# A GROWTH-BASED APPROACH TO THE AUTOMATIC GENERATION OF NAVIGATION MESHES

by

David Hunter Hale

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2011

## ABSTRACT

DAVID HUNTER HALE. A growth-based approach to the automatic generation of navigation meshes. (Under the direction of DR. G. MICHAEL YOUNGBLOOD)

Providing an understanding of space in game and simulation environments is one of the major challenges associated with moving artificially intelligent characters through these environments. The usage of some form of navigation mesh has become the standard method to provide a representation of the walkable space in game environments to characters moving around in that environment. There is currently no standardized *best* method of producing a navigation mesh. In fact, producing an optimal navigation mesh has been shown to be an NP-Hard problem. Current approaches are a patchwork of divergent methods all of which have issues either in the time to create the navigation meshes (e.g., the best looking navigation meshes have traditionally been produced by hand which is time consuming), generate substandard quality navigation meshes (e.g., many of the automatic mesh production algorithms result in highly triangulated meshes that pose problems for character navigation), or yield meshes that contain gaps of areas that should be included in the mesh and are not (e.g., existing growth-based methods are unable to adapt to non-axis-aligned geometry and as such tend to provide a poor representation of the walkable space in complex environments).

We introduce the Planar Adaptive Space Filling Volumes (PASFV) algorithm, Volumetric Adaptive Space Filling Volumes (VASFV) algorithm, and the Iterative Wavefront Edge Expansion Cell Decomposition (Wavefront) algorithm. These algorithms provide growth-based spatial decompositions for navigation mesh generation in either 2D (PASFV) or 3D (VASFV). These algorithms generate quick (on demand) decompositions (Wavefront), use quad/cube base spatial structures to provide more regular regions in the navigation mesh instead of triangles, and offer full coverage decompositions to avoid gaps in the navigation mesh by adapting to non-axis-aligned geometry. We have shown experimentally that the decompositions offered by PASFV and VASFV are superior both in character navigation ability, number of regions, and coverage in comparison to the existing and commonly used techniques of Space Filling Volumes, Hertel-Melhorn decomposition, Delaunay Triangulation, and Automatic Path Node Generation. Finally, we show that our Wavefront algorithm retains the superior performance of the PASFV and VASFV algorithms while providing faster decompositions that contain fewer degenerate and near degenerate regions.

Unlike traditional navigation mesh generation techniques, the PASFV and VASFV algorithms have a real time extension (Dynamic Adaptive Space Filling Volumes, DASFV) which allows the navigation mesh to adapt to changes in the geometry of the environment at runtime.

In addition, it is possible to use a navigation mesh for applications above and beyond character path planning and navigation. These multiple uses help to increase the return on the investment in creating a navigation mesh for a game or simulation environment. In particular, we will show how to use a navigation mesh for the acceleration of collision detection.

## ACKNOWLEDGEMENTS

Looking back on the last few years I can say with confidence that without guidance provided by my committee members, the support of my family, and help from friends this dissertation would never have been completed.

I would like to express my deepest gratitude to my advisor Dr. G. Michael Youngblood for his guidance, encouragement, and most importantly his patience (especially during late night reviewing sessions). I would like to thank Dr. Barnes who assisted with the development of the proofs present in this document. Without Dr. Akella's suggestions regarding the difference between growth and event-based expansion, I would likely have not have followed the path that lead to the Wavefront algorithm. Dr. Subramanian provided me with the firm grounding in graphics and advanced rendering techniques that served me in good stead while researching and developing the computational geometry techniques presented here. Dr. Hetyei provided a solid set of advice and guidance in forming the final version of this document. My friend and coworker Fritz Heckel provided me with motivation, advice, and on more than one occasion helped me find and fix troublesome bugs in my code. I would also like to thank my first three undergraduate computer science instructors Dr. Holliday, Dr. Luginbuhl, and Dr. Shultz for providing me with an excellent foundation of knowledge upon which to build. Finally, I would be remiss if I did not thank the many people who proofread some or all of this document.

TABLE OF CONTENTS

LIST OF FIGURES

# LIST OF TABLES

# CHAPTER 1: INTRODUCTION

The representation of space and how non-player characters move through it is one of the primary challenges faced when creating characters for games and simulations. The characters in these multimillion dollar games and simulations who will be seen and evaluated by potentially tens of millions of people [11], are supposed to represent human beings. A human relies on vision to determine where they can and cannot travel in the environment. We naturally learn how to evaluate space as walkable or unwalkable. Unfortunately, despite the huge budgets that go into creating these virtual characters they do not have eyes with which to see their world. Instead they are dependent on some classification or representation of space provided to them by the Artificial Intelligence programmer. There are many possible representations of space that could be provided to the characters, but it is not possible to generate an optimal one (this is an NP-Hard problem [44]). In recent years there has been some movement towards a single type of representation based on a delineation of space into a series of connected convex areas called a navigation mesh, but there is still no unifying theory as to how to best generate these navigation meshes from the raw world geometry (level, map, and world can be used interchangeably to refer to these virtual environments). We present three growth-based methods to generate these navigation meshes called Planar Adaptive Space Filling Volumes (PASFV), Volumetric Adaptive Space Filling Volumes (VASFV), and the Iterative Wavefront Edge Expansion Cell Decomposition (Wavefront). These algorithms work by placing unit regions into the world and then expanding these regions outward until they encounter obstructions. These algorithms allow us to generate navigation meshes that are of a high quality (i.e., higher order polygons, fewer degenerate regions, and higher coverage) than existing techniques. The PASFV algorithm decomposes 2D representations of game or simulation environments (similar to how a blueprint can represent a building). The VASFV algorithm is similar to PASFV but consumes the 3D geometry of the game environment instead of a 2D representation of the game environment. The Wavefront algorithm improves on the two previous techniques by dramatically accelerating decomposition speeds. Using these algorithms we will show that we can generate better navigation meshes than such techniques as Delaunay Triangulation, Hertel-Mehlhorn Decompositions, and Space Filling Volumes.

There are many different types of representations available to express a virtual world to an agent. The two primary categories of representations are sparse and dense coverage representations. A sparse representation stores only key control points or regions and the pathways between them. A good example of sparse representations is a waypoint map, which is composed of some number of points chosen in the world, which are known to be valid locations for the agent to stand (see Figure 1.1). Valid paths between these points are then added to create a graph style representation of where the agent can walk in the world. Unfortunately, when using sparse representations agents are not provided with any information about the world outside of the small pieces captured in the representation. Also, waypoint-based navigation techniques tend to move from point to point and the graphs are constructed such that there are always unobstructed paths between nodes.

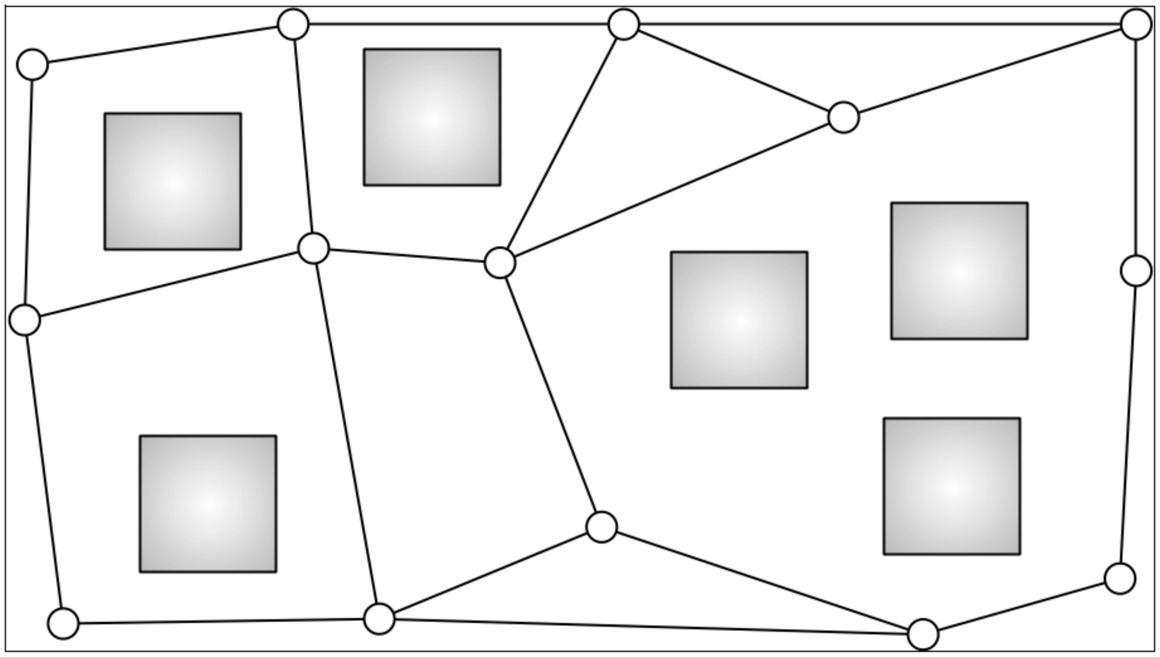


Figure 1.1: A sparse representation of a simple game world using a waypointing system. Obstructions are shown as gradient filled thick bordered areas while the waypoints are the connected circles.

Dense coverage representations of the world work to classify all (ideal case) or nearly all (not ideal, but common) of the space present in the world into walkable and unwalkable areas. These higher coverage representations provide several advantages when compared to sparse representations. Since the representation contains all of the traversable space in the world, the agent never has to enter unknown or unclassified areas. This is not the case for sparse representations of the world where the agent must employ some other path finding method to traverse unclassified space if they are allowed to traverse it at all. In addition, since a dense representation of the world is complete, a search will always return the shortest path. This is not assured in the case of sparse representations (e.g., a waypoint approach could miss a critical link sending the agent around a traversable area rather than through it). Some commonly used examples of dense coverage spatial representations would include: Binary Space Partitioning trees, Quad-Trees, Oct-Trees, Kd-Trees, Voronoi Diagrams, Dirichlet Domains, or Navigation Meshes (these representations are described fully in Chapter 2).

In general, dense representations are effective due to three principles. First, they consolidate areas of the map such that many homogeneous points on the map can be represented as a single region and that every point in a map is represented in exactly one region. Secondly, a good representation is able to determine which of these regions contain any given point in linear time. Finally, the representation has some notion of these regions being connected or passable from one to another. Together these principles combine to allow agent path planning algorithms to locate the agent in one region, the agent's destination in another region, and then search for a path between the regions in the representation rather than through the entire world. Once the agent follows this path to the region containing the target location, or if the agent and destination are in the same region initially, then the agent can enter some form of localized path planning to reach their destination.

Creating the impression of realistic characters requires that they do a plausible job of planning a path through a world from the characters' current locations to goal areas. This planning problem has traditionally been a well-studied area of AI and it essentially can be reduced to an issue of search. There are many search algorithms available, ranging from the traditional A* to more advanced or special purpose algorithms such as D* or memory limited A* [60].

However, characters do not search the entire world; instead they just search a sparse or dense representation of the world. The quality and type of representation can result in dramatically different levels of search performance. Deciding how to construct this representation from the many different types available is one of the fundamental problems for developing agents [64].

Building characters who are able to move around in 3D virtual environments in a rational manner is especially important since these characters have become commonplace through the games and simulation industries. How these characters move through and interact with the world is also important, as these actions are what the majority of users will see and use to evaluate the quality of the character as well as the quality of the game or simulation. Therefore, it is critical that these characters have the best possible representation of the world in order to improve their ability to search and path plan through it.

## 1.1 Navigation Meshes

Navigation meshes are generally considered to be derived from the concept of the meadow map used in early robotics [49]. A meadow map is a complete decomposition of all of the traversable space present in an area a robot would operate into a series of convex polygons (see Figure 1.2). The robot can then quickly determine which of the polygonal regions contains its target location and then search for a path between regions. It is important to note the use of convex polygons in the creation of these maps. By the definition of convexity [12], we know that the line connecting any two points that lie on or within a convex polygon is contained entirely within the polygon. Navigation algorithms are able to exploit this property by just planning from edge to edge of the polygon and being confident that the path is valid. This means that characters will not have to path plan about regions that contain corners or obscured geometry.

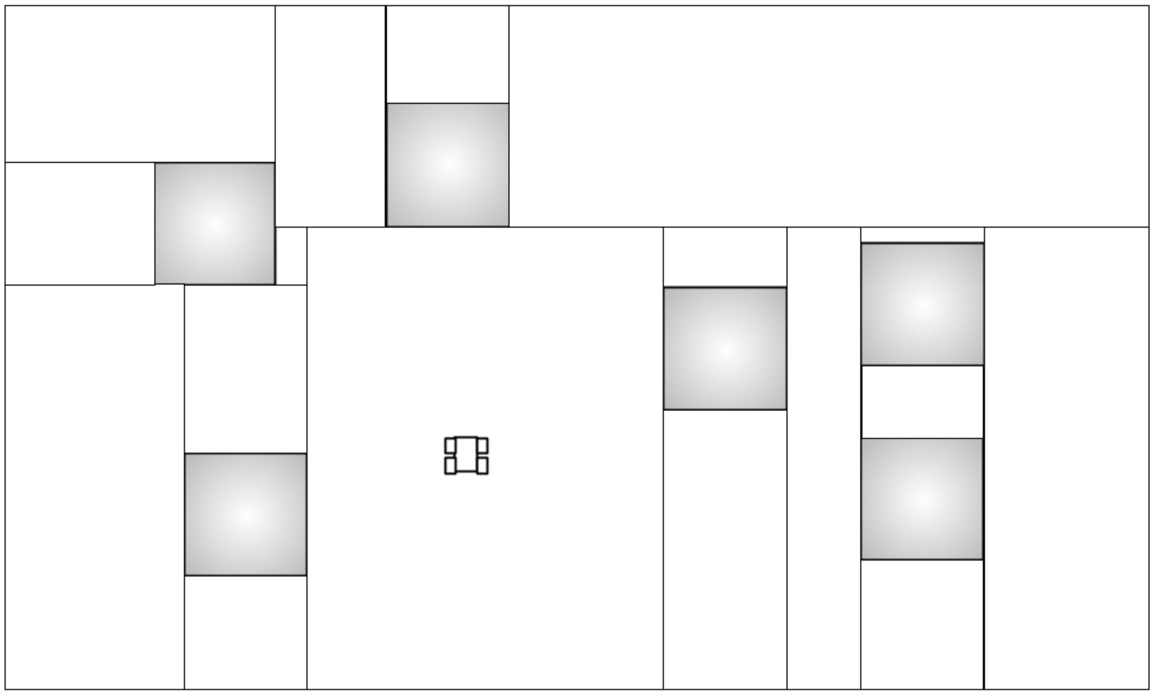


Figure 1.2: A simple navigation mesh for the same world shown in Figure 1.1.

Modern implementations of navigation meshes are limited primarily to agent navigation in games due to problems with registration of the robots position in the real world such that it corresponds to a location in the navigation mesh. Since navigation meshes the transfer from robotics to simulation and games the navigation mesh has become the dense style representation system of choice for game and simulation developers [46]. In addition, the navigation mesh spatial data structure provides benefits that go beyond simple navigation. As we will see, the spatial representation a good navigation mesh provides is a useful tool to help solve several other common problems in games.

## 1.2 Information Compartmentalization

One of the secondary advantages associated with the use of a navigation mesh is the ability to restrict an agent's knowledge of the world to the limited set of convex regions that the agent is either currently in or adjacent to. By utilizing this refinement called *information compartmentalization*, the number of objects or events the agent needs to reason about is reduced to a fraction of the total number of objects or events in the world. In addition, the degree of visibility of objects or events can be set per event to allow agents to notice and react to a large explosion on the other side of the world, but ignore a whispered conversation at relatively closer range. Unique sensor ranges can also be assigned on a per agent basis so that perceptive agents can notice events or objects at a greater range than less perceptive agents. Not only does using the navigation mesh for information compartmentalization result in a reduced set of objects to reason about, but it can also ameliorate the common complaint that the agent *knows too much about things it cannot possibly see.* This kind of reasoning reduction would not be possible if using a sparse representation of the environment.

One challenge to using a navigation mesh in this manner is that the presence of many thin triangles in the navigation mesh reduces the capability of the navigation mesh to compartmentalize objects into the regions of the mesh (as seen in Figure 1.3). This occurs because in areas where fans of thin triangles come together, it is possible and probable for objects to exist in many different regions at the same time as shown by the character in Figure 1.3, which exists in multiple regions simultaneously. This becomes worse in highly triangulated decompositions since the theoretical worst case for the maximum number of regions an object can occupy is unbounded as shown in Proofs 7.1 and 7.2 found in Sections 7.2 and 7.3. However, higher order polygon decompositions tend to avoid this problem since they can represent the geometry using fewer regions.

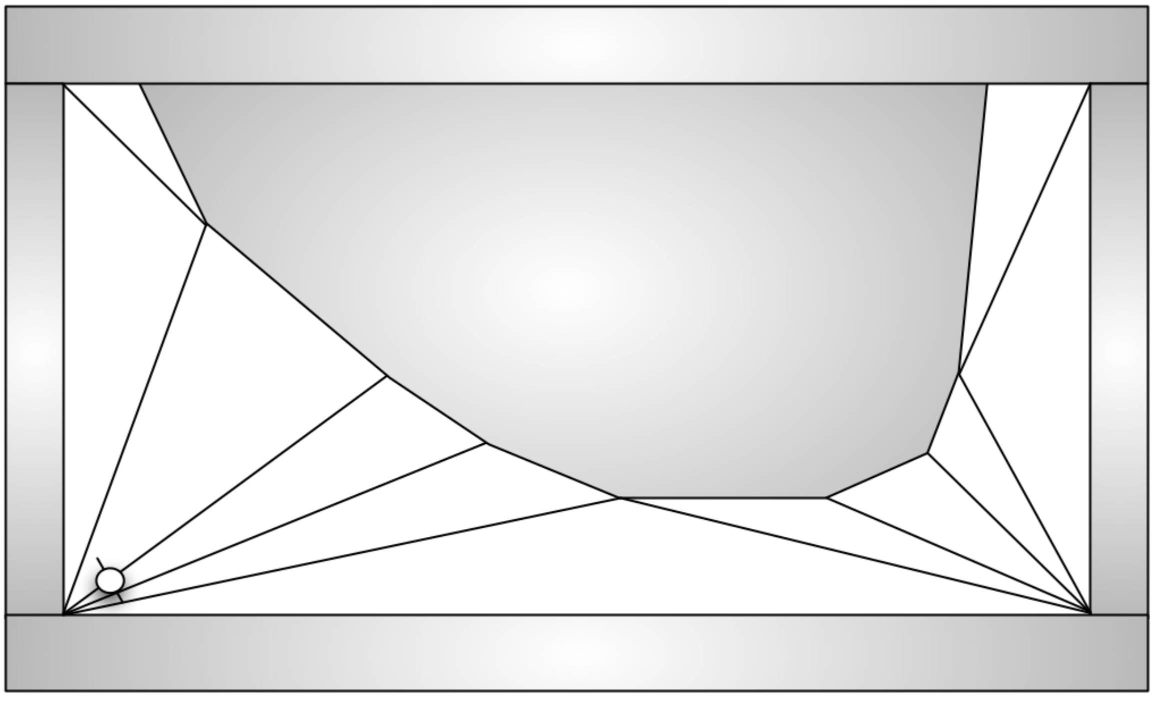


Figure 1.3: A navigation mesh (viewed from above) built from a triangulation algorithm. Notice how the triangles all converge in the corners of the world creating areas where a character might have difficulty localizing itself.

## 1.3 Collision Detection

The determination of the collisions between objects and between world geometry and objects both benefit from using the navigation mesh as a spatial data structure to filter and reduce the number of collision detection checks that need to be calculated. The first potential collision case that the navigation mesh assists with is the determination of whether an object is in collision with the world geometry. This is determined by solving the inverse problem (i.e., can we locate the object in question in a free configuration space region). If we can locate the test object inside a free configuration space region (i.e., negative space) then we know it cannot be in collision with a configuration space obstacle (i.e., positive space). The naïve solution for this test is to check the object in question against all of the free configuration space regions present in the world and if it cannot be localized to any region consider it to be in collision with the world geometry. This test can be greatly accelerated if instead of starting in a random region each object stores its last known free configuration space region position and the collision testing algorithm performs a breadth first search through neighboring regions from this last known position, since objects generally do not perform radical shifts in location from one frame to another. The second collision detection application is really a form of information compartmentalization applied to collision detection. Since objects generally fit inside a single region it is possible to just check collisions between dynamic (moving or active) objects against other objects from that same region (and neighboring regions if overlapping objects are permitted) instead of checking each dynamic object against every other object in the world.

## 1.4 Metrics

One of the open problems with the use of navigation meshes is a desperate need for a concrete set of metrics to evaluate different decompositions and compare them against each other. This is especially important since the generation of an optimal or minimal set decomposition can be shown to be NP-Hard [44] and therefore we have to use non-optimal approximations of the solution. Currently, the only commonly accepted metrics are to look at coverage of the world to check for gaps in the navigation mesh or to count how many regions compose the navigation mesh, which indicates how big the search space will be for the pathfinding algorithm [63]. These two metrics while useful are not by themselves enough to draw meaningful conclusions about how well agents will be able to navigate throughout the world using any given navigation mesh or how well a navigation mesh will be able to encapsulate and compartmentalize smaller objects that are present in the world. Furthermore, since there are few good ways to evaluate navigation meshes one of the primary benefits of automatic navigation mesh generation (the ability to rapidly generate unique navigation meshes for a given level) is not being fully utilized. If there were an effective automatic way to compare navigation meshes, it would be possible to generate a multitude of navigation meshes and choose the best one from that multitude based on optimizing the chosen metrics.

## 1.5 Navigation Mesh Generation

Unfortunately, there are still several challenges that need to be solved to completely streamline the use of navigation meshes in games and simulation environments. First, there is no standardized method to generate a navigation mesh. At present, mesh generation draws upon a wide variety of spatial decomposition, exploration, brute force, and spatial triangulation algorithms each with their own advantages and drawbacks. Some techniques generate very clean and elegant navigation meshes but take a long time to produce a mesh. A good example of this would be a by-hand decomposition of a level. Some techniques, such as Delaunay Triangulation, provide good coverage but generate a low quality decomposition filled with many oblong and irregular low order polygons. The triangular shapes can create areas where an unbounded number of regions can come together at a single point which allows objects to simultaneously reside in many regions at once (as shown in Figure 1.3). This interferes with the navigation mesh's ability to compartmentalize information, or act as an aid for collision detection (e.g., if objects exist across many regions than all of the objects in those regions must be tested). Existing growth-based techniques such as Space Filling Volumes (SFV) do not provide good coverage of the level. However, it is interesting to note the such growth-based techniques do provide advantages in terms of the very regular shapes they produce, the lower total number of regions in the navigation mesh, and the highly axis-aligned nature of the decomposition they produce. Path planning and path smoothing is easier if the agent can always assume that the boundaries between regions are axis-aligned. In particular, if an approach could be developed to better deal with the gaps in existing growth-based techniques and more closely approximate the incident geometry in the world then approaches such as Space Filling Volumes and the other growth-based systems would be ideal.

Given the widespread usage of navigation meshes, the lack of a widely accepted *best* method for generating navigation meshes, and the myriad other potential applications of navigation meshes beyond agent path planning the work in this dissertation focuses around the following central hypothesis.

*The shape and extents of the unoccupied space present in a game or simulation environment can be reduced to a series of convex regions using a growth-based algorithm, which will result in a smaller set of regions that contain fewer degenerate or near degenerate regions than existing spatial decomposition algorithms with an average higher order of polygon/polyhedron.*

We will then evaluate this hypothesis through testing of the growth-based navigation mesh generation algorithms in 2D with the Planar Adaptive Space Filling Volumes (PASFV) algorithm, in 3D via the Volumetric Adaptive Space Filling Volumes (VASFV) algorithm, and using an accelerated 2D approach via the Iterative Wavefront Edge Expansion Cell Decomposition (Wavefront) algorithm. These tests will be comparisons against what are currently considered high quality spatial decomposition algorithms (Hertel-Mehlhorn Decompositions, Delauney Triangulations, and Trapezoidal Cell Decompositions, and Automatic Path Node Generation). In addition to using standard measures, such as coverage and number of regions, we will evaluate the resulting navigation meshes using a set of newly generated metrics (e.g., evaluating the minimum interior angles, navigation mesh diameter, and region homogeneity) to determine the quality of a navigation mesh or set of navigation meshes procedurally and rate them on their ability to assist with agent path planning and information compartmentalization. Experimentation will show considerable improvements over existing methods of navigation mesh generation.

This dissertation begins with an overview of existing work in the construction of both dense and sparse spatial representations, along with an overview of some collision detection spatial data structures. We follow this discussion of existing work by introducing our new growth-based methodologies and approaches for generating navigation meshes. We then introduce our metrics for the evaluation of navigation meshes. After that we introduce our accelerated growth-based approach to rapidly generate navigation meshes and evaluate this new method using our metrics. To close the main body of this work we present the conclusions we have drawn from our experimental results.

# CHAPTER 2: RELATED WORK

There are currently a wide variety of decomposition algorithms in use among both industry and academia to produce navigation meshes. These algorithms all have advantages and disadvantages and it is hard to choose a *best* algorithm from them. This lack of a clear best decomposition algorithm can be attributed both to the lack of a concentrated analysis of the various navigation mesh generation algorithms and the lack of metrics to conduct this evaluation as well as the fact that generating an optimal navigation mesh is an NP-Hard problem.

This problem is NP-Hard due to the fact it is an optimization problem with an infinite number of possible solutions (given any "complete" set $S$ of decompositions a new decomposition could be generated by shifting one edge from any decomposition such that the resulting new decomposition would be distinct from $S$). Since it is not possible to check each possible solution to this problem and select the one with the fewest regions, and the problem can in fact be converted to the "Planar3SAT" problem (one of Karp's 21 NP-Hard problems [45]) we know that the problem is at least NP-Hard.

Where possible, we will be using a sample environment we created to illustrate each decomposition algorithm. In all of the examples we present the obstructed (occupied) configuration space in grey. Regions which are decomposing the environment will be presented in green. The environment, shown in Figure 2.1, represents a common set of obstructions and challenges to character navigation. The design of the obstructions are inspired by the sample obstructions provided by Lavelle [42] in his survey of spatial decomposition algorithms.

In addition to discussing navigation mesh generation algorithms, we will also look at collision detection acceleration data structures which could be used to construct structures similar to a navigation mesh.

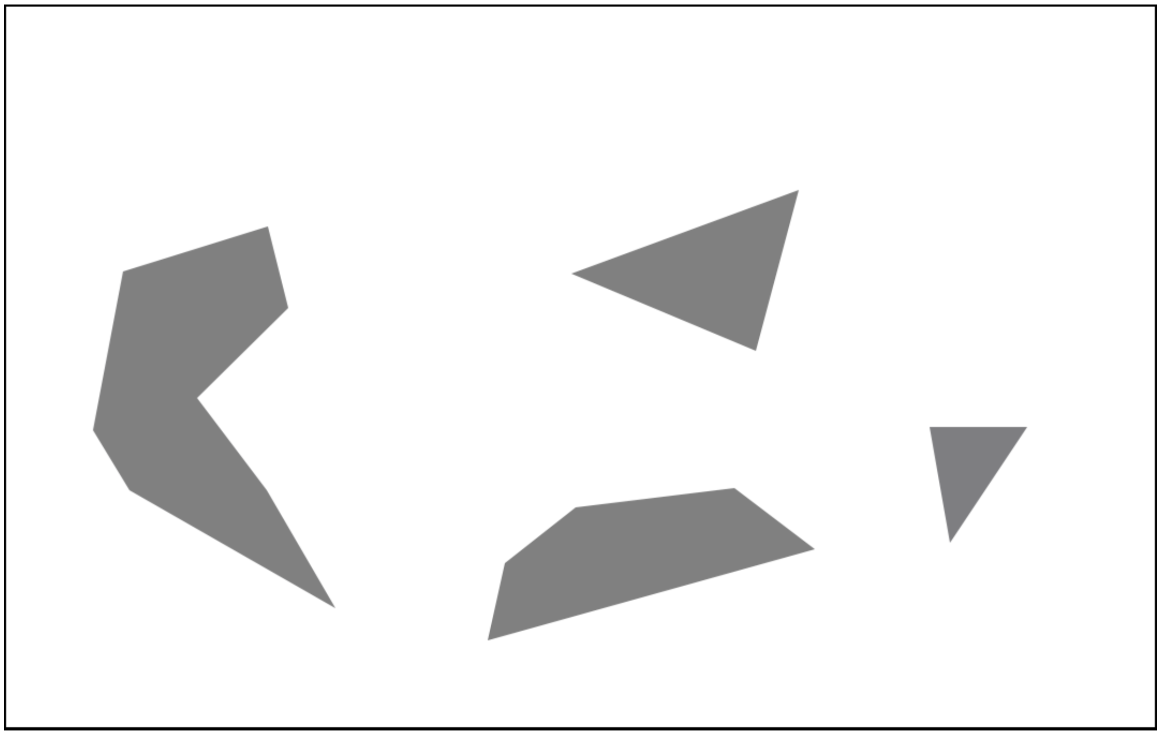


Figure 2.1: A simple level design featuring some non-axis-aligned obstructions. Obstructions are filled with a grey color. The design for this level was inspired by the sample planning problems shown by LaValle [42].

In this section we will present a brief overview of existing methods of generating spatial data structures to assist in character path planning and navigation. These data structures generally have unique generation methods; however, in the end they share a common trait in that they produce a planar decomposition of the level into convex spaces, which can be easily stored as a graph (with regions as vertices or nodes of the graph and adjoining regions connected by edges).

## 2.1 Voronoi Diagrams

Voronoi Diagrams are a form of spatial subdivision-based on dividing the world using lines drawn through the middle of all of the open space in the world such that every point on each line is equidistant from every closest pair of obstructions or more formally the Voronoi Diagram is the collection of points that is equidistant from every obstruction. This technique is based on work done by Voronoi [65, 66] though these diagrams are sometimes credited to Dirichlet [15] and referred to as Dirichlet Tessellations instead. There are two possible ways to use a Voronoi Diagram for agent navigation. The first and most common is to use the Voronoi Diagram of the open space in the world to generate a set of safe paths through the centers of all of the open space present in the world as illustrated by Russell and Norvig [54] on page 922. The agents then navigate using these paths; However, this does not actually produce a navigation mesh since only the paths formed from the Voronoi diagram are known and there is a vast quantity of unclassified free configuration space. Forming a navigation mesh from a Voronoi diagram requires that it be possible to generate a simple polygon that is composed of all of the free space present in the world [50]. The Voronoi diagram of this polygon produces the medial axis or skeleton as shown by Holleman [10, 17] of all free configuration space present in the world. Each cell of this skeleton is by the definition of a Voronoi diagram a convex region of free configuration space and as such the skeleton creates a navigation mesh, which can be used for agent navigation as shown in the work by Hollerman *et. al* [6, 26, 37].

## 2.2 Dirichlet Domains

The Dirichlet Domain Decomposition for agent navigation is derived from the Voronoi Diagram using a specific set of seed points. The free configuration space present in the level environment is seeded with a series of control points [48]. These points then *claim* all other free configuration space points which are closer to them than any other control points. This forms what is in effect a Voronoi Diagram (Dirichlet's work in this area is one of the reasons he is sometimes credited instead of Voronoi for the creation of Voronoi Diagrams) and these regions can be used as a graph for path finding. Unfortunately, the regions are not guaranteed to be convex and a line (path) from one point in one region to a point in a neighboring region (or even two points inside the same region) in a straight line does not guarantee that this path will be free from all obstructions as it would in a decomposition which produces only convex regions.

A 3D implementation of Dirichlet Domains also exists. In this version of the algorithm there is still a listing of predefined control points as in the 2D algorithm. In this case the control points will *claim* all of free configuration space that is closer to them than any other control point in 3D space instead of just looking at the points existing on a flat representation of the environment. This actually exacerbates the problem of selecting good locations for control points. For example, a control point on a lower level might grab points on the level above it in a 3D environment if it is closer to them than the control points on the upper level. This control point selection and placement problem is one of the things which prevents Dirichlet Domains from being commonly used [48] to build navigation meshes.

## 2.3 Probabilistic Roadmap

This algorithm generates a sparse representation of the world for agent navigation and planning purposes [40]. The probabilistic roadmap algorithm uses a sampling approach to randomly select points in the world. If these selected points exist in free space then they are added as vertices to the graph or *roadmap* under construction. This selection of random points continues until it reaches a user defined number of valid points for the roadmap. Alterations to the algorithm allow for the introduction of predefined points [43] or biasing the random seeding algorithm to ensure more even coverage of the open space [4, 38]. After all of the points have been selected, the next stage of the probabilistic roadmap algorithm calls for the connecting of all points to all other points which are visible to them such that the shortest paths are generated first 2.2. There is a user definable upper limit to the number of connections that can be made per node and of course it is possible to have fewer connections than this limit due to geometric obstructions. These connections form the edges of the navigation graph. The agent is then able to use local navigation to find the closest roadmap point and then travel along the roadmap to the closest point to its destination.

This algorithm works in both 2D and 3D as the point lattice that composes the navigation graph can either be established in 3D space or along some predefined plane. However, in addition to making no assurances regarding length of the paths, planning using paths generated by this algorithm expose the agent to a navigation hazard. Consider what would happen if the agent is trying to move to enter the roadmap from a point near but not on the map. In this case the agent will use local navigation to enter the nearest roadmap point. However, this map node location is not guaranteed to be accessible from the agent's current location. Instead, the agent might spend considerable time wandering around in a local navigation mode trying to find an accessible location to enter the roadmap from. Problems such as the one we outline with entering and by extension exiting the probabilistic road map prevent it from being an optimal general solution for agent navigation.

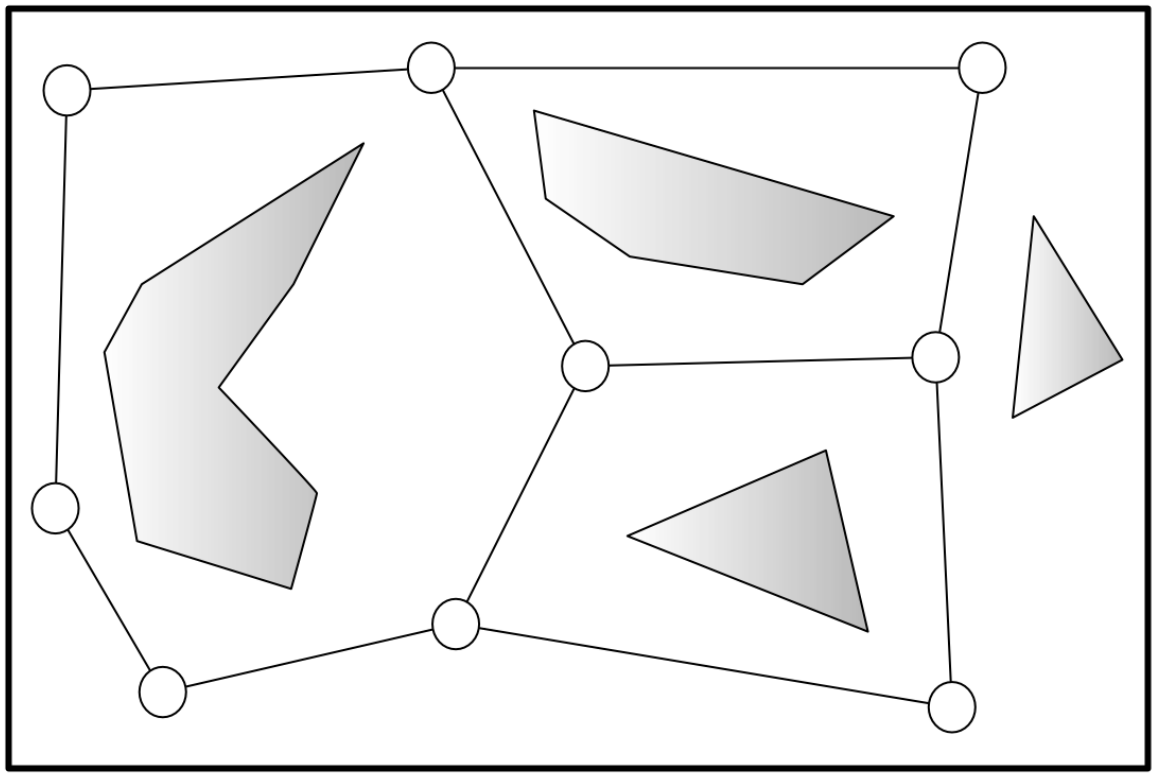


Figure 2.2: A Probabilistic Roadmap generated from our sample level. Note that is just one of many possible roadmaps for this level.

## 2.4 Delaunay Triangulation

The Delaunay triangulation algorithm is a well known method for generating a series of triangles from an input set of points. The Delaunay algorithm is straightforward—every vertex present in the world is connected to every other vertex to generate a series of triangles such that they do not intersect any triangles already created [13, 14]. The algorithm then attempts to reform the lines that compose these triangles in order to ensure that the average minimum interior angle of the resulting set of triangles is maximized. Formally, the Dealunay triangulation is the set of triangles with the maximum minimum interior angles which connect all of the vertices of obstructed configuration space. This algorithm generates an excellent coverage decomposition that works well for navigation, but can create problems with thin triangles and localizing non-point objects to a single area.

In each of the corners of the world there are many extremely thin triangles that come together at a single point. If an agent was attempting to navigate through this area the agent would almost certainly fall into several regions at the same time (we assume that agent has width and length and is not just a single point). Being in multiple regions at once makes the task of navigation much harder as the agent cannot determine which region it is in or which region it needs to go to next with any certainty. Solving this problem requires special logic and programming that adds to the overhead of agent navigation. In addition, areas with multiple thin triangles in them also reduce the ability of the navigation mesh to compartmentalize information since the information object could also lie across multiple regions. Furthermore, it is possible to show that this problem of triangles coming together at a point has no upper bound on how many triangles converge on any given point as shown in Proof 7.1 in Section 7.2.

The Delaunay triangulation algorithm can also be extended into 3D space. The 3D algorithm is similar to the 2D version since the first step is to connect all possible vertices present in 3D to form triangular faces. Connections between vertices are valid as long as they do not cross any already created triangular faces. Once all possible vertices are connected there will be a series of triangular prisms present in the world. The algorithm calls for the rotation of each face in every prism and then determining if the rotation results in an increase in the minimum average interior angle of the prism. Once each face has been given a chance to rotate the algorithm terminates yielding a navigation mesh composed of triangular prisms, which are guaranteed to have the highest possible minimum interior angle.

## 2.5 Other Triangulations

There are other forms of triangulation algorithms aside from the Delaunay triangulations we have already discussed. However, since the biggest problem with using triangular-based navigation meshes in games results from the presence of triangles inside the decomposition with low minimum interior angles (e.g., the long thin triangles). The focus of the Delaunay triangulation algorithm is to maximize the minimum interior angle of the triangulations. Because the Delaunay triangulation maximizes the minimum interior angle, any other triangulation algorithm will be inherently worse than using the Delaunay triangulations. Other triangulation algorithms follow the same basic path into 3D as the Delauney triangulation.

## 2.6 Hertel-Mehlhorn

Navigation Mesh construction via the Hertel-Mehlhorn [36] algorithm is commonly used to generate a listing of convex walkable areas [63] and for a time was considered to be near optimal for this purpose. This algorithm works by connecting all of the vertices of the world geometry that border on the walkable areas into a series of triangles. The algorithm can also consume as input a listing of triangles generated via some other triangulation method. Triangles have the inherent property of always being convex, which means we have already generated our delineation of the walkable space at this point. However, the contribution of the Hertel-Mehlhorn algorithm is optimize this listing by combining triangles into higher order polygons. The algorithm calls for the removal of an edge from a pair of adjacent triangles such that the resulting shape remains convex. The removal of lines is then repeated until the algorithm is unable to find any acceptable lines to remove. Unfortunately, this algorithm causes certain problems for information compartmentalization. The corners of the world geometry are almost always filled with slivers of thin triangles (even after combining). Since these triangles are thin they are generally smaller than the objects you are trying to place into them. This means that objects will span across multiple regions and prevent agents from fully taking advantage of information compartmentalization with decompositions created from this algorithm.

A Hertel-Mehlhorn decomposition is shown for our sample level in Figure 2.3. We used the Delaunay triangulation previously discussed as the input into the Hertel-

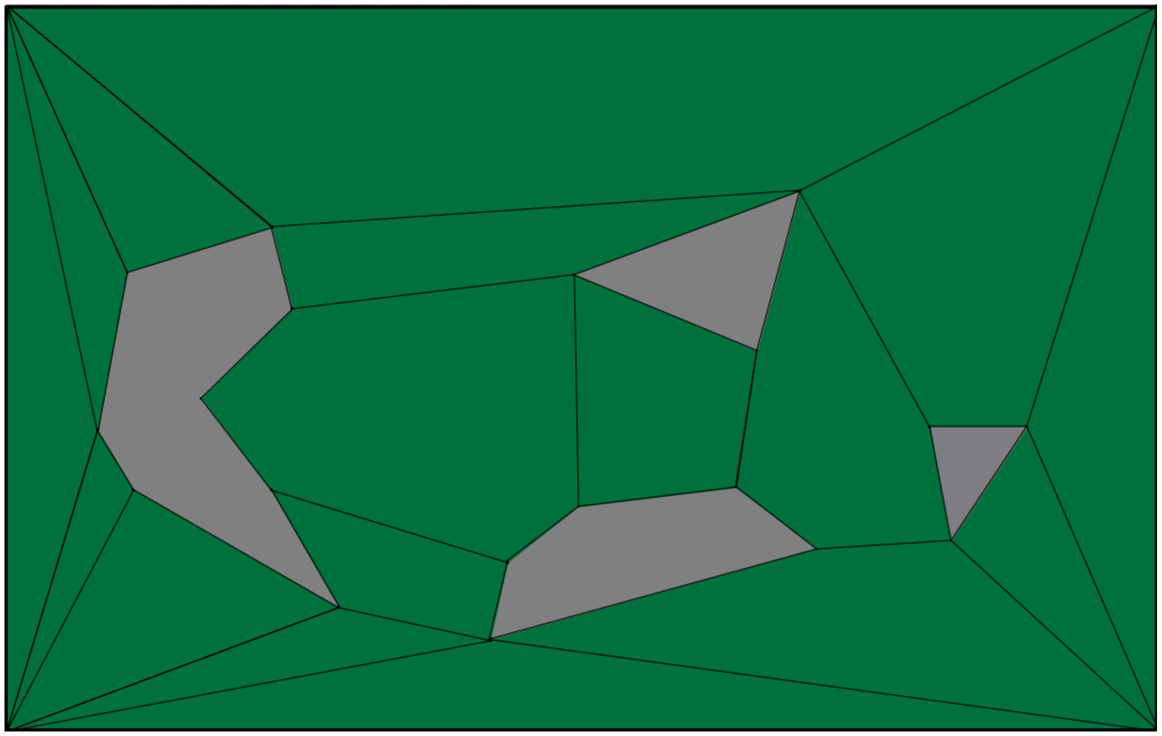


Figure 2.3: A Hertel-Mehlhorn decomposition of our sample level.

