21

Dynamic Obstacle Navigation in *Fuse*

Jan Müller

21.1 Introduction

Climbing over obstacles like walls, ledges, or ladders plays a supporting role in video games: they are not usually central to the experience, but they stand out when done poorly or omitted entirely. There are, of course, exceptions to the rule, games like *Mirror's Edge* [EA 08] or *Tomb Raider* [CD 14] that make the climbing sections, so called *traversal*, the core gameplay element. Traversal makes the AI navigation more complicated: it requires additional markup and programming for the AI to understand how to mount a ladder or jump over a wall, not to mention special animation states to control their location as they jump, vault, climb, or swim. As a result, video games with very complex or varied AI characters tend to avoid climbing altogether. Instead, AI navigation [Snook 00] is usually limited to finding paths on a continuous mesh [Mononen 12], where the actors can walk or run everywhere they need to go.

Traversal is hard enough if we have consistency between obstacles, that is, if all jumps are the same length and all walls the same height, so that there is a single solution to any locomotion problem. What if it is more complicated? What if there are hundreds of different volumes placed by either a designer or a content creation tool, each with different attributes and the AI has to decide which ones to consider based on the real time context? In the game *Fuse* [IG 13], from Insomniac Games, four hero agents with different abilities fight cooperatively. Of those four actors, up to three can be under AI control at any

given time. Furthermore, we had a complex and highly varied environment, making locomotion more challenging. In contrast to many games with cooperative NPCs, we wanted to make our AI characters follow the same rules as a human player. They aren't tougher to kill, can't teleport to keep up with the player, and need to be able to follow the same gameplay mechanics with regard to activities like climbing a ladder or manning a turret. It is crucial for the game balance that the AI plays as well as a human—not better or worse. The rule we ended up with is "50/50," meaning that half of the total damage is dealt by human players and the other half should be dealt by the AI (assuming that at least two of the four heroes are AI controlled, of course).

To make things even more challenging for the AI, *Fuse* features the *leap* ability, which allows human players to switch from their current character to any of the other heroes that isn't already being controlled by another human player. The game does not restrict leaps to any specific time or place; you can leap in the middle of combat just as easily as during a traversal section. Furthermore, players can drop in or out of the game at any time, and all of this has to work in a peer-to-peer online environment.

The result of all of this is that there can be no cheating. Whatever the human player is capable of doing, the AI has to be able to do it too. Whatever situation the human player is in, he or she can leap to another character (or drop out of the game), leaving the AI to deal with it. This circumstance leads to volatile AI characters that must be able to function smoothly when they initialize halfway through a nontrivial traversal section. Imagine waking up to find yourself hanging one-handed from a ledge 50 ft above the ground, while your enemies are firing semiautomatic rifles at you and a teammate is in need of medical attention down at the bottom. In such a context, the AI cannot rely on markup alone, but needs to evaluate dynamically changing traversal paths at runtime.

21.2 Fuse Traversal Setups

Fuse has numerous, wide-ranging traversal sections. To give players a break between fights, the protagonists will often have to climb obstacles like a mountain side or sewage pipes below a medieval castle in India. These setups usually have three or four possible ascents, which intersect at various points. In addition to the complexity of the ascent itself, there are often enemy bots or turrets that can shoot at you during the climb. When a character is hit, he or she can fall down to the ground and become incapacitated. If that happens to the player, the AI has to be able to find a path down to revive him.

On the content-authoring side, these traversal setups are represented as a group of volumes with different markup options. The most common types of such volumes are vertical pipes and horizontal ledges, as well as variations of those such as ladders. Depending on their attributes and the situation, you can perform different climb animations. For example, you might hang from a ledge and traverse with your hands or mount it to stand on top of it. Connections between volumes may be explicitly marked in the places where they intersect or overlap, or they are implicitly allowed through jump and drop thresholds. To determine whether an actor, be it human or AI controlled, can jump from one ledge to the next, we match their attributes and then perform distance and collision tests between the two points closest to each other. Additionally, we place custom clues that connect the navigation mesh between the start and end of the traversal section. These elements

