2007

Point seeking: a family of dynamic path finding algorithms

Andrew Fanton

Point Seeking:

A Family of Dynamic Path Finding Algorithms

Master's Project By
Andrew Fanton

Chair: Jessica Bayliss, Ph.D.
Reader: Zach Bulter, Ph.D.
Observer: Sean Strout, M.S.

1 Introduction

1.1 Basics of Path Finding

In the field of Artificial Intelligence, calculating the best route from one point to another, known as "path finding," has become a common problem. If an agent cannot effectively navigate through an environment – be it real or virtual – it will often not be able to perform even the most routine tasks. For example, a Martian rover can't collect samples if it can't get to them; meanwhile, a computer game is not much of a challenge if your opponents can't find their way around.

The problem of path finding has three basic aspects: map representation, path generation, and locomotion. First, the environment must be interpreted into a form which can be processed algorithmically. Afterward, a path through this environment is planned out. A list of movement instructions or locations to travel to are then produced in order to guide the agent. During both the planning and movement of the agent, an algorithm may consider the agent's limitations with regards to changes in velocity and orientation. Together, these steps serve to move an agent from its initial position to the desired location.

1.2 Traditional Approaches

A variety of path finding algorithms have been developed, some as a general solution and others for a specific application domain. Though algorithms vary greatly in how they represent the environment and choose a path, there is underlying methodology that is nearly universal. The environment is abstracted into a tree, graph, or similar data structure that stores potential movement; a set of moves is then selected that connects the starting location with the goal.

In the case of a tree, the start location serves as the root with the leaves being the goal. The branches of this tree (which represent each possible move) are then searched until an acceptable path is found to the goal.

For representing the map, the simplest approach is to generate a grid where each cell was marked as to whether it was traversable or not. Adjacent traversable cells would be connected in the tree. Optionally, costs can be added to the cells to provide more detailed representation. Cells need not be squares. Other symmetric polygons, such as hexagons, can be used, though they often increase the complexity of path generation [27].

Another common technique is to use Binary Space Partitioning[21]: the traversable area is recursively divided in half using reference lines. (These lines are usually the walls in the map.) These spacial partitions are used as nodes in the search tree. In addition, the tree can be used for other applications such as collision detection and graphics rendering.

A different approach that is frequently used involves using a predefined graph of way points[2]. Way points are traversable locations in that map that are connected to neighboring way points, possibly with an associated movement cost. These way points are typically generated by hand but allow for streamlined maps and more intuitive routes.

Each representation has its own advantages and drawbacks; there is a balance between the accuracy of the abstraction, the depth of the search tree, and the space required to store the map in memory. Generally, making the agent more aware of the details of the environment will use up more memory and require more time to find a path. Being too sparse, however, will cause the agent to ignore important subtleties in the map, missing useful routes or colliding with smaller obstacles.

Though map representation has a significant impact, the method used to travel through the search tree is typically the defining trait for a path finding algorithms. Traditional tree searching techniques, such as Depth First and Breadth First can be used but are typically undesirable. Depth First will pick an arbitrary branch in an unsorted tree. Breadth First is often slower and requires more of the tree to be loaded into memory. (Usually nodes are loaded or created only when needed due to memory constraints.) Algorithms that make an informed decision regarding what portion of the tree to search are desirable.

For this reason, an algorithm known as A* [14][12] has become a popular solution. It is designed to minimize the amount of the search space explored while ensuring an optimal path is generated. While traversing the search tree, A* calculates the least cost to reach a given node and an estimated cost to the goal using a specified heuristic. The node with the least combined cost is given priority; its children are added to the list of nodes to consider. Once the goal node is found, the branch is traced back upward to create the path. In effect, those parts of the tree that likely lead to an optimal solution are searched first, but the algorithm is capable of recovering from dead-ends and sudden detours. This reduces the computation without compromising the path quality.

Due to the success of A*, many modifications have been developed, each with its own particular strengths and weaknesses. These include, LRTA* and D* Lite [11] as well as DynamicSWSF-FP and LPA* [13] and dozens more. Each utilizes a different means of traversing the tree and/or determining the heuristic. Some are designed to be readily altered to handle dynamic environments.