Mehlhorn algorithm instead of generating a new triangulation. This results in a simpler and cleaner navigation mesh than the original Delaunay triangulation, which is to be expected since it has fewer regions. In addition, by reducing the number of thin triangles in the mesh, there are far fewer total regions present than in the original triangulation. It is also worth noting that the perfect coverage of the level provided by the Delaunay triangulation is still maintained. These results are typical for these two decomposition methods and show that the Hertel-Mehlhorn algorithm produces a better decomposition than the Delaunay triangulation alone. However, there are still several problem areas where several triangles come together at a single vertex. We will show that these areas will always be a part of any triangulation-based navigation mesh in Proof 7.2 in Section 7.3.

A Hertel-Mehlhorn decomposition solution exists in 3D as long as we can acquire a 3D triangulation of the environment to use as an input into the algorithm. We already know that the Delaunay triangulation also exists in 3D so this is not a problem. Starting from a 3D Delaunay triangulation the 3D Hertel-Mehlhorn algorithm iteratively attempts to remove a face present in the triangulation such that the resulting shape formed from the combination of the two figures which shared the removed face remains convex. This iterative removal of faces continues until there are no faces that can be removed without generating a concave region at which point the algorithm terminates. The 3D navigation meshes generated through the use of this algorithm share all of the advantages and disadvantages of the 2D version.

## 2.7 Render Generate

Recently, work has been conducted to create 3D navigation meshes using a rendering-based approach called Render-Generate [2]. This approach works by iteratively rendering depth maps of the world and using these maps to calculate the locations of the floors and ceilings along with the positions of any obstacles. Using the slopes and obstructions present in these depth maps it is possible to detect and delineate areas the agent can stand in. By connecting adjacent standable areas a walkability map can be generated. However, decompositions generated by this algorithm are limited to constant cell sizes, usually the size of the agent that navigates the world (so that the agent can stand in every cell), and no simplification is done on the resulting graph. This tends to produce meshes in which relatively small areas have a large number of regions. There is no pure 2D version of render generate as the algorithm requires height data as input to create the navigation mesh. Unfortunately, this algorithm is currently encumbered with a patent [3] which prevents it free use and implementation in commercial software releases.

## 2.8 Space Filling Volumes

Space Filling Volumes [64] (SFV) is a growth-based algorithm for generating a navigation mesh and formed the basis of our own algorithm. The Space Filling Volumes algorithm works by placing unit-square seeds throughout the environment and then iteratively growing each seed outward in the direction of the normal of each edge of the square. In the case of a collision with the world geometry all growth in the direction of the edge which collided ceases. As a post-processing enhancement generated regions can be combined if the resulting shape would still be convex. This helps to simplify the resulting navigation map. This technique works well for worlds where all of the geometry is axis-aligned, but fails on worlds with arbitrary or complex geometry.

The results of a Space Filling Volumes decomposition are shown in Figure 2.4. As this figure makes clear, the Space Filling Volumes algorithm does not provide a high coverage decomposition due to the presence of gaps and otherwise inaccessible areas inside the decomposition. For these reasons the Space Filling Volumes algorithm is rarely used to generate navigation meshes in commercial applications. However, our algorithm is a derivative of Space Filling Volumes and incorporates all of the base features of SFV.

The 3D version of Space Filling Volumes is also easy to conceptualize and imple-

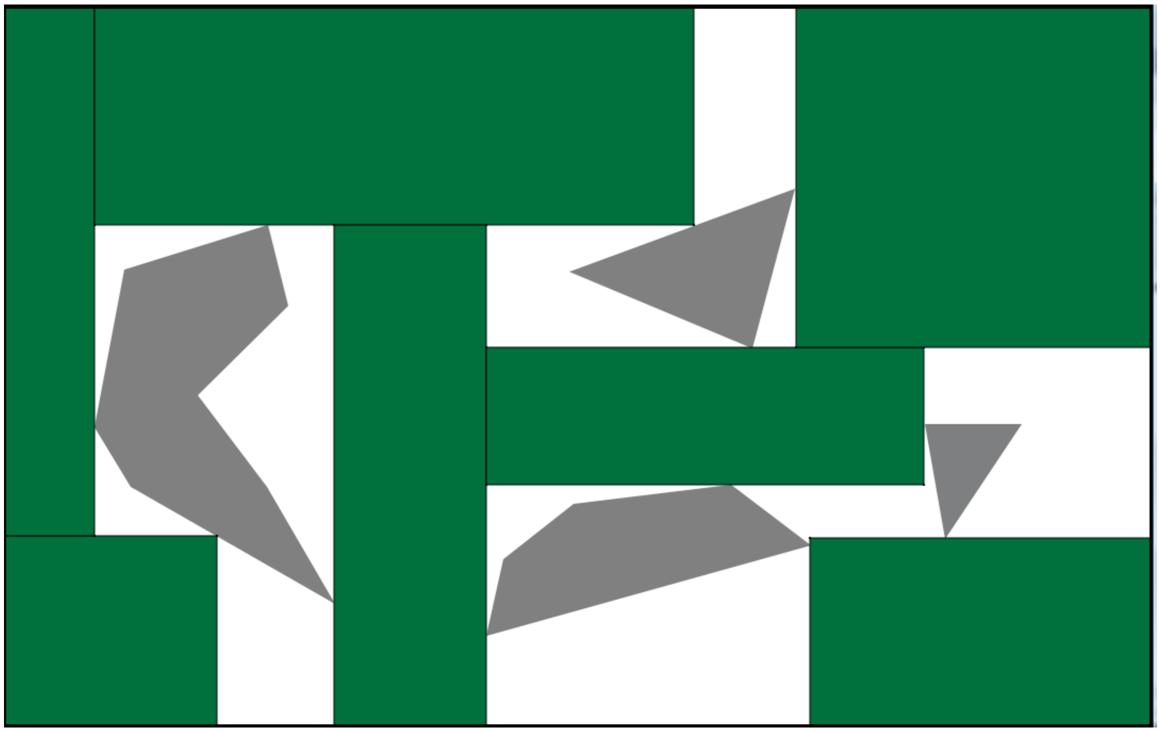


Figure 2.4: Space Filling Volume decomposition of the sample level.

ment. Instead of expanding a square or rectangle in 2D the algorithm also expands the top and bottom of each region to form a growing structure in 3D (cube or rectangular prism). The 3D version of the Space Filling Volumes algorithm results in the same type of spotty navigation mesh as 2D that tends to contain gaps and otherwise impassible holes in the mesh. In fact, since there is an additional variable added to the seeding of initial regions ($z$-axis) and the space to be represented is generally more complicated, the problems of gaps in the navigation mesh are generally worse than in 2D.

## 2.9 Automatic Path Node Generation

Automatic Path Node Generation [53] is a purely 3D algorithm for navigation mesh generation. In this algorithm, the world is tessellated into a series of triangles. This list of triangles is culled down to just the triangles that a character in the game world could stand upon. At this point the algorithm finds the centeroids of each triangle. These centeriods are transformed into rectangles by following simple space filling volume rules we presented in the previous section. These new rectangles are checked for collisions with world geometry and any invalid rectangles are discarded. Next, the algorithm attempts to calculate paths between these rectangles by trying to walk a character through the game geometry and seeing which rectangles are accessible to each other as shown in Figure 2.5. This walk is governed by the game's physics and collision detection system to determine which paths are legal, which means that the navigation mesh generated using this algorithm is only as accurate as the simulation engine in the game. This information is used to build the final connectivity graph, which creates a navigation mesh (or a series of connected disjoint meshes). This approach works well for agents that just walk from point A to B, but does not inherently handle cases where the agent can move via methods other than walking such as jumping or climbing. This is one of the few algorithms that was initially developed in 3D and 2D implementations are just simplifications of the initial work.

## 2.10 Manual Spatial Decomposition

By-hand decomposition of virtual worlds is still used in many areas of game and simulation design and programming. Such hand decompositions are generally done following a heuristic (e.g., draw lines between groups of three or more vertices to create as large of convex regions as possible) to determine precisely how the decomposition

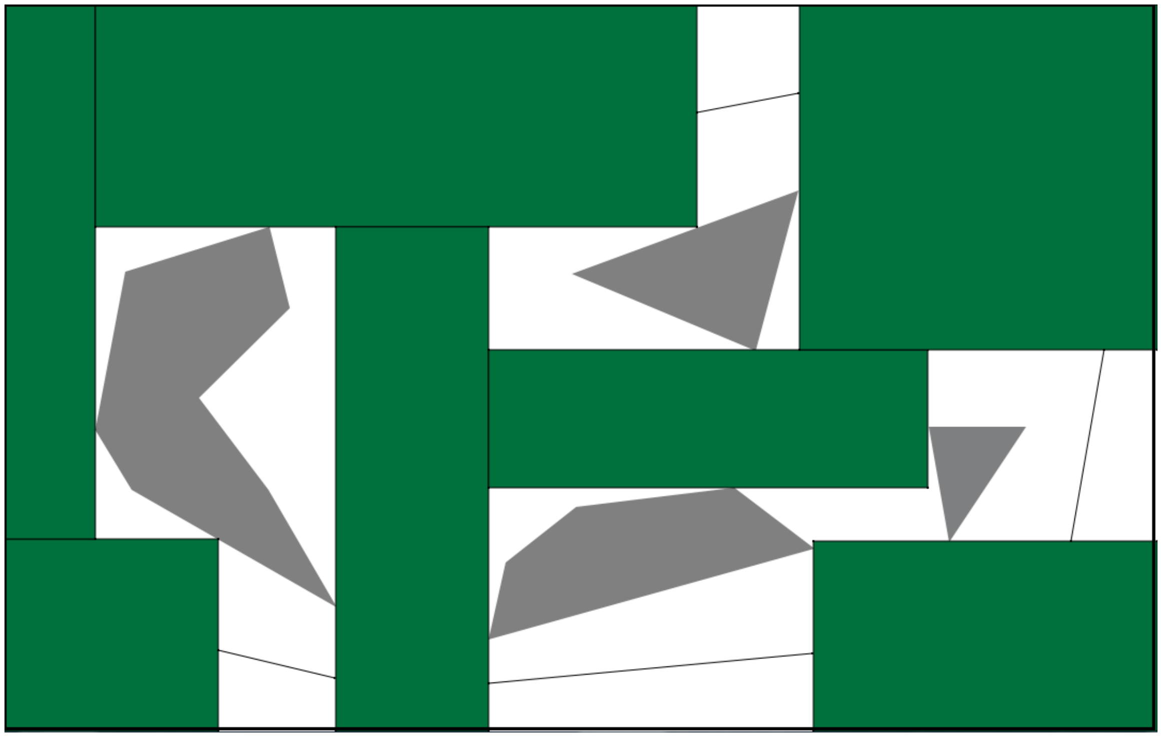


Figure 2.5: Automatic Path Node generation for our sample level.

should occur, which results in decompositions similar to Hertel-Mehlhorn. These hand decompositions do provide excellent representations of world space and have both high coverage and good navigation potential. However, this method's biggest drawback is the extreme time requirement (several days per environment) to properly construct a hand decomposition and as such a better method is needed. Manual decompositions are generally constructed in 2D to simplify the problem of building the decomposition since a human is doing assembly instead of a computer. The 2D navigation mesh is then extruded to form 3D structures if a 3D navigation mesh is required.

## 2.11 Augmented Manual Spatial Decomposition

Half Life 2 and the related Source engine games sit in an interesting middle ground between hand generating navigation meshes and automated navigation mesh generation systems. In Source engine games like it the navigation mesh is created by a level designer in real time using a rendered view of the environment to place the convex regions of the navigation mesh. After creating a given game level the designer will load that level in a special editing mode which assists in the placement of the convex regions. This arrangement is generally effective at covering the most commonly traversed areas of a level, but does not provide any assurances of complete coverage and relies far too much on the skills level of the person doing the decomposition. Augmentation of manual decomposition does allow for the generation of native 3D decompositions as the user is able to specify $z$-axis locations in the same manner as the $x$-axis or $y$-axis locations. In addition, this system is generally faster than a purely by hand decomposition since the person doing the decomposition does have some placement and recording aides in the computer.

## 2.12 Volume Sweep Techniques

The developers of the game Left4Dead (built off the Source engine) felt that the augmented manual approach which we just discussed, for the generation of navigation meshes, left something to be desired. Instead of requiring manual decompositions they produced an interesting algorithm for automatically decomposing a level using a sweeping algorithm. In this algorithm the level designer places an axis-aligned-bounding box which represents the player into a ”known good” location in the level. Then this agent sized bounding box is shifted in one box increments in every direction. So for example the box is moved to the north. If this location is a valid one for the box (determined by running a collision test with the geometry) then this new location is added to the navigation mesh as a valid point for the agent to stand. Once a location is established as valid, all neighboring locations are then added to list of locations to test. In this manner the entirety of the level will eventually be tested and any valid locations where the agent could stand will be discovered. Once this portion of the algorithm terminates the resulting list of boxes are combined where possible such that two boxes are combined into one region, which can then be combined again as long as the resulting shape is still convex. This will eventually result in a navigation mesh for any given level geometry—though it might take some time for larger or more complex levels. This technique is often extended into 3D by projecting all 3 dimensions of the agent's bounding box into the world and then sweeping up and down as well as the more traditional forward, back, left, and right. When this technique is extended to 3D it often referred to as voxelization.

This algorithm produces a result that is outwardly similar to the Space Filling Volumes algorithm but which has a higher degree of coverage. Also, the shapes used in the decomposition are more uniform and provide a more normalized navigation mesh. The primary advantage of this algorithm over Space Filling Volumes is that it does not produce large gaps or holes in the decomposition. Any un-decomposed space will be distributed evenly around the edges of the navigation mesh.

## 2.13 Approximate Cell Decomposition

Approximate Cell Decomposition (a.k.a., Proximate Cell Decomposition) is originally a robotic navigation and path planning technique for moving through known environments [39, 8]. This algorithm is similar to several other algorithms we have discussied bearing a resemblance to both Volume Sweep techniques and Binary Space partitioning trees 2.17. Generating an Approximate Cell Decomposition uses a recursive spatial subdividing algorithm. The world is initially split into four equal quads. Each quad is then checked to see if it contains any configuration space obstructions. If the quad is empty then the algorithm terminates for that quad. Otherwise, if there is an obstruction the quad is further subdivided into a smaller set of 4 quads. This new set of quads is then checked for obstructions and if necessary subdivided further. The algorithm continues to break quads containing obstructions down into smaller units until a minimum size threshold is reached at which point a quad will not be subdivided further and that quad will be marked as impassable and discarded. The resulting collection of passable (empty) and impassible quads are then used to construct the navigation mesh. Adjacent quads can be connected to show passability between them and the standard graph representation of a navigation mesh can be established using this technique. The resulting navigation mesh is somewhat similar to one generated using a spatial partitioning tree, except the representation is a planar graph instead of a tree structure, which is easier to search since adjacency information is provided. The primary problem for this algorithms is that it does not deal well with non-axis-aligned geometry and tends to fill diagonal areas with lots of little boxes. In addition, since there is no combining stage in this algorithm there will be a larger number of nodes present in the navigation mesh than algorithm such as Hertel-Mehlhorn or even space filling volumes. Finally, there will also be small gaps of un-decomposed space present near non-axis-aligned geometry where the minimum size threshold for further subdivision was reached before properly describing the occupied space and quads were marked as impassible that contained both obstructions and walk able areas. These impassible areas do not have a large effect on agent navigation since they are generally near walls and do not cause gaps in the navigation mesh, but it would still be nice to have a perfect coverage decomposition.

This algorithm can also be extended into 3D environments. Instead of subdividing the world into four equal quads the 3D version of Approximate Cell Decomposition generates a set of eight cubes per subdivision. Aside from this, the algorithm proceeds in the same manor in 3D as in 2D with the recursive subdivision of each cube into smaller sets of cubes until the obstructing configuration space objects are contained in a single cell or until the cell size hits the minimum threshold for splitting into further smaller objects. The resulting 3D navigation mesh looks remarkably similar to the 2D version even including the small fringes of non-decomposed space against any non-axis-aligned geometry that might be present in the world.

## 2.14 Vertical Cell Decomposition

The vertical (or trapezoidal decomposition) splits the free space present in a level into a series of vertical or horizontally aligned polygons [9, 42]. It does this through the use of a planar sweep algorithm [18, 7, 12] to identify the event points (i.e., the vertices of obstructed configuration space) in a certain order (from lowest $x$ coordinate to highest). At each event point this algorithm either adds each of the points it finds to a list of points it will create polygons from later or it pops points off the list to create a convex trapezoid (note the trapezoid can be degenerate and therefore a triangle). Due to the manner in which the plane sweep algorithm traverses the world (it is parallel to the y axis) all of the vertical free space edges of the trapezoids are guaranteed to be parallel to each other and the $y$-axis. This property of vertical edges is only true for edges that are adjacent to other free space regions—it is usually not true for edges that are adjacent to obstructions. This algorithm can be extended to 3D at which point it becomes a cylindrical decomposition and the planar sweep algorithm becomes recursive. Unfortunately, this algorithm does not generate a minimal set of regions as most of the triangles it produces could be combined with other adjacent shapes as shown in Figure 2.6.

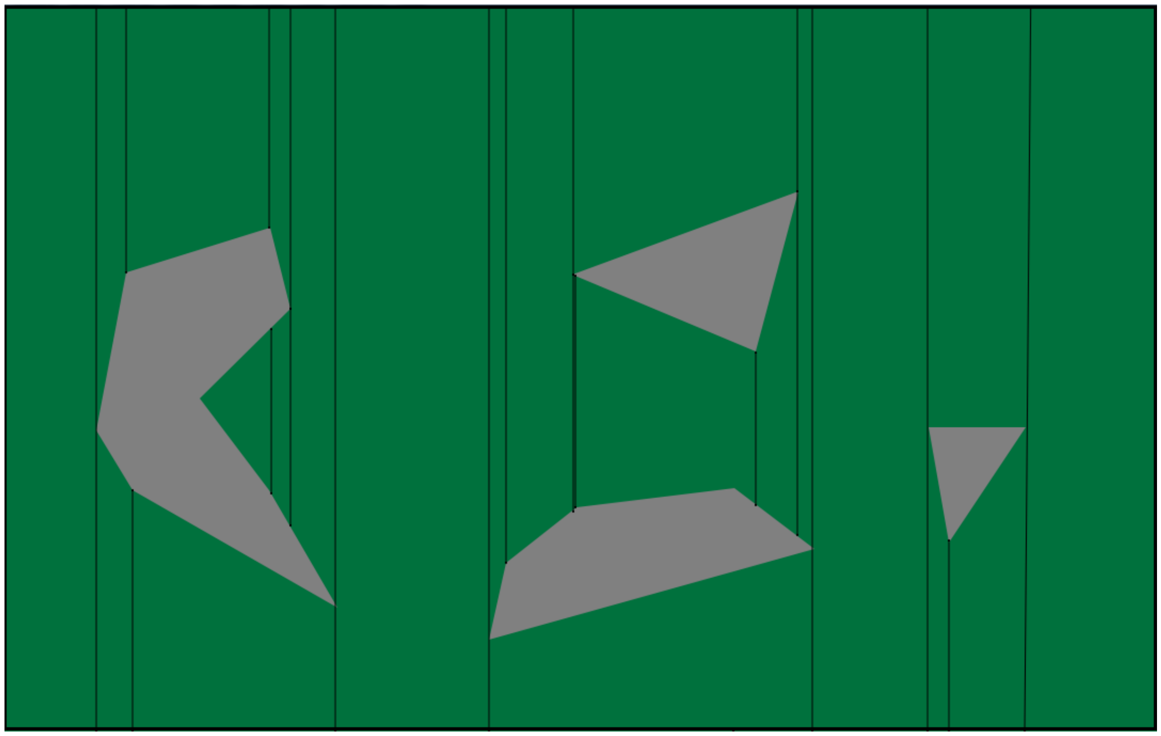


Figure 2.6: A Vertical Cellular Decomposition of our sample level.

## 2.15 NavSphere

The Navigation Sphere (navsphere) system for generating a navigation mesh is a augmented manual generation approach used commercially to generate low overhead navigation meshes [52, 51]. In this approach the level designer places a series of overlapping spheres into the game or simulation environment. Connectivity between multiple navigation sphere regions is established when two spheres overlap. In addition, it is possible for the author(s) of the level to manual establish links between two separate navigation spheres by the use of “navfeelers”, which are known good paths between two separate navigation spheres. In this manner, a level designer can establish a navigation mesh that suits their level using the minimal number of regions on the navigation mesh to achieve the desired coverage level. The navsphere system is specifically designed to provide a minimal overhead system of navigation mesh representation. This system is low overhead because each sphere present in the system can be represented using just an ID number, a radius, and a center point. The “navfeelers” used by the algorithm are represented as origin and destination sphere ID numbers and the series of control points that define the known good path. This system is designed primarily for lower power embedded or home entertainment systems, which do not have a large memory or cpu budget. However, it is handicapped by the fact that the initial generation of the navigation spheres in the world requires manual placement. The navsphere algorithm is a native 3D algorithm as originally conceived; However, it would be trivial to convert it to a 2D algorithm by dropping the $z$ coordinate off all of the data structures and just using circles instead of spheres.

A navsphere decomposition looks similar to the automatic pathnode generation decomposition, but it is manually generated and instead of rectangular regions it uses spherical regions.

## 2.16 Recast Navigation

The Recast method for generating navigation meshes focuses on the use of voxelization and then combination to generate a proper navigation mesh. Initially, Recast calls for the complete breakdown on the environment in a series of voxels, which fully represent the available and walkable areas of free space [20, 16]. A voxel is in this algorithm a well defined and self contained 3D area of space (not a 3D pixel as is commonly used in graphics). At this initial point, the algorithm does not care about the complexity or convexity of the voxels present in the world although that will eventually change. After the voxelization completes then the voxels are converted into simple 2D regions. Each region exists as a bounded plane somewhere in the world. Note that these planes while 2D do not necessary lay flat parallel to the $xy$-planes and that they can in fact be of any orientation in the world [41]. Finally, after generating these bounded planar regions they are subdivided into walkable convex regions and the gateways between each region are recorded to create the navigation mesh [68]. This algorithm is a native 3D algorithm and there are no pure 2D implementations. It is important to note that this separation has a minimum size threshold, which can result in the discarding of small areas of walkable space from around the edges of obstructions if including those fringes would require the addition of new regions that are below the minimum size threshold.

This algorithm results in the creation of high coverage navigation meshes with only small areas of walkable space remaining un-decomposed. In addition, the implementation for this algorithm is available for free at http://code.google.com/p/

recastnavigation/. However, it does still have the problem of producing highly triangulated final decompositions since the algorithm used to produce convex shapes from the mid-stage planar regions is a triangulation algorithm.

## 2.17 Binary Space Partitioning Trees

The binary space partitioning tree is a spatial data structure developed originally by Fuchs [24, 23] to provide a spatial ordering of objects present in at 3D environment. The algorithm works by constructing a binary tree to recursively subdivide all of the space present in the world without distinguishing between free and obstructed configuration space though the use of halfplane divisors. These halfplanes are created by selecting a face of a configuration space obstacle such that it maximizes the number of objects in each group (i.e., if there are fire objects present in the world and the choice is between dividing them into groupings of one and four or three and two then it will select the three and two grouping). If the choice of a splitting plane would bisect another configuration space object then that object is subdivided into two new objects. Once this algorithm concludes, the leaf nodes present on the BSP-tree are completely composed of free configuration space regions or completely full obstructed configuration space regions. These regions can then be used for navigation and planning by traversing up and down the tree to locate non-obstructed paths between two arbitrarily selected points as shown by Tokuta [62]. The primary problem with constructing navigation meshes in this manor is that the connectivity between regions is not easy to establish. For example, two adjacent open space regions might lie on entirely different branches of the tree and require a traverse all the way to the root to establish a connection between them.

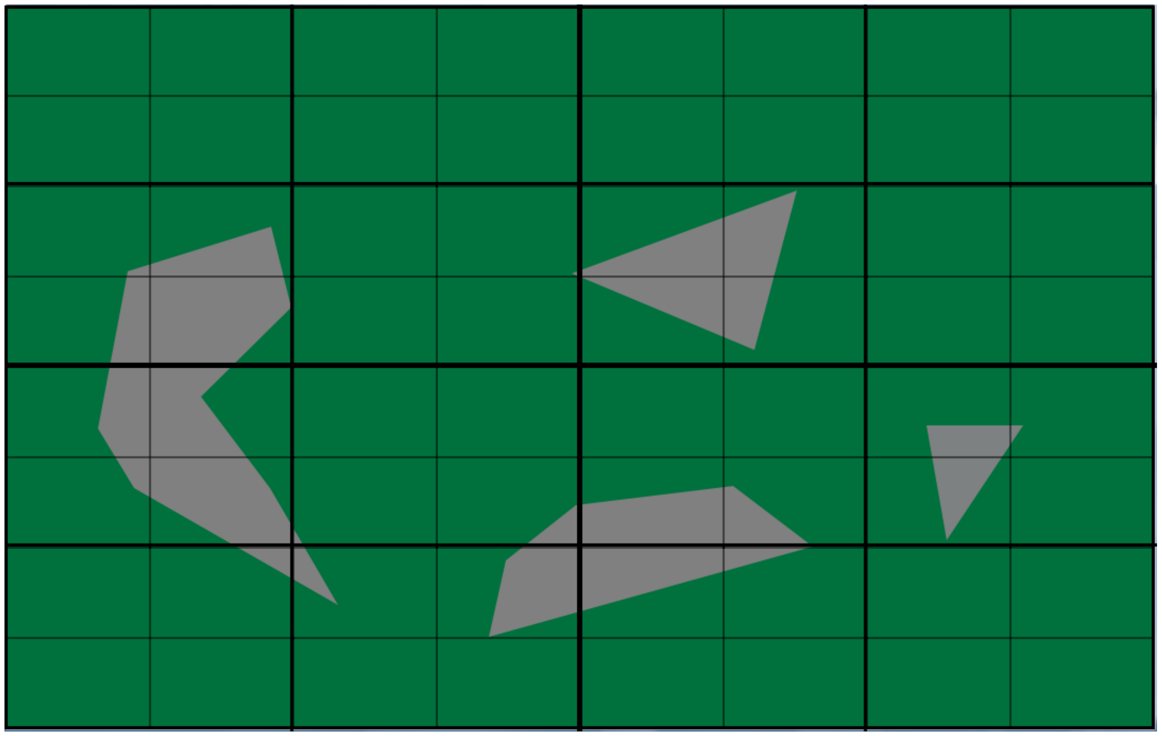


Figure 2.7: The Quad Tree decomposition of the sample level. Higher level nodes of the tree are shown as darker lines.

## 2.18 Quad / Oct Trees

The Quad-trees and Oct-trees are specialized forms of Binary Space Partitioning trees [1, 55]. The Quad tree functions by splitting the world into well-defined partitions; however, unlike a BSP tree the Quad tree is an order four tree and the splitting planes are locked to be axis-aligned. The Oct tree functions the same way in that it is limited to axis-aligned splitting planes however it splits in three dimensions and each node has eight children areas [56, 57, 58]. Like BSP trees these methods can be used for agent navigation since they do provide a full mapping of the free and obstructed configuration space, but they also experience the same problems due to the tree structure and optimizations of the data structure instead of a flat graph that clearly specifies which regions are adjacent to each other. This spatial subdivision technique is shown in Figure 2.7. This algorithm is less efficient than the normal graph representation most navigation mesh generation algorithms produce and as such it functions primarily as a culling and graphical algorithm [61, 19].

## 2.19 kd-Trees

The kd-tree is another highly specialized form of the BSP tree [1, 57, 58]. The kd-tree is a multiple dimensional data structure initially proposed by Bentley [5] to quickly sort and classify two or higher dimensional space. Like the Quad/Oct tree the halfplanes used to classify and subdivide the world are also required to be axis-aligned in the kd-tree. However, unlike the Quad/Oct tree the order of the insertion of the splitting axis is defined in advance and must be rigorously followed. For example, a given kd-tree with a dimensionality of three might split on the y-axis, the z-axis, and then the x-axis. It would then continue to split in this order until the world is fully classified. As expected, this classification system will fully decompose the world and generally provides a balanced tree of obstructed and free configuration space. It can be used as a navigation mesh in much the same manner as the other tree based data structures.

## 2.20 Watershed Algorithm

The watershed transform [47, 25] is a image processing algorithm that can be used to subdivide a given set of configuration space obstructions into a series of regions and listing of the gateways between regions. It is important to note that the regions generated by this algorithm are not guaranteed to be convex and as a general rule are in fact concave. The algorithm comes in both 2D and 3D versions. We will examine the 2D version first. In a 2D watershed transformation the distance between each point of free configuration space and the nearest configuration space obstruction is calculated. This results in the Voronoi diagram of the freespace present in the environment. Then this distance value is negated and used as a height value to define a sloping surface inside the free space areas. In effect, each room of the decomposition becomes a pit or pool shape. These pits are referred to as “catchment basins” in the algorithm. Next the algorithm simulates filling these basins with water by creating a small hole at the bottom of each basin and evenly sinking the entire world in a larger body of water. As each basin floods there will be line of water climbing the sides of the basin which represent a sweep line moving through the level. Whenever a local maxima between to catch bases (a saddle shape) is detected using this sweep line it is marked as a portal between two regions. In this manner the algorithm will fully define each catchment basin as a region of the decomposition and the local maximums as the portals between each region.

The 3D version of this transform [67] is a straightforward addition of another parameter—the calculation of the distance equation. Instead of determining the distance of the closest obstruction in the $xy$ plane (assuming $z$ is the up vector) the 3D watershed algorithm determines the distance to the nearest obstruction in real space. This does result in a order of magniture more difficult decomposition can approach $O(n^2)$ runtimes. The determination of catchments and saddle points is the same.

While this algorithm is effective and does guarantee a complete coverage of the area to be decomposed with a minimal number of regions, it suffers due to the lack of convexity of the regions it generates. This means that many of the best (fastest) algorithms for localizing objects on the navigation mesh would be unavailable since they all require that the navigation mesh contain nothing but convex shapes.

## 2.21 Spatial Hashing

In addition to the tree based data-structures we discussed earlier, spatial hashing is also commonly used to accelerate collision detection [33]. In a spatial hashing algorithm every point in world space maps into a hash-table. However, each point does not map to its own unique location in the hash-table. Instead, a square or cubical section of points in world space of user definable size all map to the same location in the hash-table. Objects that are within this space are all considered to be inside the same bucket of the hash table (unlike a normal hash-map more than one object has to be stored per hash location). Furthermore, large objects that overlap the borders of one or more hash regions are considered to be in all of the hash regions they touch. Determining which hash region an object falls into is a simple matter of taking the $x$-coordinate of the object (or the corners of the object's bounding box) dividing it by the size of the cell and then adding that result to the product of the $y$-coordinate divided by cell size times the width of the world (as shown in Equation 2.1). Once all of the objects have been placed into a bucket (at a cost of O($n$) for n objects) determining collisions becomes a matter of iterating over the map and resolving collisions between all objects in a single bucket, since only objects within the same bucket are close enough to be colliding. Interestingly, there have been some efforts to assist with AI and character planning using spatial hashing. This work has focused on taking advantage of the knowledge of how regions border one another to provide proximity information on objects to AI characters as they move through the world to allow them make more realistic decisions based on available information [34]. Additionally, this work also presented the concept of using the spatial hash grid to store information about general world conditions which might be of interest to the agent, although later work has shown that such information is more efficiently stored in a navigation mesh [35].

$$f(region) = hash(\frac{x}{cellsize}, \frac{y}{cellsize}) \tag{2.1}$$

# CHAPTER 3: PLANAR ADAPTIVE SPACE FILLING VOLUMES

Our approach for the generation of navigation meshes via spatial decomposition is based off of a simple physical event. The regions we place in the world resemble marshmallows that have been placed in the microwave. They expand dramatically to fill the available free configuration space and then follow the contours of any obstructed configuration space they encounter. This form of expansion allows us to establish high degrees of coverage even in complex worlds.

## 3.1 PASFV Algorithm

The Planar Adaptive Space Filling Volumes PASFV algorithm, as shown in Algorithm 3.1 can be broken down into several simple steps. We have several assumptions and invariants we must maintain in order for our algorithm to be effective. First, we assume that all of the obstructed space regions provided as input are convex. Our own generated regions must end every phase of growth in a convex state. Finally, once a free area has been claimed by a region, then that region must maintain its ownership of that area.

Our algorithm begins in a state that we refer to as the initial seeding state by planting a grid based pattern of single unit regions across the environment to be decomposed. If the proposed location of a region is contained within an obstructed space area it is discarded. Our regions are initially spawned as squares with 4 sides given in a counterclockwise order from the northwestern point when viewed from above. These squares are generated to be axis-aligned. After being seeded into the world each region is iteratively provided the chance to grow. There are two possible cases for successful growth. The simple case occurs when all obstructed space (impassable) regions are convex and axis-aligned. The more advanced growth case allows for non-axis-aligned convex obstructed space areas. Worlds which violate our assumption that all of the obstructed space input regions be convex are not handled by our algorithm at this time and will not be evaluated. However, by subdividing non-convex geometry into convex shapes it is possible to convert any world into something our algorithm can process.

First, we shall examine the *base case* for growth in the PASFV algorithm. Each region is selected and provided the opportunity to grow once each frame. Growth occurs in the direction of the normal to each edge in the region. We attempt to move the entire edge a single world unit in this direction. We then take our proposed new shape and run three collision detection tests. We want to ensure that no points from our growing shape have intruded into another obstructed space or another region and that no points from either of the aforementioned obstructions would be contained within our newly expanded shape. Finally, the region performs a self check to ensure it is still convex in its new configuration. Given that all those tests return acceptable results, we will allow the shape to finalize itself into that new configuration. If any of those results are unacceptable then it means that we had a collision or become concave. Because of the axis-aligned properties in this state we know that we were parallel to, as well as adjacent to, the shape we have collided with in our prior extents, which is the desired ending condition for region growth. In this case we return to our previous shape and set a flag to never attempt to grow this region in that direction again. We then allow every other edge in the shape to grow in the same manner. Once each edge in a shape has been provided the chance to grow a single unit, we proceed to the next shape. This method of growth is sufficient to deal with all cases for axis-aligned obstructions.

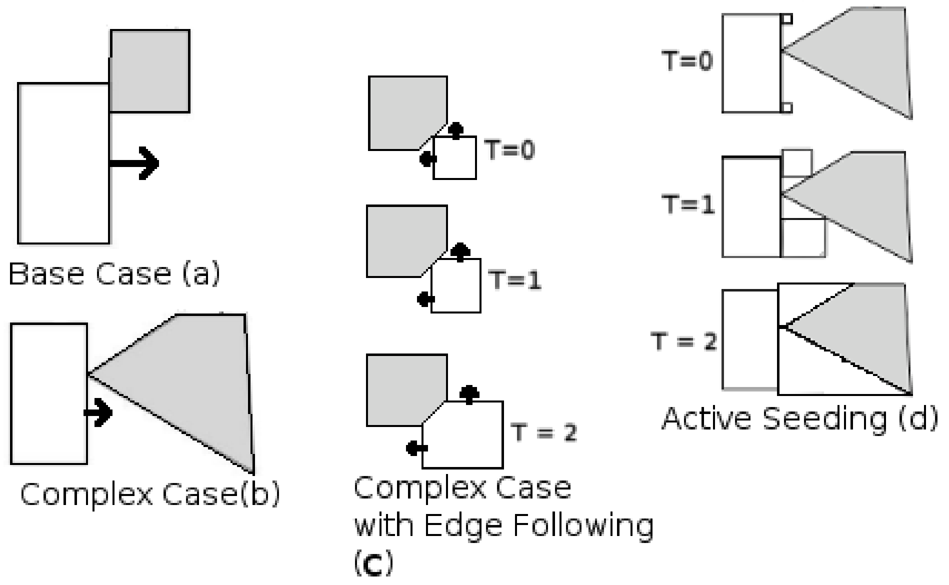


Figure 3.1: The various cases present in PASFV. All growing free configuration space regions are shown in white. Primary direction of growth is shown with an arrow. (a) shows the basic growth case. (b) shows the complex case where growth is stopped by encountering an edge. (c) shows the complex case where the free configuration space region enters contour following mode. (d) shows an example of seeding to generate new free configuration space regions.

The advanced case algorithm, as shown in Algorithm 3.1 and Algorithm 2, is more complicated, but it is also able to deal with non-axis-aligned obstructed configuration space areas. It begins by incorporating everything contained in the base case algorithm and then expanding on it. Again we cycle through each region and provide each edge in that region a chance to grow. This time, however, since we cannot assume that we will automatically be parallel to what we have collided with we need to take an additional step and ensure that we follow the contour of the region we have collided with if possible. We have three basic collision cases to consider.

**Algorithm 3.1:** PASFV ALGORITHM

```
void startDEACCON(List NegativeSpaceRegions) StillGrowing = true ;
/* Perform Initial Seeding to populate world with regions based on
   user settings */
if NegativeSpaceRegions.isEmpty() then
    seedWorld();
while StillGrowing do
    StillGrowing = false ;
    for NegativeSpaceRegions in World do
        NegativeSpaceRegion.grow();
/* List of points that will have seeds placed in them */
List seedPoints = new List();
for Regions in World do
    /* Run Find Adjacent Open Space algorithm */
    seedPoints.append(Region.findOpenSpace());
/* This method will add new regions to the world */
World.addSeeds(seedPoints);
if seedPoints.isNotEmpty() then
    /* Re-start growth algorithm */
    World.startDEACCON(seedPoints);
/* Run combining and cleanup algorithms */
World.combineConvexShapes();
World.removeColinearPoints();
World.removeDeginerateEdges();
```

The first is the basic parallel line case as shown in Figure 3.1(a). We can test this by comparing the equations of the edge we are advancing and the edge with which we have collided. If we enter this case, we proceed in exactly the same manner as in the case presented above since we have, in effect, shown that locally we are in the above *base case*. The next case occurs when a point from the object we are colliding with would lie within the newly proposed bounds of our region. In this case we must stop further growth in this direction in order to preserve the convex property. In the case of this collision we are unfortunately forced to accept a poor proximity to the edges of the obstructed configuration space region as shown in Figure 3.1(b). We deal with this later by seeding.

**Algorithm 3.2:** PASFV Algorithm - growthMethod()

```
for Edges in this.Edges do
    /* Get a new Edge one unit forward in the direction of the old
       edge's Normal */
    newEdge = currentEdge.Advance();
    if newEdge.isNotColliding() then
        StillGrowing = true;
        currentEdge = newEdge;
    else
        /* We are adjacent to a obstructed configuration space region
           */
        /* Check to see if we are in advanced case */
        if usingContourFollowing then
            if newEdge.isSplitableCollision() then
                /* Adapt to follow Edge */
                /* Determine which Vertex collided */
                /* Add in extra point and edge */
                newEdge.splitPoint();
                /* Overide direction of growth to the */
                /* equation of the edge of the object */
                /* it collided with */
                newEdge.setGrowthDirection()
            else
                /* Not possible to grow in this direction */
                currentEdge.canGrow = false;
```

The final collision case is the most interesting and occurs when one of the endpoints of the region edge would lie within an obstructed configuration space area. We are able to split the vertex of our region into two points. These points will follow along the contour of the obstacle, expanding the region and increasing the order of the polygon in the process. We accomplish this by inserting a new edge into our polygon of length zero at the point of collision. We then alter the direction of growth of each of the points we inserted such that it is following the equation of the line with which it collided. We cap growth at the extent of the edge we are following so that we do not create additional non-axis-aligned exposed edges to deal with later, which would be the case if we allowed modified edge growth to proceed past the edge it is following. Once we have overwritten the direction of growth for the contact points and limited their extent, we can return to following the *base case* and the region will grow to follow the obstructions as shown in Figure 3.1(c).

By following this method we are able to fit a region to non axis-aligned shapes such that the boundaries with other regions will be axis-aligned. Once every region reports that it is unable to continue growth we can proceed to the next step of the algorithm.

The above growth methods alone do not ensure a complete coverage of potential free space, but we employ a second component of the PASFV algorithm as shown in Algorithm 3.3 to improve our results. We call the second component *seeding* and it works by locating areas adjacent to our regions that have not been claim by a region of free space as shown in Figure 3.1(d). We determine where to seed on an edge by determining which parts are in contact with obstructions. Once we locate every obstruction in contact with a given edge we can find each section of free space adjacent to that edge. A seed region is then placed into the midpoint of each of these free space areas. This seeding process will result in a high degree of coverage for the world. Once every edge has had a chance to seed we will re-enter the growth phase if there were any seeds generated. This allows our new regions the chance to fill any empty space and improve our decomposition. This is especially effective in collision cases where we were forced to stop growth due to collisions such as that in Figure 3.1(b). We continue to perform this cycle of grow and seed until we have filled in all reachable free configuration space and we do not place any new seeds.

Finally, we run a cleanup and combining algorithm on our set of regions. We first go through and check for any regions that can be merged into one single convex region. After merging all allowable regions we remove any degenerate zero length edges or colinear points to provide as clean an output as possible.

**Algorithm 3.3:** Locate points to add as seeds in open space

```
List findOpenSpace();
/* Locate possible seeding locations */
for Edges of Region do
    /* Determine all objects that intersect this edge */
    /* Determine midpoints of all free areas between  */
    /* intersections.  Compose a list of these points */
    return Edge.getOpenAdjacent();
```

## 3.2 PASFV Experimental Evaluation

The PASFV algorithm was tested and evaluated using five maps from a popular Quake 3 modification, Urban Terror (www.urbanterror.net). These maps were randomly chosen to cover a wide variety of environment types, ranging from one map that is the interior of a building to another map that shows wide open spaces with general building geometry.

We will be discussing PASFV in reference to one of the environments, called

"Lakes" as shown in Figure 3.2 (A/B - Second from Left). We performed a series of decompositions on this level using three different decomposition methods. The first decomposition was performed with the Hertel-Mehlhorn (HM) algorithm [36]. Figure 3.2 (C) shows the final HM decomposition. This method produced 42 regions after combining on the 2D ground plane.

Next, the fully automated technique of Space-Filling Volumes (SFV) as discussed previously was utilized to decompose the same environment. A sufficiently dense uniform seeding to allow near coverage was used, however it did experience connectivity and coverage issues. These issues are inherent to the SFV algorithm and were not caused by our implementation. The decomposition is shown in Figure 3.2 (D). This method produced 28 regions on the 2D ground plane after running a simplification algorithm that combined regions where possible.

Finally, PASFV was used to decompose the same environment. Figure 3.2 (E) shows the final decomposition. This method produced 73 regions after combining on the 2D ground plane.

After the free configuration space was decomposed, an analysis program was run that builds the navigation map between the centers of each region and the midpoints of common gateways (shared edges between regions). The navigation map for PASFV and the other decomposition methods for the "Lakes" map are presented in Figure 3.2 (F-H).

Fifteen decompositions across 5 different maps were used for agent navigation comparison tests. A simple agent using A* search [32] was used to plan a path from the start location to the goal location using the centroids of the connecting gateways

Table 3.1: Comparative Agent Performance on Decompositions Across Multiple Levels. * indicates statistical significance with p-value of less than .05.

| Algorithm | Avg Distance | Avg Turns | Coverage |
|---|---|---|---|
| HM | 497.4 | 5.2 | 100% |
| SFV | 505.3 | 3.97* | 90% |
| PASFV | 442.7* | 4.12* | 100% |

between regions and the centroids of the regions for navigation. This form of navigation map construction provides a base line for agent path planning. The actual path a character would travel would be constructed from this navigation map using smoothing algorithms to provide a more natural looking path. An equivalence table was constructed to determine which map regions correspond to each other between different decomposition methods. Regions are considered to be equivalent to each other if their centers and extents are roughly the same. Eight paths of approximately the same length between randomly selected regions were then created, and the distances to travel those paths were calculated using the A* agent. This provides us with a total of 40 unique paths for each of the 3 decompositions. These results are then compared across all maps using a repeated measures F-Test design over the three different types of decomposition, which are shown in Table 3.1.

