Automated Path Prediction for Redirected Walking Using Navigation Meshes

...

Conference
2016 IEEE Symposium on 3D User Interfaces (3DUI)

Authors

1. Mahdi Azmandian*
2. Timofey Grechkin*
3. Mark Bolas *†
4. Evan Suma*

* USC Institute for Creative Technologies
† USC School of Cinematic Arts

Preface

- To simulate a large virtual environment in a fixed small physical tracked area.
- How to simulate an infinite corridor in a small room?
- How to provide an illusion of a huge space in a small room?
- How to steer the user away from the physical area's boundary?

Redirected Walking (RDW)

- Introduce subtle, unnoticeable discrepancies between the user's physical and virtual motions.
- Discrepancies build up over time - physical and virtual trajectories diverge.
- Strategically steer users away from the boundaries of the physical tracked space to enable exploration of large virtual environments.

Example - Infinite corridor

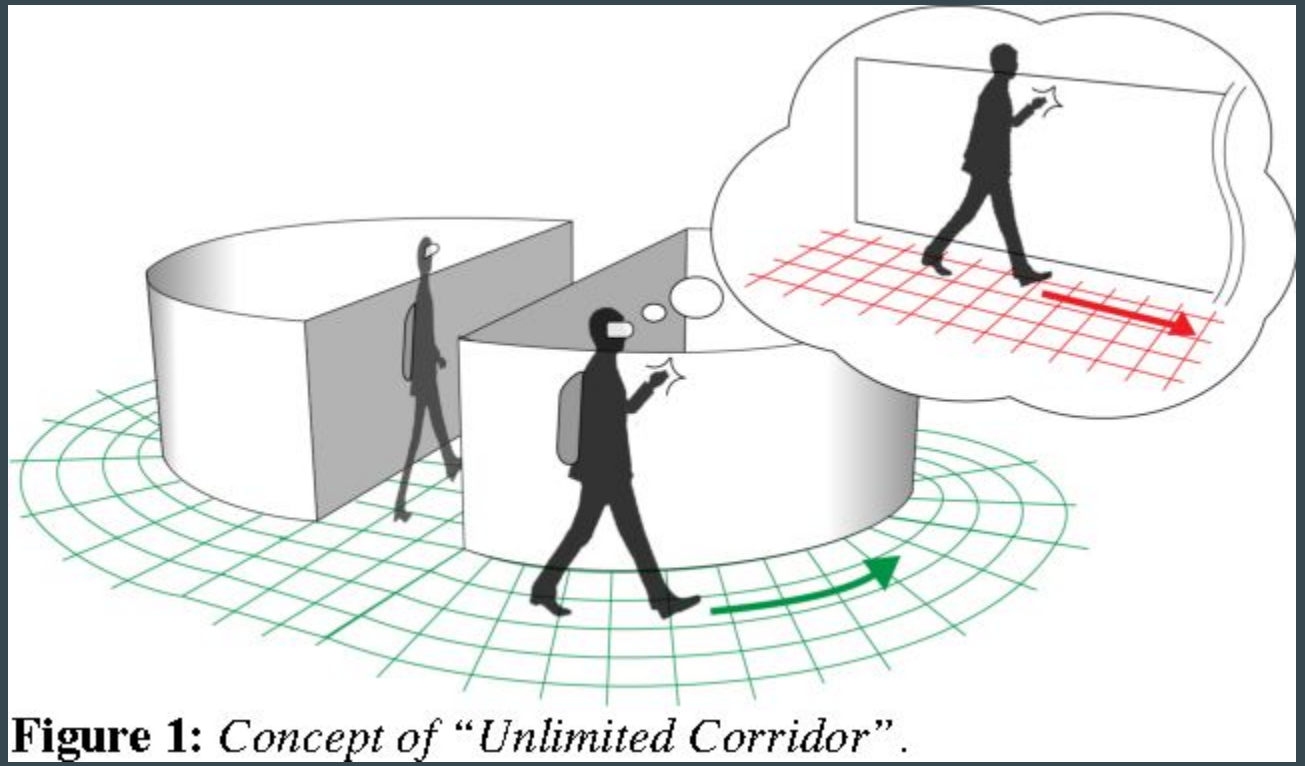
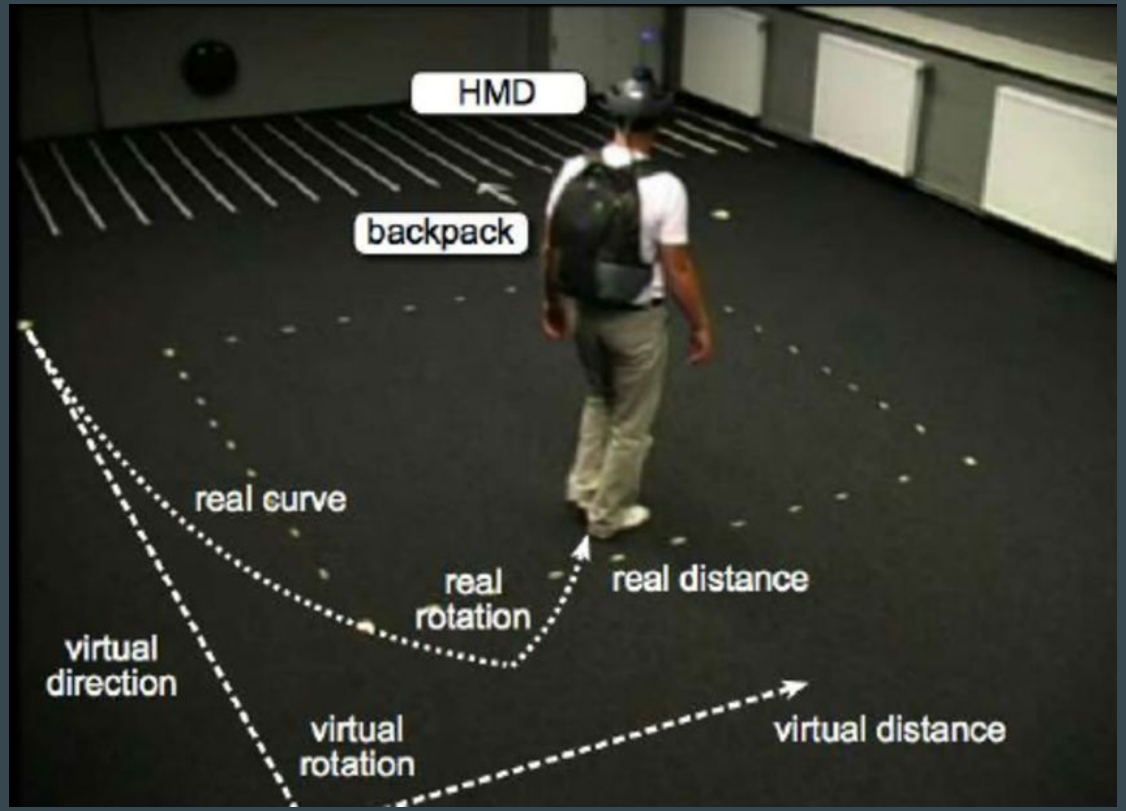


