Using Navigation Meshes for Collision Detection

D. Hunter Hale and G. Michael Youngblood
University of North Carolina at Charlotte, Computer Science
9201 University City Blvd., Charlotte, NC USA 28223-0001
{dhhale,youngbld}@uncc.edu

ABSTRACT
Traditionally, tree-based spatial data structures such as k-d trees or hash-based structures such as spatial hashing are used to accelerate collision detection, and navigation meshes are used for agent path planning. In this paper, we present a series of algorithms to replace the traditional tree-based spatial data structures with the graph-based navigation-mesh. The advantages of using a single data structure for both agent navigation and collision detection acceleration are twofold. First, the costs of constructing and maintaining two unique data structures are cut in half if a single data structure provides both spatial groupings for rapid collision detection and search space reduction for path planning. Second, using one spatial structure, development time can be shorter and, at runtime, there is generally less memory overhead. We present the results of an experiment that compares a navigation mesh as a collision detection accelerator to two popular and commonly used forms of spatial data structures, the k-d tree and the spatial hash map. We also compare its performance to a world without any spatial data structures to provide a baseline of performance. Our results show a fifty percent decrease in collision detection time between dynamic objects in comparison to k-d trees. In addition, until the number of objects present in the world exceeds three thousand the navigation mesh accelerated collision detection outperforms spatial hashing accelerated collision detection across all tests.

Keywords
Navigation Meshes, Spatial Decomposition, Collision Detection

1. INTRODUCTION
In recent years there has been a movement in the games and simulation industry to create more physically accurate and photo realistic environments. This movement is constrained by the limitations of current consumer hardware. In order to overcome one of these limits, spatial acceleration algorithms have been introduced to speed up the processing of potential collisions in the game or simulation environment. These acceleration algorithms create data structures that provide a sorting or compartmentalizing of the objects present in the environment. With this compartmentalization, it is possible to reduce the number of collision tests performed each frame from an $n^2$ problem to a more manageable one. Even considering advances in graphical and processing power, it is still widely believed that acceleration algorithms will always be required [1].

The primary purpose of a spatial data structure is the compartmentalization of information and space for the acceleration of intersection tests between objects in the game world. Game objects can generally be divided into two groups: static objects that remain in place during the entire runtime of the application (e.g., generally buildings or other large immovable objects), and dynamic objects that can move through the environment (e.g., the player, artificially intelligent characters, and interact-able objects). Using a spatial data structure these objects can be sorted into well-defined groupings based on their coordinate location in the game environment. This sorting is generally hierarchical in nature, which results in many of the commonly used spatial data structures having some form of tree structure (e.g., k-d trees, Binary Space Partitioning Trees, Quad Trees, and Oct trees). Sorting is accomplished by partitioning the world into smaller and smaller chunks of space until some minimum size threshold is reached. Objects present in a chunk of space are then stored on the leaf nodes of this tree structure. This allows for faster collision and intersection tests because objects can only collide with other objects if they both reside in the same or neighboring leaf nodes on the tree.

Most games also maintain a secondary spatial data structure called the navigation mesh to provide connectivity data for agent path planning [6]. The navigation mesh divides the game environment into two general types of space: occupied space, which describes areas of the world that are filled by level geometry, and unoccupied space, which includes any traversable areas in the level. The mesh subdivides these two classifications of space into disjoint convex regions and builds a connectivity graph showing possible movement between each convex region. This graph is defined such that a region of unoccupied space and everything within it is a node on the graph, and traversable adjacencies between regions are represented as links on this graph. Characters perform path planning by locating their target and current locations in this graph and performing a graph search to lo-
cate a series of nodes to traverse to reach their destinations. Since the regions that compose this graph are convex, characters can move freely within and between adjacent regions without leaving unoccupied space and without colliding with occupied space obstructions.

