20

Precomputed Pathfinding for Large and Detailed Worlds on MMO Servers

Fabien Gravot, Takanori Yokoyama, and Youichiro Miyake

20.1 Introduction

Precomputed solutions for pathfinding were common on old generation consoles, but have rarely been used on current hardware. These solutions give the best results in terms of computation cost, as all path request results are precomputed in a lookup table. However, they have two drawbacks: memory cost and loss of flexibility. Currently most games use dynamic algorithms like A* for navigation.

In the context of MMO games, however, precomputed solutions are still used. While the corresponding servers are typically equipped with ample memory, they have very few CPU cycles available for each request. This article will show how precomputed pathfinding has been implemented for *FINAL FANTASY XIV: A Realm Reborn*.

This game has very large and detailed maps (about 4 km^2 each), with cliffs from which the agent can fall (unidirectional path). We will present an accurate navigation system with navigation mesh autogeneration and a component based hierarchical lookup table. We define a component to be a group of connected polygons (at the lowest layer) or connected sublayer components (at other layers). We will first give an overview of the whole

navigation system, followed by a brief explanation of the navigation mesh autogeneration, finally focusing on the precomputed data generation.

20.2 System Overview

The navigation system presented in this article has been developed for *FINAL FANTASY XIV: A Realm Reborn*, "a Massively Multiplayer Online Game," developed by SQUARE ENIX. The choices made for this system are mostly driven by this game's needs; however, the techniques presented here can also be used in other environments.

The game performs all the navigation path computations on the servers. One server can simulate several maps. One map can be as large as 4 square kilometers with several thousand NPCs and players. Figure 20.1 shows an in-game screenshot of part of the world to explore. Because the pathfinding system must be very fast, meaning A* was not an option, a precomputed path lookup table was chosen.

Another important requirement was to have the pathfinding system be mostly automatic. It has to be able to generate accurate navigation for hundreds of maps without expert or manual input, but if necessary, it must be possible to edit the data directly. As a game rule, any NPC can go where the player can go (except when specified by the level designer). The navigation system must be as close as possible to the collision system used by the player. Since players can jump or fall off (from almost anywhere) the navigation system must support those features. It also means that large NPCs can go through narrow passages if the player can use them. The NPC size is not used to find a path, but it is still used to smooth it at run-time. To fill the role of world navigation representation, a navigation mesh was chosen.

The precomputed navigation data is generated from the navigation mesh, and this generation is the core concept of this article. The precomputed data is stored in a hierarchical lookup table. This approach was chosen to reduce the memory footprint while maintaining very fast computation time. These tables are used on the server to perform all the navigation system tasks.

Figure 20.1

FINAL FANTASY XIV: A Realm Reborn screenshot.

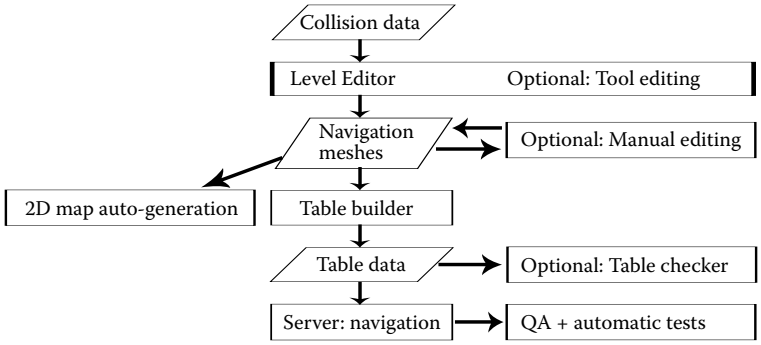


Figure 20.2

Tools flow chart from the collision data to the table data used by the server navigation system.

20.3 Tool Chain

The main tools used to generate the navigation table data are the Level Editor and the Table Builder (Figure 20.2). The Level Editor gathers all the collision data to generate the navigation mesh. The navigation mesh auto generation is done by Recast, an open source software library developed by Mikko Mononen [Mononen 12], released under the MIT license. The mesh is generated after a voxelization phase to identify the walkable areas [Miles 06, Axelrod 08]. The main modification made to Recast was the addition of the game-specific "falling mesh" into the generation process.

The level editor also allows editing the navigation mesh through various tools. For example, it is possible to remove the mesh generation inside a box or mark an area as only accessible by the player. Doors can also split and tag the mesh underneath, thus forbidding motion when closed. The most important feature is the navigation mesh seed point. It allows designers to remove all the polygons that are not connected to this point. Falling polygons are kept only if they allow the connection between two valid walkable polygons.

The navigation mesh by itself is simply a 3D polygon mesh model that can be edited or manipulated by other appropriate tools. For instance, it can be used by the 2D map autogeneration tool (Section 20.9.4).

Figure 20.3 shows a screenshot of a navigation mesh generated by this system, viewed in Maya. It is possible to edit it manually, but doing so invalidates further autogeneration. This option, if used, must be done in the final stages of the project.

The main purpose of the navigation mesh is to be used to compute table data through the Table Builder. This data is used in the server navigation system, which is checked by QA and other automatic tests (Figure 20.2). It is also possible to analyze the table data directly with the Table Checker tool. All of these quality checks expose problems in either the input data (collision data) or the algorithm. Quality checking is one of our main concerns in creating a system as robust as possible for autogeneration of navigation data.

20.4 Mesh Generation

Early in the project, navigation meshes were chosen for the navigation system. They describe a free moving space usable for steering.