Figure 1: *Concept of "Unlimited Corridor".*

Example - RDW


HMD
backpack
real curve
real rotation
real distance
virtual direction
virtual rotation
virtual distance

Existing Algorithms for RDW

- Steer-To-Center
- Steer-To-Orbit
- MPCRed
- FORCE

Why the need for automation?

Need for automation

- The adoption of such algorithms requires annotation of virtual environments with graphs describing possible user trajectories.
- When done manually such annotations can be both tedious to generate and insufficiently flexible at run-time.
- Significant deviation from expected path while exploring open areas - affects performance.

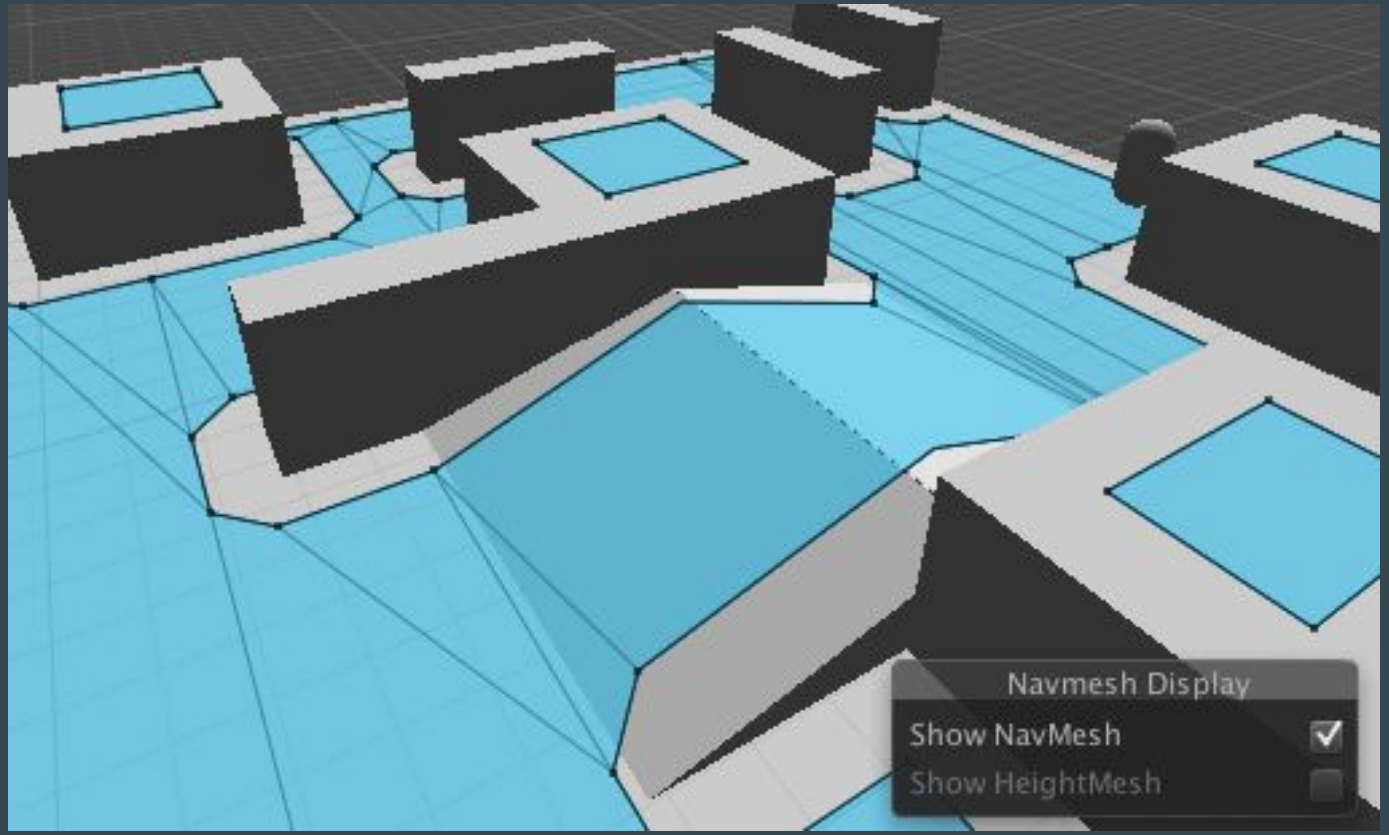
Proposed Solution

- Automatically predict user's short term trajectories using Navigation Meshes.
- No need to manually annotate the environment to a graph.
- Dynamic adjustments to path predictions based on current user position.
- Generate a short term graph at user's position and use existing algorithms on it.