Another such variation is Iterative Deepening A* [1][16]. Whereas standard A* simply searches for the solution by traversing the tree once, IDA* works by progressively increasing the search depth. If the depth of the node plus the estimated cost to goal exceeds the current limit, the path is terminated. There are several advantages to this approach. First is the conservation of memory; IDA* can be implemented without the second “closed” list of nodes. Second is the manner in which paths are generated. A reasonable partial path can be constructed without traversing the full depth of the tree. The cost of each individual step will be minimized, not just the overall path.

It should be noted that standard A* and most other traditional approaches do not address the issue of locomotion. Though numerous techniques exist to plan the path of objects with limited mobility [10], they are usually too computationally intensive for standard applications and are subject to the specific properties of the entity in question. Paths are either post-processed to accommodate or the agent is simply assumed to not be subject to any locomotive restrictions.

For the remainder of this paper, the later is done as it keeps the complexity of handling dynamic environments to a minimum and creates no overhead.

1.3 Problem Statement

The traditional methodology described above works in many common applications. However, it has a fundamental weakness. The path is generated then it is followed. This means a path can become obsolete as the entity is transitioning from the start to the goal. Recalculating the path can be a costly endeavor and proves to be extremely inhibitive when there is limited processing power and/or a large quantity of entities to find paths for. This can lead to delays in the processing of the program [6]. It is also very inefficient because large portions of the paths are calculated but never used.

This, of course, is not a concern in a static environment. However, in a dynamic environment where many of the obstacles are frequently moving, traditional path finding becomes impractical. In the extreme case, all objects are moving rapidly and unpredictably, virtually guaranteeing that the path generated will be compromised.

To overcome this problem, an algorithm must be able to respond to changes in the environment, modifying or replacing the agent's current path in a timely fashion. It should also reduce wasted calculation as this will conserve resources and allow the system to be more responsive.

2 Previous Work

To compensate for changing environments, numerous techniques have been developed. Many are modifications of or additions to existing algorithms.

2.1 Obstacle Avoidance

Of the numerous techniques have been employed, perhaps the simplest is obstacle avoidance[22][7]. The entity traveling along the path looks ahead for collisions that might occur in the near future along its current trajectory. If one is discovered, the entity steers itself accordingly and resumes its original path once the obstacle is cleared.

This works well if the obstacles are similar in size to the path finding agent and there is sufficient room for maneuvering. Larger obstacles may cause the path finding agent to deviate its path such that it cannot recover. Even if it does recover, the optimality of the agent's path will have been reduced. Meanwhile, if space is limited, either due to narrow passages or because of the multitude of obstacles, the entity will likely collide or, again, deviate from the established path.

The focus on immediate obstacles can present another difficulty. There is no means of predicting and averting future collisions that are less than imminent. Obstacle avoidance simply adds a minor course correction; it is not capable of adjusting to changes not within the entity's immediate area.

In short, obstacle avoidance is not sufficient in of itself. In some instances, special maps are created that help guide an agent by attracting and/or repelling it [17]. Typically, though, it is used in conjunction with another algorithm

2.2 Adjusting the Scope of Searches

One approach to reducing the complexity of path finding is to limit the scope of the searches. Variants of this technique include path hierarchies and the use of local areas.

With a hierarchical model [23][4], the whole path is searched at a coarse grain with the path segment closest to the agent being refined with additional searches. This can be used to reduce the computation spent finding a path in areas the agent is not likely to reach prior to the next calculation; such areas can be omitted from finer grain searches. The assumption is that when a hierarchy is used in this manner, the path can and will be recalculated. If the path is only generated once, the lesser detail of the later portion provides no benefit.

There are different ways of utilizing a hierarchy. The data structure representing the map may be layered, thus allowing quick access of the appropriate level of the hierarchy. However, to conserve memory, a single layer map may be used, with varying levels in the hierarchy being calculated as needed.

Searches can also be limited by restricting updates to a local area. Any change in the map that occurred beyond a specific area around the agent will be ignored. This prevents the system from adjusting to changes that will unlikely affect the final path of the agent or are only part of a larger, longer change.

