Emergency Response Routing

Project Documentation

Adhurim Thaqi

ADCREA, Algorithms and Data Structures for Creative Technologies
BSc. Immersive Technologies | HSLU | FS 2026
Meta Quest 3 | Unity 6 URP | C#

1 Project Overview

Emergency Response Routing is a VR educational demonstrator built for Meta Quest 3 in Unity 6. It renders a **68-node city road network** in world space and lets a player compare three pathfinding algorithms, **A***, **Dijkstra**, and **BFS**, in real time. The scenario frames the graph as an urban street grid: each node is a road intersection, each edge is a road segment with a travel-time weight, and the player is an emergency dispatcher choosing the fastest route between two intersections.

Data Story

Emergency dispatch routing is a credible, real-world optimisation problem. Response time directly affects outcomes, so choosing the right algorithm matters. The three algorithms give visibly different behaviour on the same query, A* ignores irrelevant quadrants, Dijkstra expands uniformly in all directions, and BFS finds fewest road segments without regard to travel time, making the trade-off between search cost and path optimality immediately legible without text explanation.

What the Player Does

- On startup, the player sees a menu with three algorithm buttons (A*, Dijkstra, BFS) and a GO button.
- After pressing GO, a 68-node graph appears in world space. The player aims the right-controller ray at any two node spheres, pressing the trigger to set a start node (turns blue) then an end node (turns red).
- The chosen algorithm runs and animates step-by-step: visited nodes pulse amber then settle teal, and the final path highlights orange.
- An info bar shows live statistics: nodes visited, path cost, and path length. A speed slider controls animation pace.
- Pressing RESET returns to the algorithm-selection menu for a new query.

Property	Value
Graph nodes	68 (64 grid + 4 hub shortcuts)
Edge weights	Horizontal 30 s | Vertical 35 s | Hub diagonal 18 s
Platform	Meta Quest 3, OVR SDK, Unity 6 URP
Input	Right controller trigger or right-hand index pinch
Audio	Fully procedural, no audio asset files
Scene management	Single scene, two phases via SetActive/Destroy, no LoadScene

2 Data Structures

The project uses seven distinct data structures, each chosen for a specific algorithmic reason. This section justifies every choice.

2.1 Dictionary<int, List<GraphEdge>>, Adjacency List (CityGraph)

The core graph topology is stored as an adjacency list: a dictionary mapping each node ID to the list of edges leaving that node. This gives O(1) average-case lookup per node expansion, which is critical because Dijkstra and A* call GetEdgesFrom() on every node they settle, up to 68 times per query. A 2D array alternative would waste memory on a sparse graph (68 nodes with at most 4 neighbours each).

2.2 Dictionary<int, GraphNode>, Node Metadata Store (CityGraph)

Node positional data (graph coordinates, label) is stored in a separate dictionary keyed by node ID. This decouples topology from spatial data: traversal loops iterate the adjacency list without touching position vectors, and A*'s heuristic looks up position only for the specific nodes it needs to evaluate. The separation also makes it trivial to load a different graph layout without changing the pathfinding code.

2.3 List<int>, Path and VisitedOrder (PathfindingResult)

Both the final path (ordered node IDs from start to end) and the visited order (node IDs in the order they were settled during the search) are stored as List<int>. An ordered sequence is essential: the step-by-step animation coroutine indexes into visitedOrder to colour each node in the exact order the algorithm processed it, and the path list is iterated in order to highlight the route. Random-access by index rules out Queue or LinkedList.

2.4 Queue<int>, BFS Frontier (BfsAnalyser)

BFS uses a Queue<int> (FIFO) as its frontier. FIFO ordering is not incidental, it is the property that makes BFS correct. Nodes at hop distance k are all enqueued before any node at hop distance k+1 is dequeued, guaranteeing that the first time the destination is reached it has been reached via the fewest possible hops. Replacing the Queue with a Stack would produce depth-first behaviour and lose the hop-optimality guarantee.