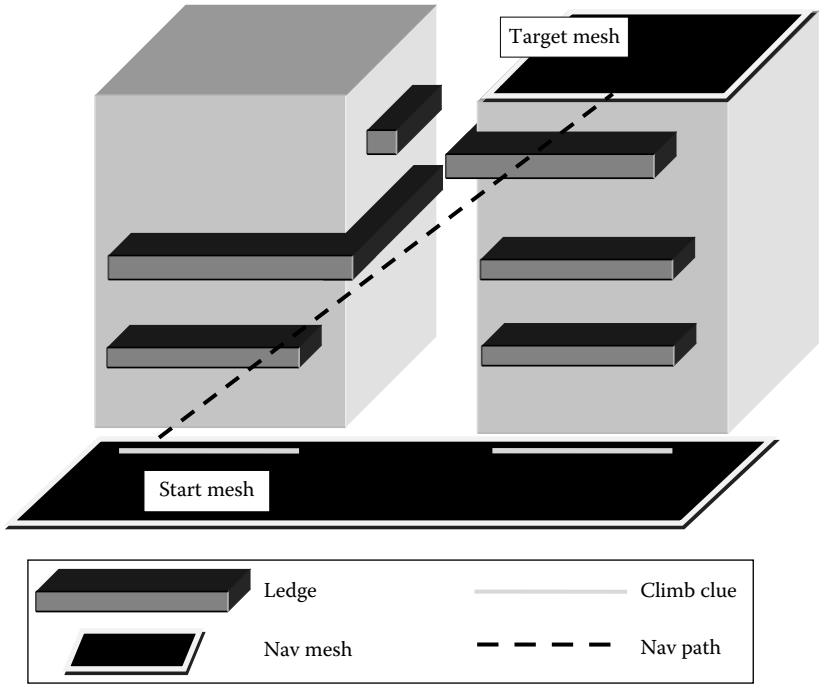

Figure 21.1

A traversal section with ledge markup and navigation path between two meshes.

together provide enough information for the AI to parse the traversal and generate a climb path procedurally. For a typical traversal section in *Fuse*, refer to Figure 21.1. For simplicity's sake, we exclude vertical elements, like pipes or ladders. However, their connections within the climb mesh work the same way as the horizontal elements.

The navigation path in Figure 21.1 connects two disjoint navigation meshes. It creates a link between the start and the end point of the traversal. The path is a straight line between these two climb clues, since the navigation system has no further information about the connecting climb.

21.3 Climb Mesh Generation

The first step of the procedural traversal system is to generate the climb mesh. It collects all possible traversal elements from a group of traversal volumes and creates connections between them. Each connection can link to multiple other traversal elements, but the most common case is that there are no more than one or two connections per node. The system first considers touching or overlapping volumes, which explicitly link two elements together. If two elements touch each other at their end points, they share one merged node in the climb mesh and create explicit connections. For example, if a ledge touches a pipe at a cross section, they share a node at their intersection that forms edges to the start and

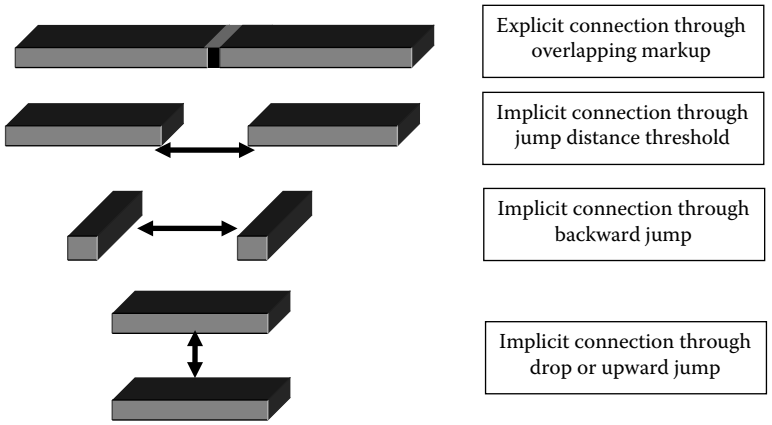

Figure 21.2

Possible connections between two ledge elements in the climb mesh.

end of the ledge as well as the pipe element. If two ledges intersect at their end points, they form an explicit connection between them. Examples of two ledges generating links between each other can be seen in Figure 21.2.

