Interior Design Application: Final Project Report

Arnav Jaiswal
Yoga Srinivas Reddy Kasireddy

April 18, 2025

1 Introduction

This project documents the development of an interactive interior design application built in Unity for the Meta Quest 3. It enables users to create, customize, and visualize 3D walls using AR and natural controller-based interactions. The motivation behind the project is to empower real-time, immersive architectural prototyping in mixed reality environments.

2 Background

While VR/AR is gaining popularity in architecture and design, many solutions lack intuitive interactions and real-time customization. Our application fills this gap by integrating procedural mesh generation, dynamic UI, texture mapping, and passthrough support in Unity. This facilitates direct environmental feedback and immersive user interaction.

3 Methods

3.1 Technologies Used

- Unity (C# Scripting, URP, Meta XR SDK)
- Meta Quest 3 + Meta All-in-One SDK
- Meta Building Blocks + Passthrough
- Meta Interaction SDK + Nappin Asset Library

3.2 How Does It Work?

Meta Camera Rig

: The Meta Camera Rig Building Block comes integrated with Passthrough functionality which allows user to perceive the AR space

Controller Mapping

A C# script maps the Meta Quest 3 controller buttons: A & X toggle the menu canvas, while Y initiates wall creation after placing origin and top-right points.

Materials and Texture Loader

Scripts collect all materials and textures, assign them to dynamically created prefab buttons, and place them evenly in scrollable material canvases. A Meta Building Block Cube is instantiated with collider and interaction components.

Color Screen Canvas

Includes RGB sliders (0–255). Once a color is selected, paint is applied to the wall.

Object Search and Location

Since walls are generated at runtime, scripts dynamically find the target wall object to apply selected user properties (color, texture, material).

Toggle Canvas and Reset Functions

Each menu canvas can toggle visibility. Reset buttons allow wall texture and color resets.

Canvas Ray Interaction

Enables interaction via ray pointers by attaching Meta Ray Interaction SDK to the canvases.

Wall Reset: Users can reset wall materials and colors to defaults.

3.3 Wall Generation

Normal Wall: Two points (origin and top-right) define a diagonal on the X-Z plane. A cube primitive is scaled accordingly. Thickness is set at 0.5f. The wall is generated perpendicular to the ground.

Origin Point and Top Right Point are Meta Building Block components initialized with Ray Grab and Hand Grab components on them to be controlled by Meta Quest controllers.

Upon placing the origin point at the bottom left corner and the top right point at the top right corner of the wall to map, the user can press Y on their controller to generate the wall. The wall creation mechanics are as follows:

The two points represent the diagonal vertices of the wall. A 3-D cube object is created, and it is scaled to align with the vertices. The thickness of the wall is an arbitrary constant of 0.5f.

However, 2 points in a 3-D space lead to infinite plane possibilities.

For our use case, we force the wall to be perpendicular to the floor (X-Z plane) and hence can create a wall with just 2 points.