Details

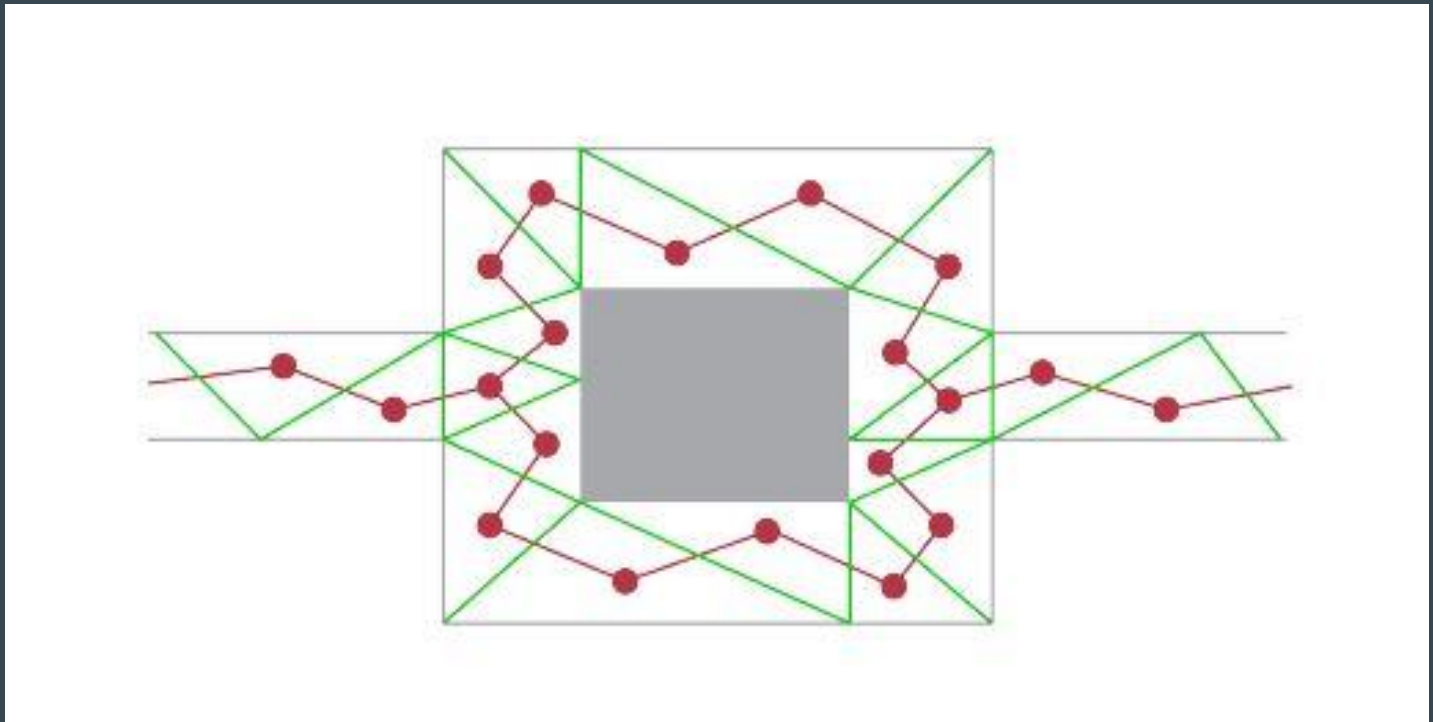
- Model the environment as navmeshes.
- These can be generated automatically - well known algorithms exist.
- Breaks the environment down into convex polygons
- Readily available in commercial game authoring platforms such as Unity3D and Unreal.

Navigation Mesh



The Algorithm

Step 1: Model the navmesh as a navgraph - each polygon is a node.



The Algorithm - continued

Step 2: Find the node *S* in the graph corresponding to the user's current location.

Step 3: Use this graph to find all possible paths originating from *S* and not exceeding path length *d*.

Step 4: Obtain a set of *branching nodes B* and set of *terminal nodes T* and the *connectivity function prevPG[]* between the nodes in *B* and *T*.

The Algorithm - Last Step

Step 5: Construct the prediction graph. Define V as the union of starting node S, branch nodes B, and terminal nodes T.

Using the information encoded in *prevPG[]*, we add appropriate edges between the vertices. We also add edges lost between neighbours in the original navgraph.

Step 6: Perform navmesh funnelling which yields shortest path from the polygons.

Step 7: Add the shortest path between each vertex v and its predecessor *prevPG[v]*.