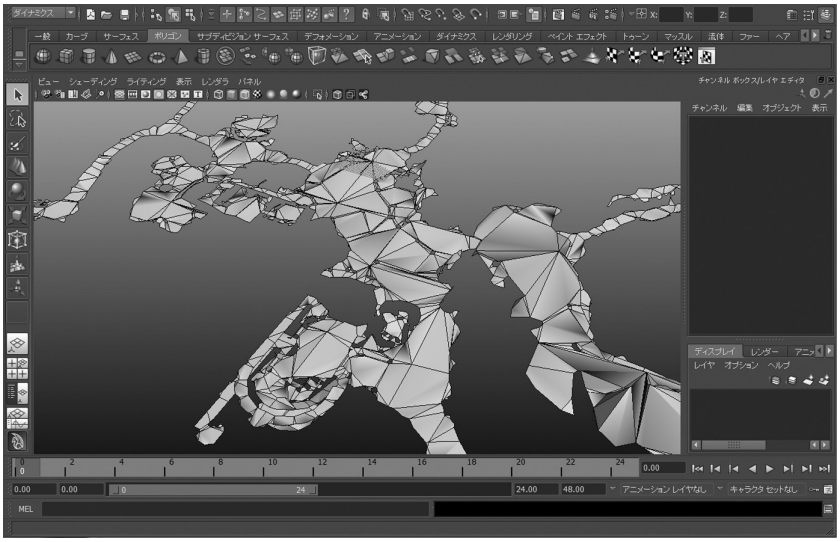


Figure 20.3

Navigation mesh of walkable areas generated for a town-like environment, viewed inside Autodesk® Maya®.

Since the world is very large, the navigation mesh generation splits it into a regular grid. Each tile generates a navigation mesh independently. We have chosen to keep this grid information in the precomputed data using small tiles of 32x32 meters.

Figure 20.4 shows a more detailed view of the navigation mesh. At its borders, the mesh is shrunk by the player radius, so that NPCs, or more precisely their centers, can move freely on the entire mesh surface. The generation process must try to minimize the number of polygons and match the collision data as much as possible.

Figure 20.4

Navigation mesh autogenerated around a tent.

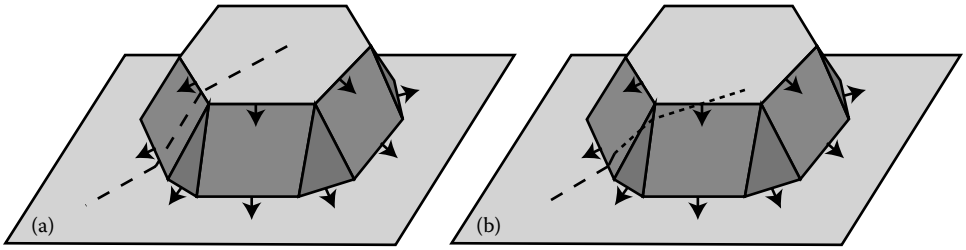


Figure 20.5

The falling mesh, in dark gray, and the desired output direction; (a) shows the unidirectional edges and how pathfinding is done; (b) shows how knock-back motion can cross falling polygon boundaries (dotted line).

As mentioned, we added falling meshes to the mesh generation algorithm. The generation of these is outside the scope of this article but some details are useful as it strongly influenced the Table Builder algorithm. The player can fall from anywhere from any height without taking damage. NPCs must be able to follow the player and avoid long detours. Designers can add an invisible wall to prevent the player from falling.

Moreover we added support for NPC knock-back, giving the player the ability to knock an NPC off a cliff. Figures 20.5a,b show how the falling mesh is used to support knock-back functionality, covering all possible falling directions. Figure 20.6 shows a falling mesh generated from a game map.

It is possible to use the mesh connectivity for the knock-back length (dotted line in Figure 20.5b). If the knock-back path stops in the middle of a falling mesh, a falling motion is appended to it.

The advantages of using falling meshes for knock-back in place of collision checks are computation speed and the guarantee that the NPC will always end up in a valid pathfinding position on the navigation mesh. The main drawback is the complexity of the autogeneration process.

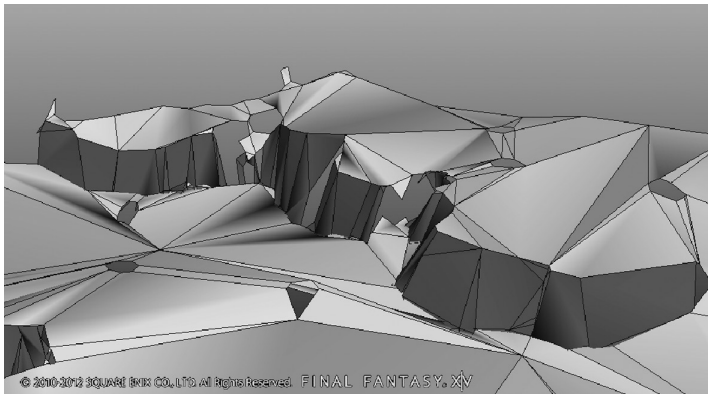


Figure 20.6

Falling mesh generated for a game world. The falling polygons are darker. In those areas the slope of the collision data is too steep to allow walking.

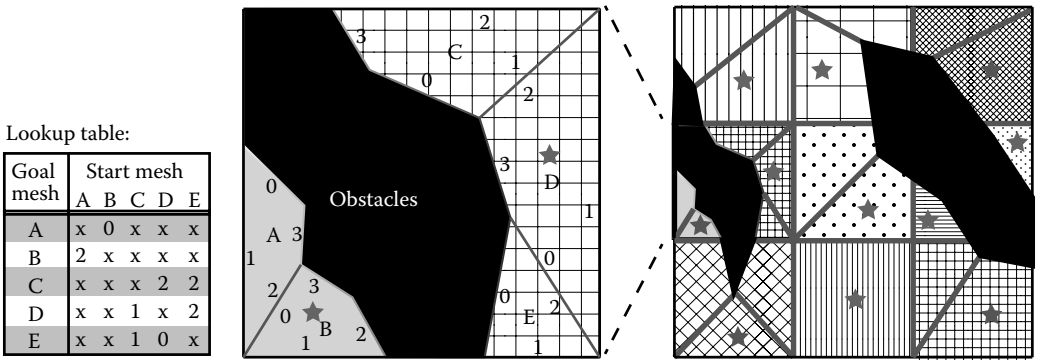


Figure 20.7

Black shapes represent obstacles, while polygons within the same component have the same pattern. The gray stars show the polygons that are used as component centers. Figure (b) shows the detail of one tile of the whole navigation mesh (c). Figure (a) shows the lookup table of (b) with the polygon letter and the output edge number, or 'x' when there is no path.