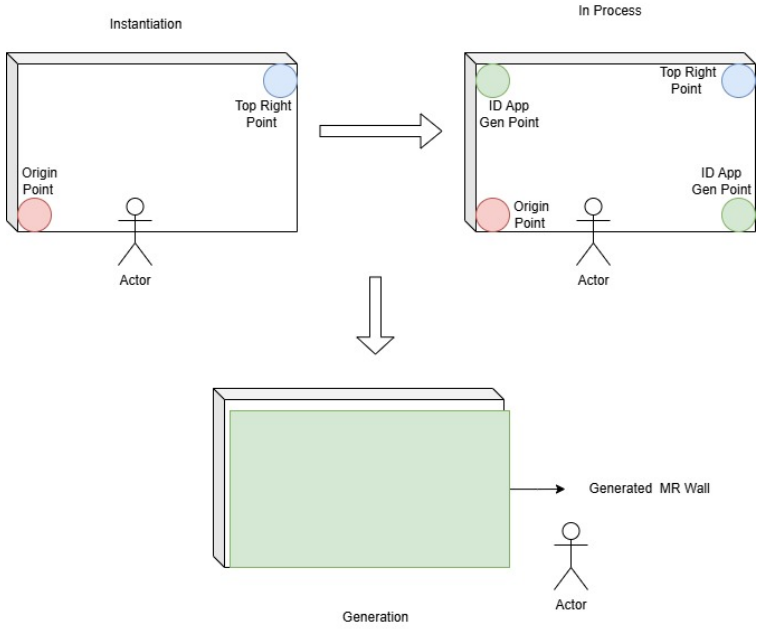

Figure 1: Graphic Display of Normal Wall Generation

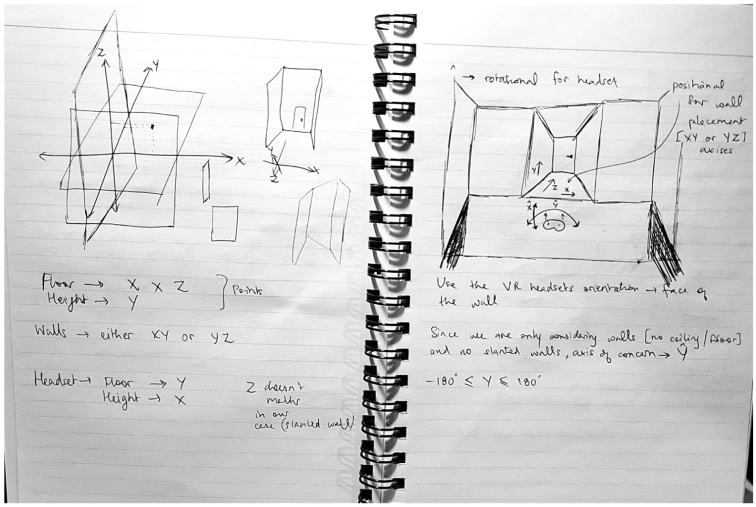

Figure 2: Trying to figure out orientations by hand to decide constraints and correct the wall creation through Meta's VR simulator

3.4 Custom Wall Generation

Arguably the most complex feature of the app, **Custom Wall Generation** allows users to define up to 10 points to map irregular or designer walls.

- **Mode Activation:** Users can enter build mode by long-pressing the 'A' button on the controller for 3 seconds. A green pulsing indicator appears in the bottom-right corner, signaling activation.
- **Point Placement:** In build mode, points can be placed using the trigger button. Each point spawns in front of the right controller. Up to 10 points can be placed.
- **Exit Build Mode:** A second long-press of 'A' exits build mode, indicated by a red pulsing indicator. The wall is then automatically generated based on the placed points.

Below are the mechanics that drive mesh generation from the user-defined points:

Plane Projection (Best-Fit Plane)

When users place points, the points may be scattered across different planes perpendicular to the XZ plane. To ensure uniformity:

- The **centroid** of all placed points is calculated.
- **Principal Component Analysis (PCA)** is performed to determine the direction of least variance among the points, which defines the normal of the best-fit plane.
- All points are **projected** onto this plane, resulting in a clean snapping effect.

Clockwise Sorting

To ensure that the generated mesh forms a valid polygon, the points must be sorted in a clockwise manner:

- The **center point** (centroid) of all projected points is calculated.
- A local 2D coordinate system is defined on the plane, where the **tangent** acts as the x-axis and the **bitangent** acts as the y-axis.
- Each point is mapped to this local 2D system and sorted by the **angle** it makes with the center using the `atan2` function.

Mesh Creation

Naive Approach: Initially, a convex hull was created by connecting the sorted points directly. However, this method failed to support walls with *intentional cutouts* (e.g., occlusions), as it force-connected all points regardless of spatial context.

Improved Algorithm: A custom brute-force algorithm was developed to support complex geometries. Due to the constraint of 10 points, efficiency was not a concern.

1. Given the points in sorted order, first connect them in order (1→2→3...N→1). These edges are called **boundary edges**.

2. Starting from the third point, try to create edges from it to all the points before its parent (P3→{P1}, P4→{P2—P1}, P5→{P3—P2—P1}, etc.). These are called **interior edges**.

3. For the nth point, they naturally form triangles with edges $P_nP_{n-1}P_{n-k}$ k ranging from 2 to $n-1$.

4. Edge Validation: Interior edges must follow the criteria below:

   - They must not intersect boundary edges.
     - **Dimensional Reduction:** The algorithm projects 3D edges onto the XZ plane (ignoring height), converting a complex 3D intersection problem into a simpler 2D calculation suitable for vertical wall analysis.
     - **Parametric Solution:** Using the parametric form of line equations, the algorithm calculates a determinant to detect parallelism, then finds intersection parameters (t, u) to determine if two line segments intersect precisely when both parameters fall between 0 and 1.
   - They must not cross exterior space (when cuts are present).
     - **Custom 2D Projection:** The algorithm establishes a local coordinate system on the wall plane using cross products, then projects both the test point and polygon vertices onto this 2D system to simplify the spatial analysis.
     - **Ray Casting Implementation:** Using the even-odd rule algorithm, the system casts a horizontal ray from the test point and counts polygon edge intersections, toggling the inside/outside state with each valid intersection where one endpoint is above the ray and one below.

3.4.1 Formal Explanation

Given:

- An ordered point set $P = \{P_1, P_2, \ldots, P_n\}$ forming a simple polygon.
- Boundary edges $E^B = \{(P_i, P_{i+1}) \mid 1 \leq i < n\} \cup \{(P_n, P_1)\}$.