Algorithm 1 Generate path prediction graph

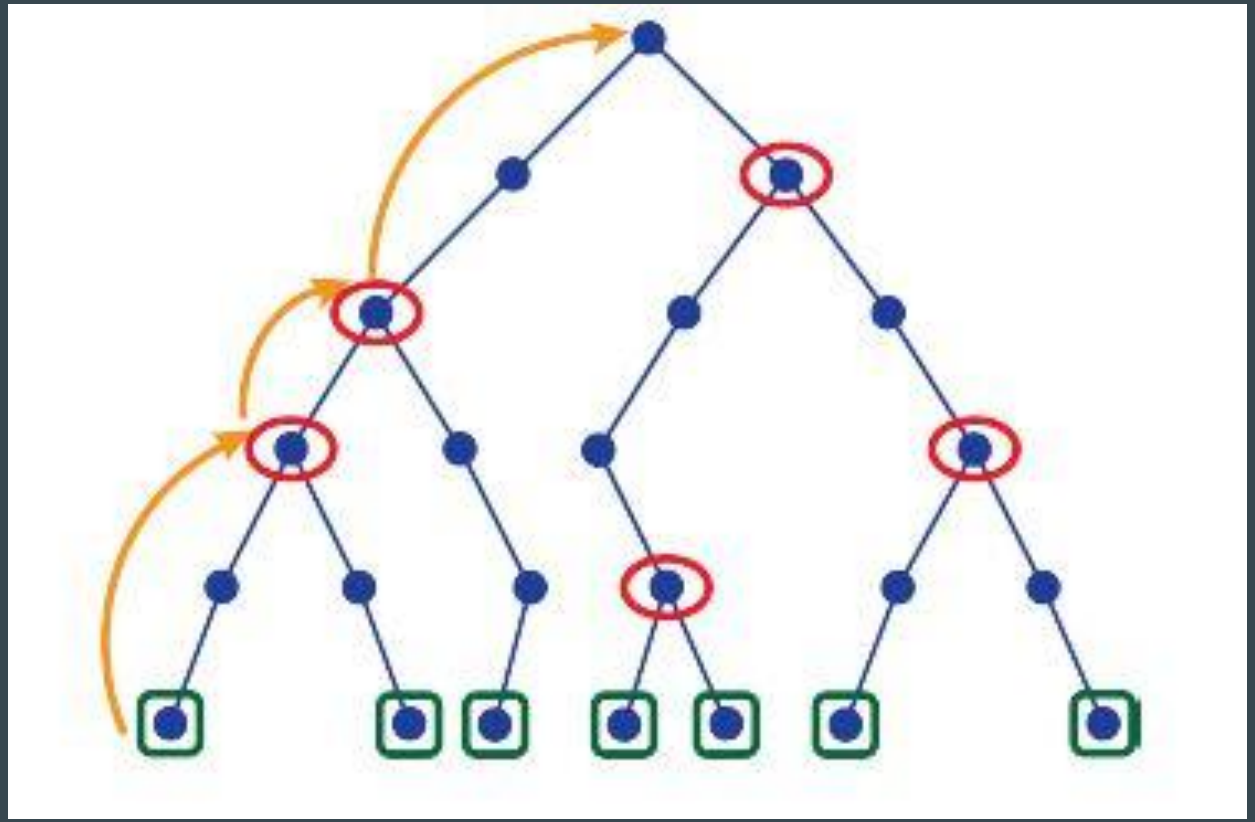
```
function PathPredictionGraph(Pos, NavGraph, d):
    Find polygon node S in NavGraph containing Pos
    S.position ← Pos
    B, T, prevPG[] ← DepthLimitedDijkstra(NavGraph, S)
    Define V ← B ∪ T ∪ S
    for all vertices u and v in V do
        if {u,v} ∈ NavGraph.edges then
                add {u,v} to E
    for all vertices v in V do
        V ← V ∪ path(v, prevPG[v]).vertices
        E ← E ∪ path(v, prevPG[v]).edges
    return V, E
```

```
function DepthLimitedDijkstra(Graph, source, d):
    Define vertex sets Q, B, and T
    for all vertex v in Graph do
        dist[v] ← INFINITY
        prev[v] ← NULL
        add v to Q
    dist[source] ← 0
    while Q is not empty do
        u ← vertex in Q with min dist[u]
        remove u from Q
        for all neighbor v of u do
                alt ← dist[u] + length(u, v)
                if alt < dist[v] then
                        dist[v] ← alt
                        prev[u] ← u
        if deg(u) > 2 then
                add u to B
        if T contains prev[u] then
                remove prev[u] from T
        add u to T
        if B contains prevPG[u] then
                prevPG[u] ← prev[u]
        else
                prevPG[u] ← prevPG[prev[u]]
        if path(source, u) is longer than d then
                break {path() return shortest navigable path}
    return B, T, prevPG[]
```

Modifications in Dijkstra's algorithm

- The depth of the tree is limited by the maximum length of the path d.
- The algorithm constructs a set of branching nodes B and terminal nodes T and keeps the track of the connectivity between these nodes within the search tree using function *prevPG[ ]*.
- If $\deg(u) > 2$, we add this vertex to a set of branching nodes B

