Automated Path Prediction for Redirected Walking Using Navigation Meshes

Mahdi Azmandian[*] Timofey Grechkin[*] Mark Bolas[*†] Evan Suma[*]

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

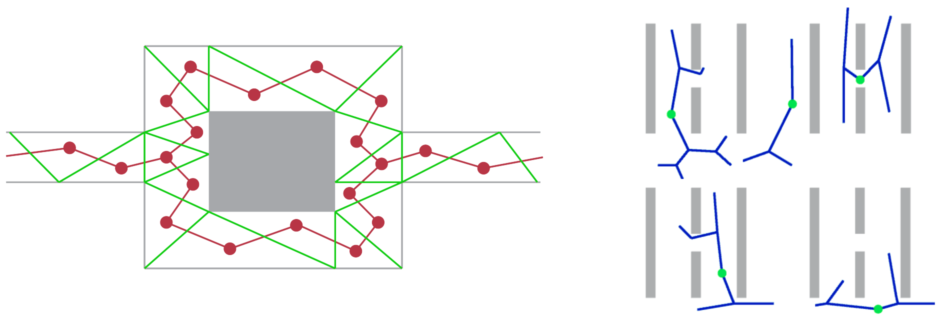

Figure 1: Navigation meshes and short term path prediction graphs. (left): Sample virtual environment overlayed with navigation mesh polygons shown in green and navigation graph shown in red. Note that edges of navigation graph represent connectivity between polygons rather than prediction of user's possible path. (right): Sample short-term path prediction graphs generated from various starting points. The starting point is shown as green dot.

Abstract

Redirected walking techniques have been introduced to overcome physical space limitations for natural locomotion in virtual reality. These techniques decouple real and virtual user trajectories by subtly steering the user away from the boundaries of the physical space while maintaining the illusion that the user follows the intended virtual path. Effectiveness of redirection algorithms can significantly improve when a reliable prediction of the users future virtual path is available. In current solutions, the future user trajectory is predicted based on non-standardized manual annotations of the environment structure, which is both tedious and inflexible. We propose a method for automatically generating environment annotation graphs and predicting the user trajectory using navigation meshes. We discuss the integration of this method with existing redirected walking algorithms such as FORCE and MPCRed. Automated annotation of the virtual environments structure enables simplified deployment of these algorithms in any virtual environment.

Index Terms: H.5.1 [Information Interfaces and Presentation]: Multimedia Information Systems—Artificial, augmented, and virtual realities; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction techniques; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual reality

1 Introduction

It is highly desirable for many practical applications to enable natural exploration of interactive virtual environments in a manner similar to the way people move in the real world. Physical walking in a tracked space has many advantages over other locomotion methods, however the size of the virtual environment that can be explored is ultimately constrained by the physical dimensions of the tracked area.

Redirected Walking (RDW) attempts to address this issue by introducing subtle, unnoticeable discrepancies between the user's physical and virtual motions. Due to accumulation of these discrepancies user trajectories in real and virtual environment diverge over time. As a result, RDW algorithms can strategically steer users away from the boundaries of the physical tracked space to enable exploration large virtual environments.

Original RDW algorithms such as Steer-To-Center consistently steer the user toward a particular goal in the real environment (such as the center of the tracked space) without taking into account the user's possible future travel direction in the virtual space. While this approach works well, the performance of RDW algorithms can be significantly improved using short-term path prediction that describes the user's navigation options given the the layout of the virtual environment. In the simplest case such path prediction assumes that all possible travel directions are equally likely. In recent years algorithms such as MPCRed [4] and FORCE [10] attempted to capitalize on the path-prediction approach to improve redirection efficiency. 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. Users can significantly deviate from expected path when navigating wide open areas or circumventing dynamic obstacles, which may impact performance of RDW algorithms.