20.5 Table Generation Overview

The precomputed navigation data is generated from the navigation mesh. For this purpose, lookup tables are built. Figure 20.7a shows such a lookup table. For instance, it indicates that traversing from C to E implies using Edge 1 of Polygon C. The lookup tables give the correct output edge to use when moving from one polygon to another. However, with the number of polygons generated (N) the table size will quickly become prohibitive ($N \times N$). A hierarchical approach is used to avoid the memory explosion [Dickheiser 03, Sterren 03]. With a hierarchical approach, we can split the N polygons into K groups and have K tables of size $(N/K) \times (N/K)$ and one portal table of size $P \times P$ where P is the number of portals connecting the polygon groups.

Instead of using portals to divide the regions, a connected component approach is used. This method is derived from the work done on *Dragon Age: Origins* (Bioware, 2009) [Sturtevant et al. 10]. A component is defined as a set of fully connected polygons. This component will be a node for the upper hierarchical level table. Since we use a hierarchical approach, we will use the term "node" in place of polygon when the description can be generalized to an upper hierarchical layer.

20.5.1 Component Approach

Figure 20.7b shows how 5 nodes are gathered into two components inside one tile. Since there is connectivity between the nodes C, D, and E, it is possible to gather them as one component. The stars show the nodes that are used as component centers. The component centers represent the upper layer nodes. They replace portals in describing component connectivity in our implementation. In general, the component center is the node which minimizes the distance to the other nodes inside this component.

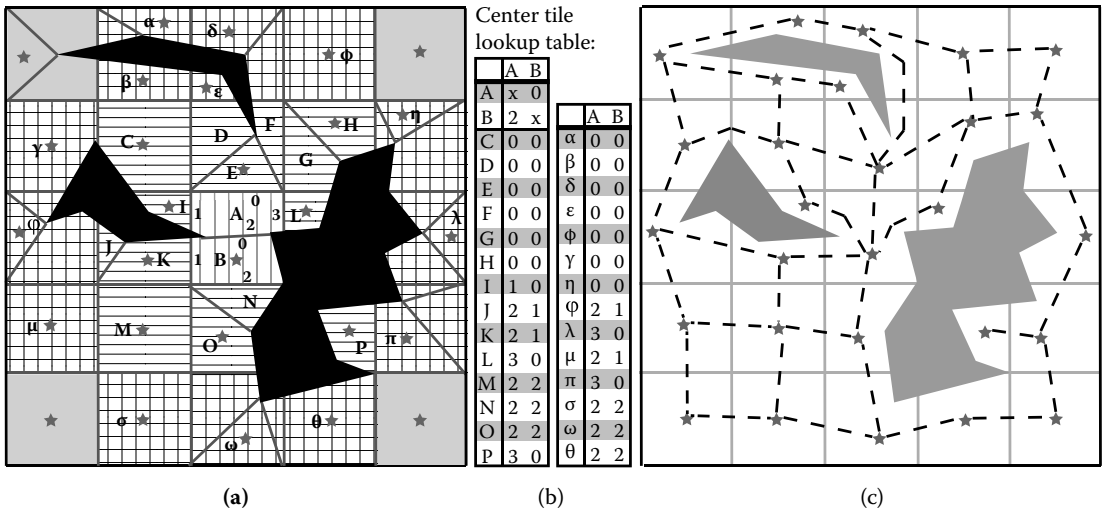
Due to the size of the world, the navigation mesh autogeneration splits it into a regular grid and generates the mesh for each tile. The lowest level of the lookup tables is based on these tiles (Figure 20.7c). We define the following elements:

- *Node*: a polygon or a sublayer component.
- *Component*: a group of connected nodes. There is always a path between any two nodes in a component.
- *Tile*: a cell in the grid partitioned layer. It can have any number of components or nodes.

20.5.2 Table Connectivity

Each tile has a lookup table describing connectivity between all of its nodes. To avoid discontinuities at its borders, each tile also has a lookup table describing connectivity between its nodes to all its neighbor tiles' nodes (Figure 20.8b, left table). This ensures that when moving from tile to tile, there is always a detailed table showing which edge to choose.

Each tile also has a lookup table that describes connectivity between each of its nodes and the component centers two tiles away (Figure 20.8b, right table). As we will see in the next subsection, having this last table increases the quality of the pathfinding heuristic given by the upper layer. Figure 20.8a shows 25 tiles with the navigation mesh. The central tile (vertical line pattern) has a lookup table describing connectivity between its two nodes with each other, with all the 15 nodes of its eight neighbor tiles (horizontal line pattern) and with the component centers (14 nodes with gray star) of its 12 next neighbor tiles (grid pattern). In summation, the central tile has a lookup table from its two nodes to 31 (2 + 15 + 14) nodes.



	A	B
A	x	0
B	2	x
C	0	0
D	0	0
E	0	0
F	0	0
G	0	0
H	0	0
I	1	0
J	2	1
K	2	1
L	3	0
M	2	2
N	2	2
O	2	2
P	3	0

	A	B
α	0	0
β	0	0
δ	0	0
ε	0	0
φ	0	0
γ	0	0
η	0	0
φ	2	1
λ	3	0
μ	2	1
π	3	0
σ	2	2
ω	2	2
θ	2	2

Figure 20.8

Figure (a) shows the local table connection levels. Latin letters represent polygons. Stars or Greek letters represent component center. The edge index of the polygons A and B is also shown. Figure (b) shows the center tile lookup table. It has a node-to-node table (left side) for the nine center tiles (line pattern) and node-to-component center table (right side) for the 12 border tiles (grid pattern). Figure (c) shows the top level node graph corresponding to Figure (a). Component centers of the lower layer, shown as gray stars, are the nodes of the top layer. The connections are shown with dashed lines.

20.5.3 Falling Meshes Outside the Table

When falling, the player loses control. In the same way, moving through a falling mesh changes the NPC's motion. We decided that for the purpose of simulating an agent falling, the falling mesh's polygons would each have only one output edge. The Table Builder uses this property to optimize the lookup table size. It first reorders the falling polygon edges so that the output edge is the first one. Thus, the output edge of any falling polygon will be Edge 0 for any goal in the lookup table (i.e., the lookup table column of a falling polygon is a null column). Moreover, we decided that any goal on the falling mesh will be changed to its ending fall point. This allows predictive motion and removes the need to have the falling polygon as a goal in the lookup table (i.e., the lookup table rows). This means that the falling mesh doesn't need to be stored in the lookup table. This method reduces the size of the lookup table by up to 20%.