Once all explicit connections have been generated, the system searches for implicit links like jumps or drops. Players can jump between two elements at any time, as long as certain distance and angle thresholds are not exceeded. For simplicity's sake, the *Fuse* AI considers such transitions in the climb mesh only from the end points and between the center points of the volume markup, although other transitions would be possible. Thus, for every traversal element, the algorithm tests three nodes for implicit links to nearby elements. Alternately, you can generate these links at fixed intervals, for example, one per meter, but that has an impact on both the mesh complexity and the parsing resolution. We discuss those implications in the section on generating a virtual controller input (VCI).

The node connections are stored in the climb mesh. Cyclic connections are allowed and are resolved in the path generation step by marking already visited nodes. At this level of abstraction, the actual animation that moves a character from one climb element to the next does not matter. That means a connection between two elements could result in a jump animation for one character and in a swing animation for another. Only the type and distance of the connection, as well as line of sight tests between its elements, are considered. Figure 21.3 illustrates the climb mesh that the example in Figure 21.1 would generate, including all of its nodes and connections.

21.3.1 Path Following on Climb Meshes

As mentioned before, the climb meshes are independent from our navigation system, which only knows about the existence of the traversal, not its layout. When an actor needs to pass through a traversal section, the navigation system returns a path that includes two special clue edges that connect the start and end points of the climb. The clue edges are placed by the level designer and represent the traversal section within the navigation system.

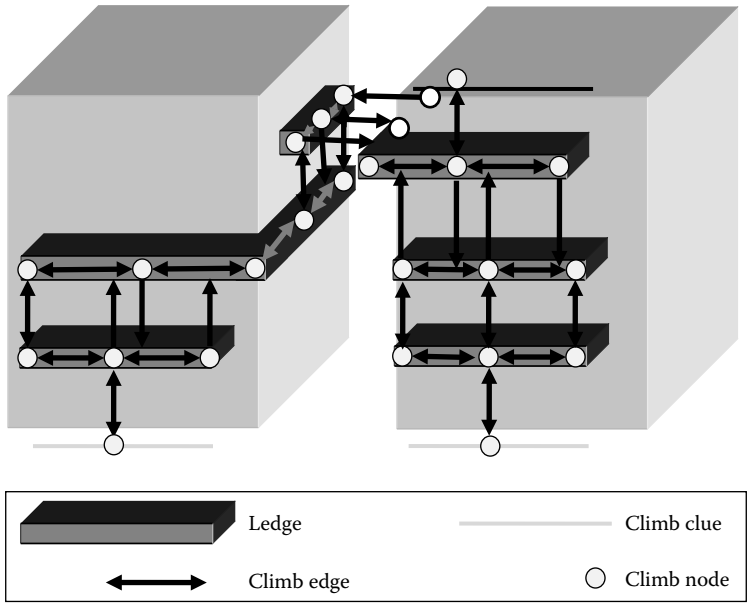

Figure 21.3

The fully generated climb mesh with all possible transition edges.

It does not store any further information about the climb elements or connections. The navigation system only knows the distance between two climb clues and that a climb has to happen to move the character from edge A to B and continue the path from there.

Within the climb mesh, the closest point from the climbing actor on the custom clue is added as the initial climb node. From there, the traversal system triggers a secondary path query that follows the traversal nodes between the start and end clues. Path following on climb meshes utilizes a standard A* search. It searches from the initial climb node and iterates through the neighboring edges, starting with the least-cost candidate, meaning the closest to the target node. Each step of the pathfinding parses the links from the current node to connected nodes in the climb mesh, marking those it already visited as closed. When a path to the target node is reached, then all nodes of that path are returned as a set of 3D vectors. The result is the shortest path on the climb mesh that connects two custom clues on the navigation mesh.

21.3.2 Caveats of the Climb Path Generation

The traversal system in *Fuse* does not consider different costs per traversal element and does not store how long specific animations take to play back. The edge costs are generally normalized within the climb mesh and increased only if other bots use the same climb elements on their path. This means that the shortest path does not necessarily take the least amount of time. However, since there are usually only a few routes to choose from in *Fuse*, avoiding other characters is of higher importance than finding the shortest-time path.

At one point during development, there was a plan to support combat during climbing. This involved shooting from climb elements as well as being incapacitated by hits. If this

were to happen to the player while he or she was hanging from a ledge, the character would hang on for dear life until another teammate came to the rescue. In order to allow this, the AI characters had to be able to parse partial climb paths to and from any location on the mesh. While the feature was cut and didn't make it into the final game, it enabled us to add other useful behaviors such as dynamic obstacle avoidance. If a player turns around on a traversal section and blocks the path, the AI can simply reparse the climb mesh and find another way to reach its goal. Without it, AI actors try to avoid each other during path generation by increasing the costs of the used climb edges. But the system is not updating dynamically based on the player's behavior, which can lead to traffic jams along blocked paths.