In this work we propose an algorithm for automatically predicting the user's possible short term trajectories, which relies on *navigation meshes*, a method commonly used for path planning by computer-controlled characters in gaming applications. Our algorithm can be automatically deployed in an arbitrary virtual environment without the need for manual layout annotation and can dynamically adjust path predictions relative to actual user position at run-time. Instead of imposing a fixed layout graph to the environment, we dynamically generate a local short-term path prediction

[*]e-mail: {mazmandian, grechkin, bolas, suma}@ict.usc.edu

graph with fixed search horizon originating at user's exact position. This path prediction graph can be directly passed to existing redirection algorithms. Our proposed pipeline is efficient, standardized, and easily deployable.

2 Background

2.1 Redirected Walking

Redirected walking was first introduced by Razzaque [5] as a potential solution to the problem of exploring large virtual environments using natural locomotion within a limited physical tracked space. This is made possible by the human vision dominating the vestibular system, which means that visual illusions can be used to "steer" unsuspecting users away from the boundaries of physical tracked space.

The method works by injecting subtle discrepancies between the user's real and virtual motions. These discrepancies may include gains (faster or slower movement in virtual environment, compared to the real world movements) for both rotations and translations. If the differences are sufficiently small, they remain unnoticed by most users. Stenicke et al. [7] used psychometric methods to estimate perceptual sensitivity thresholds for translation, rotation, and curvature (rotations of virtual environment around moving user) gains for RDW. When gains are applied over time, user's real walking trajectory is decoupled from her virtual trajectory. As a result, a long virtual path may correspond to a different physical path condensed to fit into the bounds of tracked space. At the same time user remains unaware of the applied manipulations.

RDW algorithms can be classified into two categories: *reactive* and *predictive*. Reactive algorithms such as Steer-To-Center and Steer-To-Center use the current user's physical location and direction of travel within physical tracked space to determine the best action at each calculation step. For example, Steer-To-Orbit algorithm attempts to apply a combination of rotation and curvature gains to steer the user towards the center of the physical tracked space. In contrast, *predictive* algorithms [10, 4, 1] attempt to leverage information of user's future path in the virtual environment to optimize redirection parameters. This approach involves investigating the outcome of the user's future actions under various parameter choices and choosing the settings that perform the best based on some utility function. Studies suggest that information about user's future short-term action possibilities enables predictive RDW algorithms to outperform traditional reactive algorithms.

Predictive RDW algorithms typically estimate the user's future trajectory by relying on representation of virtual environment's layout in the form of bidirectional graph. This annotation provides a high-level abstract description of possible paths that a user can take. Edges of such a graph represent the general expected direction of travel for possible paths in a particular area of the virtual environments, while nodes correspond to decision points where potential paths diverge. At run-time, the user's actual position is approximated using the nearest point on the graph. When the user is approaching a fork, all outcomes are assigned a certain probability weight (equal weights are commonly used). Most predictive algorithms operate with predictions based on a fixed search horizon, defining how far into the future the algorithm will inspect possible user paths.

The layout annotation graph can be manually constructed for a particular environment from arbitrarily shaped segments or using a predefined set of path primitives (straight lines and arcs conforming to predefined specification). This approach can be tedious and requires significant preparation before an algorithm can be deployed in a new virtual environment.

2.2 Navigation meshes

Artificial intelligence algorithms for autonomous agents in computer games often face a similar problem of describing possible paths through navigable areas in 3D virtual environments. This problem has been long studied and a variety of methods have been designed to represent a navigable environment such as navigation meshes [9].

Navigation meshes [6] represent navigable surface of the environment as a polygonal mesh. Agents can freely navigate across polygons with shared edges. In addition, special links can be used to represent connectivity between polygons that do not share an edge. For example, such links can represent adjacency when an agent can jump from one platform to another or step vertically from a lower lying surface to a higher ground (or vice-versa). An attractive property of navigation meshes is their ability to represent the free space available adjacent to a path in the environment, which enables pathfinders to perform local obstacle avoidance.

Navigation meshes can be generated automatically. A well-known algorithm for automatic navigation mesh generation was introduced by Tozour [8]. It first determines walkable polygons in a 3D environment by comparing their normals with the up vector, and then iteratively merges together as many polygons as possible. Since then, various implementations have been proposed by both academic researchers and game engine developers extending pathfinding capabilities and optimizing the performance. Navigation meshes have been widely adopted as the standard solution to navigation in gaming engines and are readily available in commercial game authoring platforms such as Unity3D and Unreal.