2.5 Stack<int>, Path Reconstruction (All Three Algorithms)

All three algorithms record the search tree as a cameFrom dictionary (each node maps to the node it was reached from). Path reconstruction walks backwards from end to start, pushing each node onto a Stack<int>. When the start is reached, popping the stack yields the path in forward order without any array reversal. This is both cleaner and O(n) in one pass.

2.6 SortedList<double, int>, Priority Queue (Dijkstra and A*)

Dijkstra and A* need to always expand the node with the lowest cost (or f-value). SortedList<double, int> maintains sorted order automatically; index 0 always holds the minimum-cost entry, giving O(1) dequeue and O(log n) insert.

Why double, not float: A float has a ULP (unit in the last place) of approximately 0.00005 at values around 600. The tiebreaker suffix added to each key is insertSeq x 1e-9. At float precision, this suffix becomes invisible above ~84 s of accumulated cost, causing key collisions that freeze the SortedList. Using double (ULP ~1e-12 at 10000) makes the suffix distinguishable for up to 10^9 insertions, well beyond any realistic path cost in this graph.

Key uniqueness scheme: key = cost + (insertSeq x 1e-9). Each insertion increments insertSeq, so identical costs never produce duplicate keys, and the sorted order remains stable.

2.7 HashSet<int>, Settled / Visited Sets (Dijkstra, A*, BFS)

Once a node is settled (optimal cost confirmed), it must not be re-processed. HashSet<int> provides O(1) Contains() checks, called once per neighbour per expansion. With up to 68 x 4 = 272 such checks per query, using a List (O(n) Contains) would be measurably slower and semantically wrong for a set operation.

Data Structure	Used In	Reason for Choice
Dictionary<int, List<GraphEdge>>	CityGraph adjacency list	O(1) node lookup, memory-efficient for sparse graph
Dictionary<int, GraphNode>	CityGraph node metadata	Decouples topology from spatial data
List<int>	PathfindingResult path and visitedOrder	Ordered sequence, random access by index for animation
Queue<int>	BfsAnalyser frontier	FIFO guarantees level-by-level expansion and hop optimality
Stack<int>	Path reconstruction	Natural reversal of cameFrom chain in O(n)
SortedList<double, int>	Dijkstra and A* priority queue	O(1) min dequeue, O(log n) insert; double avoids float ULP freeze
HashSet<int>	Settled / visited sets	O(1) Contains check prevents re-processing settled nodes

3 Algorithms

3.1 Breadth First Search (BFS)

File: Assets/Scripts/Algorithms/BfsAnalyser.cs

Internal structures: Queue<int> frontier, HashSet<int> visited, Dictionary<int,int> cameFrom, Stack<int> for reconstruction

BFS expands nodes level by level. It dequeues the front of the queue, marks it visited, and enqueues all unvisited neighbours. Because FIFO ordering means nodes at hop distance k are fully exhausted before hop distance k+1 begins, the first time BFS reaches the destination it is guaranteed to have used the fewest possible road segments (hops).

Edge weights are ignored. BFS treats every edge as cost 1. After finding the fewest-hops path, the actual travel time is calculated separately by summing edge weights along that path and stored in PathfindingResult.totalCost for comparison. This proves that a fewest-hops route can have a longer travel time than the weighted-optimal route.

Visualisation: Visited nodes appear as concentric green rings spreading outward from the start. The info bar shows Hops (road segments in path) and Weight (travel time of that hop-optimal path).

3.2 Dijkstra's Algorithm

File: Assets/Scripts/Visualization/PathfindingService.cs, RunDijkstra()

Internal structures: SortedList<double,int> frontier, HashSet<int> settled, Dictionary<int,float> dist, Dictionary<int,int> cameFrom, Stack<int> for reconstruction

Dijkstra always expands the unsettled node with the lowest accumulated cost from the start. It updates neighbour distances if a cheaper route is found, and settles each node exactly once. Because it uses no heuristic, it must explore in all directions uniformly, on a corner-to-corner query it typically settles 60-68 of 68 nodes before reaching the destination.