The results show that there is a statistically significant reduction (p-value $< .05$) in overall path distance using PASFV. This is due to the less angular shapes that are produced compared to the more triangular decompositions of HM, and the higher coverage percentage over SFV. The number of sharp (greater than 90 degree) turns an agent has to make to complete each path was also tracked and evaluated. The paths generated by PASFV contained statistically significantly (p-value $< .05$) fewer turns than HM. These turns can cause delays in agent navigation and require additional path planning and consideration in order to compute a natural looking path even with smoothing algorithms. While PASFV did not have fewer turns than SFV, in general SFV is a poor choice with which to build navigation maps. In this example SFV was given a large advantage in that all chosen paths were fully accessible to it. Normally, there will be areas of the map that cannot be reached using SFV. Overall, performance for world traversal by an agent was improved over HM and SFV due to the shorter paths and fewer turns.

The analysis of the runtimes and completeness of PASFV are presented in Section 4.4.

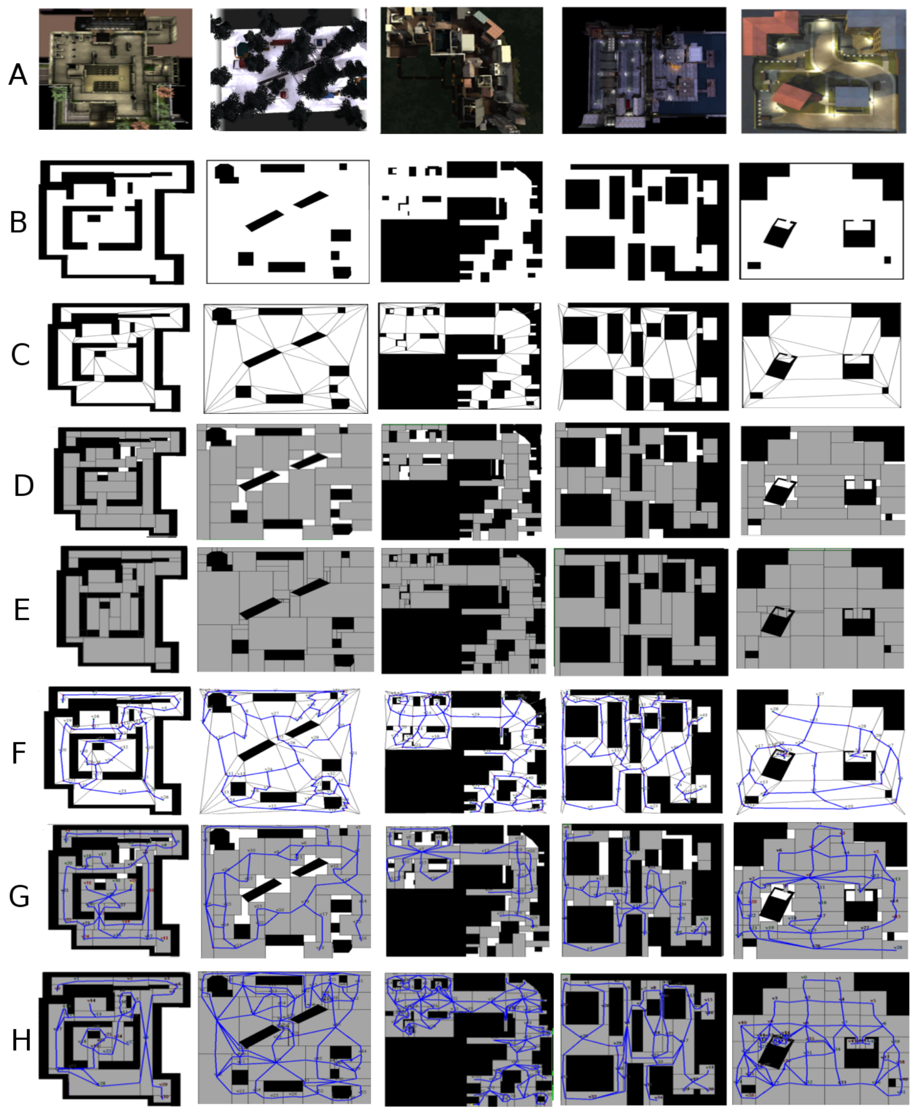


Figure 3.2: Images of the basic worlds (A) with each column representing a separate and unique world, the geometry we decomposed (B), the Hertel-Mehlhorn decomposition (C), Space Filling Volumes (D), PASFV decompositions (E), and finally the last 3 rows show the connected navigation graph generated by each decomposition algorithm (F is Hertel-Mehlhorn, G is Space Filling Volumes, and H is PASFV).

# CHAPTER 4: VOLUMETRIC ADAPTIVE SPACE FILLING VOLUMES

We next present an improved solution for decompositions of 3D environments that does not require the world to be simplified into 2D planes and instead performs a 3D decomposition, which subdivides the open space present in the world into a series of 3D regions. We provide a new method inspired by the Planar Adaptive Space Filling Volumes (PASFV) algorithm to allow spatial decompositions to operate on 3D geometry. Since we are drawing on PASFV for this algorithm many of the positive features that PASFV decompositions contain such as convex high order polygons, high average minimum interior angles across all regions, good object containment, information compartmentalization, very high to perfect coverage of the level geometry, and a low number of total regions are also present in our 3D decompositions. In addition, we can decompose levels with multiple ground planes or complex geometry, which is not represented well in 2D. We accomplish this by transforming PASFV from a 2D algorithm that grows a series of quads into a 3D algorithm that grows a series of cubes. Each cube, like the quads the preceded them, can morph into higher order polyhedrons to better adapt to world geometry. In addition, we altered the manner in which additional regions are added to the world via seeding in order to allow a more natural and usable fit to the affordances provided by the geometry.

## 4.1 VASFV Algorithm

Volumetric Adaptive Space Filling Volumes (VASFV), as shown in Algorithm 4.1 can be explained via several simple steps similar to the ones used in ASFV. There are several input constraints and invariants that must be followed to ensure the success of the decomposition. First, we assume that all of the obstructed configuration space areas provided as input are convex. If the input regions are not natively convex they can be subdivided into convex regions. Secondly, our own generated regions must end every phase of growth in a convex state. Finally, once a free area has been claimed by a region, then that region must maintain its ownership of that area. These are the same constraints present in the 2D version of the PASFV algorithm

VASFV begins in a state that we refer to as the initial seeding state by planting a grid based pattern of single unit regions across the environment to be decomposed. If the proposed location of a region is contained within an obstructed configuration space area it is discarded. This grid extends upward into the $z$ plane as well. After placement, the seeds are allowed to fall in the direction of gravity until they hit either the ground or an obstruction at which point they stop. If this falling results in multiple seeds landing in the same place then duplicates are removed. Our regions are initially spawned as cubes with six faces given in a clockwise order from the closest vertex to the origin point and then the top and bottom faces. These cubes are generated to be axis-aligned. After being seeded into the world each region is iteratively provided the chance to grow. There are two possible cases for successful growth. The simple case occurs when all obstructed configuration space (impassable) areas are axis-aligned. The more advanced growth case allows for non-axis-aligned convex areas to be present among the obstructed configuration space areas.

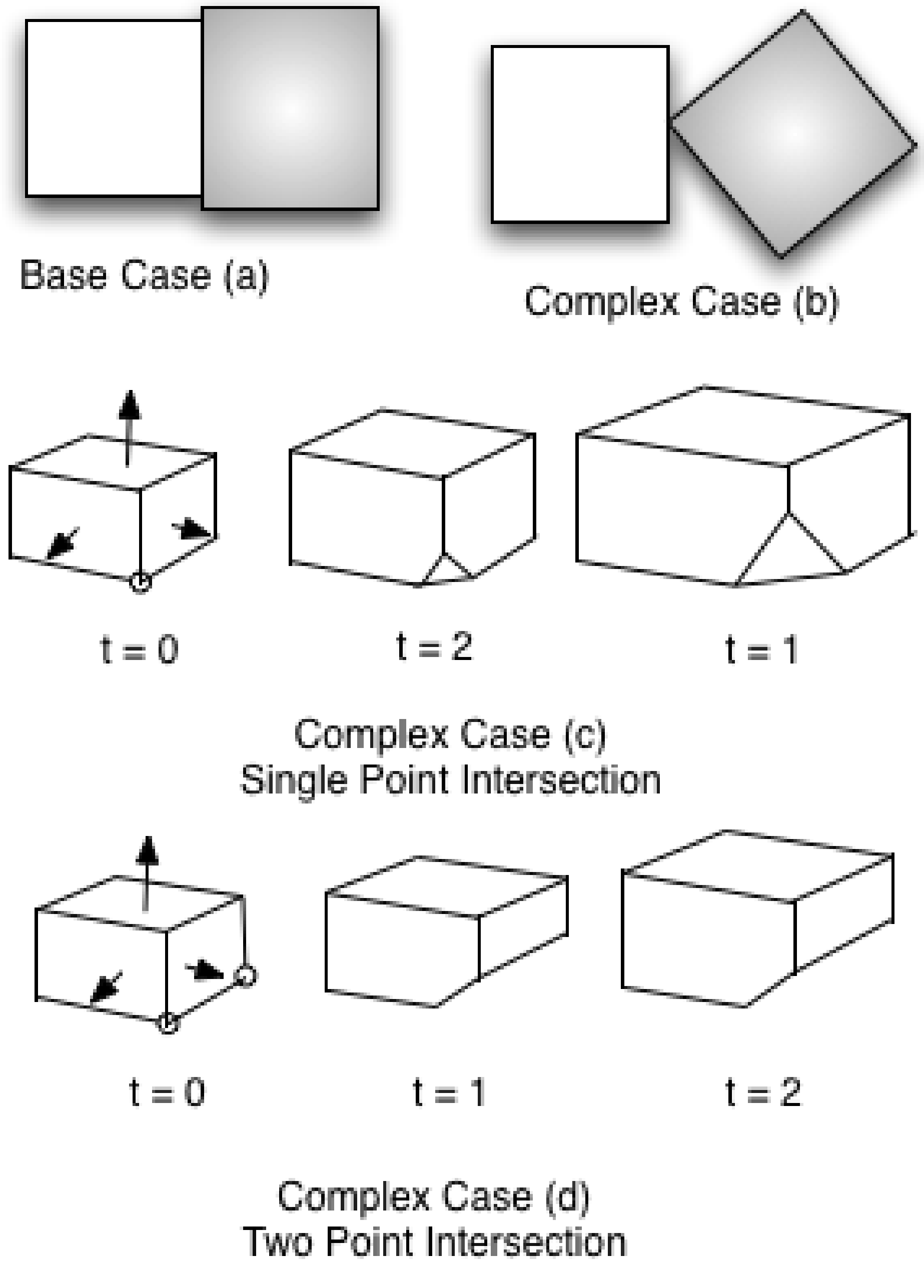


Figure 4.1: Here we see the possible growth cases for VASFV.

First, we shall examine the *base case* for growth in our algorithm. Each region is iteratively selected and provided the opportunity to grow once each pass. Growth occurs in the direction of the normal to each face of the region. We attempt to move the entire face in a single unit in this direction. We then take our proposed new region and run three collision detection tests on it. We want to ensure that no points from our growing region have intruded into a obstructed configuration space or another region and that no points from either of the aforementioned obstructions would be contained within our newly expanded region. Finally, the region performs a self check to ensure it is still convex in its new extents. Given that all those tests return acceptable results, we will allow the region to finalize itself into that new configuration. If any of those conditions are unacceptable then it means that we had a collision. If the world is axis-aligned when we reach this collision state we know that we were parallel to, as well as adjacent to, the shape we have collided with in our prior extents, which is the desired ending condition for region growth. In this case we return to our previous extents and set a flag to never attempt to grow that face again. We then allow every other face in the shape to grow in the same manner. Once each face of a shape has been provided the chance to grow a single unit, we proceed to the next shape as seen in Figure 4.1(a). This method of growth is sufficient to deal with all cases for axis-aligned obstructed configuration space areas.

The advanced case algorithm is considerably more complicated, but it is also able to deal with a greater variety of potential obstructed configuration space areas. It begins by incorporating everything contained in the base case algorithm and then expanding on it. Again we cycle through each region and provide each face in a region a chance to grow. This time, however, since we cannot use the properties of the axis-aligned world to assume we are parallel to the object we have collided with

**Algorithm 4.1:** VASFV

```
void startVASFV(List NegativeSpaceRegions) StillGrowing = true ;
/* Populate the world with the initial user defined seeds */
if NegativeSpaceRegions.isEmpty() then
    seedWorld();
while StillGrowing do
    StillGrowing = false ;
    for NegativeSpaceRegion in World do
        for Face in NegativeSpaceRegion do
            /* Translate the Face one unit forward in the direction of
               the face's normal */
            Face.Translate(Face.Normal);
            if Face.isNotColliding() then
                StillGrowing = true ;
            else
                /* See if it possible to deal with the collision by
                   increasing the order of the polyhedron */
                if Face.isSplitable() then
                    /* Determine which vertex(s).  Insert a face at the
                       vertex(s) that is being split.  */
                    Face.SplitPoint();
                    /* Lock the newly created vertex(s) to plane they
                       intersected */
                    Face.ConstrainPoint();
                    /* Further growth is possible with the newly split
                       vertex */
                    StillGrowing = true ;
                else
                    /* The collision cannot be handled by splitting stop
                       growth */
                    Face.Translate($-1$ * Face.Normal);
                    Face.canGrow = false;
List seedPoints = new List ;
seedPoints.Append(World.Seed());
if not NegativeSpaceRegions.isEmpty() then
    /* Restart growth algorithm */
    World.startASFV3D(seedPoints);
/* Run clean up algorithms */
World.combineConvexShapes();
World.removeColinearPoints();
World.RemoveDegenerateFaces();
```

we will need to take some additional steps to ensure we arrive at a good coverage decomposition. We have three advanced collision cases to consider, which case we use is determined by the number of the points in the growing face that have collided with a obstructed configuration space area. Finally, there is a fourth case that arises when a vertex from an obstructed configuration space object intersects a free configuration space region.

The first advanced collision case occurs when three or more vertices on the face of a region intersect a single face of a configuration space obstruction in the same growth step. We know due to the properties of the world that we encountered a plane which we are parallel since three or more points on the growing face would lay on it. In this case, despite the fact the entire world is not axis-aligned, the two faces we are currently considering are in fact axis-aligned and we can fall back into the basic collision case in which we stop growing.

The collision case resulting from one vertex of a obstructed configuration space object intersecting the growing region is actually the simplest of all collision cases. In this case, the face that intersected the object steps back to its last non-colliding location and then ceases further growth in that direction. It might seem strange that when a obstructed configuration space region is encountered in this manner that the algorithm stops trying to decompose in that direction, but there is not a way to achieve a better approximation of the colliding object without violating one of our two invariants as shown in Figure 4.1(b). The gaps in the decomposition resulting from this case will be filled in via seeding as we will discuss momentarily.

The final two cases for collisions with world geometry both involve inserting an additional face into the expanding region to closely adapt to the obstructed configuration space object it encountered. The first case occurs when a single vertex from a growing free configuration space region intersects with a configuration space obstruction. In this case, the vertex and each edge leading to it is split into a new face composed of three new vertexes. The normal of this face is set to the negation of the normal of the obstructed configuration space face it collided with, and three points are defined to lay directly on the obstructed configuration space face. From this point forward the new points are restricted to only lay on the plane of the face they intersected. This means that when the other faces of the free configuration space regions grow they will pull this new face out across the obstruction it intersected. These new points are restricted from growing beyond the plane to prevent more non-axis-aligned geometry from being exposed to the world. The results of this decomposition are shown in Figure 4.1(c).

The next case occurs when two points simultaneously intersect the same face of a configuration space obstruction. In this case a new face needs to be inserted into the growing region in order to better approximate the configuration space obstruction. Both of the intersecting points are split in this case resulting in four new points which will form a new quad shaped face increasing the order of the growing polyhedron. Due to the properties of a convex polyhedron if exactly two vertexes intersect another another shape then the entire edge between these points also intersects that shape, which means that we are in effect splitting that edge to become a new face. This new face is once again created using the negation of the normal of the face it intersected as its normal and made coplanar with the face it intersected. These new points are locked such that they can only move around on the face that they intersected for the same reason as in the previous case. This case is illustrated in Figure 4.1(d). This case will allow near complete decomposition of the free space in close proximity of obstructed configuration space without violating any of the underlaying assumptions of the algorithm.

The above growth techniques will do a good job of decomposing the world, but do not in and of themselves assure a complete decomposition. As in traditional PASFV we employ a seeding algorithm to allow free space that might have been missed initially a chance to decompose as outlined in Algorithm 4.2. Once all of the regions initially placed into the world have been grown to their maximum extents given the algorithms above the seeding algorithm is initialized. Each face of every region is given the chance to produce a seed in the world. The seeding method we use is to locate each distinct pocket of free space adjacent to a face and place a seed in it. It is extremely important for the quality of the decomposition that these seeds are then allowed to fall according to the world gravity model stopping only when they hit an obstruction (either an area of obstructed configuration space, or a previously place region).

This gravity-based model of seeding produces a much cleaner and more usable decomposition. Consider the two examples given in Figure 4.2 which shows possible methods of seeding a staircase from a free configuration space region at the bottom of the stairs. In the case shown in Figure 4.2(b) gravity is not applied to the seeding and the initial seed grows out and slots into a stair midway up the stair case and then grows upward. Additional seeds are then placed above and below this first region

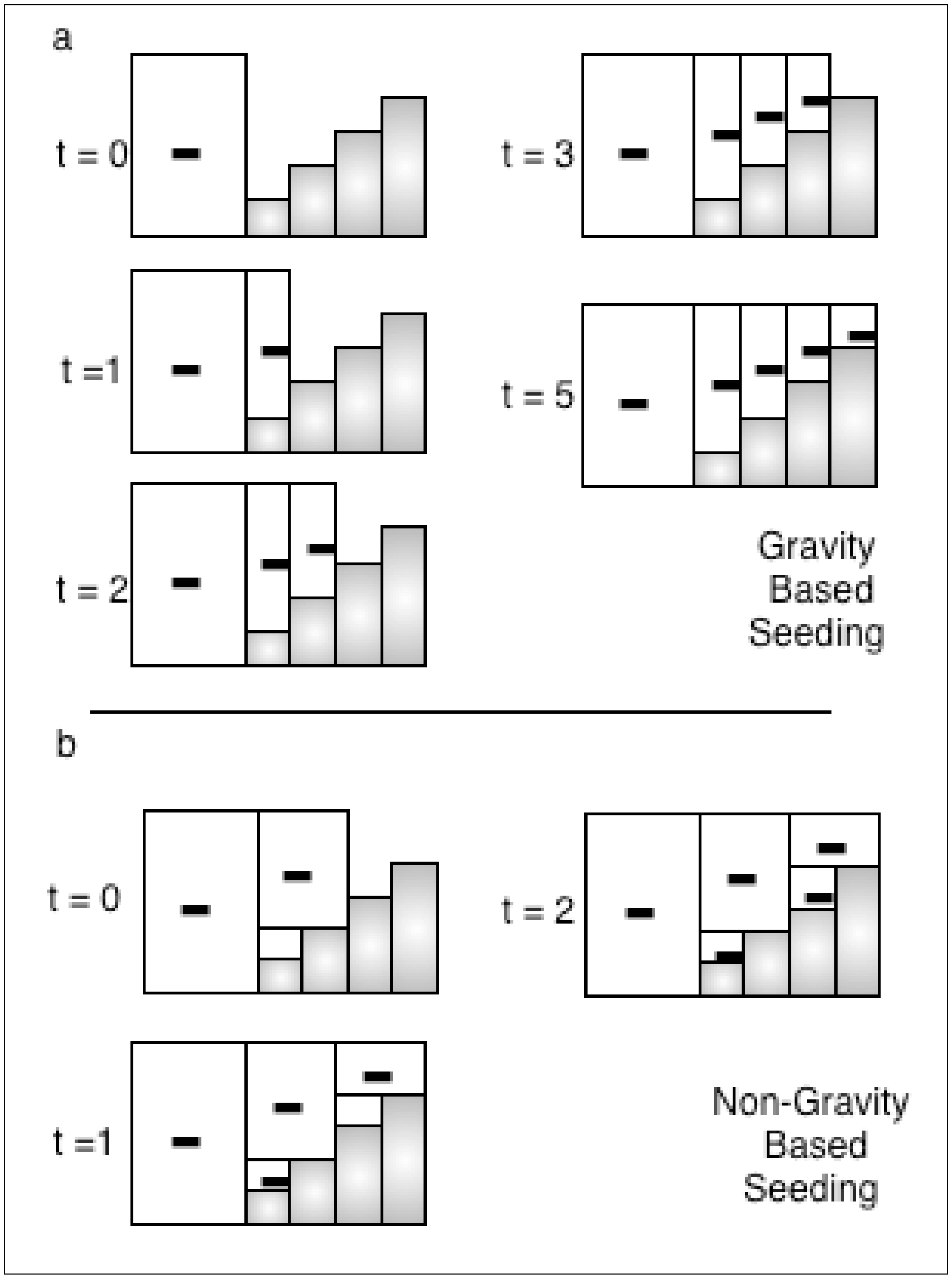


Figure 4.2: An illustration of VASFV seeding its way up a stair case. In each timestep a new seed is placed and then allowed to grow as much as possible. Obstructed configuration space regions are shown in grey, free configuration space regions are shown in white and marked with a negative sign. The world is viewed from the side and extends towards and away from the viewer. (a) Shows the results of seeding a world using gravity to modify the seeds locations. (b) Shows the results of seeding a world without applying a gravity model to place seeds.

until the entire stair case is decomposed. In the decomposition shown in Figure 4.2(a) seeds are affected by gravity. In this case a seed is generated from the first free configuration space region and then allowed to fall to the floor of the stair directly adjacent to it. This seed then grows to fill this single stair and all of the space above it. After growing, this new region will generated another seed that fills another stair completely. This cycle will continue until the stair case is completely decomposed. By comparing the two generated decompositions for the stair case it is obvious that the decomposition with gravity generates a more usable decomposition as it is possible for agents to stand in a single region on a stair. This is not possible for most of the stairs in the non-gravity based seeding algorithm as many of the regions on the stairs do not properly represent how a character would traverse a set of stairs.

Aside from the addition of gravity, the seeding system present in VASFV is identical to the 2D version and allow the algorithm to achieve complete decompositions of free space. This seeding system is especially effective in the case discussed above where an obstruction intersects the face of a free space region. After the seeding algorithm has concluded, the main growth algorithm is called again on the newly placed seeds providing them with a chance to grow. This cycle of seeding and growing continues until no new seeds are placed in the world at which point the world is fully decomposed (assuming at least one seed was placed in each disjoint area of un-configured space) and the algorithm terminates.

**Algorithm 4.2:** Locate points to add as seeds in open space

```
List Seed() List seedPoints = new List;
for FreeSpaceRegion in World do
    for Face in FreeSpaceRegion do
        /* Generate a List of potential seed points.  */
        List possibleSeeds = Face.GenerateSeeds();
        for seedPoint in possibleSeeds do
            while seedPoint.isInOpenSpace() do
                /* Move the seedPoint until it hits something */
                seedPoint.translate(GRAV_DIR)
            /* Return the point to open space */
            seedPoint.translate(-GRAV_DIR)
        seedPoints.extend(possibleSeeds)
seedPoints.removeDuplicates() return seedPoints
```

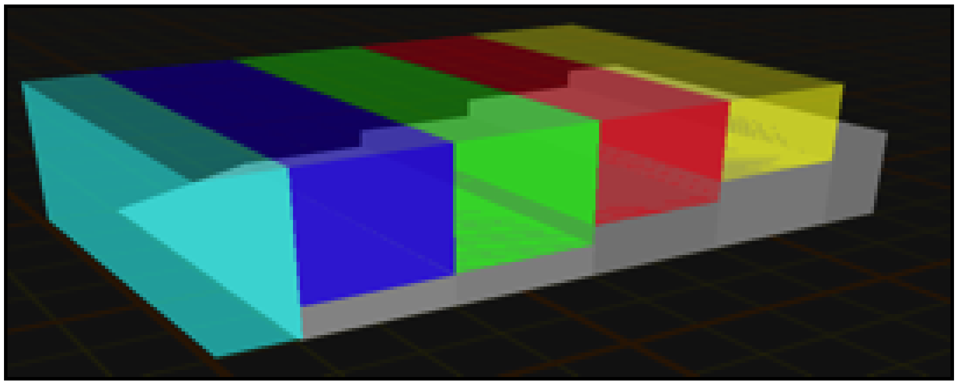
## 4.2 VASFV Evaluation

We evaluated the VASFV algorithm by comparing it with two 3D spatial decomposition algorithms: Extruded Space Filling Volumes (ESFV) and Automatic Path Node Generation (APNG). For our testing environment we wanted a game world feature that would be a stumbling block for most 2D based decomposition algorithms. Hence, we chose a staircase with a non-axis-aligned ramp leading up to it as our test environment. There are three main reasons for this. First, a set of stairs contains many walkable steps, each of which is set at a unique height above the ground, so even without any other geometry a staircase is difficult to decompose. Algorithms dependent on projecting each ground plane level into 2D and generating a separate decomposition for each level must project each step into 2D which is time consuming and results in many different decompositions, or the stair case decomposition will have to be performed by hand and linked into the different levels of the environment it connects as a special case. Secondly, while there are multiple possible decompositions for this test case, some forms of decompositions are dramatically better than others for use in agent navigation as shown above (Figure 4.2). Third, due to the number of regions present in more complex decompositions, it is hard to visualize the decomposition, so we felt a simple, but difficult test case would best illustrate the capabilities of VASFV. The three algorithms generated the decompositions seen in Figure 4.3.

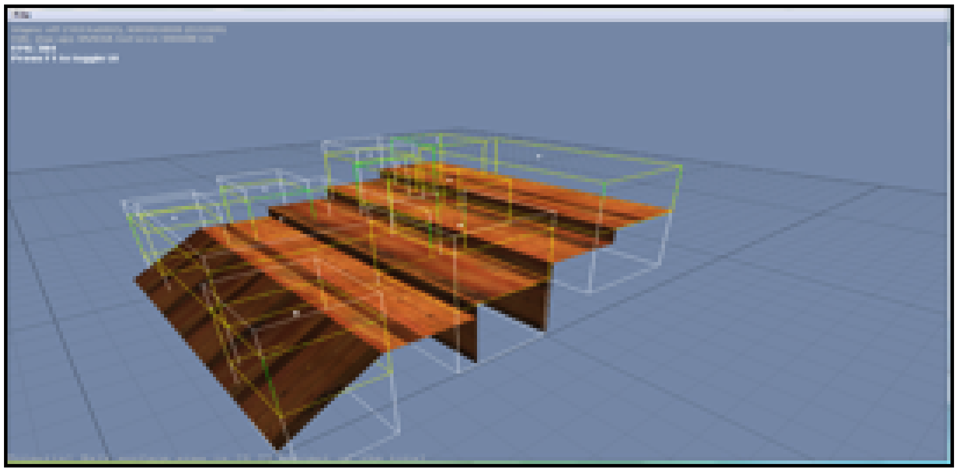
Table 4.1: Comparison of Multiple Spatial Decomposition Algorithms.

| Algorithm | Number of Regions | Coverage |
|---|---|---|
| ESFV | 5 | 70% |
| Path Node | 12 | 90% |
| VASFV | 5 | 100% |

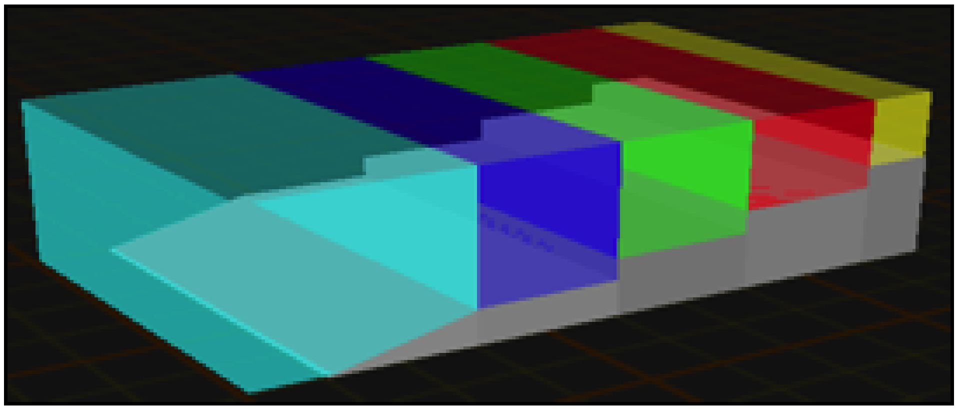
The generated decompositions for each of the three algorithms were compared in terms of how many regions were produced and the coverage (i.e., the percentage of the empty space in the world contained in the decomposition). These results are summarized in Table 4.1. VASFV outperforms both ESFV and APNG in terms of coverage, and is the only decomposition algorithm to provide complete coverage of the world. Having a high coverage decomposition is important for tasks such as pathfinding, information compartmentalization, or collision detection. This is because, as the coverage percentage drops, gaps and un-walkable areas form in the navigation mesh, which dramatically reduces its usefulness. Results also indicate that VASFV is comparable with SFV when it comes to producing the fewest regions. This is an important consideration as fewer regions means a reduced search space for path finding algorithms or other graph search algorithms. Overall, when both coverage and number of regions are taken into account VASFV produces the best decomposition



a



b



c

Figure 4.3: A comparison of multiple decomposition methods when building a navigation mesh for a stair case. (a) Extruded Space Filling Volumes. (b) Automatic Path Node Generation. (c) VASFV.

for the test case presented here.

## 4.3 PASFV and VASFV Completeness and Special Cases

The PASFV and VASFV algorithms are both complete algorithms in that they always generate a spatial decomposition if such a decomposition would exist. The decomposition might not exist due to one of several reasons that are caused by invalid input data. First, the input geometry has to be either convex closed polygons, or it has to be concave closed polygons that are made convex via subdivision. Both the PASFV and VASFV algorithms make extensive use of the *point in convex polyhedron* algorithm to perform collision detection checks while growing. If the geometry that describes the obstructed configuration spaces is not convex then these checks will fail with unpredictable results.

Secondly, either the normals or the winding order of the input geometry must be uniform and consistent across all obstructions. PASFV and VASFV depend on the normals of the obstructions (or the winding order that is used to create normal data if the normals are not provided) to define the areas of space occupied by the obstructions. If the normals contain inconsistent or incorrect data this is effectively the same as if there were concave regions in the input dataset and a violation of this assumption will result in the generation of incorrect or degenerate spatial decompositions.

The final potential failure case occurs when executing PASFV or VASFV if there are disjoint areas of the environment that do not receive region seeds or if no region seeds are generated at all. This occurs due to improperly defined grid based seeding. Effectively, the seeding grid misses all or some of the disjoint areas of the environment and those areas of un-configured space never have initial regions added to them. If a region is added to an area of un-configured space and the boundaries of that space are correctly defined then that growing region is assured of fully decomposing the area of un-configured free space it inhabits—either through its own growth, or the growth of regions it generates via seeding.

While the algorithms are complete there are still some special cases of valid input geometry that will cause PASFV and VASFV to generate poor quality spatial decompositions though these decompositions will still be valid decompositions (non-degenerate). Such cases occur when the input geometry contains curves that have been approximated through the use of many short line segments to form the required convex obstructions. We know from Proof 4.2 in Section 4.5 that a convex region defined in free space can only be adjacent to a single edge or face of a given obstruction, each of the many short lines that would be used to approximate the curve or spherical obstruction will require their own region to represent the adjacent free configuration space next to them. This means that there will be a huge number of regions required to represent such curved obstructions unless the curves are highly simplified. However, this region explosion will occur with other full coverage spatial decomposition algorithms as well since there are no algorithms to combine the space adjacent to curved objects into fewer convex regions.

## 4.4 PASFV and VASFV Empirical Runtime Analysis

Initially, consider the case of a region that is placed into an entirely empty bounded environment. This empty environment contains some amount of free space described as $x_{free}$. As the region expands, it will consume this free space until none is left at which point the PASFV or VASFV algorithm will terminate. Now consider the amount of $x_{free}$ that is consumed by the single growing region $R$ on each growth step. Initially before it grows, $R$ consumes one unit of free space. On the next growth step, R expands outwards in every direction. In the planar case (PASFV) this results in $R$ consuming 9 units of $x_{free}$. $R$ will consume 25 units of $x_{free}$ on the next growth step. This cycle will repeat with $R$ consuming ever increasing amounts of $x_{free}$ on every growth step. When plotted we see that this rate of growth of claimed area is exponential. An exponential reduction in the size of the problem that remains to be considered means that this decomposition technique will produce in the ideal case a *fractional power* relationship between $x_{free}$ and the number of regions consuming $x_{free}$—O($n^r$), where $n$ is $x_{free}$, and $r$ is between zero and one, and a function dependent on the number of regions decomposing the world. This ideal case does assume that each of the regions is able to grow outward in all four directions at once. This exponential rate of reduction in the amount of work that the algorithm still needs to perform will remain as long as any two adjacent edges of the region are still growing.

Now consider what happens to $x_{free}$ if only one edge (or two non-adjacent edges) is growing. In this case, the amount of work that is done each growth step remains the same and $x_{free}$ decreases by a linear amount. This linear reduction in $x_{free}$ means that in the worst case of a single edge of a single region growing that the PASFV and VASFV algorithm run in linear time, O($n$) where $n$ is $x_{free}$. We do not need to consider the case where no edges are growing as the region will either seed additional regions with one or more growing edges or the algorithm will terminate.

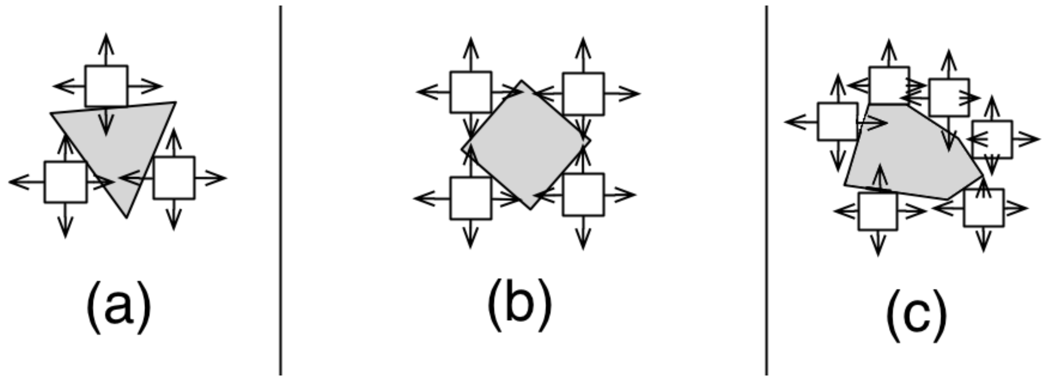


Figure 4.4: A comparison showing the results of seeding and growing regions in a world composed of one increasing complexity polygon obstruction. The obstruction is shown in gray.

Now that we have examined a single region growing in an empty environment consider the case of a simple triangle obstruction placed into the quad-bounded world environment (simpler obstructions such as a single point or line are degenerate and not decomposed) as shown in Figure 4.4(a). In this case, we know from Proof 4.2 in Section 4.5 that we will require at least one region seed ($R_n$) per obstruction edge to fully describe the world. In the worst case, we know from the above examination that the PASFV algorithm would run in linear time as there will be at least one edge of $R_n$ growing at any given time until the algorithm terminates. We also know that since only one edge of each growing region can be in collision (follows from Proof 4.2 in Section 4.5) the other three edges of each region in $R_n$ can grow outward and expand until they hit either the boundaries of the world or another growing region. While this rate of growth holds we will see a runtime approximating that of O($n^r$), where $n$ is $x_{free}$, and $r$ is between zero and one, and a function dependent on the number of regions decomposing the world, as shown above. Since we know nothing about the world, we cannot say where the collisions between regions will occur that means that for this simple case all we know is that the runtime will be between *linear* and *fractional power*. We also know that as we increase the complexity of the obstruction present in the environment that by Proof 4.2 in Section 4.5 we will be adding additional regions to $R_n$ (as shown in Figure 4.4(b-c)) and that the runtime will remain consistent as it is bounded by the same growth limitations no matter how many edges are present. Increasing the complexity of the obstruction (or adding more obstructions) does not affect the runtime of the PASFV or VASFV algorithm. Instead, runtime is determined by the amount of free space present in the world. However, doubling the size of the environment would increase the required runtime.

## 4.5 Minimal Number of Regions to Cover Environment - Upper Bound

*Theorem 4.1*: The minimum number of regions in a free configuration space decomposition generated by the PASFV and VASFV algorithms is the number of exposed obstruction edges, where exposed obstruction edges are obstructed configuration space adjacent to negative space.

*Proof 4.1*: Since obstructed configuration space is assumed to be broken down into convex polygons before it is given as an input to our algorithms, we assume that all exposed obstruction edges lie on convex polygons. Given these constraints, we can restate our theorem as the following lemma:

*Lemma 4.2*: Each exposed edge of a convex obstructed space polygon P must be "covered" by a single unique convex region in a free configuration space decomposition.

*Proof 4.2*: We note that by “covered” we mean that the edge must be contained in and adjacent to a single convex region in the free configuration space decomposition.

Suppose not. We now consider two distinct exposed edges $A$ and $B$, of the convex obstructed space polygon $P$. By contradiction, there exists a free configuration space decomposition with one convex region $R$ adjacent to both of these exposed edges $A$ and $B$. Then, since $R$ is adjacent to both $A$ and $B$, it must be adjacent to a point $a1$ on $A$ and a non-endpoint $b1$ on $B$. The line segment ($a1$,$b1$) is not contained on a boundary of $P$ because if it were, then $A$ and $B$ would not be distinct exposed boundaries of $P$. Since $a1$ is on $A$ and adjacent to region $R$, it is contained in both polygon P and region $R$. Since $b1$ is on $B$ and adjacent to region $R$, it is contained in both polygon $P$ and region $R$. Since all free configuration space regions are convex, region $R$ must be convex. Polygon $P$ is assumed to be convex. By the definition of convexity, we can draw a line connecting any two points in $R$, and that line must be fully contained within $R$, and likewise for polygon $P$. Consider the line ($a1$,$b1$). Since polygon $P$ is convex, this line is fully contained within polygon $P$. However, since region $R$ is also convex, ($a1$,$b1$) is also fully contained within region $R$. However, since ($a1$,$b1$) is not a boundary of polygon $P$, it cannot be contained within both polygon $P$ and region $R$.

*QED* : Therefore, each exposed edge of P must be adjacent to a single, unique free configuration space decomposition region.

# CHAPTER 5: DYNAMIC UPDATES TO NAVIGATION MESHES

All the methods of generating free configuration space decompositions we have discussed so far have one rather disappointing thing in common, the end product of each tends to be very static. This is unfortunate as there is a tendency towards more interaction with the world in modern games and a desire for more realistic reactions from the environment in simulations. In several recent game releases it has been possible to dynamically alter the world in response to the players actions. For example, in the recent title from Electronic Arts *Battlefield: Bad Company* it is possible for the player to demolish walls in structures and dramatically alter the passable areas of the game environment. The title *Fracture* from Lucas Arts carries this concept of a deformable world further as it allows the player to dynamically alter the terrain of the game environment. Obviously, if the player is given the total freedom to alter the world it will be impossible to predict the state of the world at any given point in the game and the existing solution of having a pre-built spatial decomposition or set of way points for every possible world state cannot hope to adapt to this level of variability. In response, there is a need to upgrade and expand spatial decomposition techniques in order to deal with this random alteration of the world.

We propose a solution for this problem that involves an extension of the existing algorithm Planar Adaptive Space Filling Volumes in order to allow it to dynamically rebuild and relink damaged areas, such that even with dramatic changes to the world the decomposition is still valid. This extension comes in two distinct parts. The first part allows for creation of free configuration space in areas that were formerly configuration space obstructions (i.e., the removal of obstructed configuration space areas). Imagine a helicopter takes off leaving an empty field, this field can now be walked through freely and the decomposition will need to be updated to reflect this. The second extension allows for addition of configuration space obstructions into areas that were formerly walkable. For example, imagine a building collapsing into a plaza area. This will render the plaza partially or completely unnavigable and obviously requires a change to the navigation map for the environment. Both of these extensions are natural progressions from the core concepts of the PASFV algorithm which are: providing a fast effective way of generating a world decomposition, providing a high coverage decomposition, yielding good quality decompositions for navigation, and providing decompositions based around quads or higher order polygons rather than triangles allowing for better information compartmentalization.

This extension to the general form of Adaptive Space Filling Volumes is provided by two algorithms that allow for the addition of free configuration space into existing configuration space obstructions or the addition of configuration space obstructions into existing free configuration space. We refer to the enhanced version of PASFV as Dynamic Adaptive Space Filling Volumes (DASFV) algorithm.

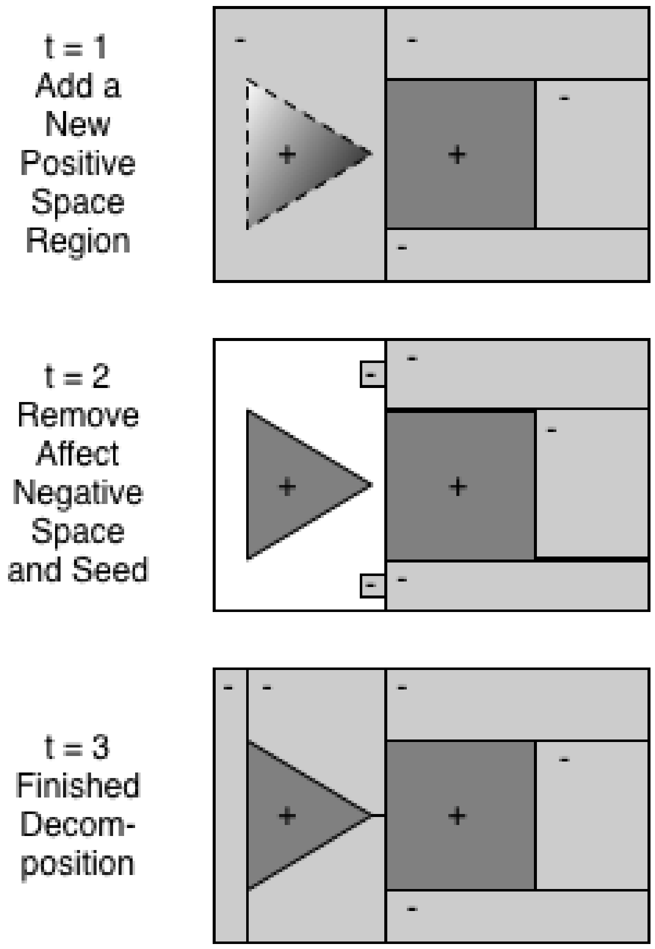


Figure 5.1: A time step progression of the addition of obstructed configuration space. The free configuration space is shown in light grey. Obstructed areas are shown in dark grey. The obstructed configuration space that is added to the world is shown with a gradient and it has the dotted outline. The second time step shows the removal of affected free configuration space regions and the addition of seeds to grow new free configuration space regions. The final time step shows the repaired decomposition.