20.5.4 Pathfinding Requests

Figure 20.9 shows an example of the pathfinding process. The path subgoals are in fact the component centers. The choice of which component center to use is made by the upper layer. In this simple example, there is only one upper layer with one lookup table in which to find the next node link to use. If a solution cannot be found at the lowest level,

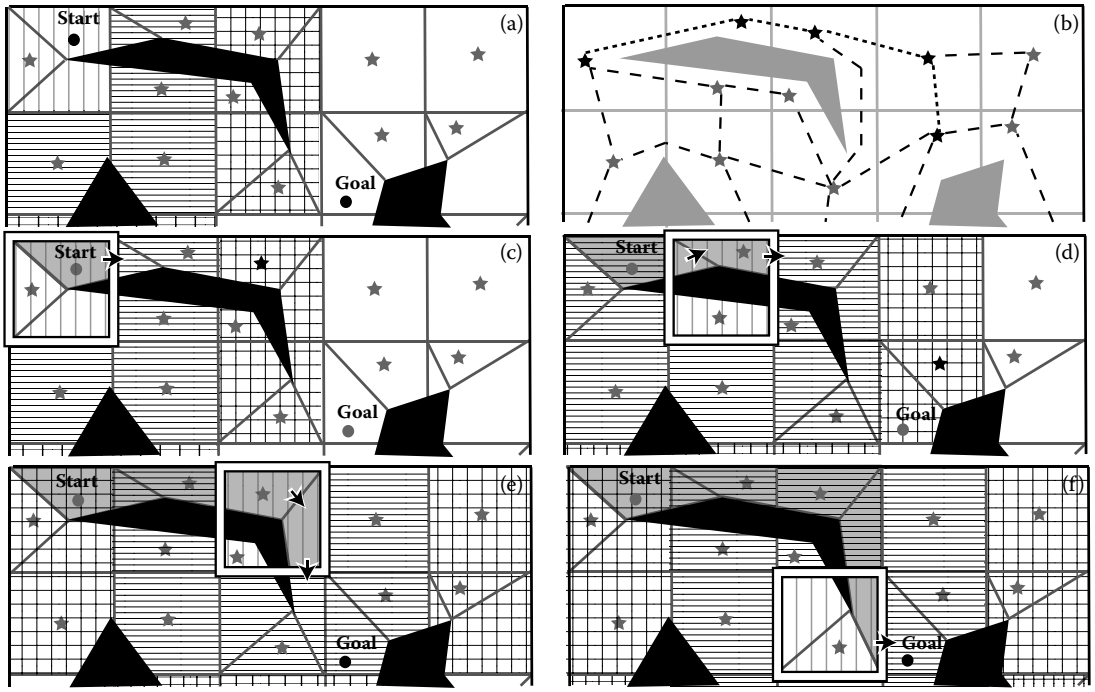


Figure 20.9

Example of pathfinding using precomputed data. Figure (a) shows the query start and goal point. Figure (b) shows the top level with the top level path. Figures (c,d,e,f) show the path planning process for the current tile (vertical line pattern) in each pathfinding iteration, as well as the edges chosen (black arrow). The resulting mesh path is shown in gray.

it is searched for in the upper level (b). Figures 20.9c,d show how the subgoal is chosen. It is the furthest component center on the upper layer path that is still inside the local lookup table (patterned tiles). The current tile's lookup table contains the next edge to use. The Figures 20.9e,f show that once the goal is inside the neighbor tiles, the direct node–node table is used. This example shows the complete pathfinding process, but if the goal is sufficiently far, it is not necessary to compute the complete path for every hierarchical layer. This allows for very fast path computation.

20.5.5 Hierarchy

In order to further reduce memory usage, we use not just two, but three hierarchical levels on the largest maps. For instance, with one of our tested maps, the table size is 16.5 mega octets with 2 levels, 8.8 with 3 levels, 8.7 with 4 levels, and more than 400 without any hierarchy.

We use the same process as previously described to create the upper level. Figure 20.10 shows the second level added to the example of Figure 20.8. In Figure 20.10, the second level has a tile size of 2 by 2 sublevel tiles. For the upper level tiles, the nodes (dots or stars in Figure 20.10a) are the component centers of the lower layer (stars in Figure 20.8a). For each tile it is possible to compute its component centers (stars in Figure 20.10a), which become the nodes of the highest layer graph (Figure 20.10c).

In our test, the mean and maximum number of components for the lowest layer (mesh) are 1.5 and 15, respectively, and for the middle layer, 3.2 and 19, respectively. This approach gives a good compression, minimizing the table size.

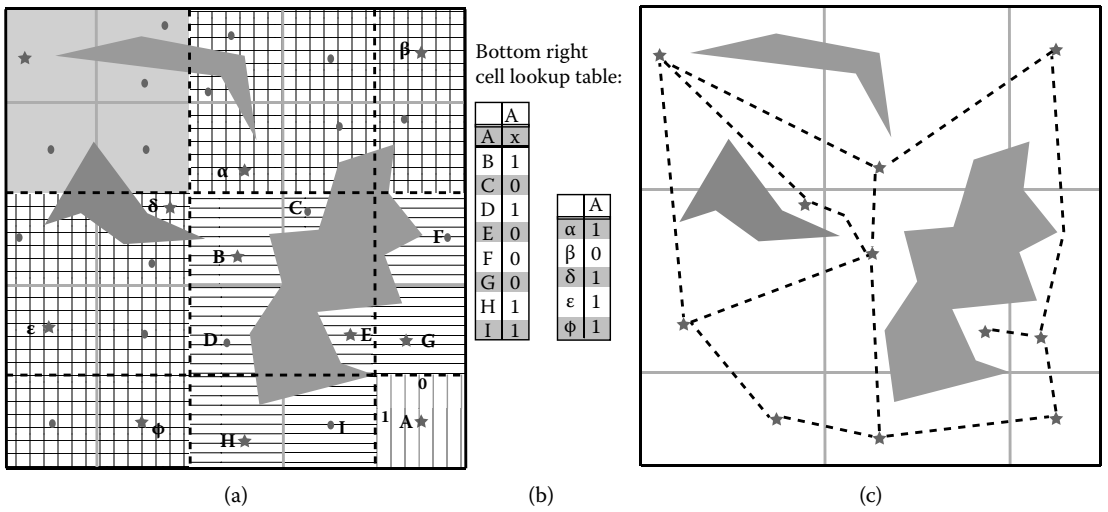


Figure 20.10

Figure (a) shows the second layer tile (dashed line) with dots for nodes and stars for component centers. The edge index of Node A is also shown. Figure (b) shows the bottom right tile lookup table. It has a node-to-node table (left side) for the center tiles (line pattern) and node-to-component center table (right side) for the border tiles (grid pattern). Figure (c) shows the top level node graph corresponding to Figure (a). Component centers of the lower layer, shown as gray stars, are the nodes of the top layer. The connections are shown with dashed lines.