2.3 Path Finding Using Splicing

There is an approach similar to hierarchical path finding described above known as Time-Sliced Path Finding [8]. (This technique is also referred to as “Line Splicing”.) It is designed to help balance the load on the system by breaking up path finding into smaller, more quickly computed segments. First is a quick path, which only calculates a path for a small number of revolutions; when its time is up, the node closest to the goal is selected as the entity's next destination. This allows for a timely response. While the agent is heading toward that position, a full path from it to the original goal is calculated in the background. Once completed, a third path is calculated from the entity's new position to the full path. This path splines the quick path to the full path, minimizing any “doubling back.”

Time-Sliced Path Finding is able to handle dynamic environments because, prior to completion, the full path can be modified as needed without interfering with movement of the path finding agent. However, once the final path is accepted, it cannot be change; a new path must be calculated if the environment changes afterward.

A related approach is referred to as Path Splicing[20]. With Path Splicing, the original path is stored. When a change is detected, the corresponding nodes are notified; the affected segment of the path is then recalculated instead of the entire path. Such changes, however, are ignored if the agent has already passed that particular segment of the line. In addition, an updated segment may be discarded and a new path generated if the change results in a considerably long path.

2.4 Adaptive A*

Another solution is a modification of A* known as Adaptive A* [15]. With traditional A*, the cost of a node is based off a static heuristic; the estimated cost to the goal is calculated with each search, reproducing the same value. With Adaptive A*, however, these heuristic are updated using information from previous searches. This improves the efficiency of future searches by reducing calculation and limiting exploration of fruitless branches within the search tree.

To appreciate this difference, consider the following scenario: path finding agents starting on one side of a map have to find a way to the other end. The agents' options are two long corridors, but the corridor on the right comes to a dead end. The first agent to explore these corridors will likely expand its search into both sides. Once it finds the correct path through the left corridor, all of the nodes in the right corridor will be updated with a much greater costs. This is because in order to reach the goal from the right corridor, agents must go back to the starting area and traverse the whole length of the left corridor. As a result of these new costs, future searches will prefer the open left corridor.

This algorithm can prove useful for dynamic environments. It retains knowledge regarding the terrain that can be readily updated. After the initial exploration of the map, the cost calculations of searches will be limited to handling only those nodes which have changed.

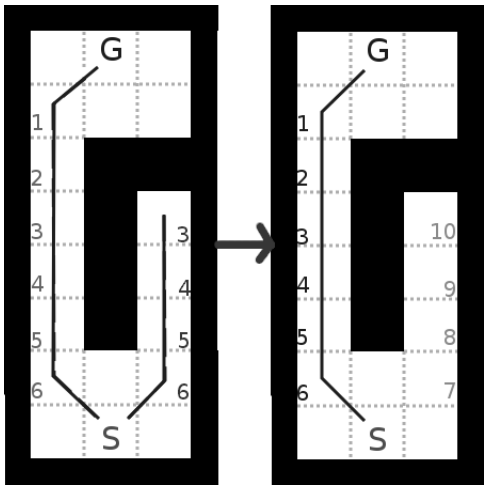


Figure 1.0 – Adaptive A* updating estimated distance from node to goal

Unfortunately, this algorithm has many drawbacks when applied to broader conditions. It benefits multiple agents seeking the same goal from similar starting points as well as a single agent which regularly recalculates its own path. Anything beyond these conditions can noticeably reduce the computational savings gained. Each stored value will be accessed far less frequently. In addition, if the rate and/or volume of change in the environment is too great, the costs will need to be change as often as they are accessed. In either case, the stored information will become less valuable, degrading the efficiency of the algorithm.

2.5 Polygonal Decomposition of Search Space

Overview

Other solutions, such as "Polygonal Path Finding" [25] and "Time Encoding" [3], focus on map representation. The primary objective is to meaningful and readily updated search space. Once this search space is produced, a standard algorithm, such as A*, is used to find a path.

Polygon Path Finding uses bounding polygons to approximate the space in which an object resides. These simple polygons are subtracted from the "C-free" space in which a path finding entity can move. Once all of the objects have been accounted for, the space is decomposed into a series of polygons (using the vertices of the bounding boxes as reference points). The nodes of the search tree are the midpoints of line segments separating open areas.

To adapt to a dynamic environment, the map is divided into layers. The static layer, which contains all of the immobile objects, can be divided once and stored. To get the full map at a given moment, the dynamic layer is superimposed upon the static one, and the existing spacial segments are divided as needed. This limits the amount of calculation required to update the map, particularly if the number of immobile objects outweighs the number of moving obstacles.