## 5.1 Adding Obstructed Configuration Space

Let us first examine the case where the building collapses into the plaza. First, the free configuration space regions that used to compose the plaza need to be removed. These are found by performing a series of intersection tests on the existing free configuration space regions and the newly added obstruction. Since all of the

**Algorithm 5.1:** Place Configuration Space Obstacle

```
/* We will assume that the area was fully decomposed */
HandleObstructedAddition(List oldNegatives, List oldPositives, newPositives);
/* Locate all areas that need to be removed */
for NegativeSpaces neg in oldNegatives do
    /* Run intersection check against each new region */
    for PositiveSpaces pos in newPositives do
        if neg.intersects(pos) then
            neg.remove();
            if connectivityKnown is true then
                /* set adjacent regions to seed */
                neg.setNeighborsSeeding(true) /* otherwise do nothing */
    if connectivityKnown is false then
        neg.ResetSeeding();
/* All conflicting regions will have been removed */
/* Assume the implementation of ASFV contains */
/* a method to start seeding */
ASFV.seed();
/* Also assume that the ASFV implementation */
/* contains a method to grow placed seeds */
ASFV.run();
/* Once the ASFV algorithm concludes the decomposition */
/* will be complete again */
```

free configuration space regions are convex and the newly added configuration space obstructions are required to be convex, the intersection test between them can be performed very quickly with a point in convex polygon algorithm [59]. Once a list of free configuration space regions to be removed has been established, the algorithm will branch into two directions depending on what information is available. If connectivity information is available between regions then the neighbors of the regions to be removed should be reset so that they will attempt to seed as per the PASFV algorithm. If connectivity information is not available then all free configuration space regions not being removed should be reset to seed again. At this point the PASFV algorithm will be started at the seeding stage of the algorithm. PASFV will run until the newly vacated area is fully decomposed at which point it will stop as per the original algorithm. Sample code is provided for this case in Algorithm 5.1 and an illustration of this algorithm in action is provided in Figure 5.1.

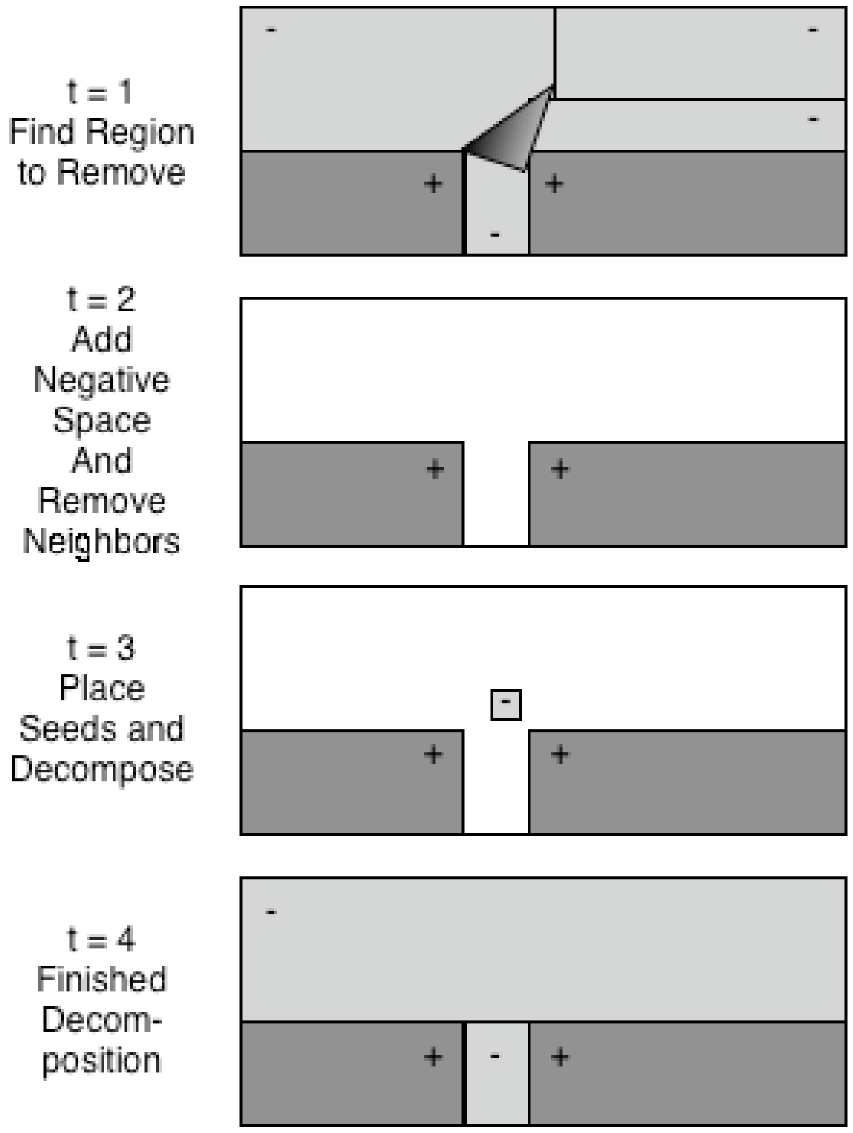


Figure 5.2: A time step progression of the addition of free configuration space. The free configuration spaces are shown in light grey. Configuration space obstructions are shown in dark grey. The obstructed configuration space to be removed is marked in a gradient. In the second time step the targeted obstruction in configuration space has been removed along with its neighboring free configuration space regions. In the third time step a seed has been added to the decomposition. In the final time step the decomposition has been fully repaired.

## 5.2 Adding Free Configuration Space

The addition of free configuration space regions works similarly to the addition of obstructed configuration space regions. To continue the example of the helicopter taking off from a field we need to quickly determine how to divide up the field it vacates for navigation. First, the listing of all of the obstructed configuration space regions to be removed is compiled. Once this has been compiled the algorithm will again branch depending on whether or not connectivity information is available. If connectivity information is known between regions then the neighbors of the affected obstructed configuration space areas can easily be located. Otherwise the neighbors have to be located algorithmically. This is accomplished by taking the affected obstructed configuration space areas and increasing their size by 0.1 percent, and then determining what free configuration space regions they intersect. Since the decomposed regions perfectly border against the obstructions this small size increase to a obstructed configuration space area will force it to intersect all their neighboring free configuration space regions. Once the targeted obstructed configuration space areas and neighboring free configuration space regions have been identified they can then be removed. Seeds are then placed at the former centroids of the removed obstructed configuration space regions to provide a starting point for the re-initialization of the PASFV algorithm. Again, once PASFV completes the newly exposed free space will be fully decomposed into high order convex polygons. The presentation of this algorithm can be seen in Algorithm 5.2. An illustration of this algorithm running can be seen in Figure 5.2

**Algorithm 5.2:** Adding free configuration space to existing obstructed configuration space regions

```
/* We will assume that the area was fully decomposed */
/* removePositive(List oldNegatives, List oldPositives, List toRemove)
   */
/* Locate all areas that need to be removed */
for PositiveSpaces oldPos in oldPositives do
    /* Run a simple equality check to find regions to remove */
    for PositiveSpaces posRemove in newPositives do
        if oldPos.equals(posRemove) then
            /* Found a region to remove locate its neighbors */
            /* Assume we possess a function */
            /* to find neighbors of a given region */
            List neighbors = oldPos.getNeighbors() for NegativeSpace neighbor in
            neighbors do
                /* Remove the negative space neighbor */
                neighbor.remove()
            oldPos.remove()
/* All conflicting regions have been removed */
/* Assume the implementation of ASFV contains a  */
/* method to start seeding */
ASFV.seed() /* Also assume the ASFV implementation contains a  */
/* method to grow placed seeds */
ASFV.run() /* Once the ASFV algorithm concludes the */
/* decomposition will be finished */
```

In the case of the removal of a obstructed configuration space area it may seem strange and counter intuitive to remove the free configuration space regions adjacent to it when they do not appear to be directly affected by the removal, but there are several reasons for doing so. The primary reason for removing neighboring free configuration space regions is related to the concept of information compartmentalization, so let us take a moment to consider how the shape of regions affects this quality. As long as we are just using these decompositions for agent navigation from point-to-point along some kind of connectivity graph and the decomposition completely covers the free space and is well connected, the ability of the decomposition to compartmentalize information is not of great importance to the agent. However, when we move beyond using the decompositions for just navigation we encounter issues with triangle-based decompositions such as those generated using the Hertel-Mehlhorn algorithm [36] or Delaunay triangulations [13]. Decompositions of environments can be used for the encoding and storage of relevant spatial details for the game or simulation. For example, objects and events can be localized to regions. This means that the agent moving around the world can reduce the amount of things it needs to reason about to those objects in its region and perhaps neighboring regions. Localized dynamic object collision detection can also vastly reduce the amount of collision tests per frame by only considering possible collision cases for objects that occupy the agent's region or neighboring regions. It is in situations like these that the triangular decomposition methods begin to have problems. There is no limit to the number of triangles that can come together at a point (see Proof 7.1 in Section 7.2) so any given triangle could have a huge number of potential neighbors. Furthermore, since triangles by their nature tend to be long and skinny in these decompositions it is harder to say that objects can be contained in only one or two regions. Reasonably sized dynamic objects in a game might span the narrow edges of many triangles or even worse might sit at the convergence point of dozens of triangles. This high overlap potential can drastically reduce potential computational savings of using decompositions of the world to reduce and localize the number of expensive computations that must be performed.

By its very nature the PASFV algorithm can only produce free configuration space regions that intersect each other in two ways. The first intersection type occurs when axis-aligned parallel edges meet each other, in this case there can only be at most two regions involved in the collision. The other possibility occurs when the right angle corners of three or four growing regions meet at a single point. This in effect limits the number of regions meeting at a single point to four, which is obviously far lower than the unbounded worst case for triangular decomposition methods. As such, PASFV-based decompositions tend to lend themselves more towards applications beyond navigation which depend on good information compartmentalization. However, in order to ensure this is the case when we remove obstructed configuration space areas we need to ensure that we do not expose non-axis-aligned edges of any free configuration space regions. Doing so would potentially introduce all of the problems associated with allowing for the possible intersection of more than four regions at a single traversable point. This is the primary reason that our algorithm calls for the removal of neighboring free configuration space regions when a obstructed configuration space area is removed. Also, since information compartmentalization tends to work better on larger regions it benefits us to clear out any neighboring regions to ensure that we get the largest regions possible. The problems that could be caused by just removing obstructed configuration space areas and not their free configuration space neighbors can be seen in Figure 5.3. Despite starting from the same initial configuration, the decomposition of the world in Figure 5.3 is clearly worse than Figure 5.2 since it contains five malformed regions rather than two axis-aligned regions.

After the decomposition of the world has been adapted to fit the changes in the world's underlying geometry the connectivity graph between regions can be quickly rebuilt. Instead of completely rebuilding the connectivity graph a shortcut using only

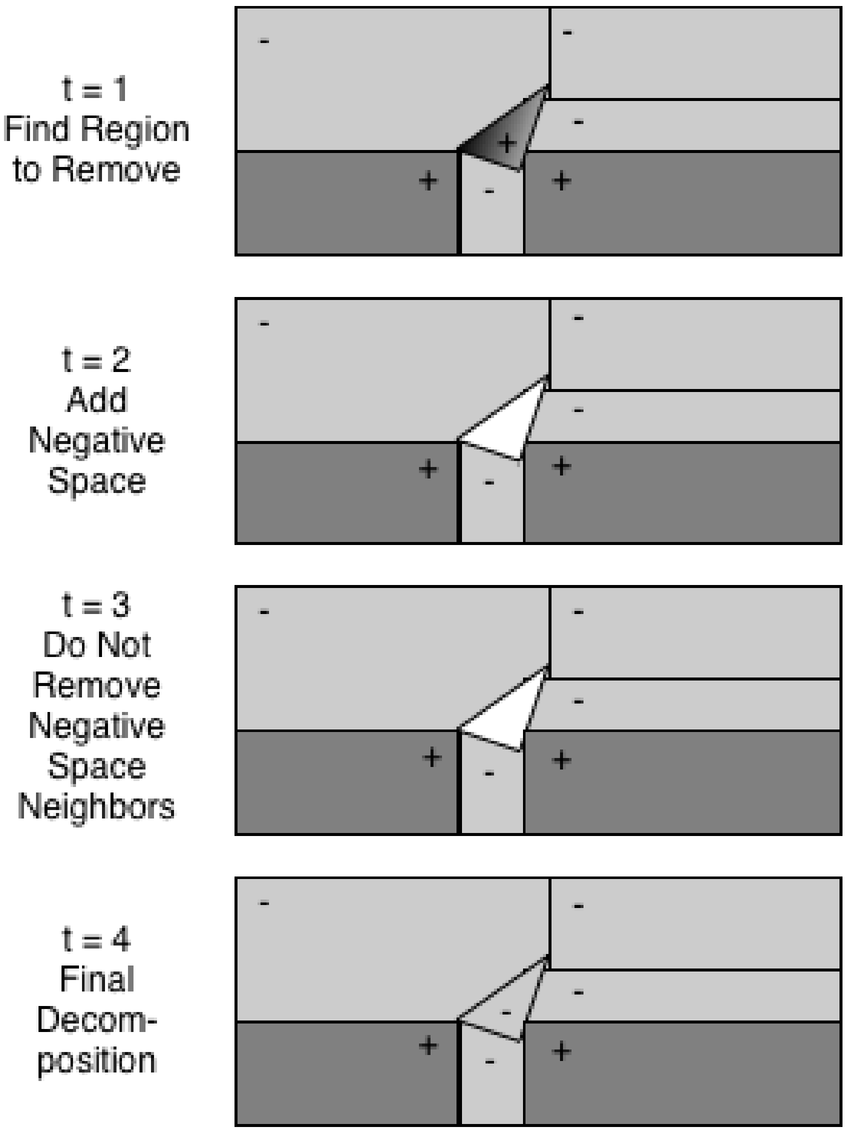


Figure 5.3: This timestep illustration shows the undesired effects of adding free configuration space to a obstructed configuration space area without removing the neighboring free configuration space regions as called for in the Dynamic Adaptive Space Filling Volumes algorithm.

two steps is available. The first step is to remove any links that are invalidated by the removal of free configuration space regions. Then it is a matter of iterating through each of the new regions that were created during the decomposition repair phase and determine which regions border it to rebuild the connectivity graph between each region.

## 5.3 DASFV Navigation Mesh Evaluation

A pair of experiments were conducted to examine the two primary aspects of dynamically altering the world after a decomposition has been generated, those aspects being the addition of free configuration space, and the addition of obstructed configuration areas. Both experiments were conducted on the same computer using the DEACCON (Decomposition of Environments for Automated Creation of Convex Navigation-Meshes) tool which implements the DASFV algorithm on a 2.13Ghz/2GB RAM computer.

The first experiment consisted of examining the effects of adding obstructions to existing areas of free configuration space. This is the "building collapsing into the plaza" example from above. This experiment was conducted on ten random maps each with increasing amounts of geometric complexity, which represent other buildings that are present in the world. The performance of dynamically repairing the decomposition using our new extensions to Planar Adaptive Space Filling Volumes is compared to the time it takes to completely rebuild the decomposition in order to adapt to changes. As can be seen in Figure 5.4 using our dynamic extension to generate new decompositions for altered environments is considerably faster compared to rebuilding the entire decomposition even as the complexity of the world in which the algorithm operates increases. These results are in accordance with our expectations that rebuilding a smaller area of the decomposition is faster than rebuilding the entire decomposition.

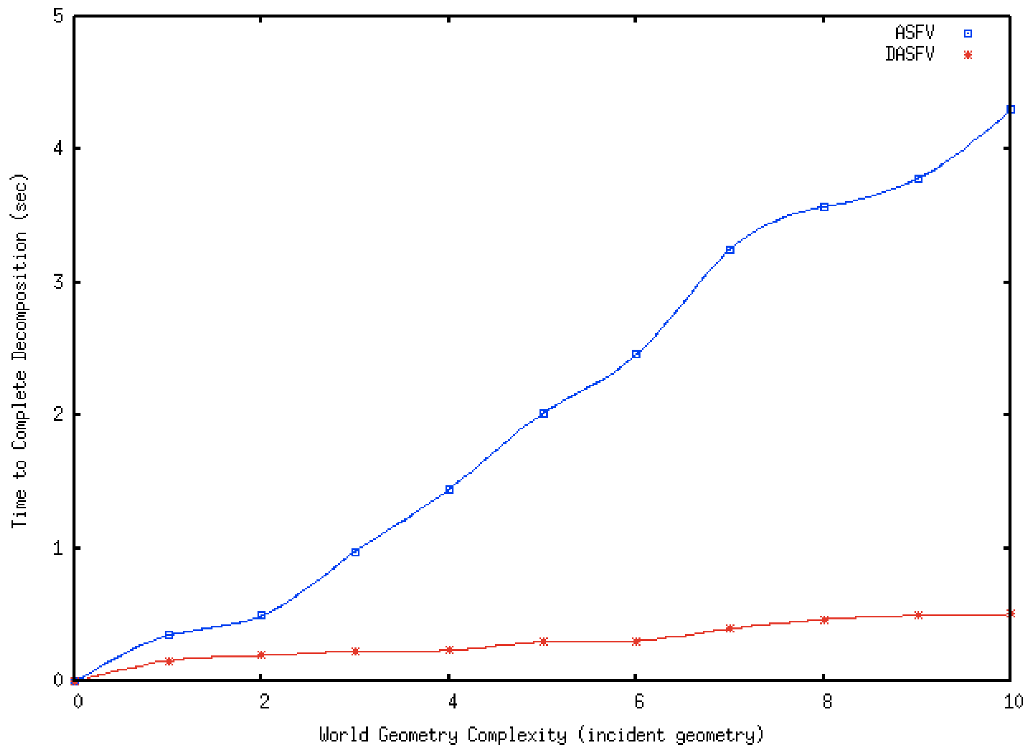


Figure 5.4: This graph shows a comparison between the time costs to repair a decomposition vs rebuilding the entire decomposition when configuration space obstructions are added to the world as the complexity of the world is increased.

The second experiment consisted of examining the effects of adding free configuration space to existing obstructed configuration space areas, in effect removing those areas, and then decomposing the newly created free space. Our helicopter taking off example would be represented by this experiment. Again, we compared the time cost to decompose just the affected areas to the cost of rebuilding the entire decomposition against a backdrop of increasing geometric complexity. The results of this experiment as shown in Figure 5.5 does show that it is better to rework just the affected area than the entire decomposition. However, there are some cases for very simple worlds containing only one or two regions where the repair takes as long or longer than rebuilding the entire decomposition. A close examination of the worlds in question yields an explanation of why this phenomenon occurs. Recall the addition of free configuration space algorithm calls for the removal of all neighboring areas of decomposed free configuration space in addition to the directly affected obstructed configuration space area. This means for very simple worlds it is possible all or most of the free configuration space regions in the world are removed along with the targeted configuration space obstruction. This results in the repair algorithm having to run intersection tests on every region in the world before discarding all of them. These intersection tests will take longer than just discarding everything to start with. However, such simple worlds occur so infrequently that is always desirable to perform a repair rather than a reconstruction.

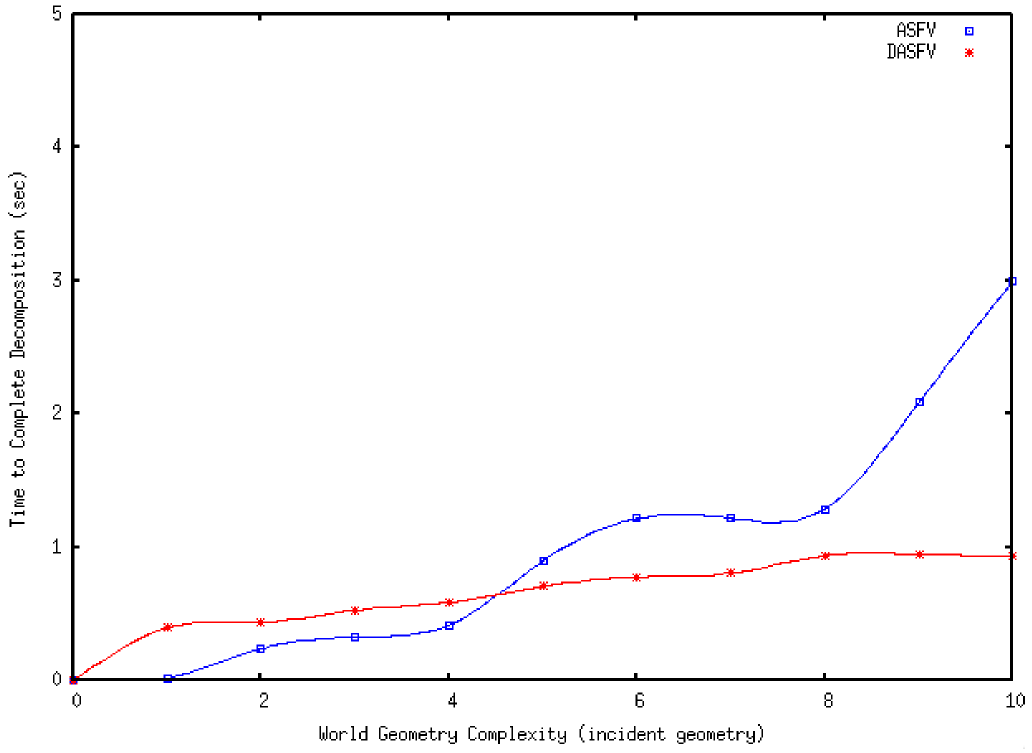


Figure 5.5: This graph shows a comparison between the time costs to repair a decomposition vs rebuilding the entire decomposition when free configuration space is added to the world as the complexity of the world is increased.

# CHAPTER 6: METRICS

Navigation meshes are being used in more and more games and simulations. For example, Fallout 3 by Bethesda (geck.bethsoft.com/index.php/NavMesh_Generation) and Neverwinter Nights 2 by Obsidian (www.obsidianentertainment.com) both incorporate navigation meshes rather than waypoints as past games by these developers did. The shift towards navigation meshes brings up the question—which of the many different mesh generation algorithms should be used for a given 3D world? In addition, as more and more automated tools for building these decompositions come into use, which can produce numerous unique decompositions from the same obstructed configuration space areas, there needs to be a way of ordering the resulting navigation meshes and choosing the best ones. Unfortunately, there are no commonly available metrics (except for simple metrics such as coverage or number of regions) to provide a game designer with a clear indication of which decomposition algorithm would produce the best set of connected regions for any given set of world geometry or prioritized navigation mesh characteristics. This is especially important as producing an optimal decomposition for a given map can be shown to be NP-Hard [44].

We address this problem by presenting a set of metrics with which to evaluate navigation meshes and then provide a clear ordering of the navigation meshes based on desired characteristics of the designer or developer. Many of these metrics can be calculated and evaluated automatically, and can be used to select among the many navigation mesh generation options.

This chapter presents a collection of metrics with which to evaluate the navigation meshes generated from world decompositions. All these metrics are designed to work on 2D representations of 3D worlds; However, in the future they could be extended to deal with full volumetric 3D decompositions. These metrics should be used to compare multiple variant decompositions of the same world geometry.

In addition to describing each of the 12 metrics listed below, we provide an example of how each one performs by running the metric using three sample decompositions of the world shown in Figure 6.1(b). The first decomposition Figure 6.1(c) was created using the standard SFV algorithm and provides an example of why SFV is not the best algorithm to use when decomposing non-axis-aligned worlds. The second decomposition shown in Figure 6.1(d) was created using the Hertel-Mehlhorn (HM) algorithm, while the third decomposition shown in Figure 6.1(e) was generated using the PASFV algorithm.

## 6.1 Number of Regions

The first metric to consider is a simple examination of the number of regions present. This can tell us some interesting things about the quality of the decomposition that we are looking at. In most cases, when evaluating a pair of decompositions, the lower this number is the better since the decomposition will be smaller and easier to search.

Looking at the output of this metric seen in Table 6.1 on our sample world, we see

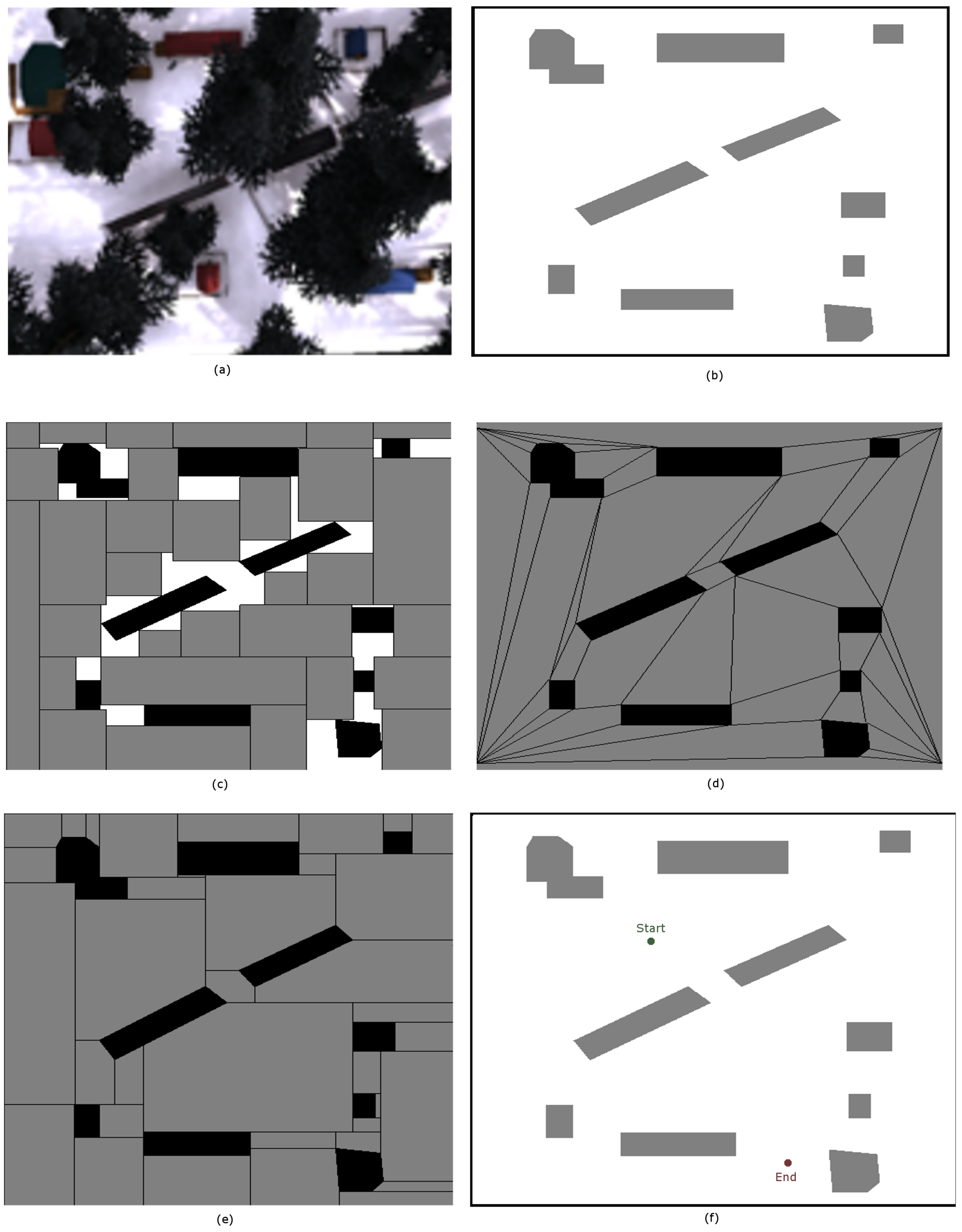


Figure 6.1: Different decompositions of free configuration space from the Twin Lakes level in Silicon Ice's Quake 3 Total Conversion Mod: Urban Terror as seen from a top view in (a) with base obstructed configuration space geometry in (b). A Space-Filling Volume (SFV) decomposition is shown in (c), a Hertel-Mehlhorn (HM) in (d), and an Planar Adaptive Space-Filling Volumes (PASFV) in (e), the randomly generated starting and ending points to illustrate flow through the level are shown in (f)

Table 6.1: The *Number of Regions Metric* for each of the decomposition algorithms.

| Algorithm | SFV | PASFV | HM |
|---|---|---|---|
| Number of Regions | 31 | 35 | 41 |

SFV produced the lowest number of regions, which would indicate that they produced the simplest navigation mesh. Unfortunately, this low number of regions is due to the fact that the SFV algorithm did not fully decompose the world. We are therefore more interested in the evaluation between the other two metrics; in this case PASFV produced considerably fewer regions to fully decompose the world than HM.

If the navigation mesh is being used in a graph-based search, then a lower number of regions would indicate fewer nodes in the graph, which will assist with calculations whose run times are determined by the number of nodes present in the graph. A good example of this type of calculation would be A* path planning. Fewer nodes on the graph yield fewer locations that the algorithm has to search when generating a path. There are few interesting exceptions to this rule of lower is better. It is possible for areas of empty space to have special rules or conditions attached to them—even if there is not a clear geometric divider to split up the empty space in the world. For example, a basketball court is composed entirely of empty space, but the regions of the court itself have meaning (i.e., baskets are worth different amounts of points depending how close the player is to the goal when they shoot). It would be possible to represent an entire basketball court as one region and then the rest of the stadium as another series of regions. It would probably make more sense to have multiple regions on the court and encode information on how they affect play into the regions to provide agents using this decomposition a better understanding of game rules. Such a representation dividing the areas of a court into many regions would most likely be generated by hand, but these metrics we present still apply to hand generated decompositions. In this specific example, the decomposition with more regions would be more useful.

## 6.2 Coverage Percentage

The percentage of free space covered by the decomposition of the map is another interesting simple metric to utilize. This metric should always be maximized to be at or near 100 percent. Coverage percentage is one of the differences that sets the complete decomposition methods (PASFV, HM, Delaunay) apart from the other algorithms that yield difficult to use or inefficient navigation meshes (SFV). The good decomposition methods will theoretically yield 100 percent coverage for any map; however, depending on the implementation, coverage might be a percentage point or two less than 100 percent, but these near perfect coverage decompositions are still usable.

This metric takes the summation of the area of each of the regions and then divides that value by the total empty area of the world that is being decomposed as seen in Equation 6.1 to generate this percentage.

$$\frac{TotalAreaOfFreeSpaceRegions}{TotalFreeSpaceArea} = CoveragePercentage \tag{6.1}$$

As we can see in the results from the coverage metric shown in Table 6.2, PASFV and HM both produce high coverage decompositions that fully define the unconfigured areas of the world. SFV produces a decomposition that is past the boundary

Table 6.2: The *Percentage of Decomposition* metric presented for each of the decomposition algorithms evaluated.

| Algorithm | PASFV | ASFV | HM |
|---|---|---|---|
| Coverage | 75% | 100% | 100% |

into unusably low coverage, which means that in most cases it would be discarded; however, we will continue to evaluate it in this work for the sake of completeness.

This metric has a straight forward application as decompositions with a coverage of under 95 percent should be discarded. This is due to the fact that low coverage decompositions will have holes in the navigation mesh that make agent navigation difficult, and they will produce noticeable spaces and dead areas that cannot be entered if the navigation mesh is being utilized for collision detection with the world. Decompositions that are above this cut off point will work for most purposes and the other metrics presented here should be used to choose between them.

## 6.3 Distribution of Area Sizes

The area covered by each region gives us an idea of the general size of the regions. It is possible to determine the average amount of space the agents and dynamic objects being used in a world will consume. By looking at the distribution of available space in each region it is possible to develop an understanding for how well objects can be placed into any given region. We are able to visualize the area of the regions by providing a histogram with the $x$-axis showing various groups of region sizes and the $y$ axis showing the number of regions that fall into each grouping. The shape of this distribution provides us with considerable information to use when comparing multiple decompositions. If there are many regions that are smaller than the average size of objects that are supposed to be placed into the regions, then there are going to be issues with objects lying across multiple regions. Because objects in multiple regions have to be tested against every other object in each region they inhabit this reduces the efficiency gains provided by information compartmentalization.

To calculate this metric two things are required: first the area of every region must be known and secondly, the area of the object or agent that is being compared against must be determined. This second value, $\alpha$, is critical for the generation of groups for the histogram. It typically represents the mean area taken up by dynamic objects that move between regions. The groups are created via the functions in Equations 6.2–6.3. This function allows the user to determine the size of the groups by setting $\Delta$ and the maximum number of groups the graph generates. This function has three stopping conditions. One for the case for values less than $\alpha$ (Equation 6.2). The second exit condition provides the critical first group values (Equation 6.3). Finally, the third exit condition produces the remaining categories up to the maximum the user requested (Equation 6.3). Groups 3–$m$ will start just larger than the maximum of the previous group (e..g., if $\alpha = 1$ and $\delta = 1$, group 1 is [0,1), group 2 is [1], group 3 is (1,2], group 4 is (2,3], and so forth).

$$
\begin{gathered}
S : SamplePopulation \\
\alpha : ThresholdAreaForComparison \\
n = \{1 \; to \; m\}, m = MaxNumberOfCategories - 2 \\
\delta = GroupSizeMultiplier
\end{gathered}
\tag{6.2}
$$

$$
s \in S \wedge s < \alpha \rightarrow < [0, \alpha)
\tag{6.3}
$$

$$
\begin{gathered}
\in S \wedge s \geq \alpha \wedge s \leq \alpha\delta \rightarrow [\alpha, \alpha\delta] \\
\in S \wedge s > \alpha\delta \rightarrow (\alpha n\delta, \alpha(n+1)\delta]
\end{gathered}
\tag{6.4}
$$

Table 6.3: The averages and standard deviations generated from the *Distribution of Area Sizes* metric.

| Algorithm | SFV | PASFV | HM |
|---|---|---|---|
| Average | 11.489 | 10.60 | 8.848 |
| Standard Dev. | 9.35 | 13.24 | 9.53 |

In our example decompositions, we see some interesting results for region size distribution. For the purposes of our analysis we are comparing these regions against a dynamic size object of one square unit. It turns out that three of the regions generated by both the PASFV and the HM algorithm are smaller than the target

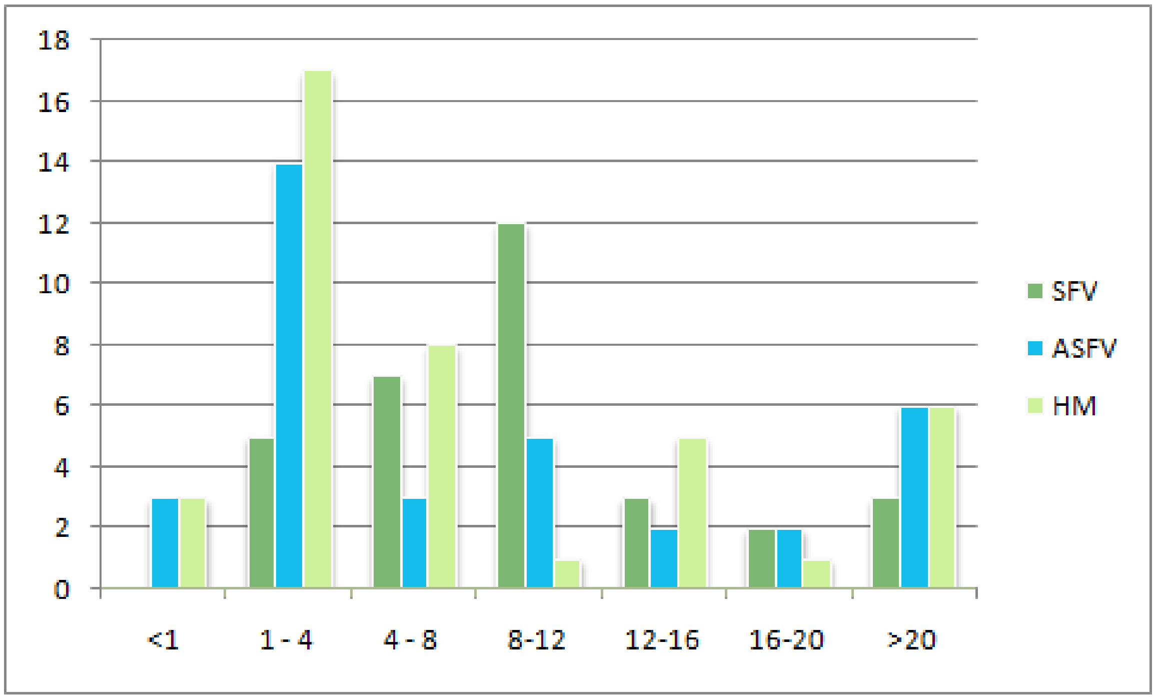


Figure 6.2: The results of our *Distribution of Area Sizes* metric. The x-axis gives the various size categories that each region fell into. We used a single square unit as our average agent/object size for this analysis. The y-axis indicates how many regions exist that fit into each size category.

object size of one unit which shows that these three regions are not very useful. Both HM and PASFV also produced a number of regions in which our test object would barely fit, which could also cause problems for information compartmentalization. Surprisingly, the simple SFV algorithm did the best on this test with no unusable regions and few borderline regions. This is partially explained by the fact that the SFV algorithm did not attempt to fill in the small areas between the obstructed configuration space areas—doing so would have resulted in the formation of more and smaller areas. In this case, SFV was actually assisted by its low coverage. When looking at the averages of the areas of the regions produced by this metric as shown in Table 6.3, the SFV and PASFV decompositions have similar values, which means that discounting a few small regions, PASFV does end up being almost as good at this metric as SFV, with the added advantage of providing a 100 percent decomposition of the world.

When comparing multiple decompositions of a world using this algorithm, the best decompositions will be ones that have few or no regions that have an area smaller than the average size of the objects the game engine would like to place into them. In addition, it is better if the regions are several times the size of the objects that will be placed in them, as this allows for multiple objects in each region and provides some room to maneuver the objects around in a region without colliding. Decompositions performing well on this metric will show distributions with a large number of large regions and a low standard deviation so that regions are generally the same size as each other. We realize that comparing the simple areas of the regions and the areas of the objects to be placed in them does not absolutely guarantee that the objects will fit in the region due to the potential for widely differing shapes, but we will further expand on the issues of placing objects into regions in a later metric.

## 6.4 Graph Degree Distribution

By conducting a graph complexity analysis on the connectivity of the different regions that make up the decomposition, we can gain understanding of how well the navigation mesh would work for its original purpose of navigation. The number of low connectivity (0-1 neighbors) regions is interesting and can tell us something about the underlying geometry we are modeling. If a level has been fully decomposed, which we will know from running previous metrics, then a region with only one connection represents a dead-end in the level geometry. As an example, suppose the level designer does not want there to be too many dead end branches that the player or AI could wander into. It is possible to compare multiple good decompositions of different underlying level geometries to each other by using the distribution of this graph and use this metric to select a level geometry that has fewer side passages. If the developer did in fact want many branching dead ends, say for a game built around a maze, then it would be possible to use this metric to pick a level design that contains many low connectivity paths. This is one of the few metrics that provides meaningful comparisons between decompositions of different sets of underlying geometry.

We create this distribution by iterating through each of the convex regions present in our mesh and then recording the number of adjacent regions. Adjacency is determined by whether or not two regions share a common edge of a user defined length threshold. We then use this adjacency data to create a graph of connectivity by plotting the adjacency information as follows. The $x$ axis shows the possible number of connections. The $y$ axis gives the number of regions having that many connections. For example, the values (3 , 5) would indicate that five regions contain exactly three connections to neighboring regions.

Table 6.4: The averages and standard deviations generated from the *Graph Degree Distribution* metric.

| Algorithm | SFV | PASFV | HM |
|---|---|---|---|
| Average | 2.96 | 3.43 | 2.488 |
| Standard Dev. | 1.179 | 1.53 | 0.588 |

As we can see in the data shown in Figure 6.3 and Table 6.4 from our example

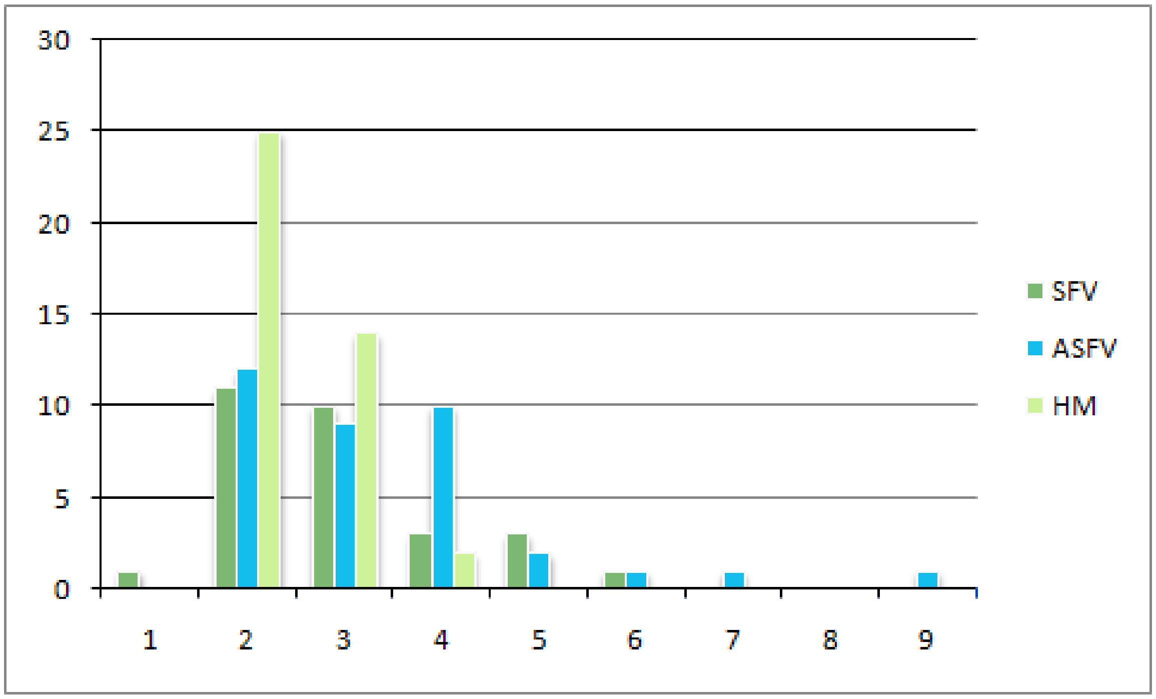


Figure 6.3: The results of our *Graph Degree Distribution* metric. The x-axis gives the number of connections to adjacent regions. The y-axis shows how many regions possess exactly that many connections to a neighboring region. Note: 0 is a possible $x$ value but was omitted in this chart due to no instances of it occurring in the data

decompositions, the HM generates an extremely high number of regions that only have two connections, indicating most of the paths through the HM algorithm generated decomposition are very linear. The SFV volume shows slightly better branching, with a few regions with four or more connections, but it also shows a problem dead end region which only has one connection. This dead end connection is especially problematic since we know that the world we are decomposing does not have any dead-ends. This anomaly is caused by the low coverage of SFV algorithm creating gaps and empty areas in the navigation mesh. Looking at this data, the best decomposition in terms of connectivity is the PASFV, which contains more than twice as many connections of size four or higher and one that contains nine neighboring regions indicating that there are many possible paths for agents to move around the world.