In this paper we explore how navigation meshes can be used to generate short-term dynamic path predictions suitable for using with RDW algorithms. These representations can be constructed at run-time and can take into account both actual user position and dynamic changes in the environment.

3 Path prediction algorithm

Our goal is to design an algorithm to automatically generate layout annotation graphs to be used with existing predictive RDW algorithms. In practice such algorithms operate with limited prediction horizon and thus typically prune the nodes of path prediction graph beyond certain fixed distance *d*. Therefore, a partial layout annotation representing short-term path prediction should be sufficient.

Algorithm 1 oulines the structure of the proposed procedure. The algorithm takes as input the current user position *Pos*, navigation graph of the environment *NavGraph*, and maximum path length *d* (corresponds to fixed search horizon). Here *navigation graph* (see Figure 1 (left)) is a connectivity graph derived from navigation mesh. Each navigation mesh polygon is represented as a node. Edges represent connectivity between polygons in the navigation mesh. Additionally, each node is assigned a location coinciding with the centroid of its polygon in the virtual environment.

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
```

First, the algorithm finds node *S* in *NavGraph* corresponding to the navigation mesh polygon currently occupied by the user. The assigned location of this node is shifted to user position *Pos*.

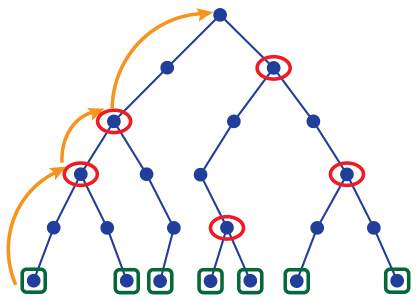

Figure 2: Sample output of modified Dijkstra algorithm. Navigation graph is explored up to a fixed depth. Branching nodes are marked in red, and terminal nodes are marked in green. Each marked node has a pointer to its previous branch (or source) defined by *prevPG*[]. For illustration simplicity only a subset of pointers is shown.

The next step is to query navigation graph for all possible paths (in any direction) starting from S and not exceeding length d. The process yields a set of *branching nodes* B (where potential paths can split), a set of *terminal nodes* T (where potential paths terminate), and the connectivity function *prevPG*[] between the nodes in the above two sets. This is accomplished using a modified version of Dijkstra's algorithm [2] which terminates when a certain distance from the source has been explored (see Algorithm 2).

Finally, the path prediction graph is constructed. We define the set of vertices V as the union of starting node S, branching nodes B, and terminal nodes T. Using connectivity information encoded in *prevPG*[] we add appropriate edges to set E. To better understand this last step it is helpful to consider the outputs generated at the previous step in greater detail.

Algorithm 2 Modified Dijkstra algorithm