21.4 Parsing Climb Paths

The second part of the procedural traversal system is the climb parser. The hero AI in *Fuse* shares about half of its implementation with the player controlled characters. This is possible by splitting hero character states into drivers and processors: drivers are responsible for the state input and transition logic. For human players, this means interpreting controller inputs and context into animation parameters. For AI actors, these transitions are controlled by their behaviors and VCI. While drivers are fundamentally different between human and AI characters, the state processors can be shared. The state processor interprets driver updates and communicates with the animation tree of the character. The reuse of hero state processors means that the hero AI automatically inherits all traversal animations from the player characters.

The climb parser generates the VCI for AI-controlled characters. The state transition logic then compares the input data against animation thresholds to determine when, for example, a ledge climb animation can transition into a jump to a nearby pipe. Once the first traversal state has been initialized, the AI behavior strictly follows the VCI and the current state's transition logic. Thus, during the traversal, there is no decision making in place beyond following the VCI, adhering to the state transitions thresholds (including terminating states like the death of the character), and checking against collision with geometry and other characters.

21.4.1 Generating Virtual Controller Input

To generate VCI, the climb parser first resolves the climb path as a Bézier spline curve [DeBoor 78, Farin 97]. Curves in Bézier form are defined as a set of four control points that span a curve between each other. A Bézier spline is a continuous set of such curves that form a smooth, higher-order shape. Such splines can be straight lines, only spanning between two points, but also complex curves with dozens or hundreds of control points. Using the path points as input data for a Bézier curve automatically interpolates the path and creates better results than the raw input data with 90° turns. The reason for this is that the additional curve points add angular momentum to the path. A simple example is a 180° turn around 2 corners: in its raw form, the path is only as long as the total length of its three sides and incorporates two sharp, 90° turns. A spline following the same control points however will extrude the curve to avoid sharp angles, which adds length and curvature to the path.

This is important because the system projects the VCI target on that curve at a fixed distance of 2.8 m ahead of the AI actor from its current position on the spline curve. A traversal move resolution of 2 m gave the best results for the climb animations in *Fuse* but might differ in other games. Many transitional animations move the character by roughly this distance and need to trigger ahead of a directional change to look smooth. The projection distance is the length of the diagonal of a 2×2 m^2 square, which is roughly 2.8 m. That also means that the climb parser is not accurate if there are multiple intersections within a 2 m radius (plus a certain margin for error) and generally speaking chooses the closest one. For example, if there were two vertical pipes intersecting a ledge 1 m apart, the parser would pick the closest one (based on the approach direction). This occurs because the Bézier curve doesn't follow the raw path data precisely, so when two elements are close together, the algorithm can't tell which one was originally on the A*-generated path. Regardless, this doesn't necessarily impact the result. When this happens in *Fuse*, it always appears correct if the parser chooses the closest element.

The 3D vector between the current actor position on the spline curve and the projected VCI target is the VCI vector as depicted in Figure 21.4. The parser generates input strength values for all three axes relative to the character. Those input values are then tested against thresholds for different transitions. A transition in this sense does not necessarily mean jumping or climbing to another element but also transitioning from an idle animation to a climb animation on the existing one. Each climb state has individual thresholds in its transition functions that define the points where characters can switch from one state