Examining this distribution tells us quite a lot about how well-connected the regions are that make up our navigation mesh. Obviously, regions that have no connections to other regions are worthless, as they are not accessible for use in agent navigation (note some games may intentionally create such areas similar to pit traps to force the user to respawn in that case the presence of zero connection regions is to be expected). These unconnected regions are a problem with the simpler decomposition algorithms such as the standard SFV algorithm. Such unconnected regions should be treated as red flags that indicate a decomposition should be immediately discarded. Ideally, regions would be well-connected with three or more neighboring regions in order to provide multiple alternate paths between regions and to avoid choke points and dead ends. Good decompositions on this metric will have strong right skew on the graph of connectivity.

## 6.5 Region Homogeneity Analysis

One of the common problems with decompositions occurs when the algorithm generates oblong shapes. For example, imagine a rectangle that is the length of the world but only one or two units wide. This rectangle would do well on all of the previously presented metrics that look at individual regions since it would have a large area size (due to its extent in one direction). It should theoretically be able to contain many objects, and its minimum interior angle will be 90 degrees, which is perfect for a quad-based polygon. Unfortunately, this rectangle would be almost useless for information compartmentalization since it is too narrow to successfully contain anything but the smallest of objects. Our Homogeneity Analysis is designed to detect such regions.

This analysis works by first gathering some basic information about each region of the decomposition. Each point in a region is examined and then the minimum and maximum $x$ and $y$ values are recorded. These are then used to determine the width and height of the region. Since these regions are required to be convex by the algorithms that generate them, we know via the definition of convexity that the lines that connect these maximum and minimum points will lie entirely within the region. We can consider these values to be a good indication of the maximum extents of the region. Now that we know the height and the width of the region, we can derive the number that we are really interested in—a ratio between the width and height of a region.

This ratio between length and width is slightly more difficult to calculate than one might expect. The larger of the two values (width or height) is divided by the smaller of the two. If the width is larger, then the resulting value is reduced by one to center the final number on zero. If the height is larger, then after dividing the height by the width, the resulting number is multiplied by a negative one, and one is added to the result again to center the results of this comparison at zero. Values below zero would indicate that the region is taller than it is wide, with values above zero indicating that the width of the region dominates the height. Values of zero would indicate that the object has the same width and height and forms either a square or a normal shaped triangle (right triangle), therefore values approaching zero are considered ideal. These values are then averaged and a box and whisker plot of the combined values for all of the regions is generated.

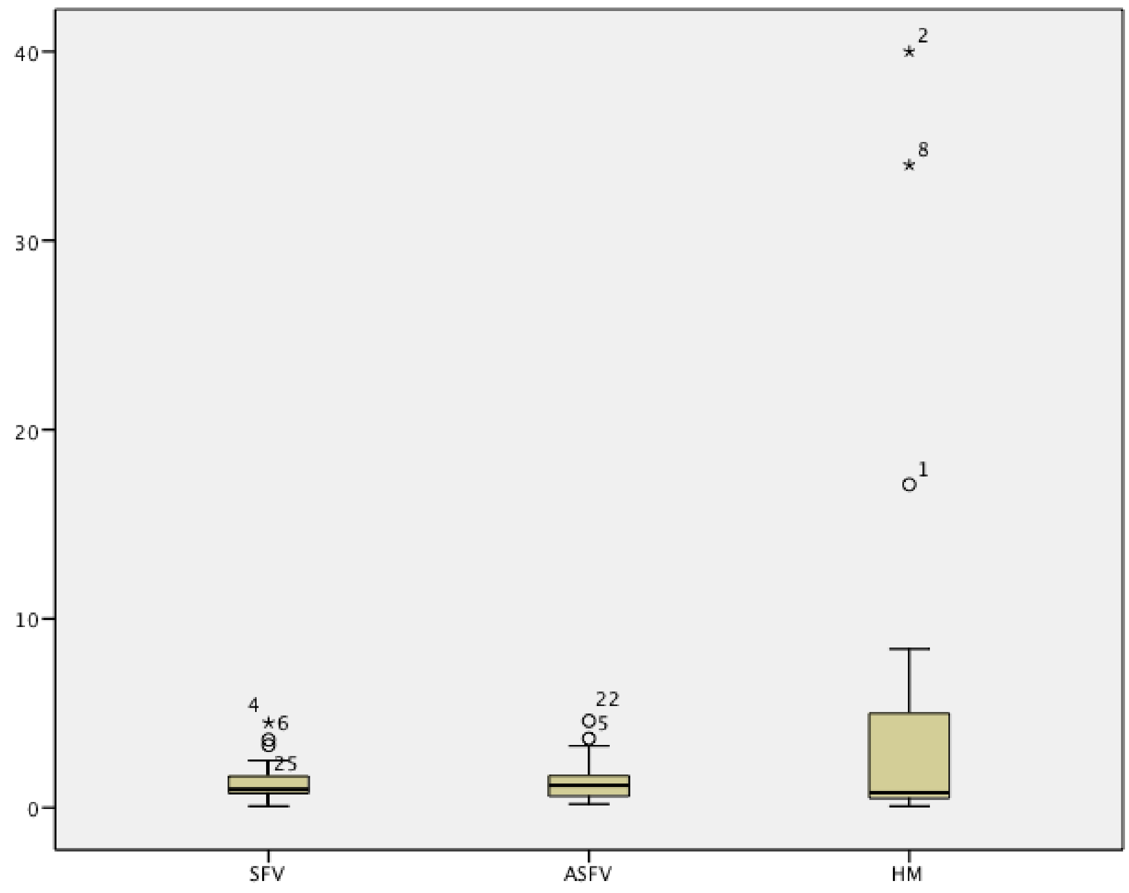


Figure 6.4: The results of our *Region Homogeneity Analysis* metric. The ratio between width and height is graphed via a box plot to produce this metric. Standard deviations are shown via the inner boxes; the average is shown as the middle line. The outer lines show the outer quartiles of the data. Outliers are marked via circles. Extreme outliers are marked via a asterisk.

This data provides us an interesting look at the homogeneity of the sample decompositions. As we can clearly see in Figure 6.4, the HM-based decomposition has major problems with outliers due to the large very wide and thin triangles that compose the top and bottom of its decomposition. Due to the lack of extreme outliers and the more balanced standard deviations we can say that the PASFV decomposition is the most homogeneous. Based on the results of this metric, SFV does generate a highly homogeneous decomposition, which is no surprise given its initial seeding with squares. For applications desirous of homogeneity and using purely axis-aligned bounding geometry SFV would probably be a good fit, but for better coverage and connectivity, PASFV would be better.

By looking at the shape of the box and whisker plot of the ratio between the length and width of regions, it is possible to get an immediate feel for how homogeneous the regions are. If the average of the data is approximately 0 and the standard deviation is low, then the regions are compact and approximate squares or right triangles—both of which are very good at compartmentalization of information. If there exist outliers or the standard deviation is large, then some of the shapes present in the decomposition are strongly elongated in one direction. This leads to regions that cannot fully contain objects and problems with information compartmentalization.

## 6.6 Common Vertex Distribution

The next metric helps to visualize how many potential trouble areas there will be for information compartmentalization. It works by looking at the locations of vertices relative to each other and then generates a histogram showing vertices that are duplicated among multiple regions.

By examining every vertex present in a decomposition, it is possible to find and graph how many other vertices share a single location. To find the base data for this graph, iterate through each vertex in the decomposition and check to see if it is equivalent to any other vertex (within some small margin of error). Record how many vertices share their locations with other vertices. This information is then graphed in the following manner. The $x$-axis indicates the clusterings of vertices at one point in space. The $y$-axis indicates how many times each of those clusterings occurred. For example, an $x$ of 1 and $y$ of 15 would indicate that there are 15 points that are uniquely located, and an $x$ of 4 and a $y$ of 8 would indicate that there are 8 points that share common locations in space with 3 other points.

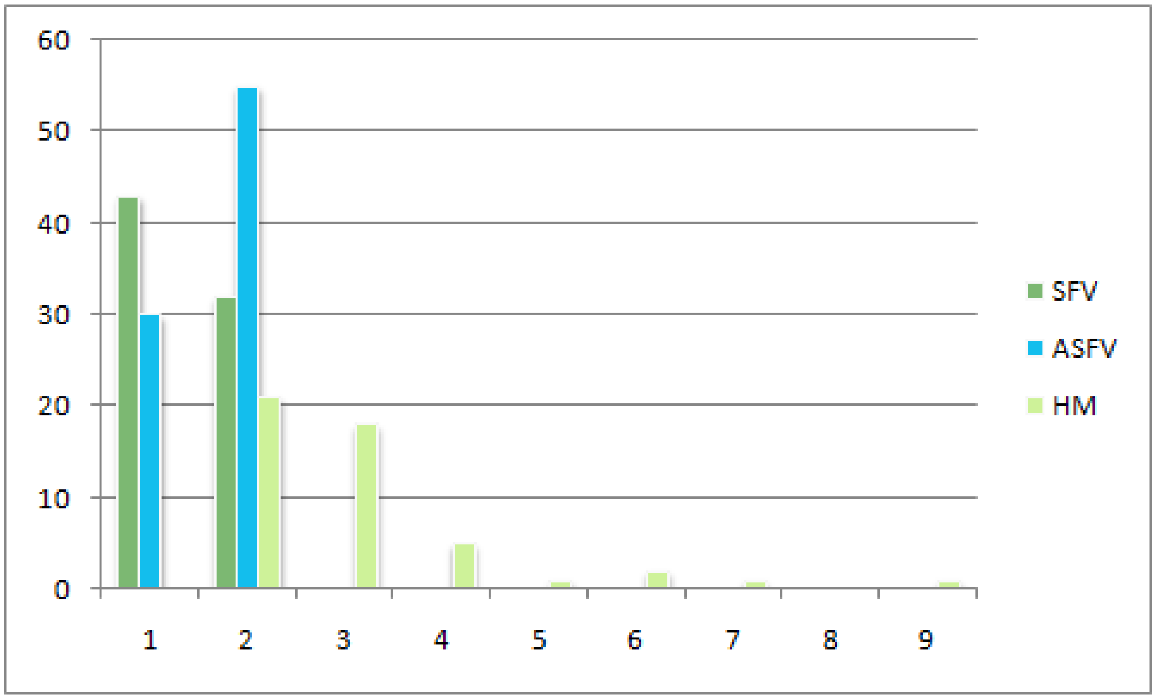


Figure 6.5: The results of our *Common Vertex Distribution* metric. The numbers of co-located vertices are given on the $x$ axis. The actual number of times this combination of vertices occurred is given on the $y$ axis.

Table 6.5: The averages and standard deviations generated from the *Common Vertex Distribution* metric.

| Algorithm | SFV | PASFV | HM |
|---|---|---|---|
| Average | 1.338 | 1.64 | 3.05 |
| Standard Dev. | .477 | .48 | 1.42 |

In the histogram showing the results of the Common Vertex Distribution shown in Figure 6.5 and the means presented in Table 6.5, we can clearly see that the HM algorithm has considerably more locations that are shared with multiple other vertices. In one instance the HM algorithm placed 9 vertices into a single location which will almost certainly be a problem area. In this case the PASFV algorithm is the clear favorite over HM and again wins out over SFV despite having to use smaller regions to attain higher coverage.

Decompositions that contain high numbers of locations where multiple points come together are a problem waiting to happen. These locations where many regions all share a common vertex are the problem areas for information compartmentalization. One of the issues experienced by triangle-based decompositions is that they tend to have more clusters made up of a larger number of vertices due to the ability of many triangles to converge onto a single point. One of the nice features provided by quad-based decompositions is that only 4 rectangular shapes can come together at a single free configuration space point. Ideally, the distribution graphs generated with this metric will show peaks in the low 1-4 range and have few higher categories. Due to collinearity of adjacent regions, good decompositions will have a high peak at $x = 2$.

## 6.7 Minimum Angle Distribution

By examining the distribution of minimum interior angles present in each region of a decomposition, we are able to see if there are any small corners that agents will not be able to enter without crossing into other regions. Regions with small interior angles have been known for a long time to present problems for information compartmentalization [13]. One of advantages provided by using the Delaunay triangulation algorithm is that it will attempt to maximize this metric. Unfortunately, despite attempting to maximize this metric, Delaunay is limited in how much it can accomplish due to its triangular base shape.

Calculation of this metric is straightforward: to do so, first iterate through every region in a decomposition and find the minimum interior angle of that region. Add all of the minimum angles to a list of angles. Then generate a series of numeric groups in the range (0,90] to drop the angles into. The best size of groups to use is entirely up to the user, but we have found 10 degree increments to be insightful. Then plot the data for minimum angles from the above list onto a histogram using the group size determined earlier for readability.

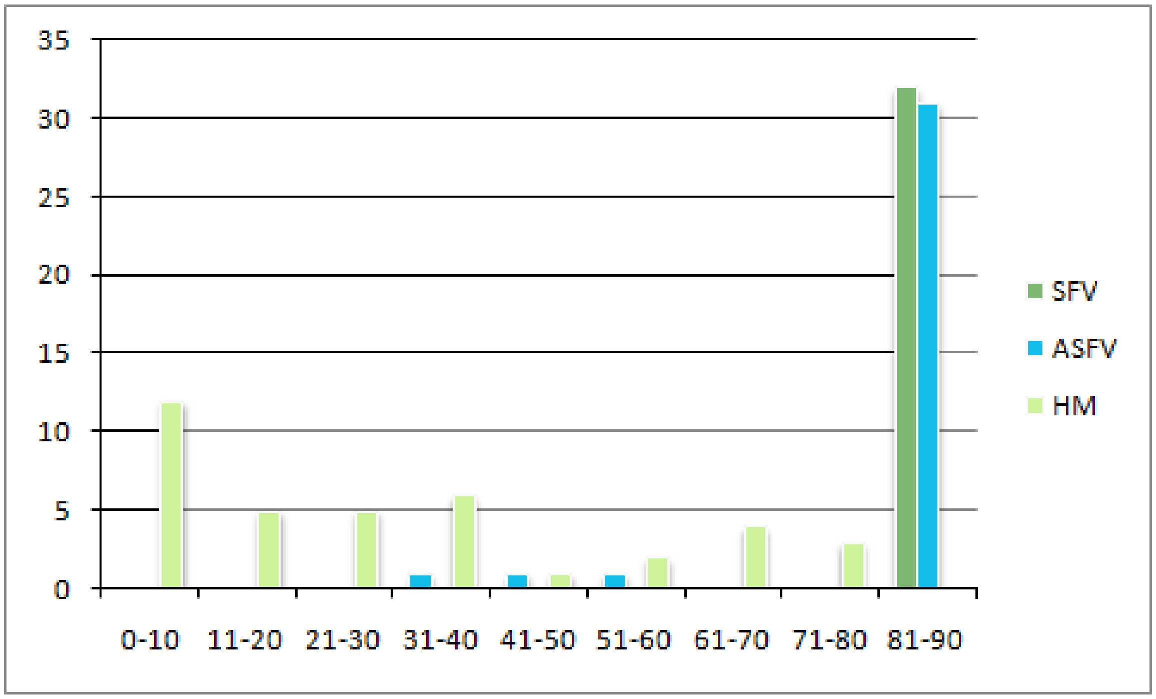


Figure 6.6: The results of our *Minimum Angle Distribution* metric. This histogram shows the distribution of the minimum interior angles of regions for different decomposition methods. The $x$-axis shows minimum angles using a group size of 10 degrees for viewing clarity. The $y$-axis lists how many angles of each decomposition type fell into that group.

The results of this metric, which are presented in Figure 6.6 and Table 6.6 , show us clearly that the HM decomposition of this world is not the best possible decom-

Table 6.6: The averages and standard deviations generated from the *Minimum Angle Distribution* metric.

| Algorithm | SFV | PASFV | HM |
|---|---|---|---|
| Average | 90 | 86.47 | 34.864 |
| Standard Dev. | 0 | 11.77 | 23.99 |

position. HM produced an abundance of very tiny interior angles, which means, it is filled with highly acute triangles. This will wreak havoc upon information compartmentalization. Once again SFV and PASFV were very close in this metric, which is no surprise given they use quads as their basic geometry. If SFV had fully decomposed the world, then it would be the best algorithm to use based on this metric, but, because it did not, PASFV is the preferred decomposition technique.

Using this metric is very straightforward: a distribution that has fewer outliers and has larger minimum angles is going to do a better job of information compartmentalization, and in previous work, decompositions with fewer small sharp corners have been shown to generate faster agent navigation paths with fewer sharp turns that require additional path smoothing [30]. For triangle-based decompositions, minimum angles of 60 degrees would be ideal as that would indicate all of the triangles are equilateral, while angles greater than 30 degrees are sufficient for most purposes. Higher order polygon decompositions generally will yield minimum angles of 45 degrees or above with an ideal minimum value of 90 degrees. In all cases, decompositions with one or more regions containing minimum angles of less than 10 degrees indicate that there exists very acute triangular shapes that are going to cause problems for agent navigation and dramatically reduce the ability of the decomposition to compartmentalize information.

## 6.8 Object Placement Distribution

The most complicated metric presented here is an exact analysis of each region's ability to contain various user-defined sized bounding boxes or bounding spheres. Unfortunately, to exactly determine how many of a given shape it would be possible to fit into a convex polygon we would have to solve a variation of the *Trunk Packing* problem (could also be considered a cutting stock problem) which is NP-Hard [21]. This means that we have to use an approximation to generate the data for this metric. The approximation generates close results that tend to underestimate, which in this case is better than an overestimation, since we need to be able to say with confidence that any given region can contain at least that many of the user-defined object. Our estimation works by laying a tight grid of objects over a region and then counting the number of bounding objects contained entirely within each of the regions. Any of the methods for approximating a solution to the trunk packing problem would work here as long as they err on the side of underestimates and the same estimation method is applied to every decomposition included in a comparison.

After determining how many of the user-defined bounding objects can be placed into each of the regions in the decomposition, this result can be graphed. Again, this data is best expressed in a distribution with the axes laid out as follows. The $x$ axis will indicate the possible number of bounding objects that a region could contain from 0 to some user defined maximum value. The $y$ axis will show how many times each of the possible conditions listed on the $x$ axis occurred. For example, a result of

(4,7) would indicate that there exists somewhere in the decomposition seven regions that by our approximation of the solution for trunk packing could contain 4 of the user defined volumes.

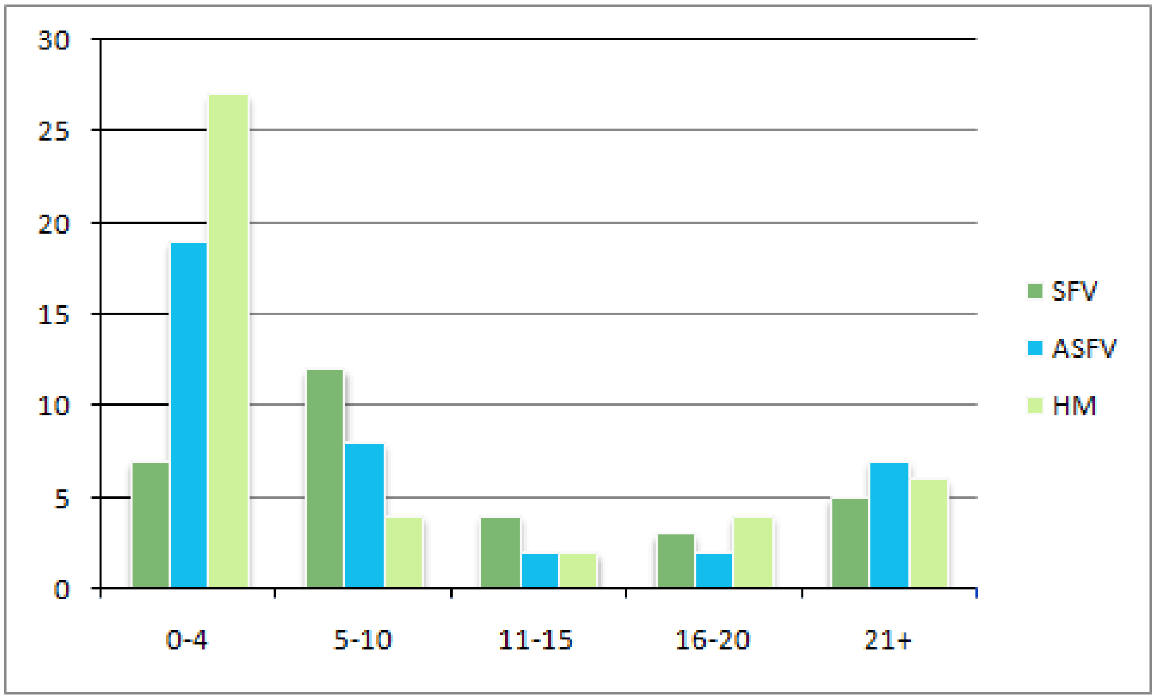


Figure 6.7: The results of our *Object Placement Distribution* metric. This histogram shows the distribution of the maximum amount of bounding spheres that can be placed in each of the regions generated by different decomposition methods. The $x$ axis again gives groups showing how many objects could be placed in a region. The $y$ axis shows how many regions had a maximum capacity of no more than the amount listed for the group.

Table 6.7: The averages and standard deviations generated from the *Object Placement Distribution* metric.

| Algorithm | SFV | PASFV | HM |
|---|---|---|---|
| Average | 13.68 | 10.06 | 8.79 |
| Standard Dev. | 6.36 | 6.69 | 6.88 |

The experimental data in Figure 6.7 and Table 6.7 shows that the PASFV decomposition can contain objects better than the HM-generated decomposition. Again, the SFV does yield good object containment numbers, but the PASFV is still a better choice, since that algorithm offers good information compartmentalization and complete coverage of free space.

Despite being a complex metric to calculate, the interpretation and application of this metric is actually very straightforward. If there are any regions that cannot be assured of at least containing a single bounding object, then there will be a problem for information compartmentalization and calculation reduction, since objects within that region are certain to be in other regions as well (i.e., existing in multiple regions simultaneously). Other problem areas are indicated by the presence of regions that can only hold a low number of objects, since those would cause problems if multiple objects attempted to move past one another. For these reasons, decompositions that minimize the number of regions that can only hold a few of the user-defined bounding objects are considered to be superior.

## 6.9 Decomposition Efficiency

After determining how effective the generated regions in a decomposition are at storing objects we move on to considering how efficient the decomposition is overall. Each environment that is being decomposed has a certain number of obstructed configuration space edges that define it, for example an empty environment is composed of the four edges that make up the bounds of the environment. The addition of configuration space obstacles to the environment will increase the number of obstruction edges present in the world. This number alone tells us how complex the world is and allows us to rate worlds according to their complexity. By combining the number of obstruction edges in the world with the number of free configuration space region edges present in a final decomposition we can assign a numeric value to how the efficiently the level was decomposed. This number represents how many free configuration space regions it took to decompose a world of any given difficulty.

To calculate this value the total number of the edges of obstructions in the occupied configuration space in the world is divided by the total number of free configuration space region edges in the decomposition. This will yield a number between zero and one. Values close to or at one indicate that one free configuration space edge is sufficient to decompose one edge of obstructed configuration space. Values approaching zero indicate that many free configuration space regions are required to properly decompose a single edge of obstructed configuration space. In this case a value of one would be ideal while lower values indicate that a less efficient decomposition. The results of running this metric on our sample level and the three decompositions of it are shown in Table 6.8.

Table 6.8: The efficiency with which each algorithm decomposed the world on a scale of zero to one.

| Algorithm | SFV | PASFV | HM |
|---|---|---|---|
| Efficiency | .40 | .373 | .378 |

The evaluation of this metric results in some interesting results. None of the decompositions presented here were highly efficient when decomposing this level. The SFV algorithm as usual generated a inflated score on this metric. As often happens in complex worlds the SFV decomposition has very low coverage. Since the world is not fully decomposed there are not as many free configuration space region edges present in present in the SFV decomposition, and that there are many edges of obstructed configuration space that have not been fully decomposed. This means that it will score well when dividing the high number of edges present a complex world by the artificially low number edges in the decomposition. Excluding the SFV decomposition since its score is deceptively high, we see that the HM decomposition scored the next highest followed closely by the PASFV decomposition. It is not surprising that the HM decomposition performed well on this metric since the HM algorithm works by generating triangles from the vertices of the obstructed configuration space and it would follow that every edge of obstructed configuration space is incorporated into a single region of free configuration space. The growth based algorithms can make no such assurances as it can take several free configuration spaces regions to fully cover an edge of obstructed configuration space as can be seen in the included PASFV decomposition in several places.

The Decomposition Efficiency Metric is particularly useful for several reasons. First, it serves as a logical check against the number of regions metric by providing some information about how hard the level is to decompose. If a decomposition has many regions and a low efficiency score than a better decomposition can almost certainly be generated. On the other hand if a decomposition contains many free configuration space regions but the efficiency score approaches one then the decomposition is acceptable and the level itself is just complicated. This metric is also excellent to use as a threshold when attempting to generate high quality decompositions. It is very easy to evaluate so it can be quickly and automatically calculated and multiple decompositions can be generated until a large sample pool has been created that meets the efficiency threshold. At this point the other metrics can be used to determine the best decomposition from this pool.

## 6.10 Navigation Mesh Diameter

The following three metrics are borrowed from graph theory and can be applied to navigation meshes with some slight adaptation. The first metric to originate in graph theory is the concept of the diameter of a graph. The diameter of a graph is measured as the greatest distance between any two vertices present on a graph. By calculating this figure we can determine the maximum search depth of most path finding algorithms for a decomposition as well as the compactness of the graph representation of the navigation mesh.

The first step to calculating this metric is to convert the navigation mesh to a graph representation: regions become vertices on the graph and the common edges or gateways between regions become the edges of the graph. Since this metric focuses on the abstract representation of the graph we do not need to worry about weighting the edges to represent geometric distances between vertices. Once the navigation mesh is converted to a graph representation find the shortest path from each node to every other node in the graph. From this collection of paths the longest shortest path is selected to generate the Diameter of the graph. If the graph has one or more disjoint vertices then the Navigation Mesh Diameter is considered to be infinite since no path exists to one or more nodes.

The lowest graph diameter indicates the most compact graph with the smallest search space so the lowest scoring decomposition is the best performer on this metric.

Table 6.9: The diameter of each of the navigation meshes.

| Algorithm | SFV | PASFV | HM |
|---|---|---|---|
| Diameter | 11 | 9 | 13 |

In this case that is the PASFV decomposition. SFV loses points due to low coverage in this metric since it is missing some of the edges which allow for shorter paths that are present in the complete PASFV decomposition. HM performs the worst in this case because the corners in this particular decomposition are filled with many small regions which makes the path from corner to corner longer than in a decomposition that fills each corner with a single polygon. HM will not always decompose a environment in this manner, but these collections of thin triangles in the corners are a common feature in HM decompositions.

The Navigation Mesh Diameter metric is like the Coverage metric a good tool to use for detecting and removing subpar decompositions from consideration in a final pool of similar possible decompositions after the other metrics have already been evaluated. Poor decompositions due to low connectivity or disjoint regions will have a very high or infinite Navigation Mesh Diameter which allows for the quick red flagging and removal of low quality decompositions. In addition, this metric can also be used as a threshold when generating a list of potential decompositions before submitting potential decompositions to a more computationally expensive metric.

## 6.11 Navigation Mesh Bottleneck Detection

The next metric looks at a navigation mesh's ability to move objects through the navigation mesh. This metric is best applied in world designs where there is a clear starting point in the world and another clear exit point somewhere else in the world. In figure 6.1(f) our sample level has been annotated with randomly chosen starting and ending locations to show this metric. The results of this metric trace the route from the start point to the ending point and then calculate the the most constricted connection between regions present along this path. This constriction indicates the maximum capacity to move objects or agents along this path before running into crowding and congestion issues and provides an upper limit for the size of objects that can follow this path.

The calculation of this metric is easier than calculating the maximum flow on a graph, which this metric is derived from, as we are primarily interested in only the path that connects the goal and destination region. This path should be calculated by that same navigation algorithm that will eventually be used in the game or simulation world. Once this algorithm has calculated the path of the decomposition it becomes a simple matter to travel down the path towards the goal and record the width of each region to region connection that the path moves through. The smallest of these recorded values is the potential bottleneck for this path and the width of it is returned for comparison to other decompositions.

Table 6.10: The width of the narrowest pass between the goal region and destination.

| Algorithm | SFV | PASFV | HM |
|---|---|---|---|
| Bottleneck Width | 1.40 | .7 | 1.6 |

When evaluating this metric the goal is to select the decomposition on which primary object flow path (start to end) has the widest possible bottleneck area. For this set of decompositions that is the HM decomposition. It is interesting to see that the PASFV algorithm which normally performs well on these metrics is so far behind the others, but if the decomposition in question is examined we see that the shortest path from start to goal passes through the highly constricted center area of the map between the two buildings, which explains this anomaly. On the SFV decomposition this narrow center area was not decomposed which changed the optimal path to be longer but somewhat wider path, while the HM algorithm decomposed the center in a manner that produced a wider passage.

The primary information provided by this metric is the evaluation of the suitability of each different navigation mesh to handle the primary traffic flow of the level. Decompositions with a higher rating on this metric will result in less milling around of agents or path finding errors. For this reason alone the metric is worth using, but it is capable of doing more. Instead of just calculating the bottleneck potential for the path from the start to the goal it is possible to calculate the bottleneck potential for the path between every possible pair of regions. This information can then be encoded into the navigation mesh under consideration and later used to do bottleneck avoidance path planning. This metric can generate information which can be used to improve the underlying navigation mesh in addition to evaluating it.

## 6.12 Navigation Mesh Maximum Total Flow

This metric is superficially similar to and derived from the previous metric, but tells us more about the ability of the decomposition to handle large crowd movements from a start point to a goal point. Unlike the previous algorithm which just determined the narrowest bottleneck when moving down the primary path between two points in a level this metric finds the total movement capacity between two points using all possible paths. This allows for the generation of decompositions, which maximize the total ability of the navigation mesh to move a crowd from the start to the goal positions by multiple paths.

As expected the calculation of this metric is considerably harder than calculating the narrowest bottleneck on a single path. Both this metric and the previous one start off the same way first the biggest bottleneck on the best path between the start and goal point is located. At this point the connection the bottleneck occurs at on the optimal route is marked as broken and edge that it represents is removed from the graph model of the navigation mesh. Furthermore every other region on this path has a bottleneck penalty applied to it equal to the bottleneck we just removed. This penalty represents the fact that the regions on this path have already committed to carrying a certain amount of traffic into the first bottleneck and cannot commit their full capacity for any other paths. Now the metric will attempt to locate additional paths from the start position to the goal point and continue doing so until it cannot find any more paths. Each time it successfully locates a path it remove the narrowest connection between regions and again applies that connection width as a penalty to all other regions on the path. If the cumulative penalties on a connection are greater the connection is wide then that connection is considered broken as well. Eventually every path from the start to the end position will be fully committed to carrying as much traffic as possible and all of the bottleneck connections will have been plotted. At this point the *Maximum Total Flow* can be calculated, this value is the summation of the width of the narrowest connection on every path we previously generated minus any penalties that might have been applied to that path. A simplified example of this showing two paths and the locations of bottlenecks on them is shown in Figure 6.8. This simple example would yield a result of 14.8. The full result set of this metric when performed on the three decompositions we are evaluating using the same start and end point as in Figure 6.1(f) is presented in Table 6.11

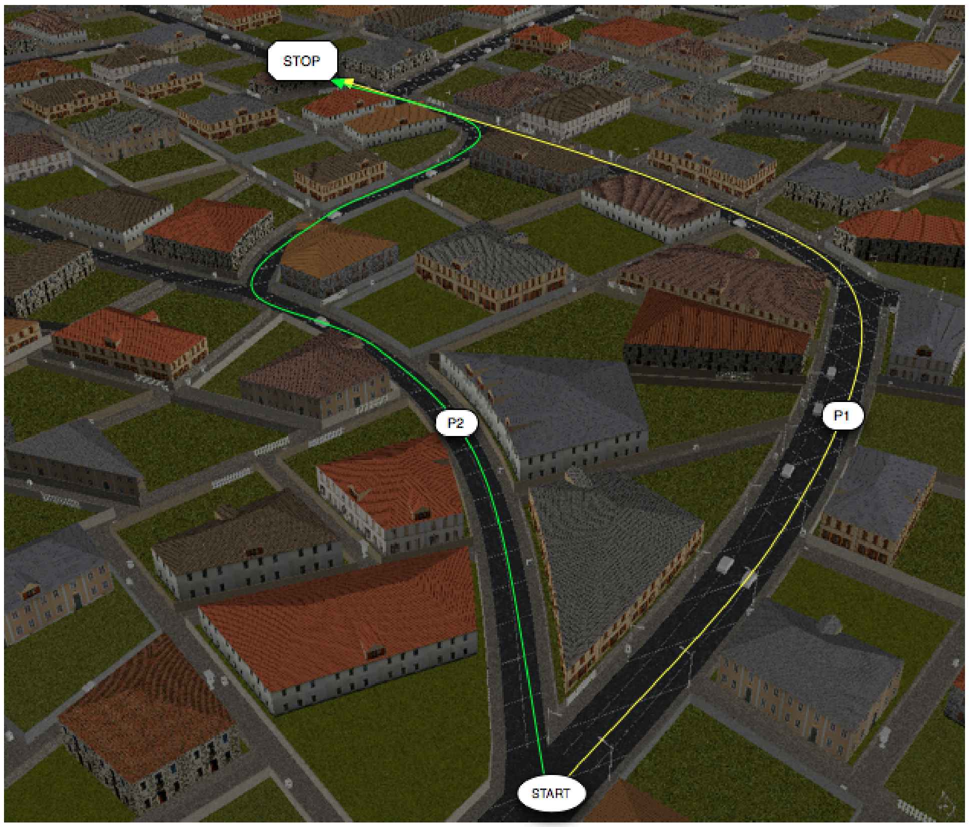


Figure 6.8: A simple level is show with both starting and goal points marked. The two unique paths to the goal are shown as p1 and p2. Both p1 and p2 would contribute in the generation of Maximum Total Flow metric despite p2 not being optimal.

In this case when alternative paths are open for consideration to avoid tight bottlenecks both PASFV and HM show considerable improvement. SFV does not see as much of an improvement, which is primarily due to the navigation mesh it generated

Table 6.11: The efficiency with which each decomposition could move agents through the environment.

| Algorithm | SFV | PASFV | HM |
|---|---|---|---|
| Bottleneck Width | 3.3 | 6.7 | 25.5 |

not being complete. In this metric HM is the clear winner, which is expected from examining the decomposition we are evaluating as HM produced regions with long adjacent edges.

The primary purpose of this metric is to look at how well any given navigation mesh can move agents and objects along all lines of travel in a level. Computing the metric is somewhat computationally complex and expensive but once done the information can again be placed into the navigation mesh for use later when agents are moving around the navigation mesh. This metric is especially important in games or simulations which involve moving large quantities or crowds through an area. For example, when simulating the evacuation of a building picking a decomposition with fewer artificial bottlenecks due to the configuration of the decomposition is important. This metric is something of an advanced metric and is not designed to be ran on every generated decomposition. It is a prime example of why some of the previous metrics are recommended as thresholds with which to generate a reduced pool of potential final decompositions. Using thresholds in this manner reduces the number of times more computationally intensive metrics such as this one need to be executed.

## 6.13 Metrics Evaluation

Once all of the metrics have been calculated, it is possible to do a final comparison and evaluation of the different navigation meshes under consideration. A good first step in evaluating the combined results of these metrics is to determine if there are any outliers that allow one of the possible decompositions to be excluded from consideration. For this comparison, Space Filling Volumes performed so poorly in the coverage metric (see Metric 2 discussion) it will be excluded from further consideration. The other two algorithms were competitive on each of the remaining metrics, which means we can analyze the results focusing on the two remaining decompositions. The second logical step in evaluating the suite of metrics is to assign a general point rating to each of the decompositions. The remaining two algorithms have been awarded points as follows: the outright winner of each metric is assigned one point, and if two algorithms tie or produce very similar results each of the tied decomposition techniques are assigned .5 points. Other schemes for assigning relative performance within each metric could also be used. The results of this comparison is shown in Table 6.12

Table 6.12: The number of metrics that each algorithm produced the best result in. Ties award .5 points to each of the tied decomposition algorithms.

| Algorithm | PASFV | HM |
|---|---|---|
| Average | 8.5 | 3.5 |

Upon evaluating the results of this comparison among the high coverage metrics the PASFV decomposition performs better in these metrics than HM decomposition having outperformed HM on more than twice as many individual metrics. This comparison assumes that metrics should be equally weighted. This equal weighting works well if there is a clear distinction between the results like the ones evaluated in this work, but if the results are more evenly divided (e.g., 6 vs 4 points) or if there is a tie then we need to move onto the third step in evaluating the results of these metrics. In the third evaluation step, we consider what the metrics measure and what advantages provided by the navigation mesh (e.g., information compartmentalization or collision detection assistance) are implemented in the program that will utilize the navigation mesh (see Table 6.12 for a summary of each algorithm). For example, the metrics related to the ability of each region in the navigation mesh to be used for information compartmentalization or collision detection can be ignored or delegated to secondary status if the target application is just going to use the navigation mesh for navigation. After discounting metrics which have features that are not considered important for this particular application, another table can be generated from the remaining metrics using the above scoring method. This table should show which decomposition is the better of the ones in comparison. If after excluding inapplicable metrics and regenerating the table there still is not a better decomposition then it is safe to conclude that the decompositions being evaluated are roughly equal and any of them can be selected.

By providing users of navigation meshes with a series of metrics with which to evaluate these meshes, it becomes possible to choose the best generated mesh from a collection of possibilities. These metrics will also serve to enhance the ability of automatic mesh generation tools to produce high quality meshes since there are now fitness functions these tools can work towards and compare against. It possible to automatically generate a very large set of potential navigation meshes using several mesh generation algorithms. Then this potentially large set of meshes can then be sorted based upon quality as determined by these metrics. Low quality navigation meshes can be quickly discarded while higher quality meshes continue to be processed. Eventually, through a combination of these metrics the best quality navigation meshes can be selected and then presented to a human to evaluate who can use the additional information provided by these metrics (i.e., the histograms and graphs) to select a final navigation mesh from this smaller set of top quality meshes. In addition, each of these metrics could be extended with minor changes to work with full 3D decompositions, instead of just 2D representations of 3D worlds or stacked levels. In conclusion, enhanced mesh quality should lead to better character navigation, better information compartmentalization, and improved collision detection with the world providing better and more organized information to character AI for motion planning and reasoning in interactive games.

# CHAPTER 7: EVENT-BASED SPATIAL DECOMPOSITIONS

To this point we have presented 2D (PASFV) and 3D (VASFV) growth-based spatial decomposition algorithms that are inspired by Space Filling Volumes. We have also presented an extension to enable these algorithms to adapt to dynamic environments. However, PASFV and VASFV still consume many processor cycles performing collision tests and validity checks as the free configuration space regions grow through empty space. Most of the time when a region is expanding via PASFV or VASFV it looks something like this: "Grow a unit. Did I hit anything? No, continue growing" repeated over and over again. Overall, there are very few interesting events that occur as these regions grow and it would be better if we could somehow calculate where the most interesting locations will be and then grow directly to them.

We propose a system by which these interesting event points can be located in advance and are used to inform the growth system such that every growth step each region takes will have a potentially interesting result. This algorithm, which we call the Iterative Wavefront Edge Expansion Cell Decomposition (Wavefront) technique (see Algorithm 7.1 is composed of five major parts (seeding, sorting edges, locating potential events, growth, and collision resolution) and is built on top of an existing PASFV or VASFV implementation. To begin with we have to alter one of the invariants we proposed and utilized for the PASFV and VASFV algorithms. The invariant in question previously stated that all of the growing regions we will place must continually be convex. In the Wavefront algorithm we change this to state that all existing regions must be in a convex state before growing any other regions. This allows regions to be temporarily concave while growing as long as they return to a convex state before they finish their growth. We retain the invariant that states that once a region has claimed an area that it must continue to occupy that area.

Previously, we would use either grid-based or obstruction influenced seeding as discussed in Appendix 8.2 to place a large number of unit sized quad (or cube in 3D) regions into the world. These regions would then iteratively be given the chance to grow and expand outward in the direction of the normal of each edge (or face in 3D—we will use edge for simplicity here since they are both effectively boundaries for occupied space). When using the Wavefront algorithm on our initial entry into the seeding phase we generate a list of *potential* seed points using the same methods. We then randomly select one of these seed points to use as our initial region. The other potential seed points will be retained for later seeding passes but will only be used if they are still in areas of unoccupied configuration space that are as yet unclaimed by any regions. If on later passes through the seeding phase this list is empty, we will attempt to refill it by looking for areas of unclaimed un-configured space adjacent to the regions we have placed, as in the PASFV and VASFV algorithms. If this list remains empty after that point then the Wavefront algorithm will terminate.

After a seed region has been generated we will proceed to the edge classification phase of the Wavefront algorithm. These next two phases are the most computationally intensive steps of this algorithm, and we only wish to perform them on valid regions that we know are going to grow. Therefore, we only grow one region at a time and we discard region seeds that are covered by earlier growth. This classification phase has no equivalent step in the versions of the PASFV and VASFV algorithms we presented earlier. During this phase we iterate through each of the edges of obstructed configuration space present in the world as well as any edges present in regions that we have already placed into the environment. We discard any edges whose normal faces away from the target seed point of the region as these edges are back facing and they cannot interact with the region. These edges are then sorted into categories based their relative spatial position when compared to the target seed location (+$x$, -$x$, +$y$, -$y$, +$z$, -$z$). Edges that span multiple categories are placed in the first applicable one depending on the evaluation order used in the implementation of the algorithm. Our implementation uses the following ordering +$y$, -$y$, +$x$,-$x$, +$z$, and -$z$. Any ordering will work as long as it is consistently followed.

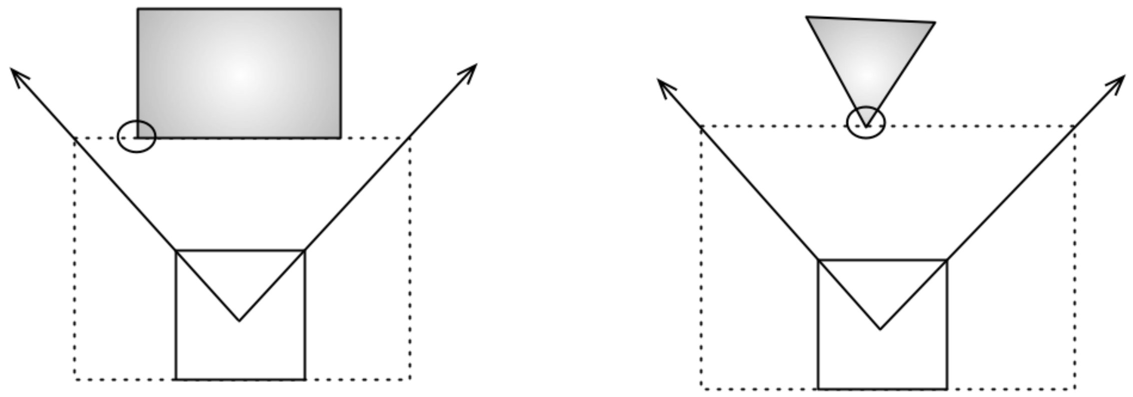


Figure 7.1: The two simple cases for event based spatial decompositions. Case (a) on the left shows growth towards a parallel element. Case (b) on the right right shows the discovery of an intruding vertex.