```
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[]
```

The Dijkstra algorithm (Algorithm 2) takes an input *Graph* and a starting node S and constructs a nearest-first path search tree describing all possible paths between the nodes of the Graph. We modified the original algorithm in two important ways. First, the depth of the tree is limited by the maximum length of the path d. Second, 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*[]. For each vertex u that is visited, if $deg(u) > 2$, we add this vertex to a set of branching nodes B. The algorithm also maintains a set of terminal nodes T. Also at each iteration step, in addition to updating the Dijkstra's *prev* function, we define function *prevPG[]* that points to previous branching node (or source S). For a vertex u being visited, *prevPG*[u] is set to *prevPG*[*prev*[u]] unless *prev*[u] is in B, in which case it will be set to *prev*[u]. The algorithm terminates when all nodes within distance d from the source are visited (see Figure 2).

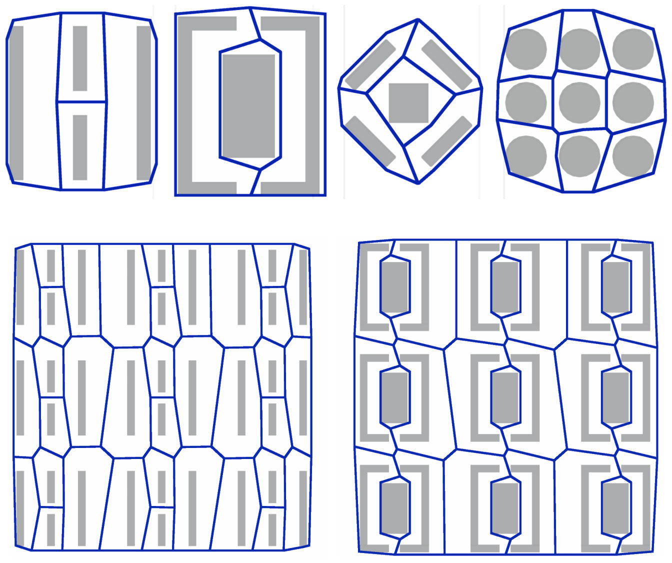

Figure 3: Automatically generated prediction graph for six sample environments.

The node sets B and T, along with node S, form the initial set of vertices V of the prediction graph. Since Dijkstra's algorithm removes cycles from the graph by returning a tree, we reintroduce the lost connections by connecting vertices in V that are neighbors in *NavGraph*. We then perform navigation mesh funneling which yields the shortest path from a series of consecutive polygons. This is required to remove artifacts manifesting from the artificially inserted edges. Then the shortest navigable path between each vertex in v and it's predecessor (*prevPG*[v]) are queried. These paths are added to the prediction graph to conclude the graph generation.

4 Implementation and Performance Assessment

We implemented proposed algorithm on the Unity3D platform. Unity provides native support for automatic generation of navigation meshes and path queries (e.g. finding the shortest navigable path between two points in the environments). However, we found it more convenient to use an add-on package from A* Pathfinding Project [3] available in the Unity asset store. The A* Pathfinding API exposes the navigation graph data structure, making implementation of our algorithm much more straightforward.

4.1 Comparison to manual annotations

To compare our method with manual layout annotations, we relaxed the search horizon constraint and generated a path prediction graph covering the entire virtual environment. Figure 3 shows the path prediction graphs generated for six test environments. In our

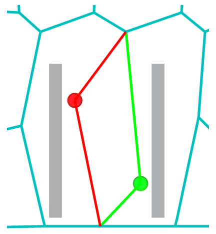

Figure 4: Dynamic adjustment of local graph. For different placements of the user (red and green), the local graph adjusts to provide a prediction that matches the user's position.

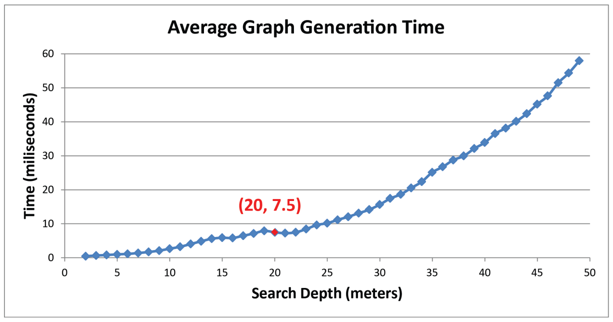

Figure 5: Effect of search depth on execution time. Values are averaged over 25 random placements of the user position in a test environment.

judgement these results represent a reasonable replacement for annotations that could be constructed manually. There are two types of artifacts present. First, extra branching nodes are created at the intersections where more than three paths meet. This is due to the fact that in the navigation graph each node can not have more than three neighbors. Second, placement of the intersections depends on the underlying structure of the navigation mesh and may differ from the intuitive placement in the manual annotation. Virtual environment authors can control this by changing the parameters of the navigation mesh. While both of these artifacts might affect redirected walking algorithms, we do not believe the resulting change in performance compared to existing methods will be significant.

In practice, our algorithm periodically generates short-term path prediction graph (see Figure 1 (right) for some examples) relative to current location of the user. Figure 4 demonstrates how the graph updates the trajectory prediction based on user position. In contrast, when a manual annotation is used, path prediction is based on approximating user position by the nearest point on the annotation graph. Given the importance of the accurate path prediction in the near vicinity of the user, we believe our method well likely improve the RDW algorithm performance compared to manual annotations.