Optimality: Dijkstra is guaranteed to find the minimum total travel time because it only settles a node when no shorter path to it can possibly exist (proven by the greedy argument on non-negative weights).

Visualisation: An amber/teal expanding disc. The search frontier is wide and uniform. The info bar shows Path length, Weight, and Visited count.

3.3 A* Algorithm

File: Assets/Scripts/Visualization/PathfindingService.cs, RunAStar()

Internal structures: SortedList<double,int> frontier, HashSet<int> settled, Dictionary<int,float> gCost, Dictionary<int,int> cameFrom, Stack<int> for reconstruction

A* extends Dijkstra by adding a heuristic h(n) to every priority key. Instead of sorting by g(n) (cost from start), A* sorts by f(n) = g(n) + h(n) (cost from start plus estimated cost to goal). Nodes in the direction of the destination are expanded first, allowing A* to skip whole quadrants that Dijkstra must settle.

Heuristic: h(n) = Euclidean distance to goal / 0.08. The constant 0.08 is the maximum road speed in the graph (hub diagonal edge: sqrt(2) / 18 s ~~ 0.0786 units/s). Dividing by the maximum possible speed gives the minimum possible remaining travel time, the heuristic never overestimates, so A* is **admissible** and guaranteed optimal.

Visualisation: A focused cone toward the destination. Typically visits 20-35 nodes on a corner-to-corner query versus Dijkstra's 60-68. Same optimal path, far fewer wasted expansions.

Property	A*	Dijkstra	BFS
Optimises for	Min. travel time	Min. travel time	Fewest road segments
Uses edge weights	Yes	Yes	No
Uses heuristic	Yes (Euclidean / 0.08)	No	No
Nodes visited (corner-to-corner)	~20 to 35	~60 to 68	~50 to 68
Guaranteed optimal	Yes (travel time)	Yes (travel time)	Yes (hop count)
Frontier colour	Amber settling to teal	Amber settling to teal	Green wave
Path colour	Orange	Orange	Orange

4 Glossary

Term	Definition
Edge weight	Travel time in seconds for one road segment. Horizontal = 30 s, vertical = 35 s, hub shortcut = 18 s. Dijkstra and A* minimise total weight along the path. BFS ignores weights entirely.
Visited nodes	Intersections the algorithm explored during its search, not the same as the final path. Shown as a ratio, e.g. 18/68. A lower ratio means a more directed search. A* achieves the lowest ratio.
Hops	The number of road segments in the path. Always equal to path node count minus 1. BFS minimises hops. A* and Dijkstra minimise total travel time, not hop count, their paths may use more segments if the weighted cost is lower.
Why BFS shows Weight	BFS finds the fewest-hops path, then calculates the travel time of that specific path afterwards for comparison. This demonstrates that fewer road segments does not mean shorter travel time, a BFS path may cross slower roads that A* and Dijkstra avoid.
Admissible heuristic	A heuristic h(n) is admissible if it never overestimates the true cost to the goal. A* is only guaranteed optimal when its heuristic is admissible. The heuristic h(n) = Euclidean / 0.08 is admissible because 0.08 is the maximum road speed, so the estimate is always at or below the true minimum remaining cost.
Settled node	A node whose optimal cost from the start has been confirmed and will never be improved. Dijkstra and A* settle each node exactly once. Settled nodes are stored in a HashSet to prevent re-processing.
Priority queue	A data structure that always returns the element with the lowest key. Dijkstra and A* use SortedList<double,int> as their priority queue. Index 0 is the minimum-cost entry.
FIFO	First In, First Out. The ordering property of Queue<int>. FIFO is what makes BFS expand nodes level-by-level and guarantees the hop-optimal path.

5 Console Output Reference

The following Debug.Log messages are emitted at runtime. All are tagged with a class prefix.