Listing 20.1. Pseudocode of the table building process, showing the main steps of the algorithm.

```
ComputeConnectivity();
FallingMeshSetup();
BuildMeshTable();
for(int i = 0; i<max_level; ++i) {
    ComponentComputation(i);
    SplitProblematicComponents(i);
    ComputeComponentCenter(i);
    if (i+1 != max_level) {
        AddHierarchy(i);
        BuildHierarchicalTable(i);
    }
}
for(int i = max_level-1; i>0; --i) {
    RemoveInvalidSubLink(i);
}
```

20.6 Table Generation Algorithm

The previous section explained the main ideas behind the hierarchical table data used for navigation. In this section we will explain the algorithm computing this data in more detail and discuss some pitfalls that need to be avoided in order to obtain reliable data.

Listing 20.1 shows the table builder algorithm pseudocode. It iterates over the hierarchy of levels and applies specific functions to the lowest level (mesh data). In order to avoid several inconsistencies in the generated tables, some additional measures are taken. We will detail all those steps in this and the following sections.

20.6.1 Compute Connectivity

The `ComputeConnectivity()` function is probably the simplest one. Since the navigation mesh generation algorithm produces a 3D mesh file, it is necessary to compute the connectivity between its polygons. Our code is based on the "building an edge list for an arbitrary mesh" algorithm [Lengyel 05]. The only modifications made were to add support for falling polygons. Since these can overlap, additional rules were added based on the falling polygon's material (i.e., falling output, falling portal between tiles, falling path from edge, etc.).

20.6.2 Falling Mesh Setup

To simplify manual editing of falling meshes, the only requirement imposed on them is that a special output marker is placed at the end of a fall (a special triangle denoting the output edge). Because it is possible to have an output edge connected to several walkable polygons, the table must split falling edges to match the underlying polygon boundaries. `FallingMeshSetup()` is also in charge of computing the falling path and setting up the falling output edge.

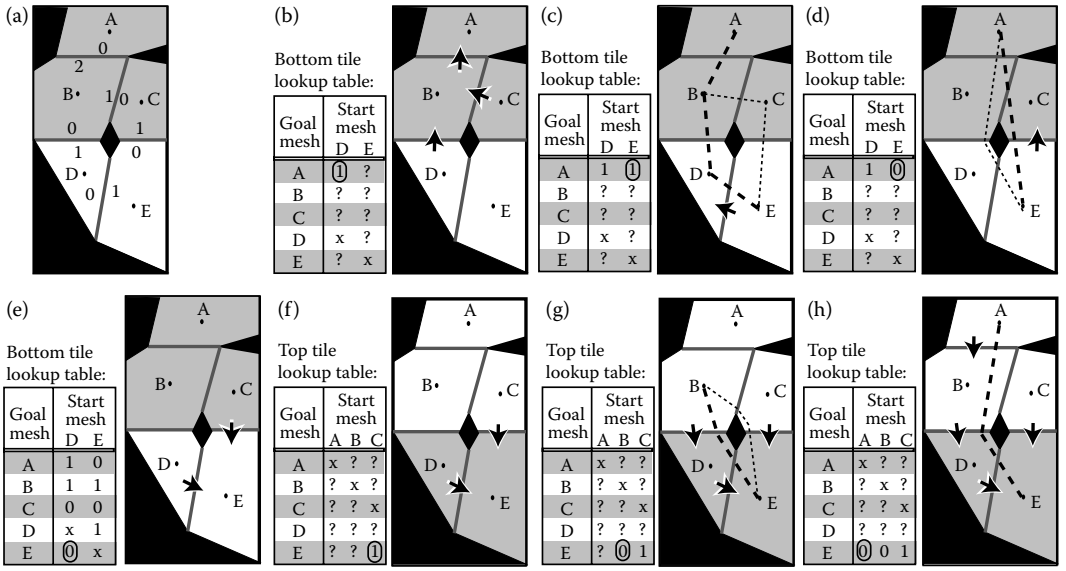


Figure 20.11

Figure (a) shows the two tiles used with the dots at the polygon center and the edge number. Figures (b), (c), and (d) show the table building process for the goal A and the bottom tile. Figure (b) shows the edge selection of D. Figure (c) and (d) show the edge selection of E, respectively, for the additional and smoothed distances. Figure (e) shows the table update for the last goal: E. Figures (f), (g), and (h) show the table building process for the top tile with the goal E for the starting polygons C, B, and A, respectively.

20.6.3 Distances Between Polygons

To build the lookup table, we tried to minimize the distance between the polygons' centers. We experimented with two ways of computing this distance: additional and smoothed.

The additional distance is the sum of the intermediate distances. Going from Polygon A to Polygon E through Polygon D, we have the distance AE equal to the sum of the distances AD and DE (Figure 20.11c).

The smoothed distance uses the smoothed path between centers as distance. In that case the distance from Polygon A to Polygon E through Polygon C is smaller or equal to the sum of the distances AC and CE (Figure 20.11d). Note that we will discuss the pros and cons of both distances in Section 20.9.

20.6.4 Table Builder

To decrease the computation time and memory usage, we decided to build the complete map lookup table data only for the highest layer (which has only one tile). For lower layers, the lookup tables take only the 21 neighboring tiles into account. In Figure 20.8a, these neighbor tiles are shown with line or grid patterns. Because we have not yet determined the component centers, a node–node table is computed instead of the node–component table.

The algorithm builds a lookup table for each tile independently (in fact, this process is multithreaded). Note that the shortest path result may change with the neighborhood; it is not possible to reuse it from tile to tile. For instance, in Figure 20.8a the path between

π and θ makes a long detour inside the center cell neighborhood. In the bottom right tile neighborhood, the path from π to θ is the shortest one.

