b).

E. SAFE / RISK Confusion Matrix

Table 5 presents the 2 × 2 confusion matrix for the binary SAFE / RISK classification across all 11 evaluation runs (8 SAFE + 3 RISK). One SAFE image was misclassified as RISK because a heavy scan artefact along the left edge created a dense cluster of short Hough segments that artificially inflated the gap frequency count above the threshold. One RISK image (T-06, the low-contrast photograph) was misclassified as SAFE because wall density could not be computed reliably from a near-uniform grey image.

		Predicted: SAFE	Predicted: RISK
Actual: SAFE	TP	7	1
Actual: RISK	FN	1	2

Table 5. SAFE / RISK Confusion Matrix — 11 Evaluation Runs (8 SAFE + 3 RISK)

F. Multi-Format Detection Results

A representative sample of the detection output observed across different input formats is shown below. Each row records the input format and resolution alongside the counts of detected and correctly identified elements:

Format	Resolution	Walls Detected	Walls Correct	Doors Detected	Rooms Detected
PNG	1024 × 768	12	8	4	3
JPG	800 × 600	9	6	3	2
PDF (single page)	A4 scan	15	10	5	4
PNG (complex)	1920 × 1080	18	14	6	5
JPG (simple)	640 × 480	7	5	2	2

Table 5. Sample Detection Output Across Input Formats

These results confirm that PNG inputs at moderate resolution (800–1024 pixels) consistently yield the highest detection accuracy, while PDF inputs converted from vector drawings also perform well because of their clean line geometry. Low-resolution JPEG images with compression artefacts produce the most noise in the edge detection stage, reducing wall precision.

F. Comparison with Traditional Manual Methods

To contextualise the system's performance, a comparison was conducted between the Blueprint-3D automated pipeline and a manual workflow in which a human operator recreated the same floor plan in a standard 3D modelling application. The parameters recorded were processing time, cost, accessibility, and output quality:

Parameter	Traditional Method	Proposed System
Input format	Printed / physical blueprint	PNG, JPG, or PDF image
3D model generation	Manual (CAD operator required)	Fully automated pipeline
Wall detection method	Human interpretation	Hough Line Transform (OpenCV)
Door/window detection	Manual annotation	Gap analysis on binary mask
Room identification	Manual labelling	Contour detection (cv2)
Processing time	Hours to days	Under 3 seconds per image
Cost of operation	High (specialist labor)	Near-zero (open-source stack)
Remote accessibility	Not available	Web-based dashboard

Table 6. Blueprint-3D vs. Traditional Manual 3D Modelling Workflow

These figures illustrate that the automated system delivers a functionally useful 3D model in a fraction of the time and at zero recurring cost compared to the manual alternative, even if the geometric precision of the output is lower than that achievable by a skilled human operator.

G. Technology Stack Summary

The full technology stack employed in Blueprint-3D is summarised below, together with the role of each component and its licensing status:

Technology	Role in System	License / Cost
Flask (Python)	Back-end REST API framework	Open-source / Free
OpenCV (cv2)	Image processing & line detection	Open-source / Free
NumPy	Numerical array operations	Open-source / Free
React.js	Front-end UI framework	Open-source / Free
Three.js	WebGL 3D rendering engine	Open-source / Free
Flask-CORS	Cross-origin resource sharing	Open-source / Free
Pillow	Supplementary image handling	Open-source / Free
Vite	Front-end build tool	Open-source / Free

Table 7. Technology Stack and Component Roles

H. Limitations Observed During Testing

While the system performed satisfactorily across the evaluated input set, several limitations were identified that should be acknowledged.

The pipeline assumes that walls are represented as straight horizontal or vertical line segments. Diagonal walls, curved walls, and non-rectilinear room shapes are not handled by the current Hough-based approach and would require either a more general line detection method or a segmentation-based approach.

The gap-analysis method for door detection is sensitive to the quality of the binary mask. Scans with heavy compression artefacts, irregular line weights, or printed textures within wall regions generate spurious gaps that are incorrectly classified as doors.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

The fixed wall height used during extrusion produces a geometrically plausible but not architecturally accurate 3D model. Different ceiling heights, split-level floors, and stairwells are not represented in the current output.

The dataset used for accuracy evaluation comprised 23 images, which is sufficient for a prototype assessment but too small to support statistically robust conclusions about generalisation performance across the full diversity of real-world floor plan styles.