Log Message	Source / Trigger	Meaning
Graph data built: 68 nodes	CityGraphFactory, Awake()	CityGraph built in memory. 64 grid nodes + 4 hub nodes. No visuals yet.
GraphVisualiser: spawned 68 nodes	GraphVisualizer.Initialize(), GO pressed	Node spheres and edge LineRenderers created in world space.
Start node: X, select end node	VRNodeSelector, first trigger press	First node picked. Node turns blue. System waiting for second selection.
End node: X, running algorithm	VRNodeSelector, second trigger press	Second node picked. Node turns red. Chosen algorithm fires immediately.
[AStar] Path: 9 nodes Weight: 160.0 Visited: 18/68	VRAlgorithmController, A* complete	A* result. 9 nodes in path, 160 s total cost, 18 of 68 nodes searched.
[Dijkstra] Path: 9 nodes Weight: 160.0 Visited: 44/68	VRAlgorithmController, Dijkstra complete	Dijkstra result. Same path and cost as A*, but 44 nodes visited, no heuristic.
[BFS] Hops: 6 Path weight: 195.0 Covered: 35/68	VRAlgorithmController, BFS complete	BFS result. 6 hops (fewest segments), but 195 s, slower than weighted algorithms.
No path found	VRAlgorithmController, pathFound = false	No route exists between the two selected nodes (disconnected graph region).
Reset to menu	VRAlgorithmController.OnResetPressed()	RESET pressed. VizUI and graph destroyed. Menu restored.

6 VR Info Bar Display

The info bar is a world-space TextMeshPro panel positioned below the graph (Y = 0.095, Z = 2.5 m). VRInfoBar polls VRAlgorithmController.StatusMessage every frame. The following table shows the text displayed in each application state.

State	Info Bar Text	Field Explanation
Waiting for node selection	Select two nodes to start	No nodes selected yet. Prompt to aim and trigger.
Start selected, waiting for end	Start: Node X, select end node	Shows ID of selected start. Waiting for second trigger press.
Algorithm animating	Searching... Visited: N / 68 Cost: X s	Live update each step. N increments as each node is settled.
A* / Dijkstra path found	Path: N nodes Weight: X s Visited: N / 68	Path = number of nodes in route. Weight = total travel time. Visited = search cost.
BFS path found	Hops: N Weight: X s Covered: N / 68	Hops = road segments in path. Weight = travel time of that hop-optimal path. Covered = BFS nodes reached.
No path found	No path found. Press RESET	Destination unreachable from start in current graph.