To avoid conflicting paths to one fixed goal, we compute the table data for one goal to all other nodes (reversed search). The search is stopped when all the nodes in the current tile are updated (Figure 20.11e).

Figure 20.11 shows the table building process for the Dijkstra algorithm at the mesh level (function `BuildMeshTable()`). As shown in Figure 20.11c,d, the smoothed-path distance can give a better result than the additional distance. However, this approach does not necessarily yield the shortest path (as for example in Figure 20.11h). Since it has been decided that going from B to E is done through D (Figure 20.11g), the path from A to E must go through D. Note also that with smoothed distance, paths are not symmetric (Figure 20.11d,h).

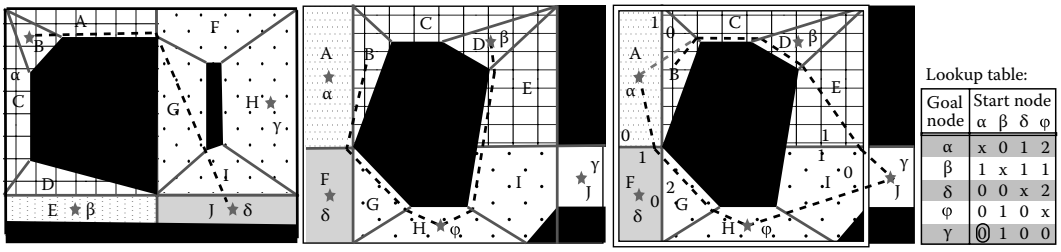
20.6.5 Hierarchical Table Builder

The function `BuildHierarchicalTable()` is in charge of the upper hierarchical levels. It is similar to the function `BuildMeshTable()`, explained previously. For this algorithm we also tried the smoothed and additional distances. The smoothed distance uses the distance computed in the lower layer between nonadjacent nodes (Figure 20.12a). This is valid only if the lower layer is not using additional distance. The smoothed distance significantly increased the quality of long paths.

Note that with smoothed distance, paths are not required to go through the component center as shown in Figure 20.12a. The shortest path from B to J is B, A, F, G, I, J. The shortest upper layer path, α, γ, δ, traverses through Component γ: {F, G, H, I}, but not through its center, Node H.

The function `BuildHierarchicalTable()` includes an additional constraint we refer to as the "subnode path" constraint. Its purpose is to ensure the validity of the algorithm responsible for solving the table inconsistencies presented in Section 20.7.2.

This constraint applies when the subnode center and one other subnode have different next components for the same goal. The Figure 20.12b shows such a situation. Imagine that the start point is B and the next subgoal is H. For the upper layer node there is a direct



Goal node	Start node α	β	δ	φ
α	x	0	1	2
β	1	x	1	1
δ	0	0	x	2
φ	0	1	0	x
γ	0	1	0	0

Figure 20.12

Problematic cases to take into account during the hierarchical table building process. The Greek letters represent the upper layer nodes (i.e., the polygon components). Mesh layer tiles have different patterns. Figure (d) shows the lookup table generated for the upper layer tile, which is a group of 2-by-2 mesh layer tiles shown inside the white border rectangle of Figure (c).

link β-φ, that is, the subnode path goes from D to H with only one change of component. However, for B, the shortest subnode path to H is B, A, F, G, H, which goes through α (A). Since the path β, φ can imply going through α, the upper layer path α, β, φ can yield a table inconsistency α, β, α (i.e., the subnode path A, B, A).

The algorithm detects all the cases where the component subnodes have a path not going directly to the adjacent components. These result in forbidden 3-node paths (e.g., α, β, φ) that are added as rule-based constraints. For instance, β cannot have α as previous node if its next node is φ. During the Dijkstra search process from goal to start, if one of these rules applies, the previous possible node is rejected. Figure 20.12c shows the table building process from the goal γ. The paths from β (β, φ, γ), δ (δ, φ, γ), and φ (φ, γ) have already been decided; α cannot be chosen as the previous node of β even if it is the shortest distance. Instead, the output of α must be δ (Figure 20.12d). Note, however, that in this example, the mesh tables have no inconsistencies between each other. The path from A to J will use the mesh table resulting in the shortest path A, B, C, D, E, I, J.

20.6.6 Add Hierarchy

The `AddHierarchy()` function is responsible for building a higher hierarchical level from the lower level component centers. The links between the upper layer nodes are defined as the paths between their subnode centers, as explained in Section 20.5.5.

20.6.7 Component Computation

Component computation is done in two phases. The first phase, performed by the function `ComputeComponent()`, calculates components using the node connectivity information see (Section 20.5.1). The only thing to keep in mind for the first phase is the use of unidirectional connections. For instance, with 3 nodes N_1, N_2, N_3, with the unidirectional connections N_1 to N_2, N_2 to N_3, and N_3 to N_1, we have N_1, N_2, N_3 in the same component.

The second phase splits those components, and will be explained in the following section.

20.7 Table Inconsistencies

If used as described previously, the table generation will produce inconsistencies. These are defined as infinite loops within the node path that ultimately prevents reaching the goal. Without unidirectional links, inconsistencies always occur between two bordering nodes (i.e., linked nodes in different tiles). With unidirectional links, the loop can include an indefinite number of nodes but at least two border nodes. These inconsistencies are due to both the local nature of the lookup table generation and the hierarchical approach, and are solved during the table building process.

20.7.1 Split Problematic Components and Compute Component Center

`SplitProblematicComponents()` is responsible for solving two possible inconsistency scenarios within the tables: the opposite path inconsistency and another one we refer to as the convexity problem. The methods of solving each of these are similar. For each node in a component, determine whether it is compatible with the other component node(s). This implies computing paths within the tile neighborhood. These paths are based on the 21 neighbor tiles in the lookup table computed in Section 20.6.