Output of modified Dijkstra's algorithm



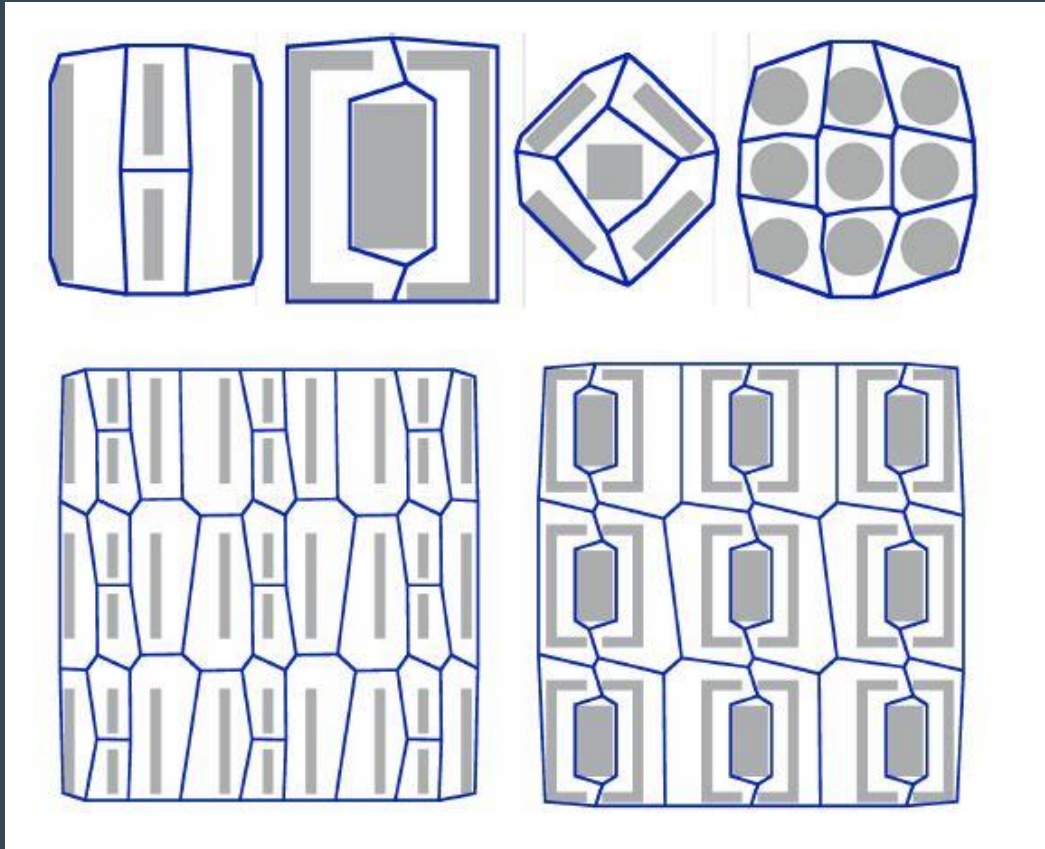
- Green nodes are terminal nodes.
- Red nodes are branch nodes.
- Each node has a pointer to its parent branch/source.
- Path length is restricted = *d*.

Implementation

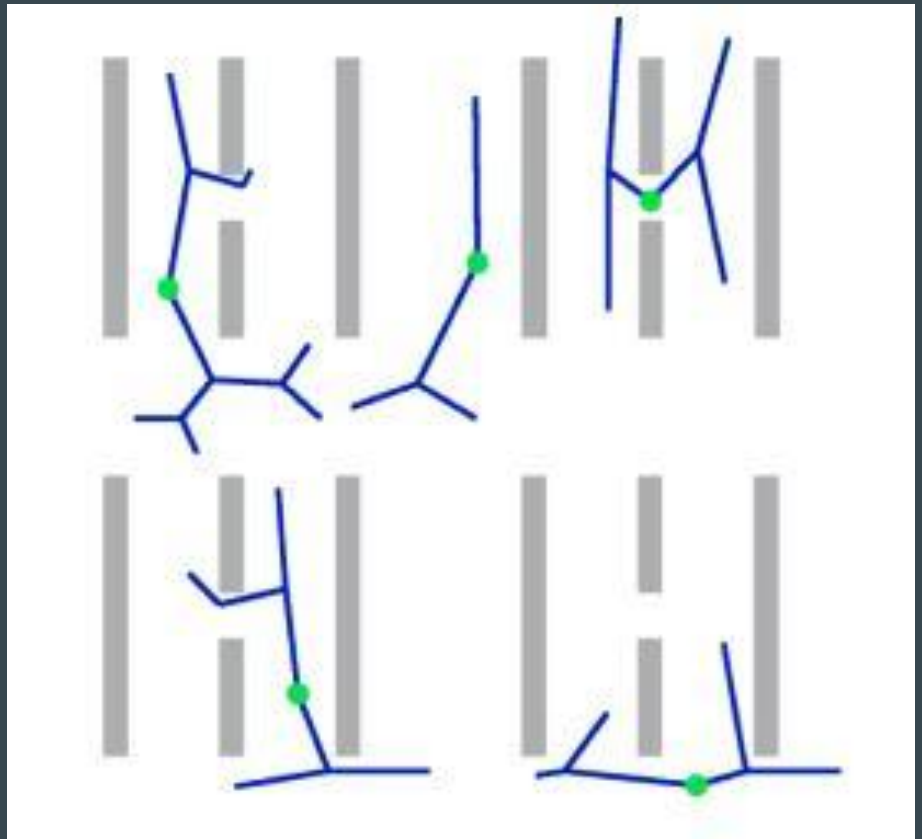
- Implemented in Unity3D
- Instead of using in built shortest path finding algorithms, we used an addon package from the Unity Asset Store.
- The A* Pathfinding API exposes the navigation graph data structure, making implementation of our algorithm much more straightforward.