Once the edges have been sorted, we locate any interesting event points. Our region will have an edge that is perpendicular to each of the sorting classifications and whose normal matches the sorting classification (we will refer to this as the classification edge). By comparing the slope of each of our sorted obstruction edges to the appropriate classification edge we can determine in advance how the growing region would interact with the obstruction. This can be visualized by thinking of a radial half-plane sweep drawn from the initial seed point and then rotated in 90 degree arcs along each edge as shown in Figures 7.1 and 7.2. This sweep line will report the orientation of the edges it finds as well as the closest point on the edge to the initial seed point. The interactions between these edges of occupied space and the edge of the region we just placed can be reduced down to a series of cases, which happen to be identical to the cases present in the PASFV and VASFV algorithms.

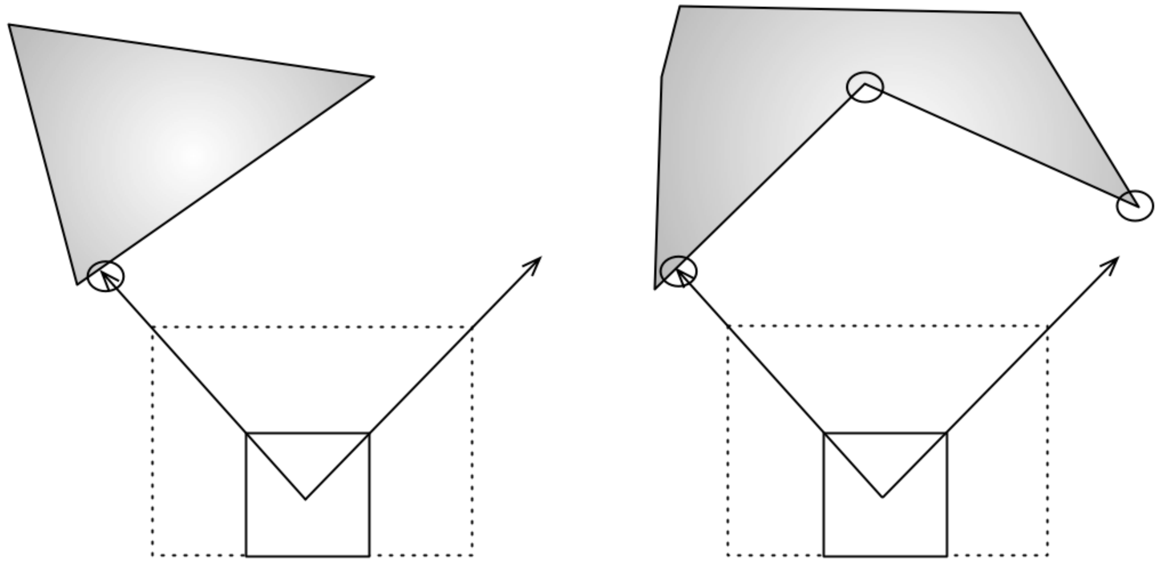


Figure 7.2: Two more complex cases for the Wavefront decomposition. In both cases the Wavefront sweep has discovered splitting events.

The first of these cases occurs if the tested edge is found to be parallel to the classification edge. This is the simple *base* case that we saw earlier in PASFV and VASFV algorithms as shown in Figure 7.1a. In this case, we will wish to move the classification edge such that it is adjacent and co-planar to the target edge. We accomplish this by calculating the closest point on the edge to the initial seed point of the region we are evaluating. We then log this point and the distance from it to the region's initial seed point as an event. Incidentally, since all of our placed regions only expose axis-aligned edges and our growing regions also only expose axis-aligned edges any events involving other regions of un-configured space will fall into this category.

A slightly more complicated case occurs when the edge is examined and found to be sloping inward towards the classification edge as shown in Figure 7.1b. In this case, we will only be able to grow such that the closest vertex of that edge lies on the classification edge without changing the slope of the classification edge. We cannot change this slope as this would result in previously claimed areas of unoccupied space being relinquished, which would violate one of our invariants. When the resulting growth for this case is examined it is revealed to correlate to the single vertex intersection case from PASFV and VASFV. This case is resolved by storing the location of the closest vertex on the edge under evaluation along with the distance to that vertex from the initial seed point of the region.

Finally, we come to the most complicated case, which might result in the potential addition of new edges to the growing region. In this case, the closest obstruction edge is sloping away from the midpoint of the classification edge, and it would be possible to move the classification edge such that one of its vertices could intersect the edge under consideration as shown in Figure 7.2. This equates to the potential splitting cases in PASFV and VASFV. In order to calculate where this split should occur the closest point on the edge under evaluation to the initial seed point of the region is found. This point is then stored as an event point again along with the distance between this point and initial seed point of the region. Additionally, we wish to store the two end points of the edge under consideration (assuming the closest point was not an end point) so that we will be able to increase the order of this region such that it adds a new edge that is adjacent to the entire length of the edge under consideration. However, instead of calculating the distance between each of these end points and the initial seed point we will treat them as if they are only slightly further away from the initial point than the point we are using to split. As we will see, the events will be processed based on the distance from the initial seed point of the growing region, and by altering the distance of these two end points we will ensure that region tries to fully encompass all of the space that is adjacent to the edge it splits on.

At this point we have a collection of potential events for our new region to grow towards; however, we need to do two things before we can begin growing. First, if the edges of the world are defined as some boundary conditions rather than actual occupied configuration space then events will need to be inserted to allow each region to grow outward to the edges of the world. Then this list will need to be sorted based on the distance between each event and the initial seed point of the region. This will allow us to process closer events first as we are more likely to reach them as further events are oftentimes unreachable due to the presence of more immediate obstructions.

With the completed event list for this region we are able to proceed to the growth phase of the Wavefront algorithm as shown in Algorithm 7.2. First, the growth rates of all of the edges of the region are reset to zero. Then, the first (closest) unprocessed growth event is selected and removed from the list of potential growth events. Then the distances that the edges of the region would have to move such that they reach this growth event are calculated. This is done by calculating the distance between the current location of the two (three in 3D) closest edges and the target growth location. This result is then broken down into its components ($x$, $y$, $z$) and if these values are positive they are set as growth rates for the edge or edges that have a normal that points toward the target growth event. Growth then proceeds as normal in PASFV and VASFV algorithms with one additional exception. Instead of growing each edge iteratively and then checking for any collisions or invalid growth conditions, all of the edges must be allowed to grow then a single set of checks for collisions or invalid growth states can be executed. This happens because there are splitting events that may result in invalid configurations if only half of the event (i.e., one rather than two edges are allowed to grow) is executed.

**Algorithm 7.1:** *Wavefront.Grow*()

```
List targetSeeds;
/* Generate a list of potential seed points and start the algorithm
   */
while worldModel.GenerateSeeds(targetSeeds) do
    while targetSeeds.notEmpty do
        if worldModel.isEmptySpace(targetSeeds.front()) then
            Region newRegion(targetSeeds.front());
            /* Pass the new region a list of edges to classify.  See
               7.2 */
            newRegion.growE(worldModel.getEdges());
            worldModel.regionList.push(newRegion);
        else
            targetSeeds.pop();
```

Once the region has finished growing any collisions with other regions or occupied configuration space objects must be resolved. Any vertices of the growing region that collided with an obstruction must be split, and the region must be converted to a higher order polygon. At this point either the PASFV or VASFV algorithm should be used to adjudicate the growth and handle any changes that this region requires. Since growth events are calculated in isolation with no consideration for other regions or potential obstructions it is possible that a collision will occur and that the region will have to retreat from a potential growth event. If this happens then the edge involved in the collision should cease further attempts to grow. The algorithm will then select another event to grow to, repeating this process until there are no more growth events or all of its edges have ceased attempting to grow due to collisions.

**Algorithm 7.2:** *Region.growE*()

```
List $targetEvents$;
/* Sort the edges in the world and classify them */
$classifyEdges( targetEvents)$;
$targetEvents.sort()$;
/* Generate a list of potential event points and start growing
   towards them */
while $targetEvents.isNotEmpty)$ do
    $zeroGrowthRates()$;
    /* This method selects the first event and calculates how this
       region would need to grow to reach it.  */
    $updateCurrentEvent(targetEvents)$;
    $grow()$;
```

After this region has finished growing, additional regions will be placed as per the seeding discussion earlier. If the algorithm enters the seeding phase, and is unable to place any new regions it terminates. This results in a collection of regions that is ready to serve as a navigation mesh. Additionally, if desired this collection of regions can be cleaned up by combining adjacent regions such that the result would still be convex. However, this combining is less necessary than in PASFV or VASFV due to the fact that the obstruction-based seeding algorithm will attempt fill each unoccupied area of the environment with a single region, and only add additional regions if they are required.

## 7.1 Wavefront Evaluation

Over the course of two experiments we have evaluated our Wavefront expansion spatial decomposition algorithm against several commonly used existing spatial decomposition approaches. We initially conducted an evaluation between the Wavefront algorithm and Trapezoidal Cellular Decomposition algorithm. We evaluated these two decomposition algorithms in a manner similar to that used for PASFV. We also verified that both decompositions covered the entire environment. For this experiment we randomly generated a series of obstructions and placed these obstructions into an environment in random positions. We evaluated these two decomposition techniques across five randomly generated environments—a sample of these are shown with the associated decompositions in Figure 7.3.

Table 7.1: This table presents the results when evaluating the Wavefront algorithm against Trapizoidal Cellular decomposition.

| Algorithm | Number of Regions |
|---|---|
| Wavefront | 22.4 |
| Trapizoidal | 31.4 |

The results of this comparison are presented in Table 7.1. We expected the Wavefront expansion algorithm to generate a decomposition providing full coverage and using few regions. This proved to be the case as the Trapezoidal decomposition re-

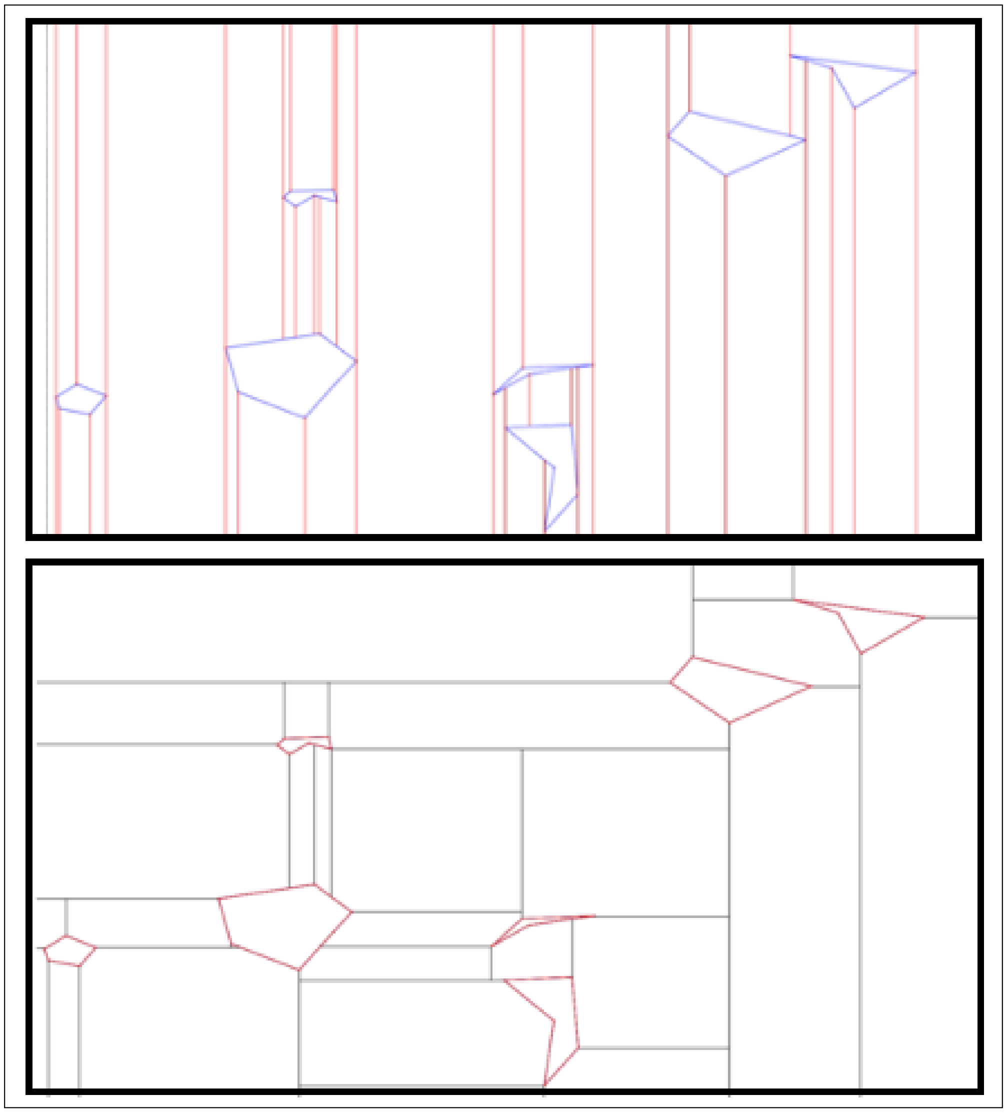


Figure 7.3: A comparison of a Trapezoidal Cellular decomposition (top) and a Wavefront decomposition (bottom) algorithm.

quired statistically significantly more regions to attain complete coverage (p-value less than .05) across our sample environments. This effect is caused by two factors. First, the direction of a Trapezoidal decomposition must be selected in advance (vertical or horizontal) and this direction must be consistently followed throughout the decomposition. However, some of the regions present in the environment (or any complex environment) would be better decomposed by one direction or the other.

This provides an advantage to the Wavefront algorithm as it can decompose around obstructed configuration space without any such pre-existing limitations. Secondly, the Wavefront algorithm produces few regions that would benefit from a cleanup and combining step post-decomposition (joining multiple convex regions into one larger, but still convex region), conversely the regions produced by one of the four base cases of the trapezoidal cellular decomposition algorithm would almost always benefit from such a step. This step is not part of the Trapezoidal decomposition algorithm and as such was not included in our evaluation. The successes in this preliminary experiment lead us to continue with the development of the Wavefront algorithm as well as encouraging us to finish the development of our metrics suite.

To complete our evaluation of the Wavefront spatial decomposition technique we compared it to a pair of commonly used navigation mesh generation techniques. First, we evaluated this technique against Delaunay triangulations as they often serve as the basis of more advanced spatial decomposition techniques, and produce good decompositions on their own. Additionally, one of our proposed metrics (Distribution of Average Minimum Interior Angles) is maximized in a Delaunay triangulation and as such this means the algorithm should perform well on this metric. Secondly, we are taking our generated Delaunay triangulations and using them to form Hertel-Mehlhorn decompositions. As we discussed earlier these decompositions are commonly used both in games and simulations and as such serve as a standard with which to evaluate against—as much as anything does due to the lack of a generally accepted best practice method for producing navigation meshes. For this evaluation, we continued to use a series of randomly generated levels. The style of these levels like those in Section 2 is inspired by the sample environments presented by Lavelle [42]. We expanded our test to use 10 such levels. Two of the base levels and all three of the decompositions generated from them are presented in Figure 7.4 for comparison.

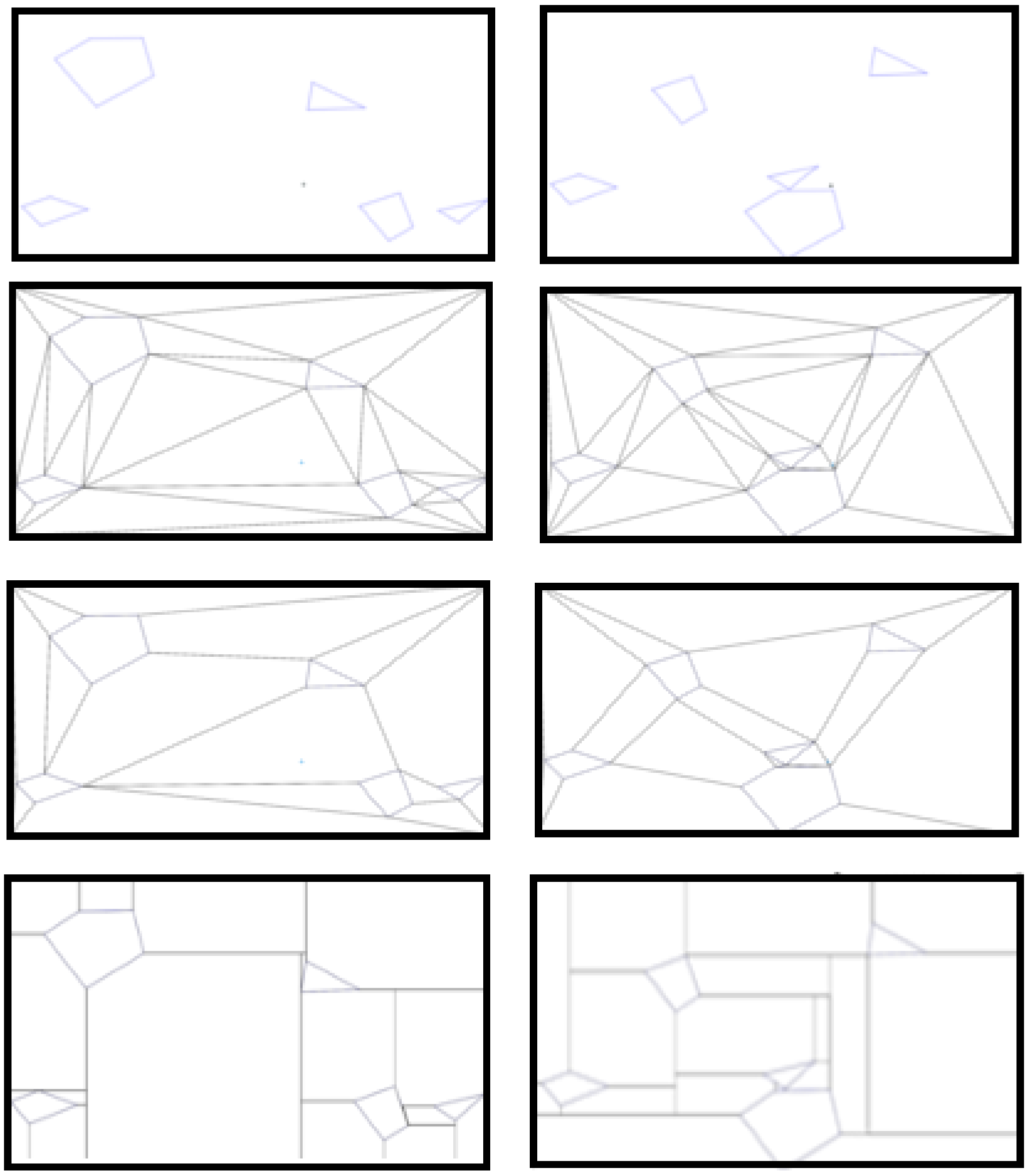


Figure 7.4: Two randomly generated environments (top), the Delaunay triangulations for said environments (second row), the Hertel-Mehlhorn Decompositions for the environments (third row) and the Wavefront decompositions (bottom).

For this set of comparisons we have deployed most of our available metrics suite only leaving out three inapplicable metrics. We initially inspected each decomposition to ensure that it provided a full coverage representation of the environment. We then evaluated the number of regions present in each type of decomposition. We calculate and present the average size of regions present in each of the different decomposition techniques, along with how "efficient" they are at decomposing the environment. The quality of the navigation mesh in general is evaluated by considering the average degree (connectivity) of regions present in the navigation mesh and the diameter of the generated navigation mesh. Finally, we examine the navigation meshes ability to store information and contain agents by evaluating the average minimum angle present in the environment (to detect slivers or other degenerate geometry), the average number of vertices shared between different regions, and the homogeneity of shape of the regions. Since we lack any associated game play data with which to base the general direction of travel through the levels on, we did not run either of the flow rate metric evaluations, and since we do not have active agents in the environment we did not consider the maximal object placement metric. The results of these metrics are presented in Table 7.2. Statistically significant (p-value of less than .05) results are marked with a *.

Our comparisons of different spatial decomposition techniques have yielded several interesting results. As expected both decomposition techniques that utilize higher order polygons yield fewer regions in the resulting navigation meshes than the technique that is restricted to triangles. Interestingly, the Hertel-Mehlhorn algorithm produced on average one fewer region per map—though this result is well within the standard deviation of the number of regions produced by the Wavefront technique.

Table 7.2: The final comparision results for event-based growth.

| Metric | Delaunay | Hertel | Wavefront |
|---|---|---|---|
| Number of Regions | 31 | 16.8* | 17.8* |
| Coverage | 100 | 100 | 100 |
| Region Average Size | 26233.08 | 49212.03* | 41020.92* |
| Average Graph Degree | 2.96 | 2.48 | 3.13* |
| Region Homogeneity | .477 | .564 | .2573 |
| Common Vertices | 3.43 | 1.985 | .59* |
| Average Minimum Angle | 19.683 | 69.39 | 83.59* |
| Decomposition Efficiency | 4.04 | 2.78* | 3.78 |
| Graph Diameter | 8.2 | 5.7 | 5.7 |

As follows from the previous metric both the Wavefront and the Hertel-Mehlhorn decompositions generate navigation meshes with larger region sizes than a Delaunay triangulation due to the fact that they both achieve complete coverage with fewer total regions. When we consider the results of the graph degree connectivity analysis we see one of the benefits of using the Wavefront algorithm over the others, as on average it produces a navigation mesh that is almost three standard deviations more interconnected than those produced by competing algorithms. The average of the minimum interior angles across the different navigation meshes also produces an surprising result. One would expect that since the Delaunay triangulation optimizes on this metric and that that Hertel-Mehlhorn decomposition builds on this triangulation that both would excel at this metric. However, that is not the case as the Wavefront algorithm performs statistically significantly better on this metric than the other two algorithms ($p$-value $< .05$). The Wavefront algorithm performs slightly below average in terms of the *efficiency of the spatial decomposition* metric. Given the performance poor reletive performance of the Wavefront algorithm in this comparison we would suspect that altering with a different seeding pattern the Wavefront algorithm might produce a better decompositions—examination of the generated decompositions reveals is the case. Additionally, we show in our *common vertex distribution* metric that the Wavefront algorithm generates decompositions with fewer regions converging to single points unlike Delaunay triangulations or Hertel-Mehlhorn decompositions, which in the worst case can be shown to be infinitely bad in this metric (Proofs 7.1 and 7.2 in Section 7.2 and Section 7.3). Finally, in terms of graph diameter both the Wavefront and Hertel-Mehlhorn algorithms outperformed the triangulation-based approach, though again this is to be expected since they use fewer and higher order polygons.

Overall, the Wavefront algorithm performs well on several key metrics (Minimum Interior Angle, Common Vertices, and Homogeneity) that measure the quality of use of the decomposition in terms of AI path planning and information compartmentalization. Across all other metrics it is competitive with the Hertel Mehlhorn algorithms and considerably better than a Delaunay triangulation.

## 7.2 Convergence of Free Configuration Space in a Delaunay triangulation

*Theorem 7.1* : There is no upper bound on the maximum number of convex regions which can come together at a single point in a Delaunay triangulation.

*Proof 7.1* : Suppose not. Then there exists an $n$ so that there is no Delaunay triangulation of any environment that has more than $n$ convex regions that intersect at a point. We will now construct an environment where every Delaunay triangulation has $n$ convex regions that intersect at the point $a$. We begin with the environment given in Figure 7.5(a), where the free configuration space is already decomposed into a Delaunay triangulation, with three convex regions intersecting at point $a$.

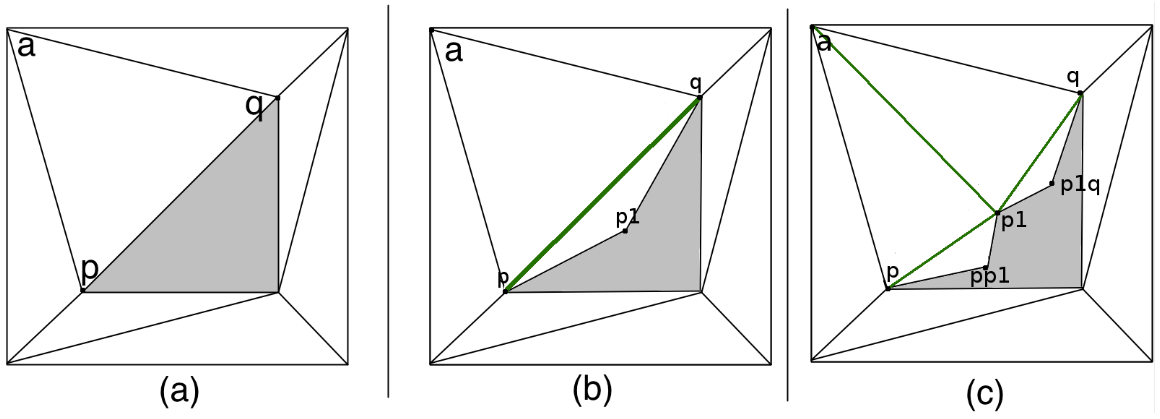


Figure 7.5: Simple level decomposed via Delaunay triangulation. The obstruction present in the level is shown in gray. We then see in (b) a alteration to the level to make it slightly more complex. Lines added by the Delaunay triangulation algorithm are shown in green

Now we construct a new environment by removing $\overline{pq}$ and adding two new edges, $\overline{pp_1}$ and $\overline{p_1q}$ where point $p_1$ is further from $a$ than points $p$ and $q$. Figure 7.5(b) shows this new environment and its Delaunay triangulation.

We inductively repeat this process on $\overline{pp_1}$ and $\overline{p_1q}$, creating new points $p_2$ (half distance between the subdivided line and point on the triangle furthest from point $a$) between $p$ and $p_1$, and $q_2$ between $p_1$ and $q$ that are further from $a$ than $p$, $p_1$, and $q$. Figure 7.5(c) shows the singular Delaunay triangulation for this level. Each iteration of this process creates two new triangles in the Delaunay triangulation as shown in Figure 7.5. We repeat this process $\frac{n}{2} + 1$ times. Every Delaunay triangulation of this environment will have $n + 1$ triangles intersecting at point $a$. Since $n$ was the upper bound on the number of convex regions intersecting at a point for any Delaunay triangulation, this is a contradiction, and there is no upper bound on the number of convex regions intersecting at a point in a Delaunay triangulation.

## 7.3 Convergence of Free Space in a Hertel-Mehlhorn Decomposition

*Theorem 7.2* : There is no upper bound on the maximum number of convex regions which can come together at a single point in a Hertel-Melhorn Decomposition.

*Proof 7.2* : This theorem proceeds in the same manner as the previous proof for Delaunay triangulations. It begins with the same supposition that if the worst case for this algorithm is bounded then we can can construct a counter example. We show in Figure 7.6(a-c) what happens when we start with a simple $n$-sided convex polygon and then add edges such that the polygon begins to approximate a circle. Consider what happens as the order of the polygon is increased by subdividing and pulling out one of the edges that faces towards the point $a$. We know that since a triangle can only be adjacent to one edge of a convex obstruction, and we are adding more edges we have to add regions of free configuration space. Regions cannot be combined into convex polygons because we are constructing the new edges so that such a combination would result in a concave region. We can repeat this subdivision an infinite number of times and each subdivision will result in one more triangle intruding more into free configuration space.

We then must show that no matter how the additional edges are distributed around the obstruction that the vertices of the world bounds will continue to deteriorate by connecting additional triangles to them. By the Pigeonhole principle, if there are $n$ pigeons and $k$ holes then there exists a hole with at least ceiling($\frac{n}{k}$) pigeons. Using this principle consider the $p$ edges from the interior shape connecting to four vertices on the outer edge of the environment since every triangle added to the decomposition

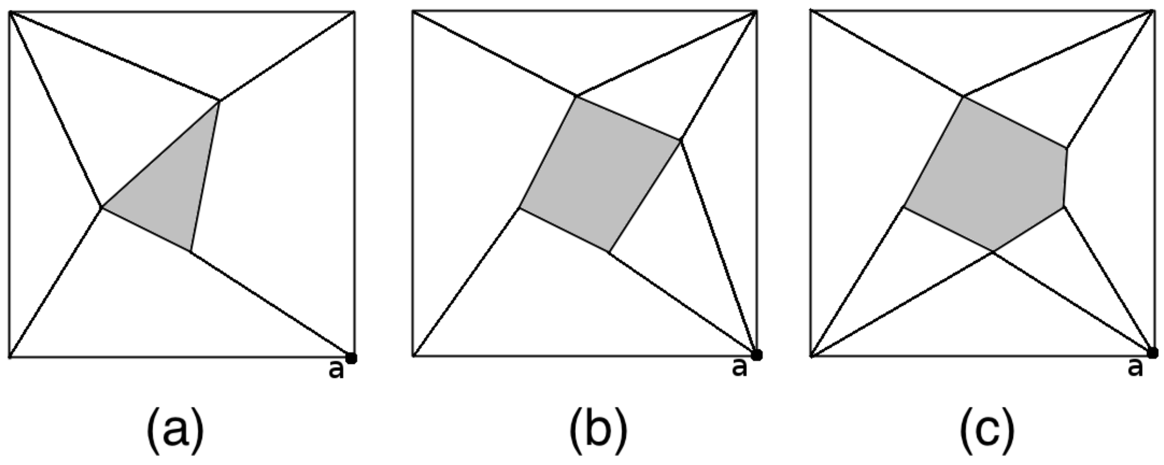


Figure 7.6: The results of subdividing a initial triangular obstruction (shown in grey) into a shape that more closely approximates a circle.

must anchor to the outside edge. No matter what decomposition we have, there exists a vertex on the boundary with ceiling($\frac{p}{4}$) edges adjacent to it. Let us say $n$ was our upper bound on the number of regions that could be adjacent to a vertex. Set $p = 4n + 1$ then the ceiling of $(p/4)$ is $ceiling(4n + 1/4) = ceiling(n + 1/4) = n + 1$. Therefore, the vertex has $n + 1 + 1$ regions adjacent to it. This contradicts our original assumption.

*QED* - Any theoretical worst case upper bound on the number of regions an agent can occupy in a Hertel-Mehlhorn Decomposition can always be made worse by subdividing obstructions into a higher order polygon therefore there is no upper bound on the worst case.

## 7.4 Characterization of Spatial Decompositions

Overall, the decomposition algorithms that we present in this document share many common features. They all are capable of consuming arbitrary geometry with minimal refinement and generating spatial decompositions. In particular, the algorithms that we present have no problems with axis-aligned geometry or other standard numeric edge cases, and do not require any mathematical adjustment to ensure that points in the input data do not lie on the same vertical or horizontal line. Trapezoidal decompositions experience problems with input data containing perfectly axis-aligned edges due to more than one point laying on the sweep line the algorithm utilizes at the same time. The decompositions generated by the PASFV, VASFV, and Wavefront algorithms are not unique to the environment (as a Delaunay triangulation would be) and like the Hertel-Mehlhorn algorithm many such decompositions can be generated from any set of obstructions. Unfortunately, the algorithms presented here like any convex spatial decomposition algorithms do not perform well when executed on geometry containing curves or spherical objects that have been approximated via a large number of short line segments. This occurs due to the fact that each of the short line segments that compose the approximation of the curve or sphere will require its own region to fully decompose the space next to it. These regions cannot be combined and this results in a large number of small regions in the final decomposition. However, these small regions would be present in any spatial decomposition that requires the output be convex as there is no way to combine them without having a free configuration space region that borders two edges of the same convex obstruction, which means one of the region must be concave. Finally, the quality of the decomposition is influenced by the quality of the input geometry. If there are many small rooms, restricted corridors, or triangular shaped rooms than there will be similar features present in the navigation mesh for that geometry.

# CHAPTER 8: CONCLUSIONS

Throughout this work we have focused on generating better representations of space in games using navigation meshes. Such representations establish and augment the perceptions of virtual characters in game and simulation environments. Oftentimes, all that characters know about the areas in which they operate comes from the representations provided to them. Using a navigation mesh representation provides an advantage over reasoning about the raw space present in the world or just considering the obstructed areas. This representation presents a smaller search space, which can improve the speed of path finding queries. Without these representations, the task of moving characters through virtual environments in reasonable manners becomes much harder, as the characters have a much larger search and planning space to consider.

## 8.1 Summary of Results

We have focused on the development of 2D and 3D growth-based spatial decomposition algorithms in order to generate high quality navigation meshes. In general, navigation meshes have become the representation of choice for game and simulation characters [46]. We have evaluated the navigation meshes that our algorithm generates against many of the most popular competing navigation mesh generation techniques (e.g., Hertel-Mehlhorn Decompositions, Delaunay Triangulations, Classical Space Filling Volumes, Trapezoidal Cellular Decompositions, and Path-Node Generation) in game and simulation environments and in all cases they show statistically significantly superior results. Our algorithms have in general fewer regions overall, produce a smaller navigation graph, generate higher degrees of coverage, and produce fewer degenerate or near-degenerate regions. Our mesh generation techniques present a high enough computational efficiency that they can be executed at run time to allow navigation meshes to be dynamically altered by runtime changes to the environment [27]. We presented a series of 12 metrics that allow us to evaluate the quality of generated navigation meshes in a manner that was previously impossible, and to extend our understanding of what makes a good navigation mesh. Finally, our new Iterative Wavefront Edge Expansion Cell Decomposition (Wavefront) algorithm is capable of generating spatial decompositions that are superior to existing and popular techniques for the creation of navigation meshes. In combination this work addresses our core hypothesis statement:

*The shape and extents of the unoccupied space present in a game or simulation environment can be reduced to a series of convex regions using a growth-based algorithm, which will result in a smaller set of regions that contain fewer degenerate or near degenerate regions than existing spatial decomposition algorithms with an average higher order of polygon/polyhedron.*

In order to prove this hypothesis, we have developed and evaluated three growth-based spatial decomposition algorithms. Through a series of experiments and proofs with these algorithms we show:

- That across multiple different environments our growth-based decomposition algorithms generate a set of convex regions that completely cover the environments.

- That through five test environments and numerous path length evaluations plotted on decompositions generated by the PASFV algorithm produces shorter paths with fewer sharp turns than Hertel-Mehlhorn decompositions and classical Space Filling Volumes[30].

- That it is possible to decompose 3D environments with the VASFV algorithm such that the resulting navigation mesh has fewer regions and higher coverage than other 3D decomposition algorithms[28].

- That using our event-based Wavefront algorithm it is possible generate spatial decompositions that are equal or better than Hertel-Mehlhorn and Delaunay triangulation generated decompositions in terms of number of regions and coverage.

- That based on our metrics suite the decompositions generated by the PASFV, VASFV, and Wavefront algorithms across multiple environments have a higher interior minimum angle, high degrees of homogeneity, fewer shared vertices between regions, and a higher degree of connectivity between regions than decompositions generated by existing commonly used algorithms—this results in navigation meshes with fewer degenerate or near degenerate regions.

Our results on the evaluation of the growth-based spatial decomposition algorithms that we developed *prove* our hypothesis statement. Utilizing these results and our hypothesis statement we can identify the four primary contributions of this body of work:

- The PASFV algorithm and the work we did to compare it to the Hertel-Mehlhorn algorithm and the Space Filling Volumes algorithm.
- The VASFV algorithm and the comparison we performed with it against the 3D Space Filling Volumes algorithm and the Path Node Generation algorithm.
- The metrics suite we proposed and evaluated that allow for better understanding of spatial decompositions.
- The Wavefront algorithm which allows for rapid spatial decompositions without wasted growth steps and the comparisons we ran on it against Delaunay triangulations, Trapizoidal Cellular decompositions, and Hertel-Mehlhorn decompositions.

In addition to our primary contributions, we have developed other algorithms and extensions that can be used in conjunction with our growth-based decomposition methods. Using our techniques, characters can be dropped into an unknown environment and can then construct navigation meshes for themselves or their teams as they explore their environment (see Appendix A)[31]. This exploration capability opens navigation meshes to previously closed domains involving procedurally expanding or on demand generated environments. We show that agents equipped with this algorithm will eventually generate a complete navigation mesh for their environment.

We also produced a pair of adversarial agent algorithms that enable an agent to use the navigation mesh to pro-actively plan modifications to the environment as shown in Appendix B. With these algorithms, agents are able to destructively modify their environments to either aid in the defense of a given area or find shorter paths through previously obstructed areas [29]. We evaluate several teams of agents running these enhanced algorithms in Capture the Flag scenarios and show that agents with a greater ability to dynamically alter their environments perform better than agents lacking one or both of the modification abilities.

Next, we have developed techniques that allow us to improve spatial decompositions by: varying the rate of decomposition in our growth-based algorithms, selecting possible seeding locations based on the obstructions present in the environment, and biasing the spatial decomposition based on the textures or terrain types present in the environment as shown in Appendix C.

Finally, we have shown a new application for the navigation meshes we produced by applying them to the problem of collision detection as shown in Appendix D. We have developed extensions that allow a navigation mesh to function as a collision detection acceleration data structure. With the extensions we show that the navigation provides superior collision detection acceleration than popular specialized data structures.

## 8.2 Future Work

We plan to extend the work presented here in several ways. First, we would like to evaluate some of the tradeoffs that are caused by using a texture aware spatial decomposition to create a navigation mesh. Such navigation meshes will contain more regions than ones generated without considering the textures or types of terrain being traversed. This will increase the amount of time required to search the navigation mesh, but alleviates the need to consider the local terrain the agent would be moving through (i.e., agents driving cars do not have to worry about remaining on the road instead they just use the “road” subset of the navigation mesh). It will be interesting to see if such a tradeoff improves path finding speeds, or if it merely reduces the burden on the agent creator since much of the path validation is done through the advanced navigation mesh. Secondly, we believe it should be possible to implement a form of portal culling in a navigation mesh (in particular, a collision enabled navigation mesh as it already contains all of the world objects) such that based on the cameras position in the navigation mesh it would be possible to quickly determine, which objects need to be rendered. The collision detection extensions we have presented and the work done by Heckel[35] shows some of the potential for a navigation mesh to influence and control the amount of information that must be considered by an agent. We believe this information compartmentalization can be taken much further. In particular, we feel that the perceptions of the agent could be quickly calculated using the navigation mesh in a manner similar to the portal culling we just mentioned. This should produce agents that act more on what they see and know about instead of omnipotent agents who are aware of everything happening in the world. Finally, we plan to look beyond convex region breakdowns of the environment. Our current research is focused on ways to decompose an environment into a series of convex regions, but we plan to examine the possibility of introducing limited concavity into select regions. Such limited concavity would allow full coverage decompositions of near degenerate environments (e.g., those containing parameterized curves or other surfaces with a high number of edges or faces) with far fewer regions than we currently require.

## REFERENCES

[1] Akenine-Moller, T., and Haines, E. *Real Time Rendering.* A. K. Peters, 2002.

[2] Axelrod, R. *AI Game Programming Wisdom 4.* Charles River Media, 2008, ch. 2.6 Navigation Graph Generation in Highly Dynamic Worlds, pp. 125–141.

[3] Axelrod, R., Berger, S., and Teler, E. System and method for the generation of navigation graphs in real-time, September 2008.

[4] Barraquand, J., Kavraki, L., Latombe, J.-C., Li, T.-Y., Motwani, R., and Raghavan, P. A random sampling scheme for robot path planning. In *Proceedings International Symposium on Robotics Research*, G. Giralt and G. Hirzinger, Eds. Springer-Verlag, New York, 1996, pp. 249–264.

[5] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (1975), 509–517.

[6] Bhattacharya, P., and Gavrilova, M. Voronoi diagram in optimal path planning. In *Voronoi Diagrams in Science and Engineering, 2007. ISVD '07. 4th International Symposium on* (July 2007), pp. 38–47.

[7] Boissonnat, J.-D., and Yvinec, M. *Algorithmic Geometry.* Cambridge University Press, Cambridge, U.K., 1998.

[8] Cai, C., and Ferrari, S. Information-driven sensor path planning by approximate cell decomposition. *Trans. Sys. Man Cyber. Part B 39*, 3 (2009), 672–689.

[9] Chazelle, B. Approximation and decomposition of shapes. In *Algorithmic and Geometric Aspects of Robotics*, J. T. Schwartz and C. K. Yap, Eds. Lawrence Erlbaum Associates, Hillsdale, NJ, 1987, pp. 145–185.

[10] Chin, Snoeyink, W. Finding the medial axis of a simple polygon in linear time. *Discrete and Computational Geometry 21*, 3 (1999), 405–420.

[11] Christian Science Monitor. Call of Duty series sales top 3 billion, Activision says, Febuary 2011. http://www.csmonitor.com/Innovation/Horizons/2009/1127/call-of-duty-series-sales-top-3-billion-activision-says.

[12] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. *Computational Geometry : Algorithms and Applications.* Springer, 1998.

[13] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. *Computational Geometry : Algorithms and Applications.* Springer, 1998.

[14] Delaunay, B. Sur la sphere vide. In *Classe des Sciences Mathematiques et Naturelle 7* (1934).

[15] Dirichlet, G. L. Uber die reduktion der positiven quadratischen formen mit drei unbestimmten ganzen zahlen. *J. Reine Angew. Math 40*, 1 (1850), 209–227.

[16] Dong, Z., Chen, W., Bao, H., Zhang, H., and Peng, Q. Real-time voxelization for complex polygonal models. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 43–50.

[17] Edelsbrunner, H. Voronoi diagrams and arrangements. *Discrete and Computational Geometry 1*, 1 (1986), 25–44.

[18] Edelsbrunner, H. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.

[19] Edelsbrunner, H. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete and Computational Geometry 5*, 1 (1990), 485–503.

[20] Eisemann, E., and Décoret, X. Single-pass gpu solid voxelization for real-time applications. In *GI '08: Proceedings of graphics interface 2008* (Toronto, Ont., Canada, Canada, 2008), Canadian Information Processing Society, pp. 73–80.

[21] Eisenbrand, F., Funke, S., Karrenbauer, A., Reichel, J., and Schomer, E. Packing a trunk: now with a twist! In *SPM '05: Proceedings of the 2005 ACM symposium on Solid and physical modeling* (New York, NY, USA, 2005), ACM, pp. 197–206.

[22] Epic Games. Current Technology Unreal Engine 3, Febuary 2010. http://www.unreal.com/.

[23] Fuchs, H., Abram, G. D., and Grant, E. D. Near real-time shaded display of rigid objects. *SIGGRAPH Comput. Graph. 17*, 3 (1983), 65–72.

[24] Fuchs, H., Kedem, Z. M., and Naylor, B. F. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1980), ACM, pp. 124–133.

[25] Goutsias, J., and Heijmans, H. J. A. M. *Mathematical Morphology.* IOS Press, 2000.

[26] Guibas, L., Holleman, C., and Kavraki, L. A probabilistic roadmap planner for flexible objects with a workspace medial-axis-based sampling approach. In *Intelligent Robots and Systems, 1999. IROS '99. Proceedings. 1999 IEEE/RSJ International Conference on* (1999), vol. 1, pp. 254–259 vol.1.

[27] Hale, D. H., and Youngblood, G. M. Dynamic updating of navigation meshes in response to changes in a game world. In *Florida Artificial Intelligence Research Society Conference* (2009).

[28] Hale, D. H., and Youngblood, G. M. Full 3D Spatial Decomposition for the Generation of Navigation Meshes. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)* (2009).

[29] Hale, D. H., and Youngblood, G. M. Adversarial Navigation Mesh Alteration. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)* (2010).