7 Scene Object Hierarchy

The scene contains two root-level objects. All logic components are on SystemRoot. OVRCameraRig is the standard Meta SDK camera rig.

```
Scene Root
  OVRCameraRig
    OVRManager               [Meta Quest SDK manager]
    TrackingSpace
      RightHandAnchor        [ray source for pointer and node selector]
      LeftHandAnchor         [unused, right hand only by design]

  SystemRoot                 [all logic lives here]
    GameBootstrapper         [scene entry point, owns CityGraph]
    CityGraphFactory         [builds 68-node graph from code]
    GraphVisualizer          [spawns / destroys node spheres and edge lines]
    VRSceneBuilder           [builds and destroys UI panels]
    VRAlgorithmController    [orchestrates algorithm selection and animation]
    VRNodeSelector           [ray-cast node picking, Phase 2]
    AudioManager             [added at runtime in Start()]
    VRPointerInteractor      [added at runtime in Start()]
    VRSlider                 [animation speed slider]

  MenuUI  [Phase 1 only, SetActive(false) in Phase 2]
    PanelMenu
    BtnAStar    + AlgorithmButtonWirer
    BtnDijkstra + AlgorithmButtonWirer
    BtnBFS      + AlgorithmButtonWirer
    BtnGo       + GoButtonWirer
    Label_SelectAlgorithm

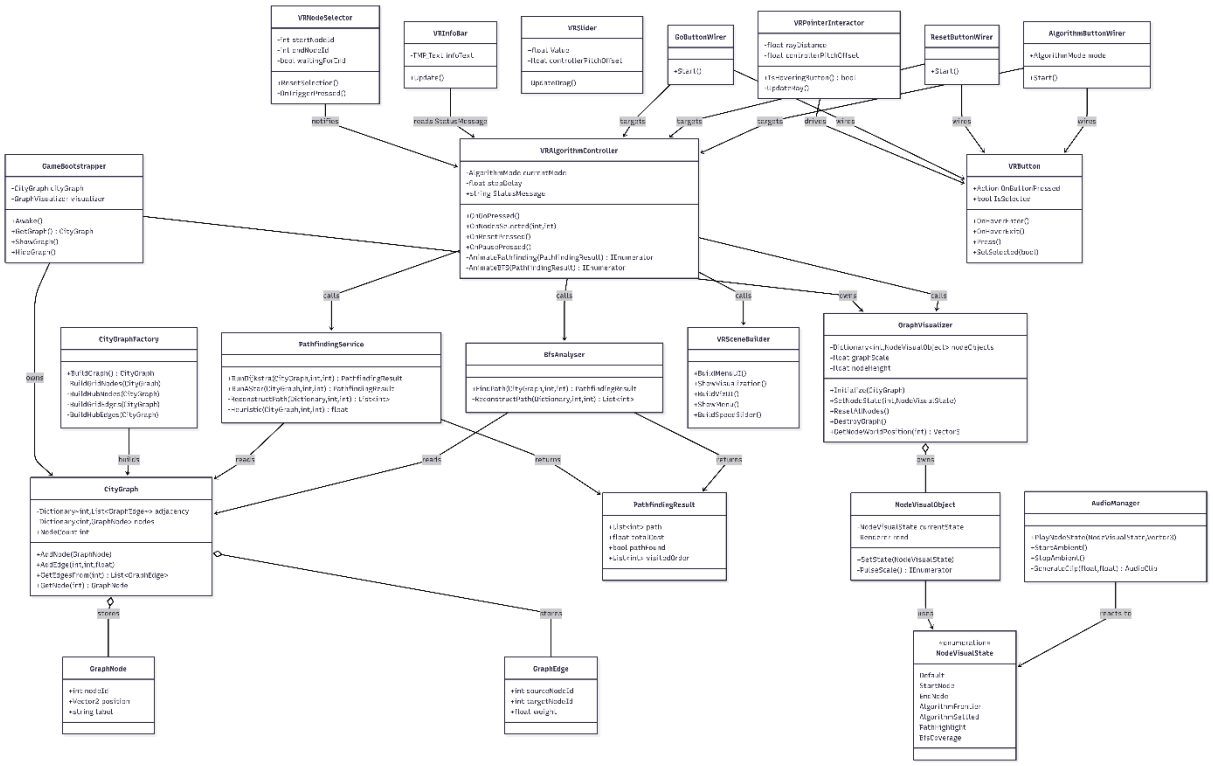
  VizUI   [Phase 2 only, created on GO, Destroy() on RESET]
    BtnReset        + ResetButtonWirer
    SpeedSliderTrack
    SpeedSliderHandle
    PanelInfoBar    + VRInfoBar
    InfoText          [TextMeshPro]

  Nodes   [Phase 2 only, Destroy() on RESET]
    Node_0_NW Corner ... Node_67_SE Hub  [Sphere + NodeVisualObject each]

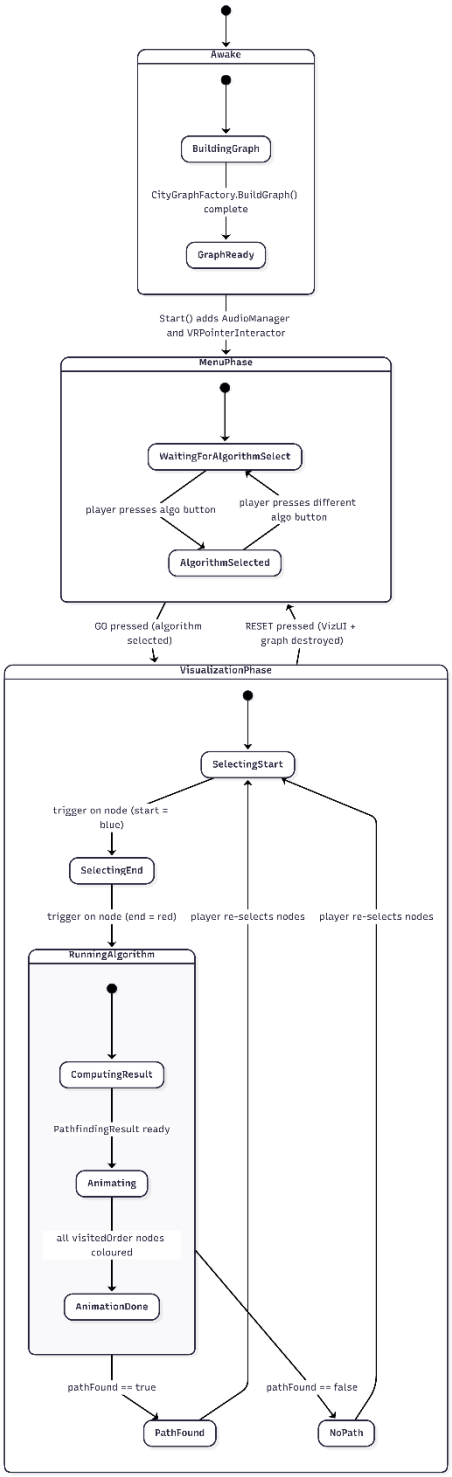
  Edges   [Phase 2 only, Destroy() on RESET]
    Edge_0_1 ... Edge_N_M  [LineRenderer each]
```