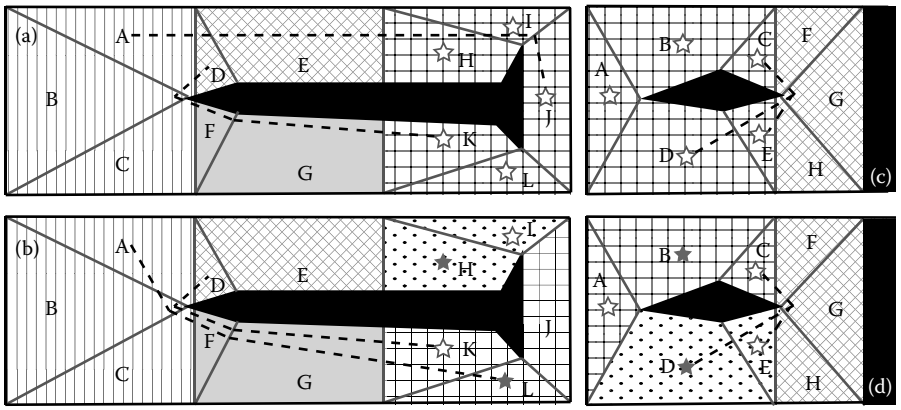


Figure 20.13

This figure shows how grid tile components are split. Areas with the same pattern correspond to the same component, while empty stars indicate potential component centers. Figures (a) and (b) show the opposite path inconsistency, while figures (c) and (d) show the convexity problem. Figures (a) and (c) show the components before and figures (b) and (d) show them after the split.

20.7.1.1 Opposite Path Inconsistency

The opposite path inconsistency is shown in Figures 20.13a,b. Going from D to K using the node–node table yields a shortest path of D, A, B, C, F, G, K. However, going from A to K is done through the node–component center table. If the center of the component including K is H, I, or J, the shortest path is A, D, E, H, I, J. This creates an inconsistency. In the same way K, L, J cannot be the center of the component including H or I. This shows that the pairs H, I, and K, L cannot be in the same component, and therefore we must split them.

This problem occurs only on the edge border of neighbor tiles (nodes D and F).

20.7.1.2 Convexity Problem

All paths inside a component are supposed to stay inside the component. An upper level node should not have a link to itself through another node. Figures 20.13c,d show such a case where the shortest path from C to D is C, F, G, H, E, D, and goes outside the component. This implies that C and D cannot be in the same component and should be split.

20.7.1.3 Component Center

Once the splitting constraints have been found the component is computed again. This time there are several possible choices of components. For instance, in Figure 20.13b we can have the components {H, I, J} and {K, L} or the components {H, I} and {J, K, L}. The algorithm selects the components with the smallest surface area first.

Afterwards, the component centers can be calculated. However, due to problems such as the opposite path inconsistency, some component centers may be illegal. For instance "J" cannot be a component center in Figure 20.13b.

Currently, the component center is chosen from within the set of allowed centers to be the barycenter of the component. It is possible to optimize center selection by, for instance, decreasing the distances between linked centers.

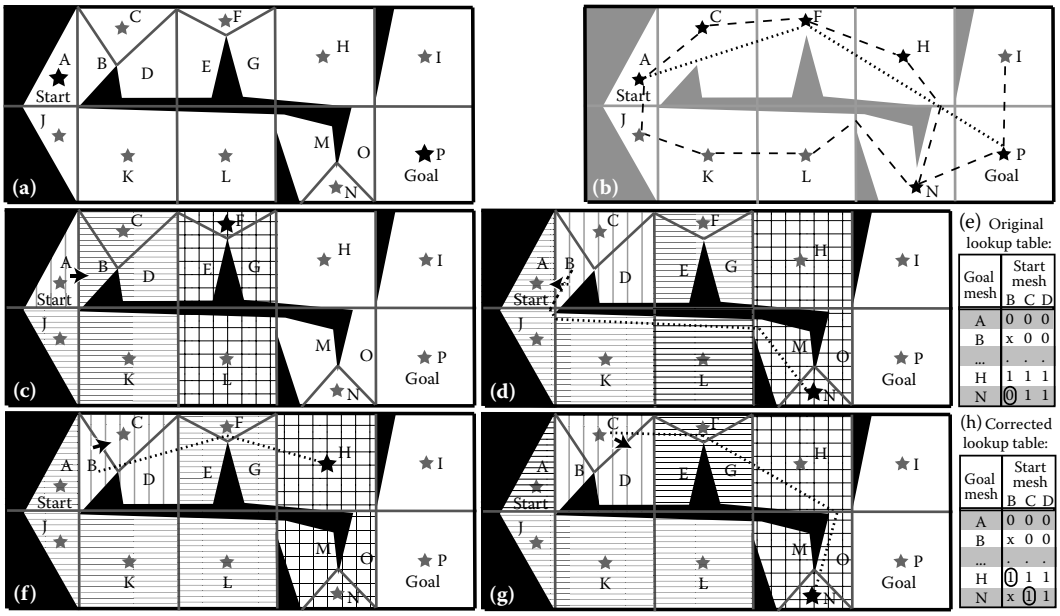


Figure 20.14

Invalid sublink. The pattern shows the table access method (line for node–node, grid for node-component). Dotted lines represent the shortest distances. Figure (a) shows the problem arising when the start point is within A, and the goal within P. Figure (b) shows the top layer nodes with links shown as a dashed line, the shortest path with black stars, and the smoothed distance is shown as a dotted line. Figure (c) shows how the subgoal F is used to go from A to B. Figure (d) shows that if the subgoal N is used from B, there is an inconsistency. Figure (f) shows that the inconsistency is solved if the subgoal H is used instead. Figure (g) shows that the path from C to N is valid. Figures (e) and (h) show the lookup table used in figures (d) and (e), respectively, before and after the sublink removal.

20.7.2 Remove Invalid Sublink

The function `RemoveInvalidSubLink()` is responsible for removing table inconsistencies due to both the local nature of the table building process and the path heuristic, which chooses the farthest component center as a subgoal. Depending on the tile neighborhood, the subgoal selection may produce a path that doubles back on itself. This is solved by disallowing the selection of the farthest subgoal by removing its link in the lookup table. Figures 20.14e,h show how the subgoal is changed by removing the output edge in the lookup table for the path B to N.

Since the hierarchical building process ensures that the upper layer tables are consistent for adjacent nodes (constraints in Section 20.6.5), removing the farthest subgoals will still result in a valid path.

This type of inconsistency can only occur on tile borders when the subgoal is changing, and is mainly a problem with long paths. In these cases, the upper layer data is more accurate than the lower layer and for this reason, the algorithm validates the layers from top to bottom.