4.2 Graph generation time

From a practical standpoint, it is useful to consider time complexity relative to maximum path length. Figure 5 shows the average execution time for creating a local graph of varying depth. Each point has been averaged over 25 random starting points in the environment on the bottom left of Figure 3. On average, our algorithm generated path prediction graphs with search horizon up to 20 meters in 7.5 milliseconds. This compares to 82.5 miliseconds reported by Nescher [4] for the average planning phase execution of the MPCRed algorithm. We conclude that our algorithm can provide updates to the RDW algorithms on every computation cycle.

5 Conclusions And Future Work

In this work, we have proposed an algorithm to automate path prediction for planning RDW algorithms. We have demonstrated how tedious manual annotation of an entire environment can be replaced with automatically generated prediction graphs local to the user's current position. Dynamically generated path prediction graphs open the door to supporting advanced prediction techniques such as forecasting user behavior in non-static environments and increased prediction accuracy in less constrained architectures. With these advanced navigation tools, our hope is to allow predictive RDW algorithms to be easily deployed in any virtual environment.

We have demonstated how short-term path prediction for RDW planning algorithm can be done automatically using navigation meshes. Our next step is to integrate this technique into existing planning algorithms such as FORCE. We can then benchmark the performance of these automatically-deployable algorithms with other algorithms such as Steer-To-Center and the original version of FORCE. We plan to conduct both computer simulations and user studies to assess the overall benefits of this work.

In addition to enhancing planned RDW algorithms, we intend to investigate how navigation meshes can further improve aspects of the general redirected walking problem. Possible avenues of research include: identifying navigation patterns for early user intent detection (e. g. knowing if the user will turn left or right at intersection) and also developing opportunistic dynamic changes in the environment to steer users away from physical boundaries.

Our algorithm relies mainly on implementation tools readily available in many game engines, and this allows for integrating predictive RDW techniques into VR applications relatively effortlessly. With the proliferation of consumer-level tracking solutions this opens the door for greatly improved end-user virtual reality experiences using efficient redirected walking algorithms.

Acknowledgements

This work is sponsored by the U.S. Army Research Laboratory (ARL) under contract number W911NF-14-D-0005. Statements and opinions expressed and content included do not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

References

[1] M. Azmandian, R. Yahata, M. Bolas, and E. Suma. An enhanced steering algorithm for redirected walking in virtual environments. *Proceedings - IEEE Virtual Reality*, pages 65–66, 2014.

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[3] A. Granberg. A Pathfinding Project. http://arongranberg.com/astar/, 2015. [Online; accessed 4-December-2015].

[4] T. Nescher, Y.-Y. Huang, and A. Kunz. Planning Redirection Techniques for Optimal Free Walking Experience Using Model Predictive Control. *3Dui 2014*, pages 111–118, 2014.

[5] S. Razzaque. *Redirected Walking*. PhD thesis, Chapel Hill, NC, USA, 2005.

[6] G. Snook. Simplified 3D Movement and Pathfinding Using Navigation Meshes. In M. DeLoura, editor, *Game Programming Gems*, pages 288–304. Charles River Media, 2000.

[7] F. Steinicke, G. Bruder, J. Jerald, H. Frenz, and M. Lappe. Estimation of Detection Thresholds for Redirected Walking Thechniques. *IEEE TVCG*, 16(1):17–27, 2010.

[8] P. Tozour. In S. Rabin, editor, *AI Game Programming Wisdom*, pages 171–185.

[9] P. Tozour. Search space representations. *AI Game Programming Wisdom*, 2(1):85–102, 2003.

[10] M. a. Zmuda, J. L. Wonser, E. R. Bachmann, and E. Hodgson. Optimizing constrained-environment redirected walking instructions using search techniques. *IEEE Transactions on Visualization and Computer Graphics*, 19(11):1872–1884, 2013.