8 Class Diagram

The following class diagram shows all major classes, their key fields and methods, and their relationships.

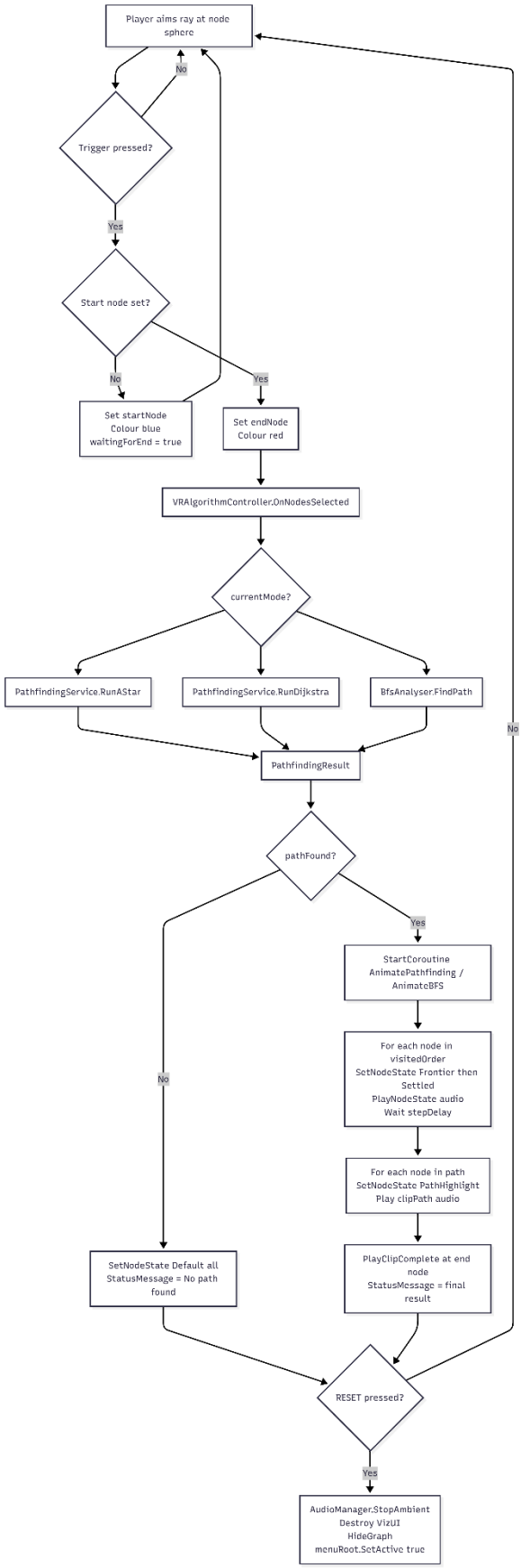


9 App State Machine



The following state diagram shows the complete application lifecycle, including all transitions driven by user input and algorithm events.