Results

Using the algorithm on full environments (very large *d*)

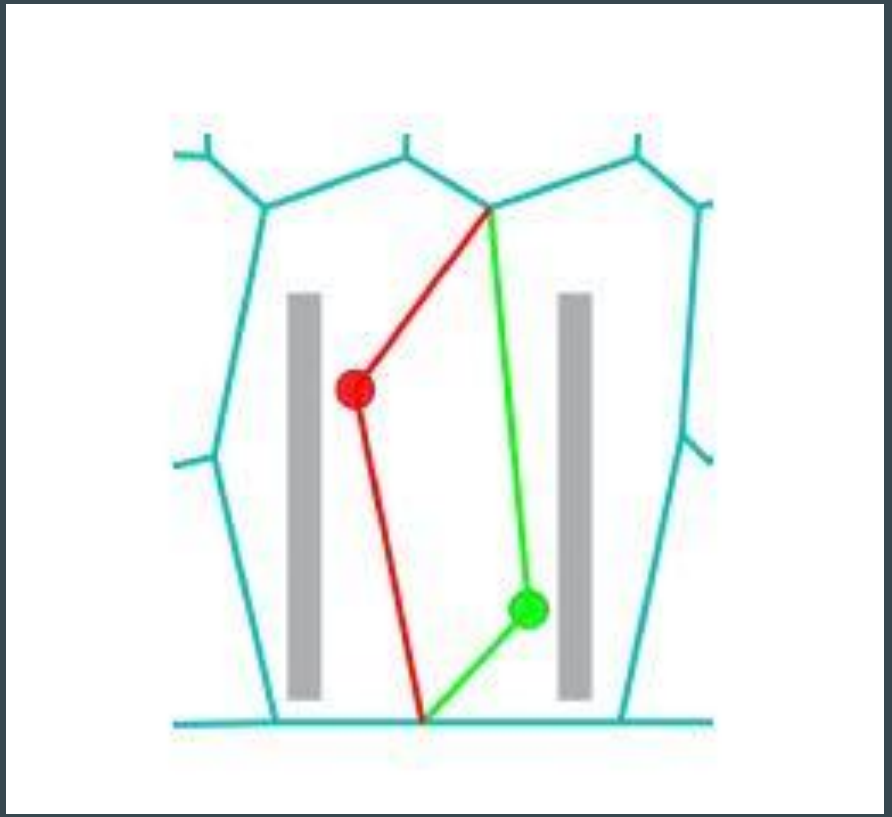


The algorithm in a practical situation (small *d*)