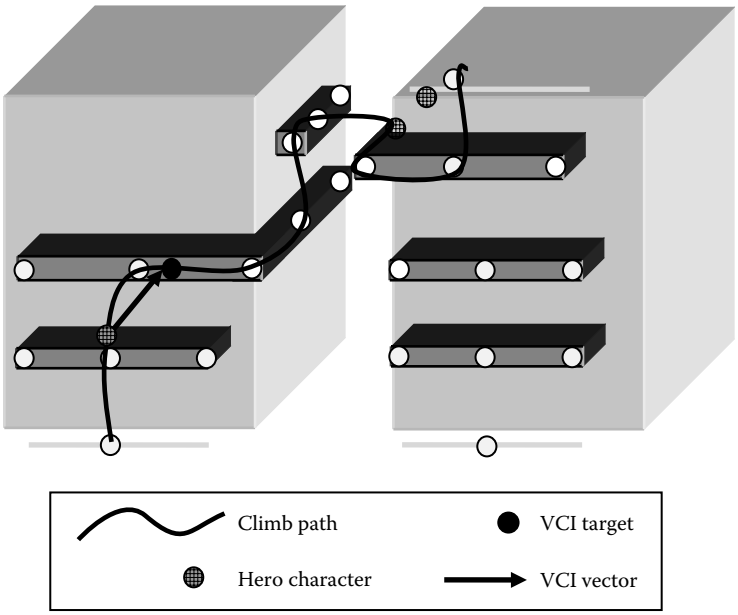

Figure 21.4

The VCI generates a 3D vector between the current actor position and the immediate, projected target position.

to another. For example, if the input vector mostly points to the relative right of the character while he or she is hanging from a ledge, he or she will start climbing to the right. If the character gets into the proximity of a pipe element and the input vector points to the right and top, he or she will transition to the vertical climb animation, attach to the pipe, and start climbing upward. This sequence of traversal animations ends when a terminating state is reached. Examples for such states are mounting the top of a wall or dropping from a ledge back onto the navigation mesh. The system also supports depth transitions such as jumping backward between two parallel ledges on opposing walls or flipping over the top of a flat wall.

There are a limited number of entry states for traversal sections, such as mounting a ladder or jumping up to a ledge above the initial climb node. This makes the transition from walking or running to climbing relatively predictable. The most common case has the VCI vector pointing up or down at the traversal start. In that case, the climb could start by attaching to a ladder or jumping up or down to a ledge. Once the traversal state has been initialized, the VCI target is projected forward on the climb path as described. The state treats the VCI data the same way it would interpret human controller input and matches the values against transition thresholds. In the example in Figure 21.4, the initial jump-up state would transition to a ledge hang state, which would be followed by another jump, since the VCI mostly points upward. Once the character reaches the second ledge, the VCI would point to the relative right side of the character, which leads to a ledge move animation. The character would follow the climb path until eventually reaching the final ledge and playing a mount animation at the top of the traversal section. That state would terminate the climb so that we can return the AI to the navigation mesh.

21.5 Conclusion

This chapter introduced an approach to generating traversal climb meshes from markup and to following the resulting climb paths at runtime. The procedural traversal system is independent of the underlying animation states and is robust against changes in the locomotion sets of the actors. In addition, the chapter demonstrated how VCI can be utilized to parse climb paths along a Bézier spline curve. Using climb paths in combination with VCI allows AI characters to handle traversal setups in much the same way as they would normal navigation meshes. It also allows the AI to share the same traversal markup and transition logic that human-controlled characters use.

There are a number of worthwhile extensions that could be applied to this approach: as mentioned previously, the algorithm can be modified so that climb paths can be generated between any two locations within the climb mesh. This allows dynamic obstacle avoidance and path replanning. Games with mid-climb combat elements would especially benefit from those features.

Furthermore, using Euclidean distance as the edge cost worked well for *Fuse*, but might not be accurate enough for more complex climbing setups. If this approach is implemented for a game with very long or complex traversal segments, then the climb path generation should consider animation playback times to accurately detect the shortest path.

References

[CD 14] Crystal Dynamics. 2014. *Tomb Raider* [PC, Xbox 360, PS3]. Redwood City, CA.

[DeBoor 78] de Boor, C. 1978. *A Practical Guide to Splines*. Springer Verlag, New York.

[EA 08] Electronic Arts DICE. 2008. *Mirror's Edge* [Xbox 360]. Stockholm, Sweden.

[Farin 97], Farin, G. 1997. *Curves and Surfaces for Computer Aided Geometric Design*, 4th edn. Academic Press, San Diego, CA.

[IG 13] Insomniac Games. 2013. *Fuse* [Xbox 360, PS3]. Burbank, CA.

[Mononen 12], Mononen, M. 2012. Recast and Detour, a navigation mesh construction toolset for games. http://code.google.com/p/recastnavigation/ (accessed July 21, 2014).

[Snook 00] Snook, G. 2000. Simplified 3D movement and pathfinding using navigation meshes. In *Game Programming Gems*, pp. 288–304. Charles River Media, Newton, MA.