10 Algorithm Execution Flowchart



This flowchart shows the complete execution path from node selection through to RESET.

11 Key Design Decisions

11.1 Wirer Pattern (No Lambda Expressions)

The course rules prohibit lambda expressions. Unity button callbacks are commonly written as lambdas (button.onClick.AddListener(() => DoThing())). To comply, each button gets a dedicated MonoBehaviour wirer class (GoButtonWirer, ResetButtonWirer, AlgorithmButtonWirer, etc.). Each wirer's Start() method resolves references to VRButton and VRAlgorithmController, then assigns a named method to the VRButton.OnButtonPressed System.Action. This avoids lambdas entirely and produces a one-class-per-button pattern that is also easier to inspect in the Unity Editor.

11.2 SortedList with Double Keys (Float ULP Fix)

Dijkstra and A* need a priority queue sorted by cost. The first implementation used SortedList<float, int> with a tiebreaker suffix (insertSeq x 1e-7). This worked for short paths, but on longer queries the SortedList would throw a duplicate-key exception and freeze.

Root cause: float has a unit in the last place (ULP) of approximately 5e-5 at values around 600. A suffix of insertSeq x 1e-7 is smaller than the float ULP at these values, the suffix is rounded away during float arithmetic, producing identical keys. SortedList cannot hold duplicate keys and throws ArgumentException.

Fix: switch the key type to double. Double ULP at 10000 is approximately 1e-12, making the suffix (insertSeq x 1e-9) safely distinguishable for up to 10^9 insertions. All path costs in this graph are below 1000 s, so double precision is sufficient with wide margin.

11.3 Procedural Audio (No Audio Asset Files)

AudioManager generates all clips programmatically in Awake() using Unity's AudioClip.Create() and filling sample buffers with sine waves, noise, and envelopes. This means the project has zero audio asset files, there is nothing to import, no Unity meta files to manage, and the entire audio system is reproducible from code alone. It also demonstrates procedural generation as a data-structures concept (audio buffers as float arrays).

All node-state clips are played spatially (spatialBlend = 1.0) at the node's world position. A custom PlayAtPoint method is used instead of AudioSource.PlayClipAtPoint because the built-in static method does not expose rolloff settings. Linear rolloff (minDistance = 0.4 m, maxDistance = 6.0 m) is required so distant nodes are audibly quieter but not inaudible.

11.4 Two-Phase Scene (Single Scene, No LoadScene)

The application has two visible states, menu and visualization, but uses only one Unity scene. Phase transitions are achieved by calling SetActive(false) on the MenuUI root (hides menu instantly) and Destroy() on the VizUI and Nodes/Edges roots (frees memory after RESET).

Advantages over LoadScene: zero load time for phase transitions; algorithm state and graph data are never unloaded between runs; the Debug console is uninterrupted; and OVRCameraRig does not need to survive a scene reload.

11.5 Runtime-Only GameObject Spawning (No Prefabs)

All GameObjects, node spheres, edge LineRenderers, buttons, sliders, and info panels, are spawned purely from code in Awake() or Start(). This was a deliberate choice to keep the Unity project self-contained: there are no prefab assets to link, no missing-reference warnings on a fresh clone, and all visual parameters (sizes, colours, positions) are defined as named constants in VRSceneBuilder and GraphVisualizer. The trade-off is that changes require code edits rather than Inspector tweaks.

11.6 Right-Hand-Only Input

The interaction model uses the right hand exclusively, either the OVRInput controller trigger or right-hand index finger pinch via hand tracking. The left hand is not wired. This constraint was intentional: a single-hand model eliminates ambiguity about which hand performs which action, simplifies the VRPointerInteractor and VRNodeSelector code, and works equally well in controller mode and hand-tracking mode without branching logic.