[30] Hale, D. H., Youngblood, G. M., and Dixit, P. Automatically-generated Convex Region Decomposition for Real-time Spatial Agent Navigation in Virtual Worlds. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)* (2008).

[31] Hale, D. H., Youngblood, G. M., and Ketkar, N. S. Using intelligent agents to build navigation meshes. In *Florida Artificial Intelligence Research Society Conference* (2009).

[32] Hart, P., Nilsson, N., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC4*, 2 (1968), 100–107.

[33] Hastings, E. J., Mesit, J., and Guha, R. T-collide: Temporal, real-time collision detection for mobile objects. In *Internation Conference on Computer Games: Artificial Intelligence, Design, and Education* (San Diego, California, USA, 2004).

[34] Hastings, E. J., Mesit, J., and Guha, R. Optimization of large-scale, real-time simulations by spatial hashing. The Society for Modeling and Simulation International.

[35] Heckel, F. W. P., Youngblood, G. M., and Hale, D. H. Influence points for tactical information in navigation meshes. In *Proceedings of the 4th International Conference on Foundations of Digital Games* (New York, NY, USA, 2009), FDG '09, ACM, pp. 79–85.

[36] Hertel, S., and Mehlhorn, K. Fast Triangulation of the Plane with Respect to Simple Polygons. In *International Conference on Foundations of Computation Theory* (1983).

[37] Holleman, C., and Kavraki, L. A framework for using the workspace medial axis in prm planners. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on* (2000), vol. 2, pp. 1408–1413 vol.2.

[38] Isto, P. Constructing probabilistic roadmaps with powerful local planning and path optimization. In *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems* (2002), pp. 2323–2328.

[39] Katevas, N. I., Tzafestas, S. G., and Pnevmatikatos, C. G. The approximate cell decomposition with local node refinement global path planning method: Path nodes refinement and curve parametric interpolation. *J. Intell. Robotics Syst. 22*, 3-4 (1998), 289–314.

[40] Kavraki, L. E., Svestka, P., Latombe, J.-C., and Overmars, M. H. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics & Automation 12*, 4 (June 1996), 566–580.

[41] Kobbelt, L., Schrder, P., che Wu, F., chun Ma, W., chou Liou, P., Laing, R.-H., and Ouhyoung, M. Skeleton extraction of 3d objects with visible repulsive force, 2003.

[42] LaValle, S. M. *Planning Algorithm.* Cambridge University Press, Cambridge, U.K., 2006.

[43] LaValle, S. M., Branicky, M. S., and Lindemann, S. R. On the relationship between classical grid search and probabilistic roadmaps. *International Journal of Robotics Research 23*, 7/8 (July/August 2004), 673–692.

[44] Lingas, A. The power of non-rectilinear holes. In *Proceedings 9th International Colloquium on Automata, Languange, and Programming* (1982), Springer-Verlag, pp. 369–383. Lecture Notes in Computer Science, 140.

[45] Lubiw, A. Decomposing polygonal regions into convex quadrilaterals. In *Proceedings of the first annual symposium on Computational geometry* (New York, NY, USA, 1985), SCG '85, ACM, pp. 97–106.

[46] McAnils, C., and Stewart, J. *AI Game Programming Wisdom 4.* Charles River Media, 2008, ch. 2.4 Intrinsic Detail in Navigation Mesh Generation, pp. 95 – 112.

[47] Meyer, F., and Beucher, S. Morphological segmentation. *Journal of Visual Communication and Image Representation 1*, 1 (1990), 21 – 46.

[48] Millington, I. *Artificial Intelligence for Games.* Morgan Kautmann, 2006.

[49] Murphy, R. R. *Introduction to AI Robotics.* MIT Press, Cambridge, MA, USA, 2000.

[50] Ogniewicz, R. Skeleton-space: a multiscale shape description combining region and boundary information. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (New York, NY, USA, 1994), IEEE, pp. 746–751.

[51] Ramsey, M. A Unified Spatial Representation for Navigation Systems. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)* (2009).

[52] Ramsey, M. *AI Game Programming Gems 8*. Charles River Media, 2010.

[53] Ratcliff, J. W. *AI Game Programming Wisdom 4*. Charles River Media, 2008, ch. 2.8 Automatic Path Node Generation for Arbitrary 3D Environments, pp. 159–172.

[54] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach*. Pearson Education, Inc, 2003.

[55] Samet, H. The quadtree and related hierarchical data structures. *ACM Comput. Surv. 16*, 2 (1984), 187–260.

[56] Samet, H. An overview of quadtrees, octrees, and related hierarchical data structures. *Theoretical Foundations of Computer Graphics and CAD 40*, 1 (1988), 51–68.

[57] Samet, H. Applications of spatial data structures: Computer graphics, image processing, and gis. *Addison-Wesley Reading* (1990).

[58] Samet, H. The design and analysis of spatial data structures. *Addison-Wesley Reading* (1990).

[59] Schneider, P. J., and Eberly, D. H. *Geometric Tools for Computer Graphics*. Morgan Kaufmann, 1998.

[60] Stentz, A. Optimal and efficient path planning for partially-known environments. In *IEEE International Conference on Robotics and Automation* (New York, NY, USA, 1994), IEE.

[61] Thibault, W. C., and Naylor, B. F. Set operations on polyhedra using binary space partitioning trees. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1987), ACM, pp. 153–162.

[62] Tokuta, A. Motion planning using binary space partitioning. In *Intelligent Robots and Systems '91. 'Intelligence for Mechanical Systems, Proceedings IROS '91. IEEE/RSJ International Workshop on* (Nov 1991), pp. 86–90 vol.1.

[63] Tozour, P. *AI Game Programming Wisdom*. Charles River Media, 2002, ch. 4.3 Building a Near Optimal Navigation, p. 171.

[64] Tozour, P. *AI Game Programming Wisdom 2*. Charles River Media, 2004, ch. 2.1 Search Space Representations, pp. 85–102.

[65] Voronoi, G. M. Nouvelles applications des parmetres continus a la theorie des formes quadratiques. premier memore: Sur quelques properties des formes quadratiques positives parfaites. *J. Reine Angew. Math 133*, 1 (1907).

[66] Voronoi, G. M. Nouvelles applications des parmetres continus a la theorie des formes quadratiques. premier memore: Rescherces sur les parallelloedres primitifs. *J. Reine Angew. Math 133*, 1 (1908).

[67] Wegner, S., Oswald, H., and Fleck, E. 3d watershed transformation on graphs. K. M. Hanson, Ed., vol. 3338, SPIE, pp. 714–725.

[68] Zhang, L., Chen, W., Ebert, D. S., and Peng, Q. Conservative voxelization. *Vis. Comput. 23*, 9 (2007), 783–792.

# APPENDIX A: USING AGENTS TO DISCOVER NAVIGATION MESHES

There are some problems with using a navigation mesh in a virtual environment. In order to generate the navigation mesh the configuration of the obstructions present in the world must be known in advance. This will cause a problem if the target virtual environment is being procedurally generated at game initialization or even worse if the world is being procedurally expanded as players or agents move outside the existing world bounds. In addition, when agents are provided with a full navigation mesh at the initiation of a game or simulation it gives them full knowledge of the environment. This makes sense if the agent is moving through an area they would logically have traversed before (e.g., a guard in a building would be familiar with the building), but does not make sense for agents that should be ignorant of the areas they are entering (e.g., a rescue worker entering a building they have never been in before). Such extra knowledge can cause the agents to behave in ways a person would not behave and this reduces the believability of the agent.

These problems have prevented the use of navigation meshes in situations where the layout of the virtual world was unknown or unknowable prior to guide agents moving through the world. We will present an algorithm that provides a solution to these problems and allows for the dynamic construction of a navigation mesh based on information gathered by agents in the world. We do this without sacrificing the benefits a navigation mesh provides to agents. At the same time agents are moving around through the world building and updating the navigation mesh they are querying and planning based on their current understanding of the navigation mesh. We accomplish this by using the Dynamic Adaptive Space Filling Volumes (DASFV) algorithm combined with sensors that are integrated into each agent to detect incorrect classifications of space in the navigation mesh versus the actual world geometry.

This section presents the Navigation-Mesh Automated Discovery (NMAD) algorithm, which allows agents to discover the navigation mesh for a game level while traversing the level. This algorithm works by making an initial assumption that the world is empty and generating an appropriate navigation mesh for this empty world. As one or more agents move through the world, each agent will detect and report any obstructing geometry (objects) they encounter. If this newly discovered geometry is not present in the navigation mesh then the Dynamic Adaptive Space Filling Volumes (DASFV) algorithm will update the navigation mesh. As these agents move through the world discovering geometry, the current decomposition (a decomposition is a breakdown of obstructed and open space in the game world in a navigation mesh) will eventually converge on the ideal decomposition.

Navigation-Mesh Automated Discovery is an extension of the DASFV algorithm to represent the limited knowledge each agent or group of agents' posses about their local environment. The NMAD algorithm as shown in Algorithm A.1 begins by initializing an empty navigation mesh composed of a single region that covers all possible un-configured (walkable) spaces present in the world. It also incorrectly classifies all of the configured (obstructed) space areas present in the world as un-configured space. While this navigation mesh is inaccurate, it is the most accurate navigation mesh possible given no additional knowledge about the world. The accuracy of the naviga-

**Algorithm A.2:** The NMAD algorithm loop

```
/* The NMAD algorithm is generally called from inside the agent
   class, and is assumed to have access to all of the agent local
   variables */
/* Check to see if any new obstructions are within range of the
   agent */
List FoundPositiveSpace;
for NewPosSpace in World do
    if Agent.CanSee(NewPosSpace) then
        FoundPositiveSpace.add(NewPosSpace);
if FoundPositiveSpace.size() != 0 then
    /* We found new positive space areas */
    for NewPosSpace in PositiiveSpaceList do
        /* Insert the positive space into the navmesh */
        NavigationMesh.insert(NewPosSpace);
    /* Regrow the affected areas with DASFV */
    NavigationMesh.regrow();
    /* Rebuild the connectivity */
    NavigationMesh.reconnect();
else
    /* Clean up the navmesh instead */
    NavigationMesh.cleanup();
```

tion mesh will improve with the addition of obstructed space locations discovered by the agents.

Once the initial (one region) navigation mesh has been constructed, the agents present in the world can use it for navigation as shown in Figure A.1(a) and A.1(b). Each agent present in the world is modeled as having a individually defined detection range to notice obstructions. As these agents move through the world, they will encounter obstructions as shown in Figure A.1(c).

When a new obstruction enters the detection range of an agent then one of two things will happen depending on how the world is represented. If the obstructions were constructed in a monolithic manner, (e.g., one obstruction represents an entire

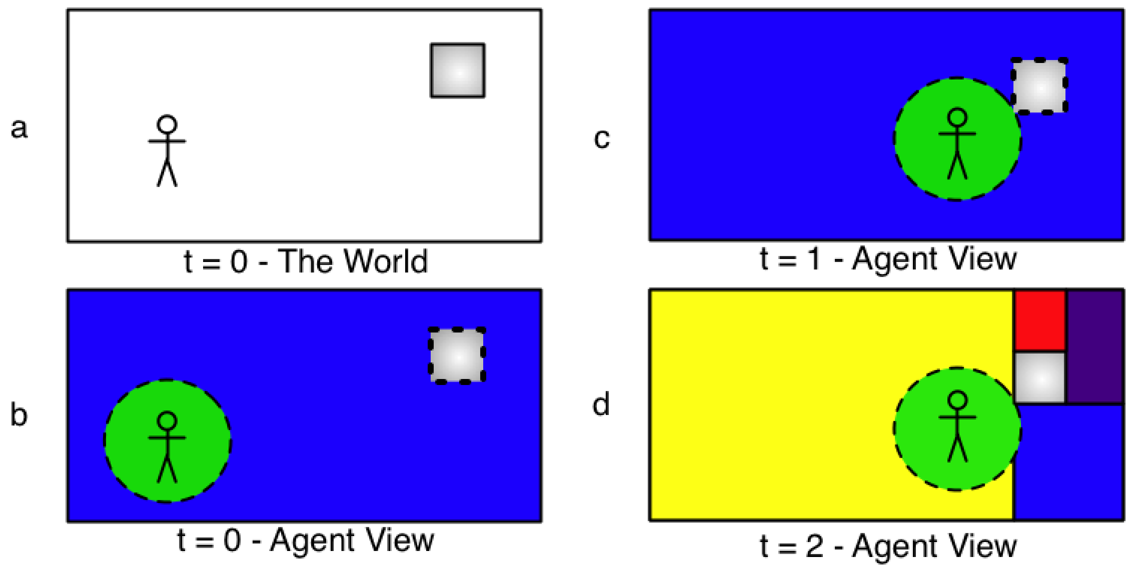


Figure A.1: This collection of images shows a sample agent traversal and discovery of obstructed space. In image (a) we see the actual state of the world. The agent is present in the lower left corner and there is a single positive space obstruction (shaded gray) in the upper right. (b) shows the initial view of the world as the agent perceives the world through the navigation mesh. The agent's detection area is shown as the (green) circle. Since no obstructions have been discovered, the navigation mesh is a single region covering the entire world (blue). (c) shows the agent moving and discovering a obstruction. At this point the obstruction will be added to the navigation mesh. (d) shows the status of the navigation mesh after the addition of the obstruction discovered in (c).

building) then the obstruction will need to be sliced into smaller components. We do this slicing because the detection range of an agent is representative of the agent's ability to see the world, and it does not make sense that by seeing one corner of a building the agent would become aware of the entire extent of that building. This can either be done in advance by creating preset splits in the object or at runtime by carving off sections of an obstruction via polygonal subdivision. However, if the obstructions are not constructed in a monolithic manner and instead are created from smaller building blocks then the blocks can be directly consumed by NMAD without subdivision. For dynamically generated worlds the obstructions list will be derived from the components used to create the world and can be subdivided based on how they would be rendered.

Once an obstruction has been identified then DASFV is used to insert the obstruction into the existing navigation mesh with the minimal possible disruption to the navigation mesh as shown in Figure A.1(d). DASFV works by first locating areas of un-configured space that intersects the area of the obstruction we are adding and then removing them. The obstruction is then inserted into the navigation mesh. The un-configured space regions adjacent to the ones that were removed are then allowed to reseed the world with more regions to fill the newly vacated areas. These newly placed regions then grow as much as possible and can generate more regions if needed to ensure that all of the newly vacated space is fully decomposed.

The algorithm then waits for another obstruction to be detected. Even if all of the obstructions present in the world have been detected, the algorithm can continue to run, at which point it becomes a form of localized DASFV as only changes in the world geometry that pass within the agents detection area will be reflected on the navigation mesh. This means that every agent on the map is not instantly aware of a new door being created in a wall or a passageway being closed off by rubble.

Finally, we have made three improvements to the base NMAD algorithm. First, when no changes are detected in the world the navigation mesh quality can be improved by combining adjacent regions when the resulting new region would be still be convex. Doing this results in a smaller, more compact navigation mesh while amortizing the cost of these improvements across multiple agent update cycles. This reduces the search space present in the navigation mesh and thereby speeds up queries. Secondly, more than one agent can feed information into the NMAD algorithm. It is possible for many agents to simultaneously search for obstructions rather than one, and by exploring the world with multiple agents it is possible to converge on a perfect decomposition faster. Finally, two or more separate navigation meshes can be maintained along with separate groups of agents who query and update just their own navigation mesh. This produces the effect of creating two or more teams of agents each with their own unique understanding and view of the game environment.

We performed a series of evaluations to asses whether the NMAD algorithm correctly built a navigation mesh based on an agents observations of the surrounding configuration space obstructions. To perform these evaluations we constructed a sandbox test environment. This environment consisted of 7225 meters square (85m * 85m) of open space that the agent could traverse. This is roughly the size of two football fields placed side by side for comparison. This world was then randomly populated with obstructions. The obstructions were composed of three different sizes of cubes (1 meter, 2 meter, 5 meter). The cube placement was restricted such that two cubes cannot overlap each other. An agent was then placed into the world. This agent had a detection radius of 10 meters for obstructions. The test agent moved at a rate of 1.25 meters per second. At this speed traversing the level along an edge would take would take 68 seconds.

The agent randomly chose destinations in what it believed to be unoccupied space somewhere in the level. Navigation to these randomly selected points was controlled by two distinct methods. The first local navigation method was used when both the target location and the agent were located in the same region of the navigation mesh. In this case, the agent will move in the direction of the destination at a normal walking pace. The second form of navigation occurs when the target point location and the agent are in different regions of the navigation mesh. In this case, the agent searches the navigation mesh to locate a path from its current location to the goal region. This path is then stored and the agent will move through the centers of the shared edges between connected regions on this path. When the agent is in the same region as its target and it enters local navigation mode. It is worth mentioning that for simplicity of implementation the agent uses a breadth first search to find a path rather than a more complicated best first search algorithm.

Our agent implemented the NMAD algorithm presented in this dissertation to update the navigation mesh as it moved through the world. We performed 10 passes through this world where we measured the error present in the navigation mesh in the form of incorrectly classified regions. One of these passes through the world is illustrated in Figure D.2. We assumed that all of the free configuration space and discovered configuration space obstructions would be correctly decomposed because the basic algorithm underpinning NMAD generates near perfect coverage (except for completely disconnected regions which will not be represented in the navigation mesh) spatial decompositions to use as navigation meshes. Recall that initially, and until it learns otherwise, the NMAD algorithm considers all unknown space to be free configuration space. This means that the only incorrectly classified regions would be obstructed configuration space areas that the agent has not yet discovered. Initially, since the agent starts with no knowledge of its surroundings all of the geometry in the world was incorrectly classified giving our algorithm a one hundred percent

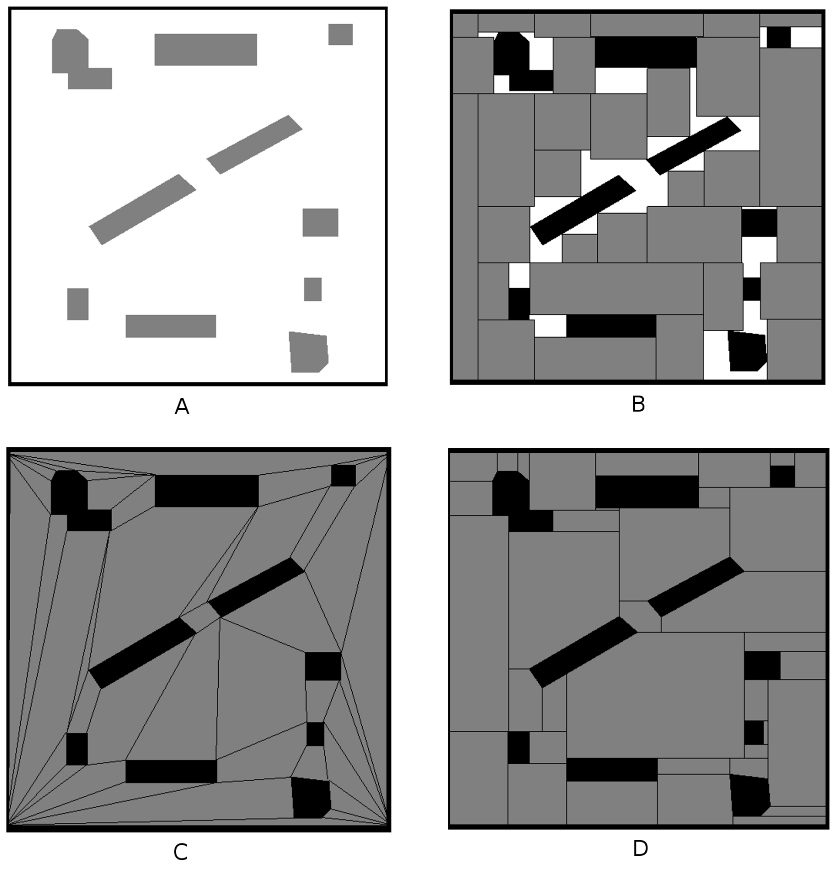


Figure A.2: This collection of images was taken during one of the test runs of the algorithm. The agent is the small gray figure in the middle of the green circle. The green circle represents the range of the agent's ability to detect errors in its navigation mesh. The randomly colored areas in the figure are regions of the navigation mesh. The obstructions are visible as boxes sitting above the decomposition. Image (a) shows the initial state of the algorithm. In this image the navigation mesh (blue) assumes that the entire world is walkable and can be represented as a single region. In Image (b) the wandering agent has encountered the first configuration space obstruction. Image (c) shows NMAD as it is the process of building new regions for the navigation mesh. The grid texture is free configuration space that has not yet been claimed by any decomposition. Normally this section of the algorithm would run almost instantly, but the it was intentionally slowed down to capture this image. Image (d) shows the final navigation mesh after all of the configuration space obstructions have been discovered and correctly classified.

misclassification rate at time zero. We then measured the misclassification percentage at ten second intervals while the agent wandered the world on each of the 10 passes we performed.

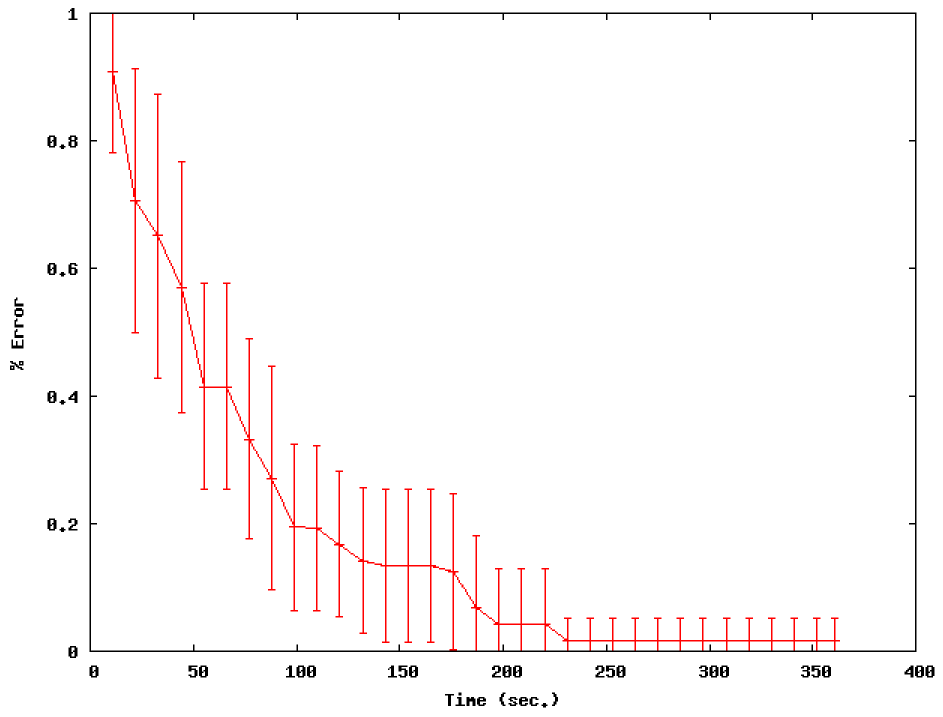


Figure A.3: A graph showing the convergence of incorrectly classified configuration space obstructions to zero over time averaged across 10 random walks through the world by an agent. The y-axis gives the error as a percentage of incorrectly classified configuration space in the world while the x-axis shows the traversal time. The bars on the graph provide the standard deviation across the multiple agent traversals; the standard deviation is rather large because the random nature of the agents movement can quickly discover all the obstructions in the configuration space or this process can proceed very slowly.

The results of this experiment are presented in Figure A.3. The incorrect classification quantity converged on zero taking on average 225 seconds.

After conducting ten walks through the same world using a random walking agent, we then implemented an agent who performs a spiral pattern search from the center of the world outward. This agent running the NMAD algorithm was then allowed to traverse five randomly generated worlds. These worlds contained between six and twelve randomly placed and sized obstructions. In addition, the agent's speed was increased to 3.6 meters per second in order to reduce the time required to run this experiment. The graph shown in Figure A.4 shows the results of this experiment. The agents spiral pattern was designed such that the entire world would pass through the agent's visibility radius so if NMAD is working correctly all space in the world will be correctly classified. From this graph, we see that NMAD will discover and correctly classify all of the space in the world within 90 seconds.

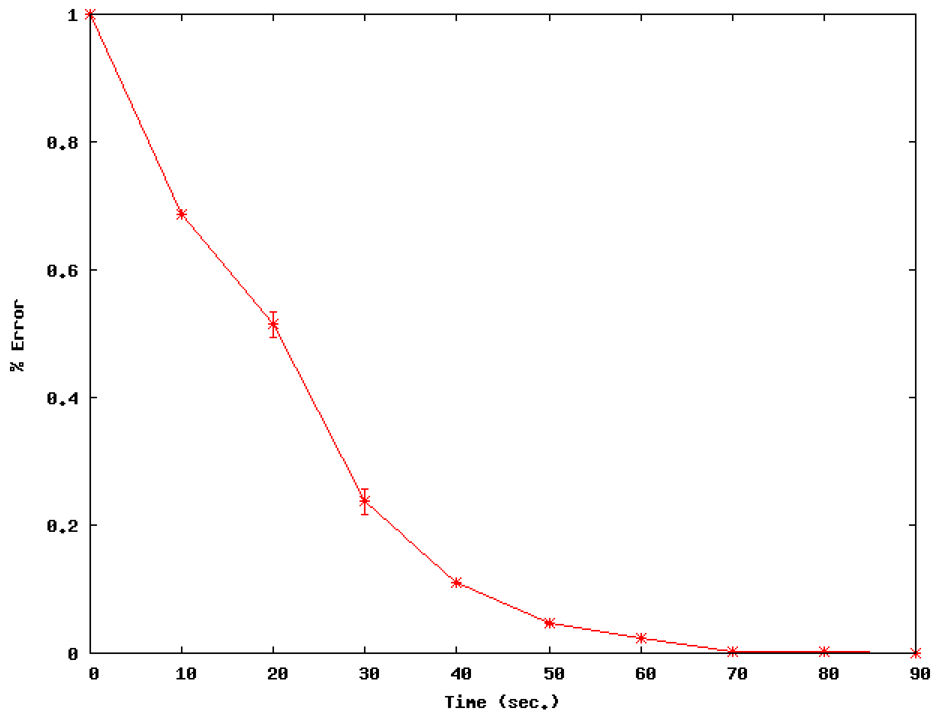


Figure A.4: A graph showing the convergence of incorrectly classified space to zero over 5 spiral searches through randomly generated worlds. The percentage of misclassified configuration space obstructions present in the world is shown on the y-axis while the x-axis shows the agents travel time in seconds.

The primary purpose of these two experiments is to show that the NMAD does correctly classify all of the space present in a game world. In both the random pathing and the random world generation experiments the misclassification in the navigation mesh eventually reached zero. By converging to zero error across many different traversals of the world, we show that the algorithm does consistently and reliably classify all of the space in the game world.

# APPENDIX B: INTEGRATING MESH ALTERATION INTO PLANNING

Our previous work with dynamic navigation mesh alteration has focused on allowing virtual characters to adapt to changes in the environment initiated by the player. We now present an algorithm that allows these characters to initiate changes to the environment. The Adversarial Navigation Mesh Alteration (ANMA) algorithm contains two component algorithms. The first subcomponent (attacker) of the ANMA algorithm slots into most heuristic-based path finding algorithms to improve agent navigation via the conversion of obstructed configuration space into additional free configuration space regions when it would benefit the agent. The second subcomponent (defender) is a higher level extension to the agent's reasoning ability that highlights and suggests regions in a navigation mesh or navigation graph representation, which it would be advantageous to the agent to remove. This removal is designed to reduce the number of potential approaches to a position the agent is supposed to defend. These algorithms work best, and in the case of the defender subcomponent require, an adversarial context for the agents to fully utilize them in planning.

## B.1 Path Planning Through Obstructed Configuration Space

The first part of the ANMA algorithm (attacker) attempts to answer the question of "When should an agent plan to use its ability to destroy obstructed configuration space to assist with path planning?" To answer this question the ANMA algorithm provides a simple extension to popular path planning algorithms such as A* or D* which allows them to consider the effects of altering the navigation space representation when determining the optimal path the agent should take to reach his goal.

Doing so allows the agent to create new paths which can be shorter than paths entirely in existing free configuration space regions.

Consider the example of a building that the agent wants to enter, which the agent is standing next to. The area around the building is fully described in the agent's navigational representation, so it is rather trivial to plan a path around the building to the door and then enter it. However, consider what would happen if the building is rather large and the agent is on the other side of the building from the only door. In that case, the agent would take considerably longer to enter the building. However, what if the agent did not have to go around the building, but had some capability to create its own doorway in the building near where it currently is (e.g., a sledge hammer to take out a wall, a brick to break a window, or an explosive charge to put a bigger hole in the building). Now the question becomes whether it would be faster to walk around the building to an existing door or to install a new door (this is also what tends to limit this algorithm to adversarial games or simulations since non-adversarial contexts tend to frown on impromptu doorway installations via explosives).

Our algorithm to accomplish this task of path planning through obstructed configuration space can be included as a few extra function calls in most existing search algorithms as shown in Algorithm 11 in the context of A* search (for a good survey of search techniques see [54]). When a search algorithm is evaluating a potential path to a goal, it calculates the benefits of moving to any given region using a heuristic. This heuristic generates an approximation of how far a potential move would leave the agent from its destination, and when added to a stored value indicating the cost to get to that potential location, allows an agent to rank potential paths through the

**Algorithm B.1:** The ANMA algorithm looking for potential obstructed configuration space regions to convert to free configuration space during a search. The example is in the context of a node expansion in an A* search.

```
// Sorted List of Regions to consider
$List\ OpenList$;
// List of Regions that have already been considered for path
   planning
$List\ ClosedList$;
// Check of the first node in the open list
$FirstNode$ in $OpenList$;
for $Neighbor\ of\ FirstNode$ do
    if $Neighbor.isPositiveSpace$ then
        // Calculate the cost to convert this node to free
           configuration space
        $float\ CostToConvert = Neighbor.findConversionCost()$;
        // Generate normal heuristic costs to traverse this node
        $Neighbor.f = FirstNode.f + Neighbor.findH()$;
        $Neighbor.f += CostToConvert$;
        $OpenList.append(Neighbor)$;
    else
        // The node is free configuration space treat it normally
```

environment. This system works very well for movement through free configuration space, and it makes sense to extend it to movement through obstructed configuration space. There are two parts to this extension: the first is calculating the raw cost of moving through the obstructed configuration space region as if it were free configuration space—this can be done using the same calculation as for free configuration space. The second, harder part comes from calculating the cost of converting an obstructed configuration space region into free configuration space. This is where having a navigation mesh spatial representation (or some other representation with a listing of obstructed configuration space) is very useful. Using this representation, the size and composition of the target obstructed configuration space region can quickly be determined. These factors combined with the agent's ability to manipulate obstructed configuration space (agents with explosives can perform faster manipulations than agents with wrecking bars) can be used to generate a cost to convert any given region of obstructed configuration space. This cost of conversion is then added to the cost of movement through the obstructed configuration space region, and this total value can then be used in the path planning algorithm like a normal free configuration space region.

## B.2 Planning for the Obstruction of Free Configuration Space

The second, defensive half of the ANMA algorithm deals with a slightly higher level problem, which occurs less often. This portion of the algorithm answers the question "How can an agent tasked with defending an area from other agents or players alter his local environment to restrict the number of potential access points?" Our approach to this problem uses conventional search algorithms on the world space representation to locate potential entry ways into the area the agent is in charge of defending, and then prioritizing the order in which these free configuration space regions should be converted into obstructed configuration space regions through the introduction of obstructions.

Consider the example of a building which an agent has been assigned to defend from other agents and players. In particular, there is a single room on the second floor of the building which the agent must defend at all costs. By examining the navigation representation of the environment, the agent can determine the list of entry ways into the area they need to defend, in this case, let us say there is a ladder connecting to a second floor window from outside, the obvious front door entrance to the building, and finally a hole some impatient person blew in a wall instead of coming in the front door. Now the agent needs to determine, which of these potential enemy attack routes they should close off first (patching the hole in the wall, locking the front door, or hiding the ladder leading to the second floor entry).

**Algorithm B.2:** This is the defensive half of the ANMA algorithm, which determines the most important free configuration space nodes to seal to protect an area.

```
// Initially the algorithm starts with its 3 requirements
$Region\ regionToDefend$;
$Region\ attackerOrigin$;
$List\ PossibleRegions$;
// We will also assume we have path finding algorithms on hand
$PossibleRegions = findPath.(regionToDefend, attackerOrigin)$;
// Validate the target region is in the assigned guard area
repeat
    // Select the most connected region using distance from agent as
       a tie breaker
    $DestroyRegion = PossibleRegions.getHighestDegree()$;
until $!DestroyRegion.isInArea()$;
```

In order to the use the defensive sub-algorithm of ANMA, we first require three pieces of information. First, we need to know the extents of the area the guard should remain in. Secondly, we need to know which direction the other agents or players we will be defending against will be approaching from. Finally, we need to know the main point we are supposed to be protecting. Using this information, we are able to determine which sections of the navigation representation it would make the most sense to alter and invalidate to prevent or delay the enemy moving through them. These requirements are also what limit the defensive portion of the algorithm to adversarial situations. While the attacker portion of ANMA can be used to create non-adversarial path planning agents (albeit ones who have little respect for the condition of the world as they move through it) it is unlikely a non-adversarial scenario could supply the three requirements to implement defensive-ANMA in an agent. The defensive sub-algorithm works by calculating the optimal path(s) from the location the agent is supposed to defend to the areas the enemy is expected to advance from using a search algorithm as shown in Algorithm 12. It then finds the free configuration space region along this path(s) with the lowest degree in the navigation graph representation of world space (the free configuration space region with fewest neighboring regions). If two or more regions have equal degree then the closer one to the defender is selected. Additionally, this search for a target region is restricted to the areas of the world where the agent is supposed to be guarding. This is to ensure that our defensive agent does not go running off to try and block the exits of the enemy base and in fact stays inside the area he is supposed to be defending. After locating this target region, the agent will move to it and attempt to use its ability to alter the world geometry to block passage from this node towards the enemy approaches by introducing new obstructed configuration space areas. A final restriction on the possible target regions for the agent to alter is that the agent needs to have the ability to introduce sufficient configuration space obstacles to block off an area. This can be determined in advance by comparing the data on the free configuration space region stored in the spatial representation of the world to the agent's ability to manipulate free configuration space regions (e.g., does the agent have concrete barriers it can deploy to block roads, plywood it can nail up to block doorways or windows, or just some rope it can string across a path). The agent then repeats this process of finding a low degree free configuration space region on the approach to the area it is supposed to guard and sealing the area off until there are no open approaches or it runs out of blocking materials, at which point it can fall back to its non-ANMA behavior.

## B.3 ANMA Evaluation

We performed a series of experiments using our agents running the Adversarial Navigation Mesh Alteration (ANMA) algorithm to determine their effectiveness against more traditional forms of agent behavior. To do this evaluation in an adversarial context, we decided to use the Capture The Flag (CTF) game type with two teams of agents. Our implementation of the CTF game was as follows: A CTF game is played on a field containing some randomly distributed quantity of open space which players can move around in, as well as obstructions which block line of sight and movement. This space is broken down into two evenly distributed halves, with each team having "control" of one half. Additionally, each team possesses a base inside the area they control that contains a flag. Each team was composed of 10 agents. Of these 10 agents half of them were assigned to attempt to capture the enemy's flag. The other half of the team was tasked with defending their own flag. A flag capture occurred if an agent picked up the other team's flag. If an agent entered the same free configuration space area as an opposing agent on the side of the field the other team controlled, then they were captured. Captured agents were removed from the game and after a short delay returned to a random location on their own side of the playing field. Games were concluded after a single capture was scored. In order to prevent stalemates, agents were required to move from one area to another each turn if possible.

Instead of spending time implementing a fully fledged graphical CTF game to test our agents in, we designed and built an agent test bed to simulate CTF games that allows us to perform agent testing and comparison very quickly. In this simulator, we use a representation of the world instead of an actual world model; this allowed us to rapidly and procedurally generate many random worlds. In this representation, free configuration space regions are represented as nodes. The gateways between free configuration space regions are represented in this simulator as edges between the nodes. In this manner the simulator represents and maintains the navigation graph of the playing field in question without having to worry about perfectly updating the underlying navigation mesh. Additionally, obstructions present in the configuration space between any two given areas of free configuration space were also maintained and represented in the simulation. These configuration space obstructions were represented as inactive links, which are not traversable to agents. Using this simulator, we are able to procedurally construct random levels to evaluate our agents in. Our procedural world generation algorithm works as follows. First, eighty to one hundred free configuration space regions are seeded randomly in the world with a bias to ensure that they are not clumped too closely together. Then all of the neighboring free configuration space regions are determined using a simple distance metric. After this determination, neighboring regions are randomly determined to be either connected one to another with a gateway, or adjacent but obstructed one from another by configuration space obstructions. The results of this determination are then stored as links for the simulation. Finally, the two bases are selected, one for each team from the available pool of free configuration space regions, such that the base is within the back 10 percent of each team's territory (remember that each team is considered to be in control of half the map) and the chosen base has at least three adjacent free configuration space regions connected to it via gateways to help ensure it is accessible. This simulated representation allowed us to evaluate our proposed agent design on a larger number of worlds than would have been possible if we were using a full fledged CTF game engine. In particular, we were able to evaluate our agents on 180 unique test levels. One of the test levels generated by our tool is shown in Figure B.1.

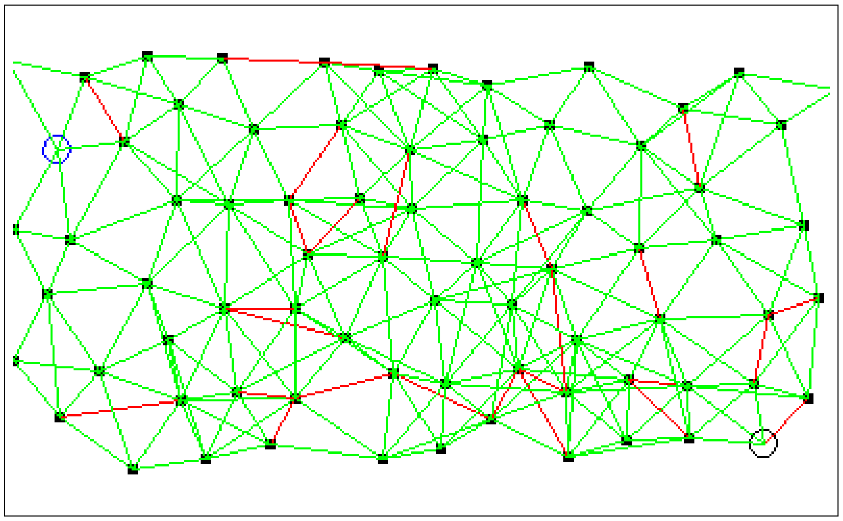


Figure B.1: This image shows one of the procedurally generated levels used in our simulated CTF game. Free Configuration Space areas the agent can occupy are shown as black squares, active gateways are shown in green, blocked gateways are shown in red, and the bases with each team's flags are shown as circles.

We evaluated two types of environmentally manipulative agents as well as one non-manipulative adaptive agent. The first type (the Builder) is able to employ the path planning through configuration space obstruction algorithm to create new gateways and regions in order to facilitate quicker movement through the environment. However, the Builder is unable to generate new configuration space obstructions to close off potential routes through the level. The second type of environmentally manipulative agent we call the Universal agent. This agent is capable of both planning paths through and removing configuration space obstructions and dynamically placing new configuration space obstructions into the world. This allows the Universal agent to close off potential access routes to its flag and set up choke points on approaches to the territory it is defending. We did not evaluate agents who were able to destroy connections between nodes but not build them, due to the fact that these agents trap themselves in their base. The final agent type we tested was designed to be highly adaptable to changes in a dynamic game world. This agent was coded to use D* for path planning [60] through the world representation so that it could take advantage of changes to the underlying navigation graph produced by the more advanced agents, or adapt as best it could to obstacles thrown in it's path by the world manipulative agents which can destroy pathways.

All of the evaluated agents had the same basic behavior patterns depending on the role of the agent. The defending agents regardless of type patrolled their side of the map and attempted to move into and capture enemies they observed in adjacent regions. Additionally, agents of the Universal type used their ability to place configuration space objects in the world to barricade off gateways between regions such that in whatever region they are in, the most optimal path back to their own flag will be sealed. Adding this simple logic to take advantage of placing configuration space obstructions yields a set of agents that automatically build mazes and choke points of narrow or obstructed corridors that approach their flag. Conversely, the attackers all were rather single minded in that they took the best available path to the goal. However, both of the advanced agents (Builder and Universal) will construct paths which allow them to cut holes in configuration space obstructions or build bridges over empty areas if such a detour would result in a more optimal path to the other team's flag.

We set up test scenarios featuring all possible unique parings of these agents (Universal vs. Adaptive, Universal vs. Builder, and Builder vs Adaptive). Each pairing of agents played sixty matches on our CTF simulator. The results of these matches are given in Table B.1 shown below. After 30 of the 60 matches, we switched the side of the map each team was on. We also included the results of running each agent type against itself an additional 60 times as a logical check to verify the integrity of the simulation and show that neither side has a positional advantage, since if the same type of agents compose both teams then the win/loss ratio should be approximately even.

Table B.1: Comparing the performance of different types of agents on randomly generated CTF levels (showing wins to loses). Matches vs. the agents own type are provided to verify the integrity of simulation.

| vs | Adaptive | Builder | Ultimate |
|---|---|---|---|
| Adaptive | 31 to 29 | - | - |
| Builder | 34 to 26 | 30 to 30 | - |
| Ultimate | 44 to 16 | 35 to 25 | 28 to 32 |

When examining our results, we see that our hypothesis is verified that agents that can manipulate the environment do perform better in CTF games than basic adaptive agents. Looking through the data, we see that Builder agents win matches against Adaptive agents at a rate of 1.33 to 1. This is a slight performance improvement, and it can be explained by the fact that the only advantage that the Builder agents possess is that they have a shorter path to the opposing team's flag. However, this advantage is somewhat transitory, as the new paths the builder agents create can then be utilized by the simple adaptive agent to attack the Builder agent's own flag.

The more advanced Universal agent performs even better against the straight adaptive agent, winning games at a rate of 2.75 to 1. This is in line with expected performance, as the Universal agent design should dominate a purely adaptive agent. However, the possibility does exist that Adaptive agents would be able to capture the flag the Universal agents are guarding before the Universal agents can seal off all of the approaches to it. This is seen in the occasional Adaptive agent's victories. But most of the time, the Universal agents are successful in blocking all approaches to their base (e.g. Figure B.2). The Universal agent also fares well when compared to the Builder agent, with a win ratio of 1.4 to 1. These agents are more evenly matched, but the Universal agent gains a slight edge since the Builder agent has to stop and open passageways through blocked paths before traversing them to attack the Universal agent's flag each time they attack. Conversely, when attacking the Builder agent's flag after the first wave of agents goes in, the Universal agents will not have to open or reopen any passageways, since there are no agents creating blockages or obstructions on that side of the map.