The type of shape to use is a major consideration for this algorithm. More complex bounding shapes increase the accuracy of the map representation but also increases the cost of calculating the spacial decomposition. In particular, polygons that are concave or have many sides of varying size compound the difficulty of dividing the free space. Generally speaking, rectangles are the simplest shapes to use as they have only four vertecies and four edges, are not concave or convex, and can be easily defined. Judging intersections and comparing locations is almost trivial. Furthermore, it is easy to decompose the space along a single axis (horizontal or vertical) which leads to fewer divisions and thus fewer nodes.

Procedure

The following describes the basic procedure of Polygonal Space Decomposition:

```
General Program Flow
for each object in static layer...
      determine intersection with existing polygons
      remove intersected polygon(s)
      decompose space
      insert polygon(s) from decomposition
end
```

```
for each execution cycle...
      for each object...
            if object is path agent
                  update map
                  perform search
            end if
            update object location
      end
end
```

Updating Map

```
copy static layer into temporary layer
for each object in dynamic layer...
      determine intersection with existing polygons
      remove intersected polygon(s)
      decompose space
      insert polygon(s) from decomposition
end
```

Decomposing Space

```
if intersection is not empty
      if number of polygons in intersection > 1
            for each polygon...
                  decompose space
            end
      else
            define nodes
      end if
end if
```

Defining Nodes

```
for each object vertex...
      determine line from vertex to intersection edge
      find midpoint of line
      create node at midpoint
end

for each node...
      find neighbors
      link with neighbors
end
```

Determining the intersection will depend on the nature of the polygons employed. The implementation used for this research utilized the existing rectangular intersection technique provided within the Java Abstract Windowing Toolkit package.

"Perform search" is where the underlying tree search method is executed, using the node closest to the agent's location as the starting point.

Determining the line from the object to the edge of the intersection is also dependent upon the polygons employed. With rectangles, the approach is straight forward. Take the vertex value and substitute its x or y value with the corresponding value of the intersection edge. For example, if the top corner of the intersection is (20,10) and the top corner of the object is (40,40), the horizontal partitioning line from (40,40) would go to (20,40). The subsequent path node would be placed at (30,40).

Linking nodes is largely a matter of identifying where the nodes originated and to

which nodes they should connect. The node created in the above example is near the upper left corner of the object. That means it should be connected with the node(s) along the upper edge of the intersection and the node at the lower left corner of the object. (There is the option of adding nodes at the center of each new polygon. In this case, the example node would connect to the nodes above and to the left of the object.)

It is important to note that the map is updated prior to each search. This is due to the unordered and unmonitored nature of many systems; the objects are updated in an arbitrary fashion, so it is simplest to update the map as a whole before each path generation. It is also the most costly.

One possible alternative is to enforce strict ordering; if all standard mobile objects are updated prior to any path agents, the map can be updated once per cycle. This assumes, however, that the agents themselves will not be an influencing factor on the search space. It also makes the system more rigid in terms of task management.

Another possible alternative is to track those objects changed. When a new path is created, any portion of the existing map not affected by the tracked objects can be treated as part of the static layer. Unfortunately, this would add overhead which may not be justified in many situations.

Example

The following diagrams illustrates the process of adding objects to a map via Polygonal Space Decomposition. The black rectangles represent objects being inserted into the map with the green dotted lines showing the resulting division. The space being decomposed is shaded and the nodes added to the graph are in darker.

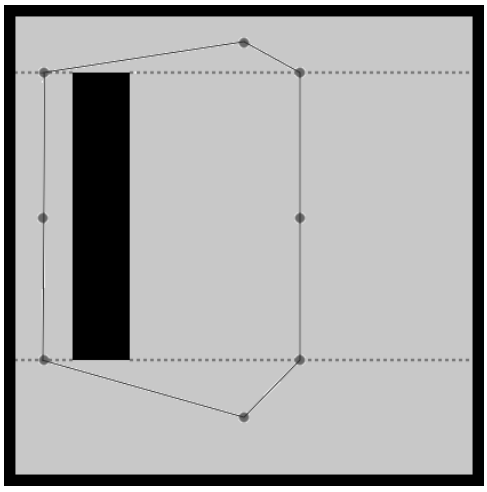


Figure 2.0 – First Step of Polygonal Decomposition

This first step is trivial. In the open area, all that need be done is calculate the location of the various nodes.