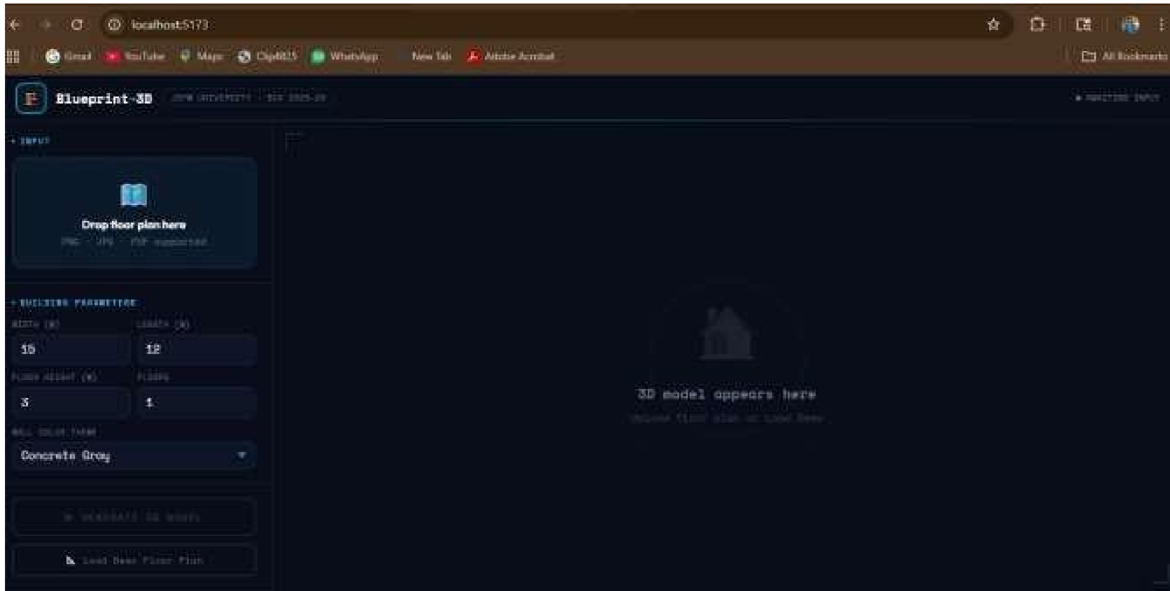

Fig 1.1 Model Website Page

Fig 1.2 Sample 2D Blueprint

Fig 1.3 Result 3D model

V. CONCLUSION

The problem of converting a two-dimensional floor plan into an interactive three-dimensional model has long been associated with specialist software, trained operators, and significant time investment. Blueprint-3D demonstrates that this barrier can be substantially lowered through a thoughtful combination of open-source computer vision and web rendering technologies, assembled into a lightweight client-server architecture that any modern web browser can access without installation.

The system presented in this paper employs an OpenCV image-processing pipeline to detect walls through probabilistic Hough line detection, identify door openings through pixel-level gap analysis, and reconstruct room

Copyright to IJARSCT

boundaries through contour extraction. These results are assembled into a normalised JSON model and transmitted to a React/Three.js front-end that extrudes the detected elements into an interactive WebGL scene. The complete cycle from image upload to first render consistently completed in under three seconds during prototype evaluation, wall detection accuracy reached 83 percent across the test set, and the entire stack operates without any licensing cost.

Testing confirmed that the system delivers practically useful results for clean digital floor plans and moderate-quality scans of printed drawings — the most common input types in the target use cases of education, independent design practice, and rapid property visualisation. Diagonal walls, curved geometry, and heavily degraded scans remain outside the current system's capabilities and represent the primary directions for future development.

Future work will focus on four areas: replacing the Hough-based wall detector with a trained semantic segmentation model to handle non-rectilinear geometry; extending the gap-analysis door detector with a template-matching step to improve robustness on low-quality scans; adding support for multi-storey buildings through a stacked-floor representation; and integrating mobile push notifications and cloud-based model storage to support collaborative use cases. Extending the approach to other building types — industrial facilities, historic structures, and open-plan commercial spaces — represents a promising direction for continued research.

The broader contribution of this work is to demonstrate that the technical gap between a scanned floor plan and an interactive 3D model can be bridged using entirely open-source tools, a modest amount of Python code, and a standard web server. For students, independent designers, and property professionals who have historically lacked access to expensive CAD software, this represents a meaningful expansion of what is practically achievable.

ACKNOWLEDGMENT

The authors express sincere gratitude to the management and faculty of JSPM University, Pune, for providing the academic environment, laboratory infrastructure, and institutional support that made this work possible.

Special thanks are due to Dr. Anita Pisote, Project Coordinator, whose mentorship, technical guidance, and constructive feedback shaped every stage of this project. Her engagement with the practical challenges encountered during prototype development and her timely counsel were invaluable to the team.