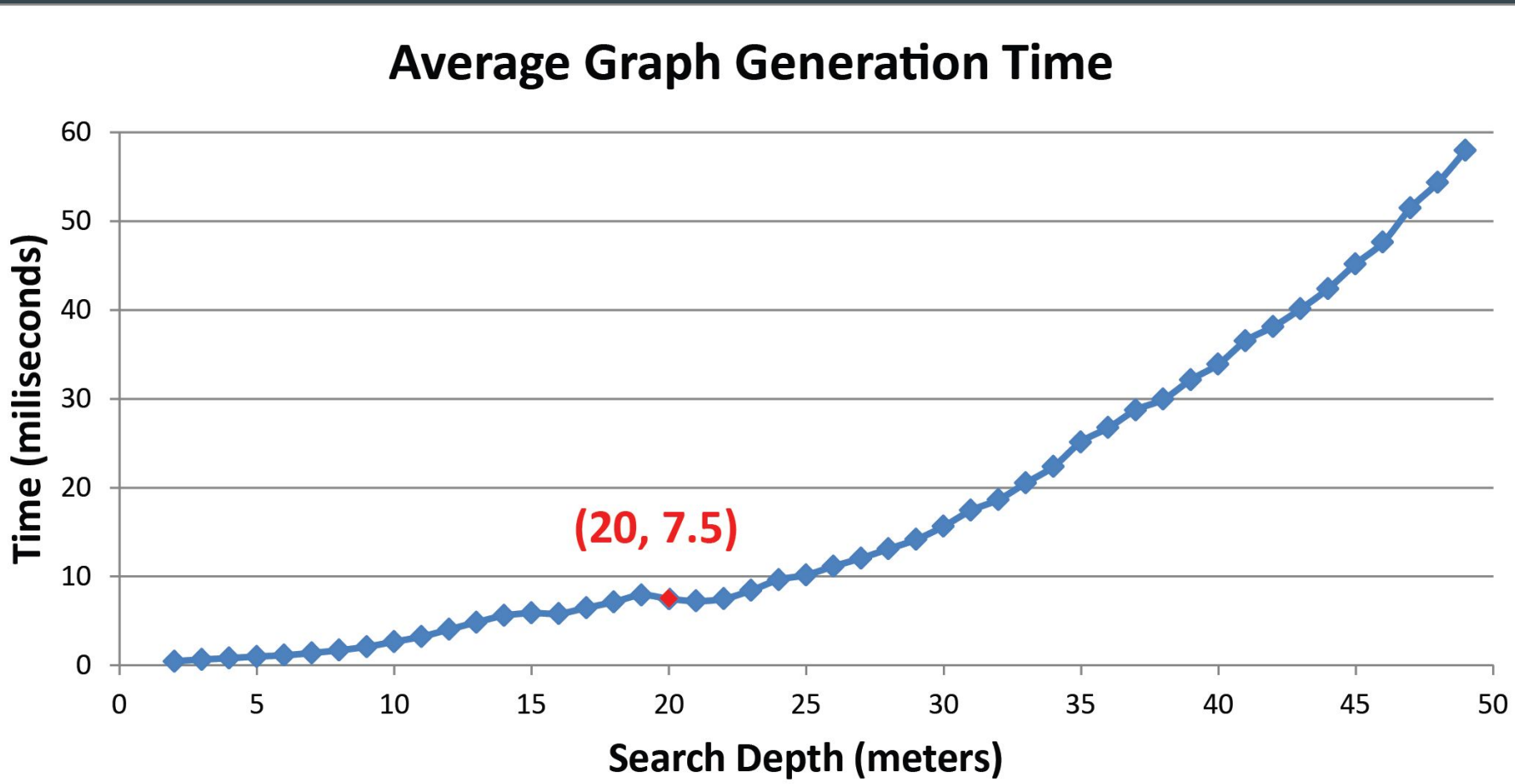
Dynamic graph generation

- When a manual annotation is used, path prediction is based on approximating user position by the nearest point on the graph.
- Our algorithm periodically generates short-term path prediction graph relative to current location of the user.



Graph generation time

- Consider time complexity relative to maximum path length.
- Each point has been averaged over 25 random starting points in the environment.
- On average, our algorithm generated path prediction graphs with search horizon up to 20 meters in 7.5 milliseconds.
- This compares to 82.5 milliseconds for the average planning phase execution of the MPCRed algorithm.
- Thus we can provide updates to the RDW algorithms on every computation cycle.


Average Graph Generation Time
Time (miliseconds)
60
50
40
30
20
10
0
(20, 7.5)
0
5
10
15
20
25
30
35
40
45
50
Search Depth (meters)

Conclusion

- We have proposed an algorithm to automate path prediction for planning RDW algorithms.
- Tedious manual annotation of an entire environment can be replaced with automatically generated prediction graphs.
- These graphs are local to the user's current position and change dynamically.
- Allows advanced prediction techniques such as forecasting user behavior in non-static environments.

Image credits

1. Matsumoto, Keigo et al. “Unlimited corridor: redirected walking techniques using visuo haptic interaction.” *SIGGRAPH Emerging Technologies* (2016).
2. Steinicke, Frank & Bruder, Gerd & Hinrichs, Klaus & Jerald, Jason & Frenz, Harald & Lappe, Markus. (2011). Real Walking through . . .. JVRB - Journal of Virtual Reality and Broadcasting ; 6(2009) , 2.
3. https://forum.unity.com/threads/converting-unity-navmesh-to-regular-mesh-via-script-leads-to-imperfect-result.276893/

Thank You