It is possible to use a navigation mesh for collision detection without any dramatic alterations to the structure of the navigation mesh. We expect that the performance of the navigation mesh when accelerating collision detection is comparable to that of existing spatial data structures such as spatial hashing. We accomplish this by providing algorithms to perform the four primary functions of a spatial data structure (insert, remove, update, find collidable objects) for the navigation mesh and then we will examine the run times of these algorithms in comparison to k-d trees and spatial hashing. Finally, we will show experimentally that with real collision checks in a sample level environment, the navigation mesh provides superior performance at isolating small groups of objects that might potentially be colliding.

2. RELATED WORK

There are several ways to generate a navigation mesh, some of which produce meshes that are more suited to work in collision detection. Any method that generates full coverage meshes would work for the technique we present. One very effective method to generate Navigation Meshes is to use the Adaptive Space Filling Volume Algorithms (ASFV). This algorithm produces navigation meshes from either 2D representation of the environment (Planar-ASFV) [5] or the native 3D structure of the environment to be decomposed (Volumetric-ASFV) [3]. Unlike other navigation mesh construction algorithms, the ASFV class of algorithms produces a navigation mesh by seeding the world with unoccupied space regions and then growing these regions outward until they have expanded as much as possible while still maintaining a convex structure. It accomplishes this growth by subdividing the initial square (P-ASFV) or cube (V-ASFV) regions into higher order polygons or polyhedrons based on collisions with the geometry of the world. This process of growth and subdivision proceeds until no additional growth is possible at which point both P-ASFV and V-ASFV place additional regions into the world, which then grow and expand in the same manner. This cycle of placing new regions and then growing the new regions repeats until it is not possible to add additional regions. This algorithm produces high quality regions which have several unique properties such as a limit on the number of regions that can come together at a single point and more uniform shapes in the decomposition. These properties combine to produce a navigation mesh that works well with our collision detection acceleration extensions.

3. METHODOLOGY

Extending a navigation mesh to support the compartmentalization of objects for the acceleration of collision detection is a straightforward process. It requires that 4 additional functions be added over and above the path planning ones that already exist. First, there needs to be a way to add objects to the navigation mesh so that when queried each region of the mesh can report its contained objects. Second, objects need to be removed from the mesh if they are no longer present in a particular region. Third, there needs to be some function to move an object from one set of collision regions into another to reflect the fact that the dynamic objects in the world are capable of movement. Finally, and most importantly, there needs to be a function which can return groupings of objects which might be in collision with each other with the minimum amount of overhead. In the following sections we will examine both the costs of these functions as well as provide some detail on the implementation in the following sections.

3.1 Insertion of Objects

Inserting objects into the navigation mesh proceeds in a straightforward manner much like conventional spatial data structures. If we assume that all dynamic objects start in valid locations in the world then insertion just involves traversing the list of unoccupied space regions that compose the navigation mesh until the region which encompasses the object’s location is located. No special case is required in the insertion step to handle objects laying across the boundary of multiple regions. This is instead handled during collision detection by checking neighboring regions as well as the one the object is primarily believed to occupy. If we cannot assume that all of the starting positions for objects in the world are in fact located in empty space, then verifying the reliability of the placement is simple. If the object we are considering is not found in an unoccupied space region and assuming our navigation mesh fully describes the world then the object must lie in an occupied space region and its placement is therefore invalid. This process can be accelerated for objects about which something is known in advance. For example, if a character removes an object from their pocket and drops it then a new object has entered the world and collision checks will need to be performed on it. Instead of searching the entire navigation mesh, it is a simple matter to pass on the region the creator of the object resides in and then perform a breadth first search until the object is located. Many game and simulation objects are spawned from objects whose position is already known (i.e., projectiles or player constructions) so it is worthwhile to consider this when adding objects to the navigation mesh. This algorithm is the only one of the four presented here that often approaches the worst case runtime. This happens because inserting an object into a navigation mesh is random and might require checking every region in the navigation mesh until the correct region is located.

3.2 Removal of Objects

The removal of objects from the navigation mesh proceeds in a similar manner except that it takes advantage of the fact that the object already knows which node it has been assigned to. The object simply looks up which region it is contained in and then tells that region to delete the object from the list of collidable objects it maintains. This simplistic method results in deletion being a constant time operation.