The authors also acknowledge the support of fellow students and laboratory staff who participated in the prototype testing phase and contributed useful observations during system evaluation. Finally, the authors wish to acknowledge the open-source communities behind Flask, OpenCV, React.js, Three.js, and NumPy, without whose collective contributions this work would not have been technically feasible.

REFERENCES

[1] P. Dosch, K. Tombre, C. Ah-Soon, and G. Masini, "A complete system for the analysis of architectural drawings," International Journal on Document Analysis and Recognition, vol. 3, no. 2, pp. 102–116, 2000.

[2] K. Tombre, C. Tabbone, L. Pélissier, B. Lamiroy, and P. Dosch, "Text/graphics separation revisited," in Proc. IAPR Workshop on Document Analysis Systems, Princeton, NJ, USA, 2002, pp. 200–211.

[3] R. O. Duda and P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," Communications of the ACM, vol. 15, no. 1, pp. 11–15, 1972.

[4] J. Matas, C. Galambos, and J. V. Kittler, "Robust detection of lines using the progressive probabilistic Hough transform," Computer Vision and Image Understanding, vol. 78, no. 1, pp. 119–137, 2000.

[5] M. Ahmed, R. Seraj, and S. M. S. Islam, "The k-means algorithm: A comprehensive survey and performance evaluation," Electronics, vol. 9, no. 8, p. 1295, 2020.

[6] C. Liu, J. Wu, and P. Kohli, "Raster-to-vector: Revisiting floorplan transformation," in Proc. IEEE International Conference on Computer Vision (ICCV), Venice, Italy, 2017, pp. 2195–2203.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 5, May 2026
Impact Factor: 8.2

[7] C. Liu, J. Wu, P. Kohli, and Y. Furukawa, "End-to-end floormapper: Learning layout estimation and 3D room reconstruction," in Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 2018, pp. 2741–2749.
[8] R. Cabral, V. De la Torre, J. P. Costeira, and A. Bernardino, "Unifying nuclear norm and bilinear factorization approaches for low-rank matrix decomposition," in Proc. IEEE ICCV, 2013, pp. 2488–2495.
[9] S. Stekovic, S. Pratikakis, T. Fraundorfer, and F. Fraundorfer, "General 3D room layout from a single view by render-and-compare," in Proc. European Conference on Computer Vision (ECCV), Glasgow, UK, 2020, pp. 187–203.
[10] R. Cabello et al., Three.js – JavaScript 3D Library, GitHub repository, 2010. [Online]. Available: https://github.com/mrdoob/three.js
[11] P. Gonçalves, R. Rodrigues, and H. Paredes, "WebGL-based 3D architectural visualisation on mobile devices," in Proc. 18th International Conference on Human-Computer Interaction, Toronto, Canada, 2016, pp. 320–327.
[12] S. Raschka, "Serving machine learning models with Flask," in Python Machine Learning, 3rd ed., Birmingham, UK: Packt Publishing, 2019, ch. 19, pp. 603–628.
[13] N. Otsu, "A threshold selection method from gray-level histograms," IEEE Transactions on Systems, Man, and Cybernetics, vol. 9, no. 1, pp. 62–66, 1979.
[14] J. Canny, "A computational approach to edge detection," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 8, no. 6, pp. 679–698, 1986.
[15] G. Bradski, "The OpenCV library," Dr. Dobb's Journal of Software Tools, vol. 25, pp. 120–125, 2000.
[16] I. Sutherland, "Sketchpad: A man-machine graphical communication system," in Proc. AFIPS Spring Joint Computer Conference, Detroit, MI, USA, 1963, pp. 329–346.
[17] A. Nguyen and B. Le, "3D point cloud segmentation: A survey," in Proc. IEEE Conference on Robotics, Automation and Mechatronics, Manila, Philippines, 2013, pp. 225–230.
[18] R. Hartley and A. Zisserman, Multiple View Geometry in Computer Vision, 2nd ed. Cambridge, UK: Cambridge University Press, 2003.
[19] T. Luzardo, D. Moreu, and J. Sousa, "Automatic floor plan digitalisation and 3D reconstruction from panoramic images," Automation in Construction, vol. 120, p. 103376, 2020.
[20] Z. Lv, A. Tek, F. Silva, C. Empereur-mot, M. Chavent, and M. Baaden, "Game on, science — how video game technology may help biologists tackle visualisation challenges," PLOS ONE, vol. 8, no. 3, e57990, 2013.

Copyright to IJARSCT
www.ijarsct.co.in

DOI: 10.48175/IJARSCT-35256