Figure 20.14 shows a problem arising from the dynamic nature of the subgoal selection. In some cases, selecting the farthest possible subgoal can lead to doubling back. In the

example shown in Figure 20.14d, the node-component table access from B to N is removed. The last valid subgoal to go to P will be H, removing the inconsistency in Figure 20.14f.

20.8 Table Use on Server

In Section 20.5.4, we saw how a path request is handled. As previously mentioned, it is not necessary to compute the polygons or the upper layer nodes path until the goal. Since no searching is required, the request is very fast.

However, this only yields a polygon sequence rather than the actual path to follow. An optimized version of the Funnel algorithm [Douglas 06] is used to calculate a smoothed path. Even if the NPC radius is not taken into account when finding the mesh path, it is used by the Funnel algorithm to push the path away from the border polygons.

The use of the agent radius for path smoothing leads to better looking paths but does not prevent large NPCs from intersecting with a collision boundary. It is possible to create a new table (not necessarily a new mesh) to avoid narrow passages, but this can result in long detours.

To further improve the path quality for small distances, a straight path search is done to the goal. If obstacles are found, the pathfinding algorithm is used.

20.9 Results

The presented navigation system provides very fast query speed at the cost of a loss of flexibility. One query takes about 4 micro seconds. The main problem with precomputed data is that any dynamic change requires a new table; therefore, this approach is best suited to static worlds. However, some dynamic changes such as opening/closing doors are supported by allowing/forbidding to cross the door polygons. This check is done at runtime.

This disadvantage is outweighed by the dramatic reduction in processing power needed at run-time by the navigation system. The most costly task is the relatively simple query for the nearest polygon, which is also facilitated by the regular grid partitioning. Even the smoothed path computation is less expensive.

20.9.1 Smoothed Distances

Using smoothed rather than additive distances for computing the lookup table (Section 20.6.3) improved the path quality.

Benchmarking with random trajectories showed that the length of more than 70% of the paths was unchanged. Of the remaining paths there were both increases and decreases in length. However, the number of trajectories reduced by more than 5% was 10 times greater than those that had increased by the same proportion.

The smoothed distance technique came at a cost: asymmetric paths, increased complexity of the algorithm, and twice the table computation time. However, the computational time is still within an acceptable range with less than 80 seconds for the largest maps tested.

20.9.2 Table Builder Algorithm

We experimented with three algorithms to compute the lookup table: Floyd–Warshall, Dijkstra, and A*. The best result in terms of path quality was given by Dijkstra using the smoothed distance. The fastest algorithm was given by the combination of A* using the additional distance. The heuristic used by A* was the distance to the tile being processed.

The Floyd–Warshall algorithm was slower, and it was difficult to add new constraints or to use the smoothed distance. It calculates all the shortest paths between all the nodes of all the 21 neighbor tiles. In the example in Figure 20.8a, this means that 42×42 paths were needed instead of 2×42 (Figure 20.8b).

The heuristic for A* with smoothed distance is the Euclidean distance to the nearest border of the currently processed tile. Even if this algorithm returned the shortest possible path for all the polygons in the processed tile, its global result would be worse than the Dijkstra version. The tiles tables' inconsistencies increased and, to resolve them, longer paths were used.

It is worth noting that all three algorithms yielded similar results when used in conjunction with the additional distance metric.

20.9.3 Table Size

The biggest table size is about 4 mega octets (Mo) for a 1.5 km^2 forest. Dungeon and town sizes are under 500 kilo octets. Note that this data is not based on the final maps, and that during the map design process, table sizes sometimes reached up to 10 Mo.

20.9.4 Alternative Uses

Other than navigation, the mesh was also used for checking the collision data, but its most interesting secondary application is the in-game 2D map autogeneration. The player has access to a 2D map of the world that must represent the areas accessible by the player. This map is basically a projection of the navigation mesh. Unfortunately, the details of its generation process are out of the scope of this article. However, it is worth showing in Figure 20.15 one of the maps that this cool feature enabled us to generate.

20.10 Conclusion

This article has described the steps taken for automatic generation of a precomputed navigation system. We have underlined ways to handle unidirectional paths and evaluated different methods of performing distance measurements. The resulting system allows very fast navigation requests for large and detailed worlds. It is a good solution for all server-based applications where there are strong constraints on security or client hardware limitations.

The whole generation system is completely automatic, freeing up the rest of the team to concentrate on more creative work. We hope that this article will be useful to others creating a precomputed navigation system and help them avoid potential pitfalls.

Acknowledgments

We would like to thank all the FFXIV team, and especially Shinpei Sakata who worked on the 2D map autogeneration, allowing us to share this unexpected use of the navigation mesh.

Figure 20.15

2D Game world map autogenerated from the navigation mesh.

References

[Axelrod 08] R. Axelrod. "Navigation graph generation in highly dynamic worlds." In *AI Game Programming Wisdom 4*, edited by Steve Rabin. Reading, MA: Charles River Media, 2008, pp. 124–141.

[Dickheiser 03] M. Dickheiser. "Inexpensive precomputed pathfinding using a navigation set hierarchy." In *AI Game Programming Wisdom 2*, edited by Steve Rabin, Reading, MA: Charles River Media, 2003, pp. 103–113.

[Douglas 06] D. Jon Demyen. "Efficient Triangulation-Based Pathfinding." Master Thesis, 2006. Available online (https://skatgame.net/mburo/ps/tra.pdf).

[Lengyel 05] Eric Lengyel, "Building an Edge List for an Arbitrary Mesh." Terathon Software 3D Graphics Library, 2005. http://www.terathon.com/code/edges.html.

[Miles 06] D. Miles. "Crowds in a polygon soup: Next-Gen path planning." Presentation on *Game Developers Conference (GDC)*, 2006.

[Mononen 12] M. Mononen, "Recast." http://code.google.com/p/recastnavigation/

[Sterren 03] W. van der Sterren. "Path look-up tables—small is beautiful." In *AI Game Programming Wisdom 2*, edited by Steve Rabin. Reading, MA: Charles River Media, 2003, pp. 115–129.

[Sturtevant et al. 10] N. Sturtevant and R. Geisberger. "A comparison of high-level approaches for speeding up pathfinding." *In Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2010*. Available online (http://web.cs.du.edu/~sturtevant/papers.html).