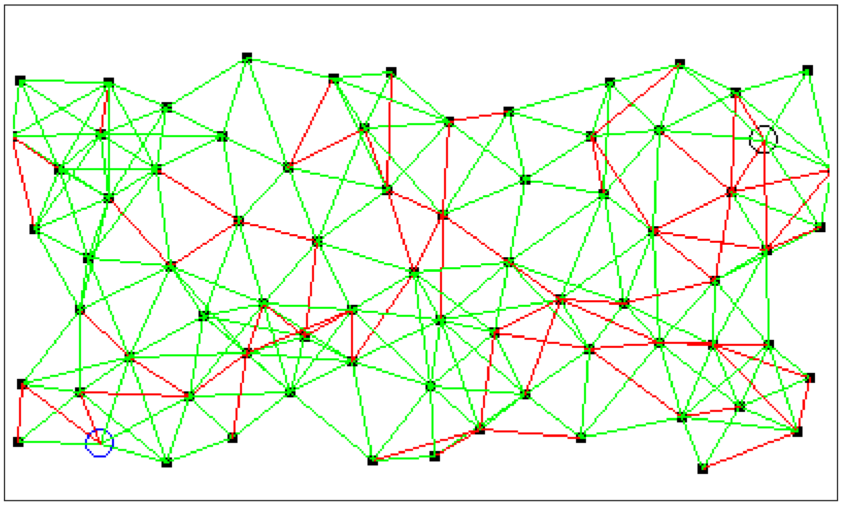


Figure B.2: This image shows one of our procedurally generated levels after two teams of Universal agents have obstructed many of the potential gateways between regions, notice all of the red inactive links in the image.

# APPENDIX C: NAVIGATION MESH EXTENSIONS

Having seen the trade-offs between accuracy and speed we must make when adjusting the size of the growth increment in PASFV and VASFV, we are prompted to ask the question, "Is it possible to rapidly decompose large environments with high precision accuracy?". In particular, consider the urban environment pictured in Figure C.1. This environment is many square kilometers in size, but contains considerable detail. The features such as sidewalks and walking paths require sub-meter accuracy for proper representation. In our traditional decomposition approach based on the scale of the world, one meter equals one unit in the world. This world and other large scale high detail environments prompted exposure of decomposition rate as a user definable parameter. However, if using a decomposition rate of .001 meter per growth step it would take approximately two months to decompose this environment using a modern computer. This obviously conflicts with our goal of providing an algorithm that can be used to rapidly respond to changes in the level geometry.

Figure C.1: A sample city generated using the UrbanPad software from Gamr7. This city contains high levels of detail and covers a large area.

In order to address this problem we have developed the Dynamic Rate decomposition algorithm that allows the decomposition to alter its rate of growth in real time based on the obstructions it encounters. The key feature of this algorithm is that the user must set a target decomposition rate and a decomposition resolution. These numbers are constrained in that it must be possible to multiply the decomposition resolution by some power of two to yield the target decomposition rate. Initially, when running a variable growth-rate decomposition, the regions in the world will grow at the specified target decomposition rate. If a collision occurs with either geometry or another growing region, instead of retreating and stopping all growth in that direction or sub-dividing the region into a higher order polygon, the growing region will retreat and reduce the rate of growth in the target direction by half. This will allow it grow closer to the obstruction. Each edge will continue this policy of attempting to grow and then retreating and halving their growth rates until the growth rate is reduced to the predefined decomposition resolution. Once that occurs then further growth in that direction either halts or the region is subdivided as called for in the PASFV or VASFV algorithms. Using Dynamic Rate decompositions allows large environments such as those generated by UrbanPad (www.gamr7.com) to be rapidly decomposed in minutes rather than days.

## Obstruction Informed Seeding

Traditionally, the PASFV and VASFV algorithms have used some form of gird based seeding in order to determine the initial placement of regions into unoccupied configuration space. This results in an even distribution of regions, but generally over-saturates the target areas and requires a clean-up stage that will combine adjacent regions if the resulting shape would still be convex. Additionally, most PASFV and VASFV algorithms also support some kind of human controlled or on demand seeding, which may generate decompositions that require less cleanup time, but these hand designed decompositions require additional time to construct compared to purely automated approaches. It should be possible to automatically generate a better seed distribution then using a pure grid or random approach. Generating an optimal seed set to minimize the number of regions present in the world would require solving an NP-Hard problem. Instead, we can take advantage of one of the properties of the convex occupied configuration space to build a set of possible seeding locations.

Consider the Proof 2 in Section 4.5 where we show that two or more of the edges comprising a convex occupied area of configuration space cannot be covered by the same convex region of unoccupied space. This means that for every obstruction edge present in the world we will, in theory, require one region of unoccupied space to describe it. In practice, this is not always the case as a region of unoccupied space can cover exposed edges of two or more separate regions of occupied configuration space. Additionally, depending on how the edges of the growing regions in PASFV and VASFV converge on each other it is possible to generate voids in the decomposition that must be filled by seeding out from the unoccupied configuration regions. However, though this proof does not provide us with an optimal set of seeding locations it does give us a good starting point. Given a distribution of seeds such that every edge of occupied space has a growing region placed adjacent to it we will ensure that the final decomposition does not have any areas without well defined regions. Furthermore, unlike grid based seeding methods this approach will ensure that there are no disjoint regions that would potentially be skipped as can happen when using the traditional grid and reseed approaches used in PASFV and VASFV.

## Texture Biased Spatial Decompositions

Consider the navigation meshes our decomposition algorithms have generated so far. All of the navigation meshes have treated all un-configured space as the same. This means that a navigation mesh for an urban environment centered around a single building with traversable roads, sidewalks, and grass (as shown in Figure C.2) would treat all of these different surfaces as the same in terms of walkability. Furthermore, regions of the navigation mesh might well cross over multiple different terrain types as long as they are all within the specified tolerance for difference in height for the spatial decomposition as shown in Figure C.3. This could result in problems for agents attempting to utilize this navigation mesh. For example, an agent driving a car though this hypothetical mesh would not know if they were on the road, the grass, or even the sidewalk based on the navigation mesh. Furthermore, paths generated using this navigation mesh might not actually be traversable depending on how the agent is supposed to traverse the environment (e.g., the path might contain only grass or sidewalk regions, which are not generally traversable if the agent is driving a car).

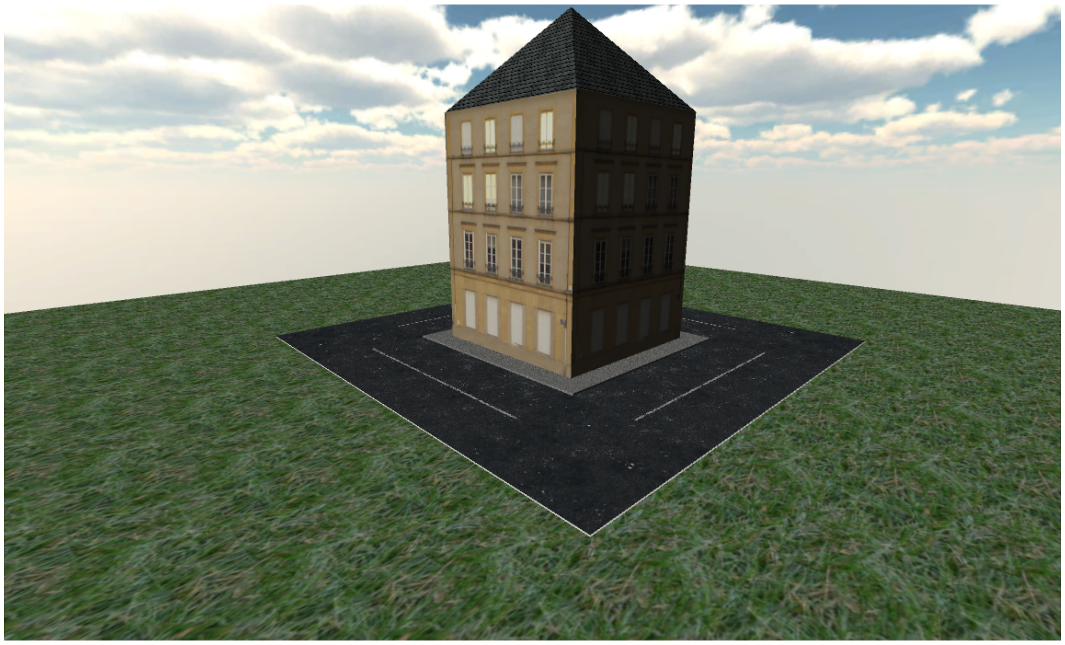


Figure C.2: A simple game environment containing multiple terrain types

Typically in game environments, different types of terrain have different textures. Since we are generating decompositions to assist with agent path planning it makes sense to group terrain textures according to the different traversal methods agents might utilize. The easiest way to accomplish this is to run the spatial decomposition multiple times altering what is and is not considered to be an occupied configuration space area and appending the resulting regions to the same navigation mesh. We rely on the user to generate a list of which textures correspond to any given traversal methods. So for instance the asphalt, road, and highway textures might all designate areas where the agent can “drive” while the sidewalk and grass textures indicate areas the agent can “walk”.

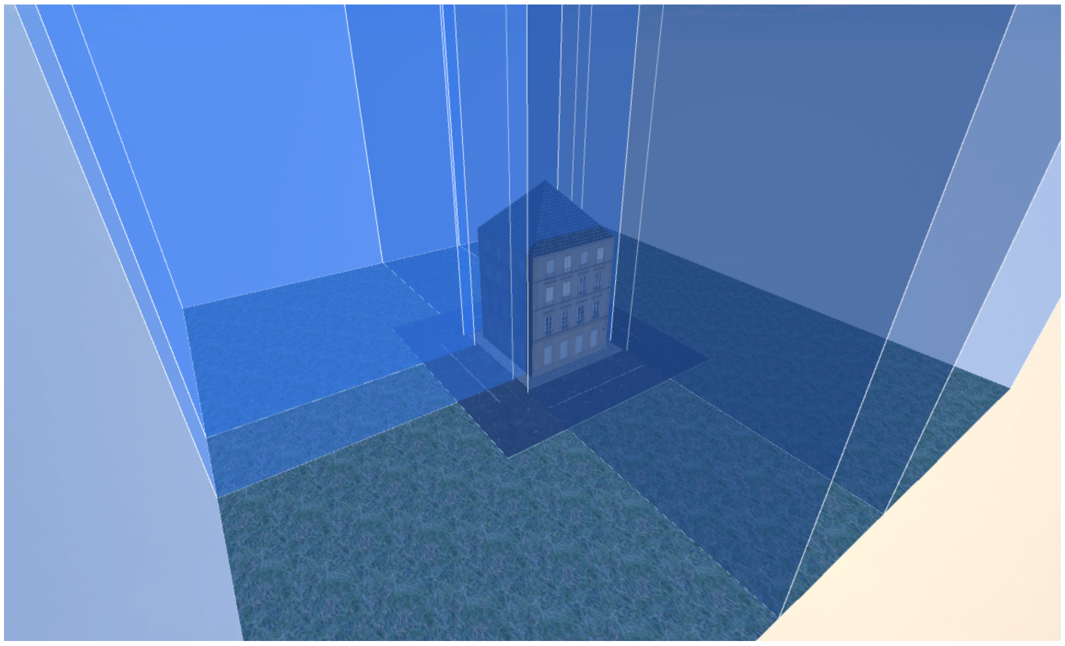


Figure C.3: A non-texture biased decomposition of the above environment

To implement Texture Aware Spatial Decompositions we first insert fake obstructions that are projected up from all of the ground plane(s) in the environment. Each of these fake obstructions is tagged with the texture of the ground plane they are extruded from. Then all of the fake obstructions associated with a user defined traversal method are toggled off to be traversable again. The PASFV or VASFV algorithm is then executed on the world in its current state (in our example images we use the PASFV algorithm). All of the regions of un-configured space generated in this cycle of the decomposition algorithm are tagged with the texture set that is defined by the current traverse method. After the decomposition algorithm finishes execution, another set of these fake obstructions is toggled off and the decomposition algorithm is ran again. Again regions that are generated in this second execution are tagged with the set of traverse textures that were just toggled to be walkable. This cycle repeats as shown in Algorithm C.4 until there are no further traversal methods to be considered and all of the un-configured space in the environment has been decomposed into regions. To utilize the navigation mesh generated by this process gateways are defined exist between regions of the same traverse types. Connections between regions which have different traverse types are referred to as boundaries and can have special traverse types (e.g., a bicyclist might be able traverse both the road and the sidewalk, but might have to "hop" to move between the two types). Such special traverse methods on boundaries must be provided by the end user.

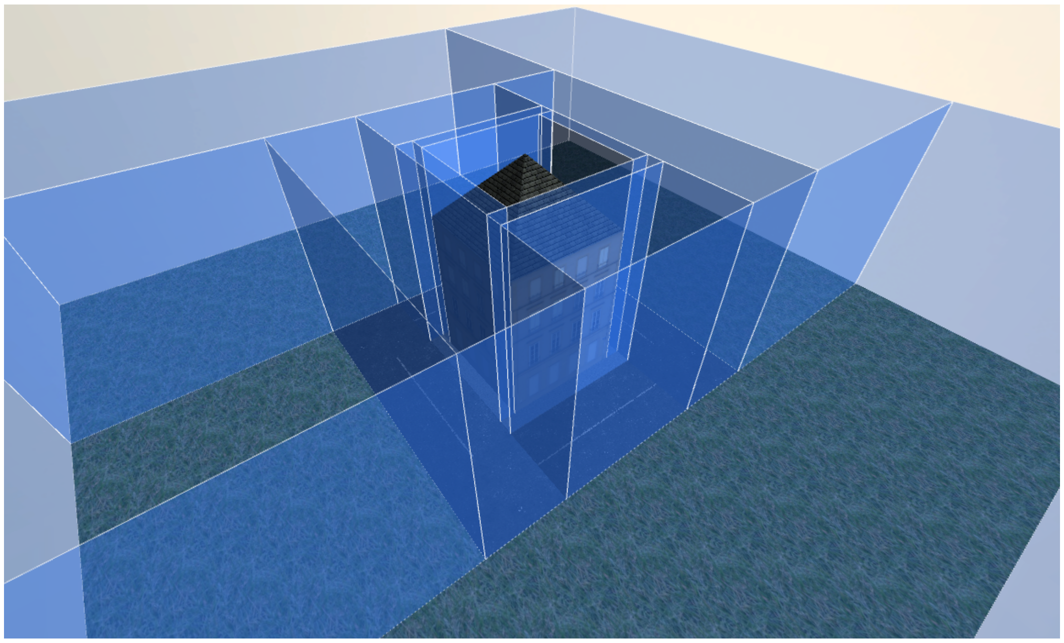


Figure C.4: A texture biased decomposition of the above environment

There are two special considerations that must be dealt with when conducting a

Texture Aware Spatial Decomposition. First, it is possible that there are certain terrains or texture types that might be present in one or more traverse types (e.g., the crosswalk would be present both in the drive and walk traverse types). This can be accomplished within the original algorithm by considering such textures to be distinct from their texture traverse groups (so there would be texture traverse groupings of drive, walk, and both drive and walk). Like the original texture groupings and traverse classifications these special considerations must be provided by the end user. Secondly, to extend this algorithm into three dimensions it is necessary to make a few changes. Instead of using the concept of fake obstructions we take advantage of the gravity based seeding that we use in VASFV and the fact that all of the seeds are resting on some terrain or other. We can use this fact to immediately tag the seeds with a unique texture as we place them. Since the seeds are in contact with a terrain piece with a specific texture we limit the growth of the region such that its entire base is associated with that texture. This means that a character standing (or riding) through that region of the navigation mesh is assured to be in a consistent type of traversable terrain.

# APPENDIX D: COLLISION DETECTION VIA NAVIGATION MESHES

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 compartmentalization 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-squared 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., 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., kd-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 navigation mesh to provide connectivity data for agent path planning [46]. The presence of these two spatial data structures, since they have internal similarities (both subdivide space into well-defined regions and allow for fast lookups of locations), raises the question *is it possible to eliminate one of these structures and use one for both agent navigation and the acceleration of collision tests?* If we consider using the tree based spatial subdivision data structures for agent navigation, we immediately encounter two problems. First, the walkable areas are not clearly delineated by the tree structures. Instead, the tree primarily stores the obstructions in the world instead. It might be possible to work around this problem by using the non-leaf nodes as regions to path plan over and using local path planning to avoid the known obstacles in the leaf nodes. This still leaves the second and larger problem; namely, *how do we determine if two or more regions are adjacent to each other since the tree structure does not store connectivity information between regions?* This problem is illustrated in Figure D.1 which shows a navigation mesh and kd-tree for a simple level. In the navigation mesh if two regions are adjacent then they share a common edge in the simulation world. This does not apply to the kd-tree as for example the leaf nodes D and E both come of the same subnode but they might well be from completely different sides of the level, while the nodes E and A which are on completely different sides of the tree might well share a common edge in the simulation environment. Applying the extensive modifications to tree based structures required to address these problems results in the generation of a new spatial data structure that, effectively, is a navigation mesh. Now that we understand why it would be challenging to use a spatial data structure for agent navigation we can ask *how difficult it is to use a navigation mesh for collision detection?*

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 datastructure (insert, remove, update, and find collidable objects) for the navigation mesh. 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.

Extending a navigation mesh to support the compartmentalization of objects for the acceleration of collision detection is a straightforward process. It requires that four 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. Secondly, 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

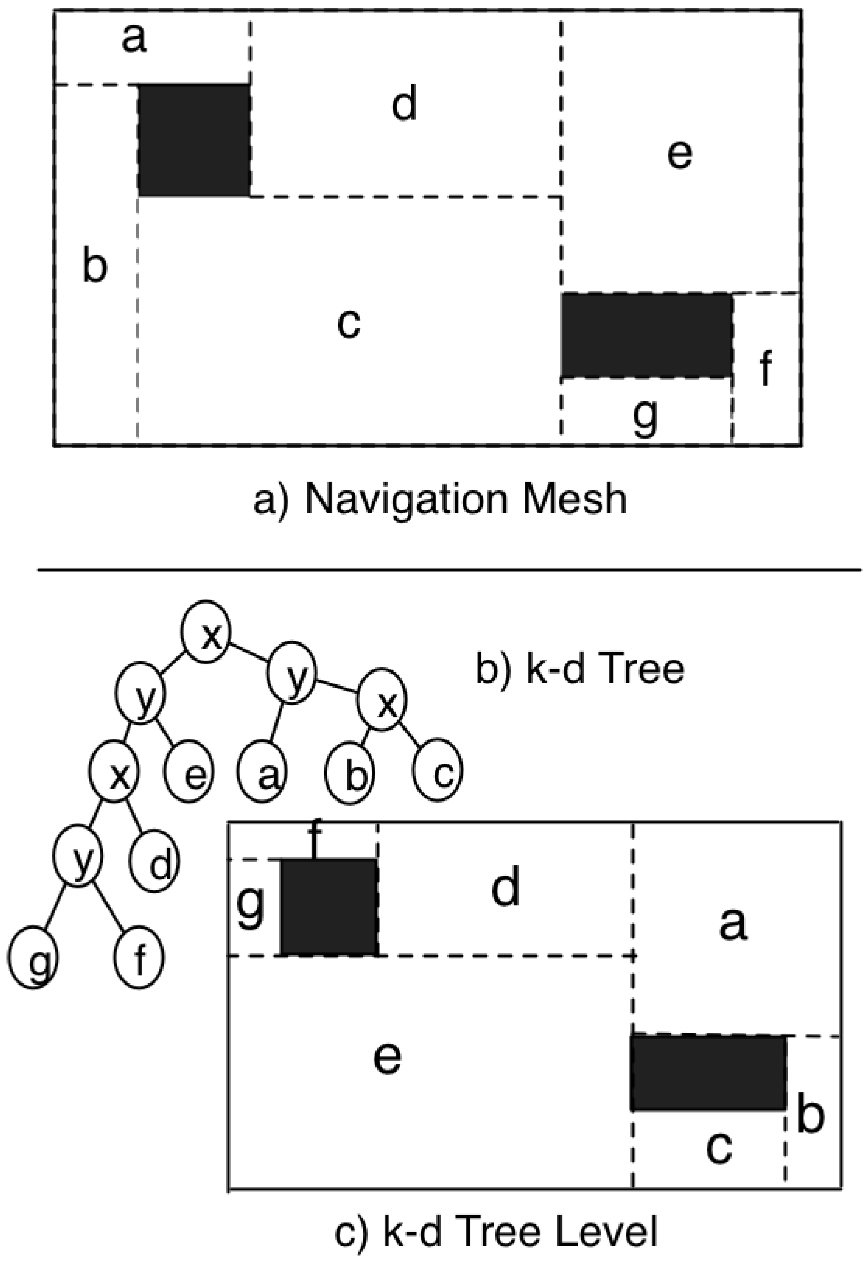


Figure D.1: Two possible breakdowns for a simple level. The first is a navigation mesh. Obstructions are drawn in solid black while the regions of the navigation mesh are shown as dotted lines. In the navigation mesh adjacent regions are guaranteed to share a common edge in the simulation environment. The second image shows a sample kd-tree with the nodes of the tree labeled along with the level associated with this tree. The x and y values in the non-leaf nodes of the tree indicate which axis is being used for splitting to build the kd-tree.

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 provide detail on the implementation.

## D.1 Insertion of Objects

Inserting objects into the navigation mesh proceeds in a 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 objects location is located as shown in Algorithm D.1. Unlike in traditional spatial data structures 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 a unoccupied space region and assuming our navigation mesh fully describes the world then the object must lie in an occupied space area 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 possible to pass on the region the creator of the object resides in and then performs 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. The runtimes of this algorithm are shown in Table D.1. 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.

**Algorithm D.1:** *NavMesh*.addObject(*ObjectToAdd*)

```
targetRegion;
/* Iterate through each of the unoccupied space regions present in
   the world until one is found that can contain the object.  */
if ObjectToAdd.hasGuessedLocation() then
    targetRegion = NavMesh.doBreadthFirstSearch(ObjectToAdd);
for UnoccupiedSpace in NavMesh do
    if UnoccupiedSpace.contains(ObjectToAdd) then
        targetRegion = UnoccupiedSpace;
        break;
if targetRegion == Null then
    /* The object location is not in open space */
targetRegion.addObject(ObjectToAdd);
```

Table D.1: Various algorithms to manipulate objects in a navigation mesh. $n$ is the number of regions in the navigation mesh.

| Algorithm | Worst Case | Best Case |
|---|---|---|
| Insert | $n$ | 1 |
| Removal | 1 | 1 |
| Update | $n$ | 1 |
| Find Potential Collisions | $n$ | 1 |

## D.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 colliable objects it maintains. This method results in deletion being a constant time operation and is given in Algorithm D.2.

**Algorithm D.2:** $NavMesh$.removeObject($ObjectToRemove$)

```
/* Call the remove function of the region the object is contained
   within.  */
$targetRegion = ObjectToRemove.currentRegion$;
$targetRegion$.remove($ObjectToRemove$);
```

## D.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 objects current location until it finds an area that could contain the object, at which 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 takes 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 and then neighboring regions should 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 as shown in Algorithm D.3. Because navigation mesh regions generally only have four or five neighbors this approach performs extremely well as can be seen in the algorithm runtimes shown in Table D.1. The worst case of $n$ can only occur in degenerate or very small navigation meshes where every region on the mesh is connected to every other region, or when an object leaves unoccupied space.

**Algorithm D.3:** $NavMesh$.updateObject()

```
/* Iterate through each of the unoccupied space regions present in
   the world until one is found that can contain the object.  */
for Object in NavMesh do
    if Object.currentRegion != Object.oldRegion then
        /* Find the new region of the object given its old region as
           hint to start a breadth first search from.  */
        Object.findNewRegion(Object.oldRegion);
```

## D.4 Find Collidable Objects

Selecting the objects that might possibly 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, which must be conducted since the objects in one region are excluding extreme cases, fewer than the number of potential collision objects in the world. 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 recombining 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 as shown in Algorithm D.4 and described in Table D.1. This works well for navigation mesh generation 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 $n$ is another 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 over very small.

**Algorithm D.4:** $NavMesh$.findCollisions($Object$)

```
/* Determine which objects are inside the same region and therefore
   potentially colliding */
List $PossibleCollisions$;
$region$ = $NavMesh$.getRegion($Object$.curRegion);
$PossibleCollisions$.add($region$.getCurrentObjects);
List $NeighborRegions$;
$NeighborRegions$.add($NavMesh$.getNeighbor($Region$);
for $Region$ in $NeighborRegions$ do
    $PossibleCollisions$.add($region$.getCurrentObjects);
return $PossibleCollisions$;
```

## D.5 Navigation Meshes Collision Detection Evaluation

We performed an experiment to validate the performance of our collision detection extensions for navigation meshes. In this experiment, we evaluated kd-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 shown in Figure D.2(a) 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. Both environments are composed of simple geometry combined to form structures that are more complex. There are 171 individual static objects present in this level plus the ground plane upon which all objects rest. We also tested out algorithms on a cityscape representation containing an enterable building as well as other non-enterable buildings and the street and alleyways between them. This level contains 51 individual static objects. We generated navigation meshes for both levels as shown in Figure D.2(a) and Figure D.2(b) using the PASFV algorithm, which our previous work has show to have excellent potential for collision detection due to the low number of regions it produces. The kd-trees for both levels were generated dynamically based on the initial positions of the collidable objects and as such differed for each test of the application and are therefore not pictured. The kd-tree in this experiment was set to have a maximum depth of five which resulted in a tree with 32 regions. This is as close as it is possible to get to the same number of regions (25) as are present in the navigation mesh for both of the environments. Additionally, this is the also approximately the same number of grid cells present in our spatial hash (30).

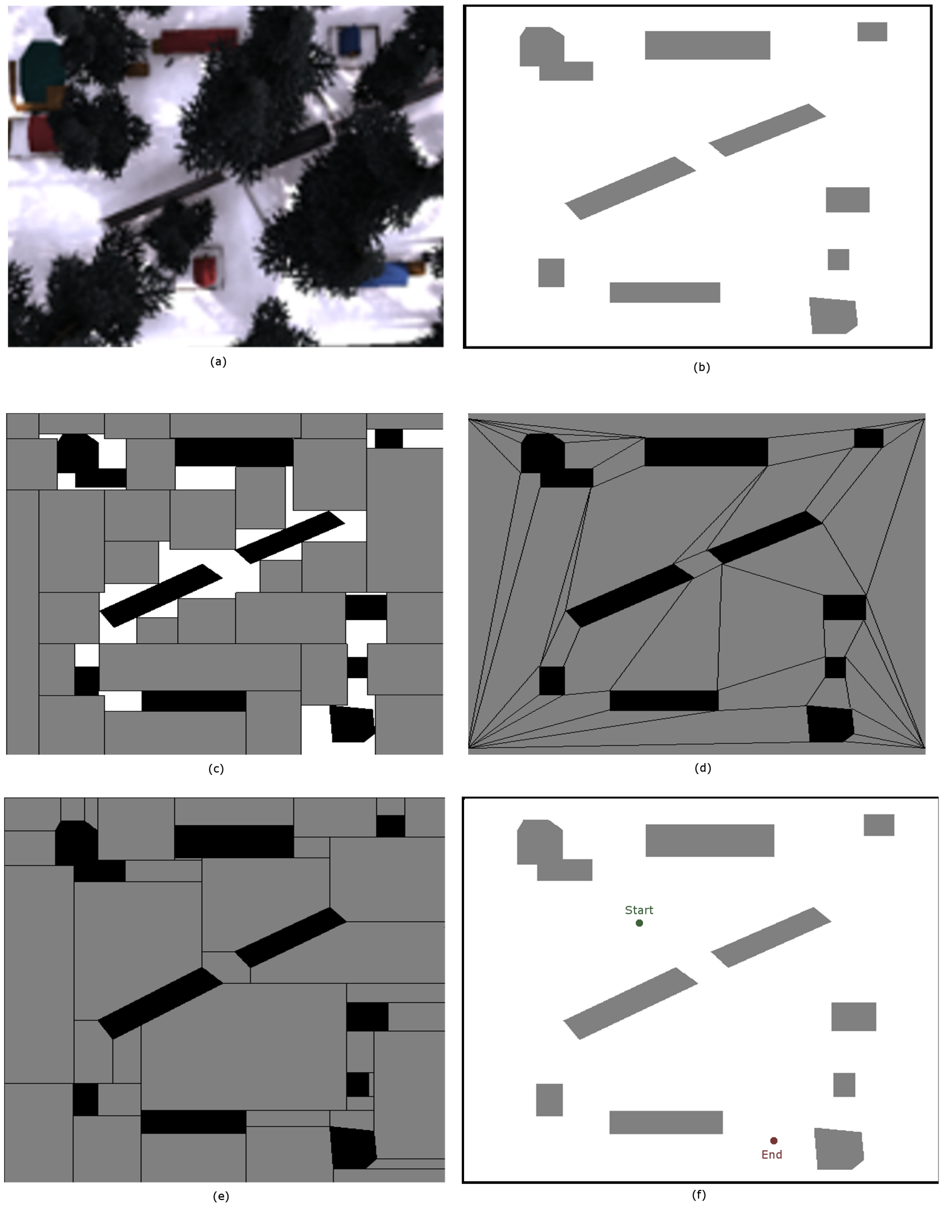


Figure D.2: We see the two levels (a and b) which we used to test our implementation of navigation meshes as collision detection acceleration data structures. The Capture the Flag (CTF) level is shown in (a) while the cityscape level is shown in (b). The navigation mesh regions are the various colored sections on the ground in the images.

We tested the ability of the navigation mesh to serve as a collision detection accelerator in two separate stages. First, we examined the ability of the navigation mesh to resolve collisions between dynamic objects. In this test we randomly placed a number of objects (200,400,600,800,1000,1500, 2000, 3000,4000,5000) into the world and then timed how long it took to resolve the potential collisions between these objects using a kd-tree, navigation mesh,spatial hashing, and a brute force all pairs collision check. We repeated this test using each set of randomly placed objects 300 times. The results of this experiment for each level are shown in Figure D.3 (CTF Level) and Figure D.4 (Cityscape Level). All times shown in the graph to detect potential collisions are given in milliseconds. In all cases the standard deviations of the results were less than 1 millisecond and as such error bars are not shown.

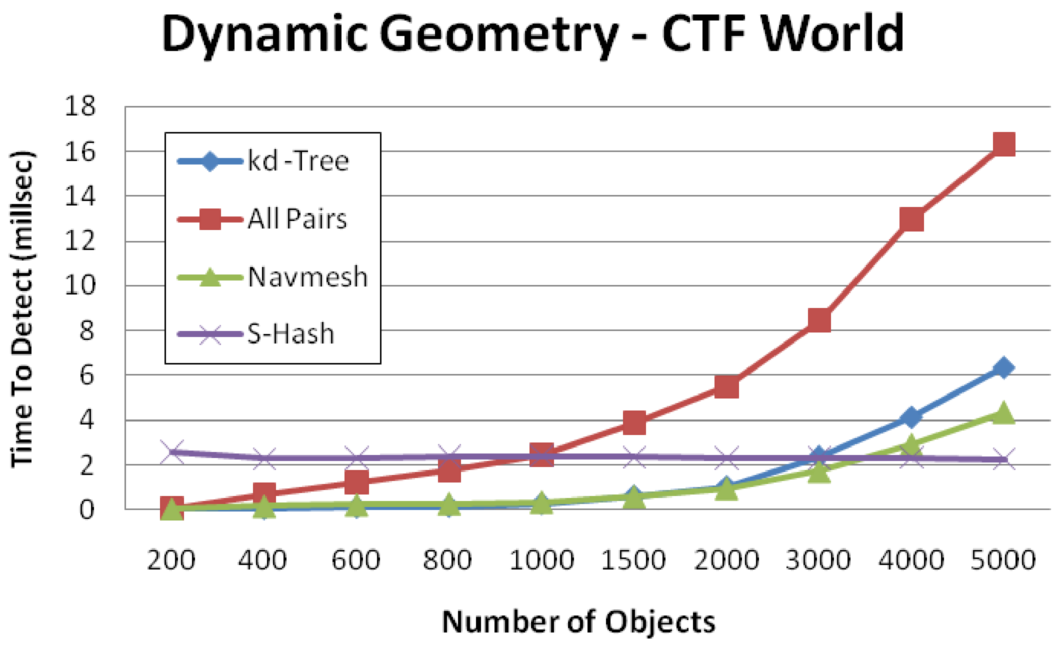


Figure D.3: The average times in milliseconds to calculate dynamic-dynamic object collisions using different algorithms in the CTF Level. In all cases the Navigation Mesh is statistically significantly faster in detecting collisions than the kd-tree and the Brute-Force approach.

In all cases the the navigation mesh provided a statistically significant improvement over both the kd-tree and the brute force approach ($p$-value less than .01, $n$

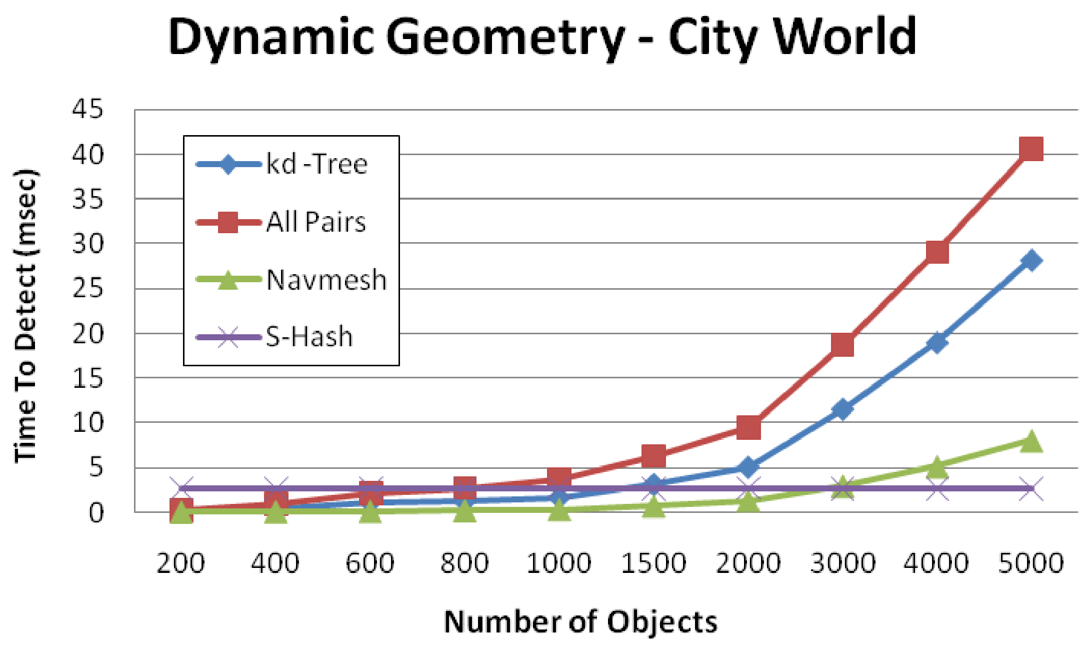


Figure D.4: The average times in milliseconds to calculate dynamic-dynamic object collisions using different algorithms in the Cityscape Level. In all cases the Navigation Mesh is statistically significantly faster in detecting collisions than the kd-tree and the Brute-Force approach.

=300). As we can see from the data presented in the figures, the navigation mesh provides an excellent compartmentalization of objects into smaller groups to accelerate dynamic object to dynamic object collision detection. It is interesting to note that the performance of the navigation mesh on the city level is somewhat worse than in the CTF level. This occurs due to the large open area present in the navigation mesh in front of the enterable building in the city level. Open areas like this in a navigation mesh would tend to degrade its ability to accelerate collision checks. If there is a random distribution of objects, it is likely a larger region will contain more objects than a smaller region. Since calculating potential collisions between objects, which are all in the same region of the kd-tree or navigation mesh is an $n$-squared operation having such large regions represents a potential time sink. In addition, the performance of the kd-tree with small groupings of objects seems to be worse than the brute force method but this is due to the extra expense of creating and maintaining a data structure versus naïvely testing all possible pairs for small groups of objects and is not unexpected. The performance of the spatial hashing algorithm is effectively constant over all of these trials, which is also to be expected since it is a bounded more by the size of the hashmap than the number of objects. Despite not being a constant time algorithm the navigation mesh actually performed better than the spatial hash map until there were more than three thousand dynamic objects in the world. This crossover effect is due to the fact that the spatial-hash has a fixed update cost since it is a constant time algorithm. On the other hand the navigation mesh has a very low upkeep cost and a collision resolution time that slowly rises as the number of objects increase. This results in performance lines that cross once the increasing collision resolution costs of the navigation mesh pass the upkeep cost of the hashmap.

Our second experiment examined the ability of the navigation mesh to quickly detect potential collisions between static objects in the world (the buildings composing the level) and the dynamic objects moving through, and interacting with the level. Again, we repeatedly populated the world with various numbers of objects (200, 400, 600, 800, 1000, 1500, 2000, and 4000), but this time instead of testing possible collisions within these groups we looked for collisions with the world geometry, which remained fixed during each of these tests. Each of the tests for the various increasing numbers of objects was repeated 300 times for each collision acceleration technique, and average time to resolve collisions and standard deviations were generated. In all cases the standard deviations of the results were less than 1 milliseconds and as such error bars are not shown.

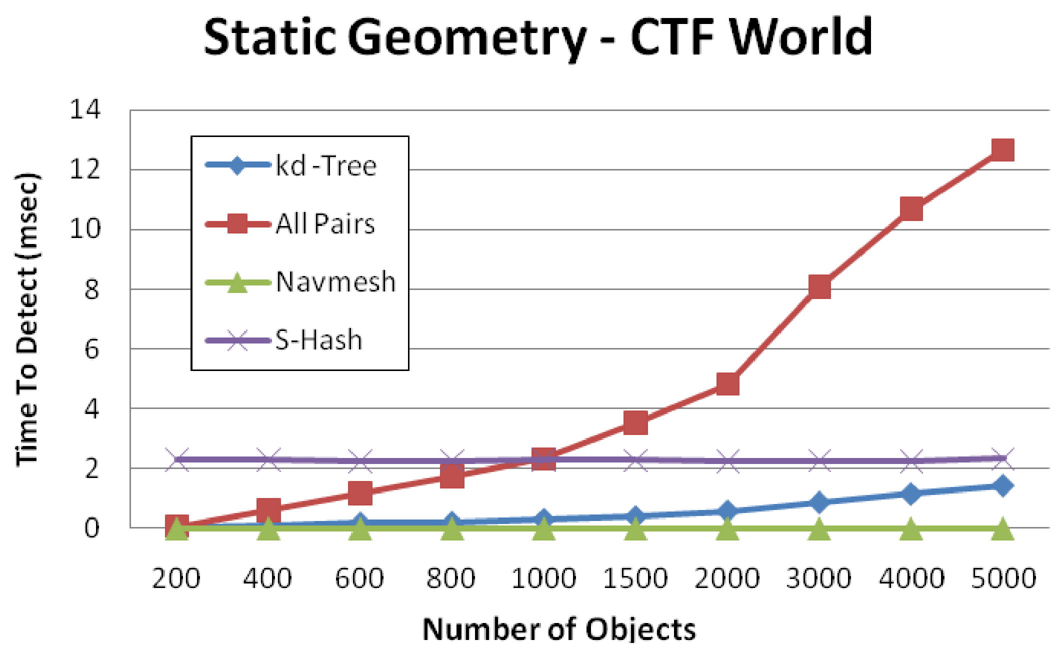


Figure D.5: The average times in milliseconds to calculate dynamic-static object collisions using different algorithms in the CTF Level.

In this set of experiments, the navigation mesh produces excellent results taking less than one msec to compute the set of possible collisions between dynamic objects and static geometry as shown in Figure D.5 and D.6. This is not unexpected as if we can locate an object in a region of free space then we can be assured that the object is not colliding with any static objects. In fact, the test to determine if an object is located in free space, can be ran in nearly constant time if we have access to the last known region the object was located in. This follows from the concept that objects do not move much from frame to frame and that by checking the last known

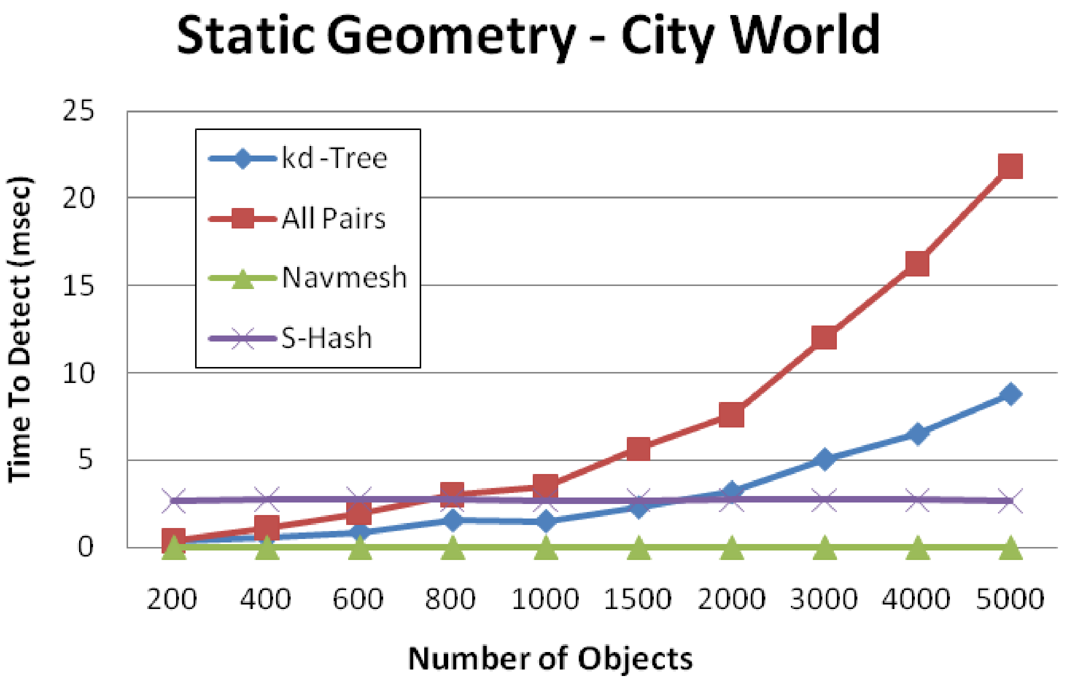


Figure D.6: The average times in milliseconds to calculate dynamic-static object collisions using different algorithms in the Cityscape Level.

region for the object and any neighboring regions we will almost certainly be able to confirm the object resides in one of those regions and from that we can infer that it is not colliding with any static objects. In the particular set of potential collisions we examined here, the navigation mesh performs statistically significantly faster than other methods of collision detection acceleration across all groupings of objects ($p < .01$ $n = 300$). The results in this case were consistent across both test levels.

By combining each pair of graphs we can determine the total time to resolve all possible collisions in a given environment. The results presented here for static and dynamic collision tests can be added together to generate an overall collision test time for all of the algorithms presented here, except for the results of the spatial-hash. The spatial-hash is a special case since the majority of the time cost for it is an upkeep cost and the full amount of the upkeep is present in both graphs. Instead the total collision cost for the spatial hash is roughly equivalent to either of the two components graphs we presented.

We used the number of dynamic and static objects present in commonly used game environments in order to select the appropriate numbers of objects to use in our collision detection tests. Based on data from the Unreal Engine 3 by Epic Games [22] 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. The upper limit of 5000 objects was selected to match the upper limit on objects supported by the Unreal Engine 3. At this upper limit spatial hashing is slightly faster than a navigation mesh for collision testing but the difference is small (3-4 milliseconds) and using spatial hashing would require two separate data structures since the spatial-hash cannot be used for agent path planning.