Interior Edges (E^I) must satisfy:

- For all $k \in [3, n]$, define $E_k^I = \{(P_k, P_i) \mid 1 \leq i \leq k-2\}$, where:
  - $(P_k, P_i) \cap E^B = \emptyset$ (Non-intersecting condition)
  - (P_k, P_i) lies in interior(P) (Non-crossing condition)

Triangulation:

$$T = \{\triangle(P_k, P_i, P_{k-1}) \mid (P_k, P_i) \in E^I\}$$

where $\triangle(a, b, c)$ denotes the triangle formed by points a, b, and c.

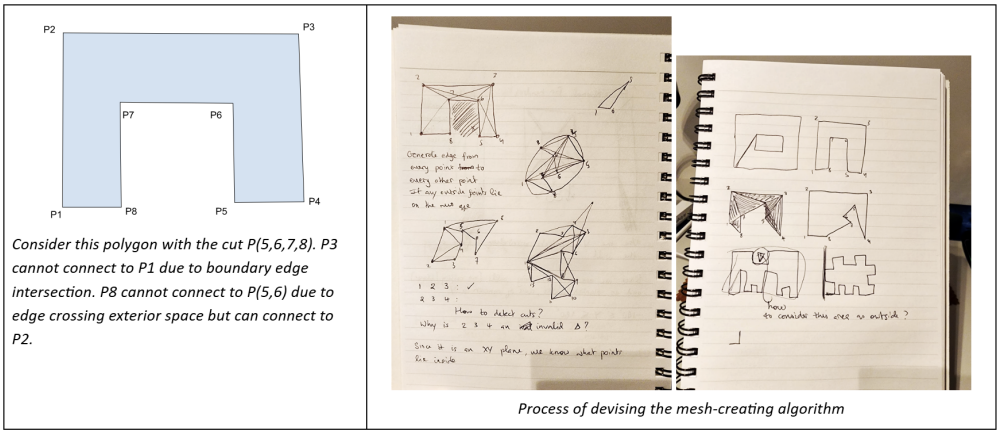

Figure 3: Graphic Display of Normal Wall Generation

3.4.2 Formal Explanation

Pseudocode: Constrained Interior Triangulation

Algorithm 1: Constrained Interior Triangulation

```
// Triangulate a polygon using constrained interior edges
Input: Ordered polygon vertices P = [P_1, P_2, ..., P_n]
Output: E^I: Set of valid interior edges, T: Set of triangles
E^B ← {(P_i, P_{i+1}) | 1 ≤ i < n} ∪ {(P_n, P_1)}
E^I ← ∅
T ← ∅
for k ← 3 to n do
    parent ← P_{k−1}
    for i ← 1 to k − 2 do
        candidate_edge ← (P_k, P_i)
        if candidate_edge does not intersect any e ∈ E^B and lies within polygon
         then
            E^I ← E^I ∪ {candidate_edge}
            T ← T ∪ {Δ(P_k, P_i, parent)}
return (E^I, T)
```

3D Extrusion

- In the previous step, the front face of the wall was created. To create the backface, we simply translate the created triangles backwards by an arbitrary constant (thickness), 0.15f in this case.
- The side faces are simple quads (2 triangles) connecting pairs of points on the front and back faces of the mesh.

Texture Mapping

- **Bounding Box Normalization**: The system calculates the minimum and maximum extents of the wall in world space, then uses these bounds to normalize each vertex's X and Z coordinates into UV space (0-1 range) through linear interpolation.
- **Planar Projection Mapping**: Texture coordinates are generated using a simple planar projection onto the XZ plane for proper scaling and alignment across the entire wall surface without distortion, regardless of the wall's shape complexity.

4 Limitations

One of the key challenges faced during the development of the application was the sensitivity of the UI elements, particularly the material and item selection menus. The scrollable sliders, while functional, were overly responsive to even slight inputs from the Meta Quest controllers. This resulted in a poor user experience, especially during demos, where users struggled to make precise selections due to unintentional slider movements. The lack of refined input filtering led to difficulty in navigating the UI smoothly, which affected the overall usability of the design customization interface.

Other Issue faced during the development was the lack of collider surfaces for repositioning the menus, the issue with this is that the canvas buttons can be interacted with using the ray controller because buttons have a default collider installed on them, but when we applied collider to the UI canvas the ray controller became confused and couldn't target the buttons or the canvas causing issues with moving the menus to a different place, this meant user has to turn around every time to face the menus and work with them rather than an elegeant solution of repositioning the canvas in question.

5 Future Goals

The ID App has multiple possibilities in the future to develop towards, since we scaled the points in X-Z plane for the wall development developing points to get them in X-Y & Y-Z planes will allow us to develop textures and materials for ground like grass, cement etc.By integrating co-routines and array of Normal and custom walls, we can go for complex constructions such as whole house and factory remodel, since these are all 3-D runtime materials generated at Unity Runtime we can have them exported as prefab and pulled into Blender or any 3D modelling software for faster modelling and printing to work on it in the real world.