3.3 Updating Object Positions

Updating the locations of objects on the navigation mesh is one of the more complex operations required to enable navigation meshes for collision detection. This is also one of the functions where the advantages of this algorithm over tree based data structures become clear. A standard tree based data structure performs updates by traversing up the tree from the object’s current location until it finds an area
that could contain the object. At that point the algorithm travels down the tree structure until it locates the smallest area that could contain the object. In many cases this results in reduced performance as simply moving from one region to a neighboring region might require searching all the way up to the root of the tree and then traversing all the way down another branch. Our update functions for navigation meshes take advantage of the tendency for objects to not move that far on a frame-to-frame basis. This implies that each object is still in the same collision region it previously occupied and that region should be checked first. Then neighboring regions should be examined if the first region no longer contains the object. We accomplish this by performing a breath-first search based on the last known position of the object. Because navigation mesh regions generally only have four or five neighbors this approach performs extremely well. The worst case of a can only occur in degenerate or very small navigation meshes where every region on the mesh is connected to every other region.

3.4 Find Collidable Objects

Selecting the objects that might be in collision with any given object is a two step process. First, all of the objects that occupy the same region as the given target object are added to the list of potential collision objects. This will account for most of the potential collisions for any given object and it also dramatically reduces the number of collision tests that must be conducted since the objects in one region are excluding extreme cases, such as navigation meshes with very few regions. The second step is to deal with the possibility that an object might extend over more than one region. Normally this is dealt with by subdividing the object into multiple parts and then tracking and recombing each part as needed based on the movement of the object. This is computationally expensive and a bit painful to implement, so our algorithm takes a slightly different approach. Instead of subdividing objects, we treat objects as only existing in a single region at a time and pull in the contents of the neighboring regions when it is time to do collision checks. This works well for navigation mesh generations techniques that can limit the number of neighbors any given region has. Once we have defined the potential set of objects the target object might be in conflict with, it is a simple matter to run a series of collision checks to determine if there actually are any collisions. The worst case for this algorithm, $O(n)$, is a condition that generally will not occur often as it again requires that every region share a common edge with every other region, which means the navigation mesh is degenerate.

4. EXPERIMENTATION

We performed an experiment to validate the performance of our collision detection extensions for navigation meshes. In this experiment, we evaluated k-d trees, collision-extended navigation meshes, spatial hashing, and un-accelerated collision detection on a pair of sample levels. The first is a Capture The Flag (CTF) environment which contains a pair of bases and some obstructions between the two bases to provide cover for players moving from one base to the other. This experiment showed that in all cases the navigation mesh collision detection outperforms k-d-tree and brute force collision detection. The navigation mesh performed better than spatial hashing until 3000 dynamic objects were present in the world. After that point the performance of navigation meshes degrades. Based on data from the Unreal Engine 3 by Epic Games [2] it is recommended that game environments have no more than 300-1000 objects in order to give reasonable performance on current systems. Within this object limit the navigation mesh outperforms all other algorithms. For a more detailed discussion of these experiments please see the extended version of this paper [4].

5. CONCLUSIONS

We have presented a system to extend the navigation meshes generated using the ASFV algorithms [5, 3] to support and improve the speed of collision queries for objects present in the game world. We have done this by providing four algorithms to extend navigation meshes to allow for object management (insert, update, remove, and find collidable objects). By utilizing a navigation mesh in this manner, a game or simulation developer is able to reduce the run time expenses in memory and CPU cycles of creating and maintaining one data structure for collision detection and another structure for agent navigation. In addition, since only one data structure is required, utilizing a navigation mesh in this manner reduces development time. Most importantly, we have shown experimentally (in [4]) that the navigation mesh actually performs better than the commonly used k-d tree data structures in all types of collision across multiple game environments. The time required to perform collision tests between dynamic objects is reduced by half. In addition, we showed that by taking advantage of the full coverage nature of the navigation mesh and the fact that objects generally do not move very far each frame we can resolve collisions with the environment in constant time instead of the $\lg(n)$ time required by k-d trees. Overall, the navigation mesh is an excellent general spatial data structure and it is ready for uses going beyond that of agent navigation.

6. REFERENCES