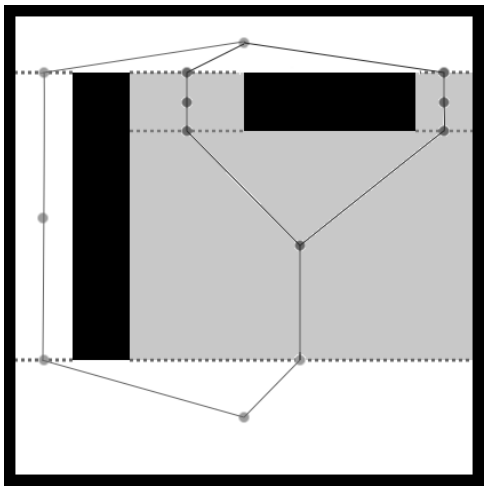


Figure 2.1 – Second Step of Polygonal Decomposition

With this second step, the object intersects the right partition. Lying along the upper edge, this object only creates three polygons. However, it does remove the node that was along the upper edge, replacing it with one to each side.

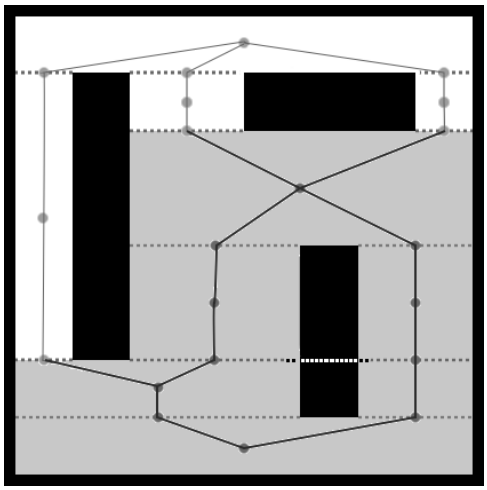


Figure 2.2 – Third Step of Polygonal Decomposition

The last object actually intersects two different polygons. To simplify the process, the object can be treated as two separate objects (as illustrated with the white dashes). Once this is done, the procedure can continue in a fashion similar to the last step, partitioning the two polygons in any order. Whichever section is decomposed first will move the nodes along the common edge. The following decomposition will simply link to these, unaware that the previous node existed.

Once this decomposition is complete, path finding is simply a matter of determine the closest node to the start and goal locations and then searching through the graph.

2.6 Time Encoding Maps

Overview

Time Encoding uses the field of an object's potential movement. This field is determined by three factors: the velocity of the object, the velocity of the path finding agent, and the distance between the two. Objects capable of greater velocities will naturally have a greater potential movement as they can reach more points in the same time frame. Meanwhile, decreasing the speed of the agent increases the time in which obstacles will have to move before the agent reaches them. The same holds true for distance; objects farther from the path finding agent will have the opportunity to move to a greater variety of locations.

Once the movement fields of the various obstacles are determined, the map is converted into a format that can be searched. Options include a boolean grid populated by probing at set intervals for collisions and a graph created using the spacial decomposition detailed above. By defining the search space so, Time Encoding causes moving objects to repel path finding entities, thus minimizing collisions.

The reason for taking this approach is simple. Traditional algorithms only work with a snapshot of the map. They cannot properly predict the future position of mobile obstacles and thus can easily guide an agent into a situation from which it must back out or otherwise adjust its course significantly. Time Encoding offers a means of handling a continuous set of states.

Though Time Encoding offers a good means of avoiding future conflicts, it is better suited for open areas. In a more constrained environment, the influence of various obstacles may deter a path finding entity from seeking certain avenues. For example, if a distance object is sitting in a narrow hallway, its influence may block off the whole passage even if the obstacle physically occupies less than half its width.

Procedure

The following describes the basic procedure of finding a path with Time Encoding:

```
General Program Flow
for each object...
      if static
            add to static object list
      else
            determine max velocity
            add to dynamic object list
      end if
end
```

```
for each execution cycle...
      for each object...
            if object is path agent
                  update map
                  perform search
            end if
            update object
      end
end
```

Updating Map

```
for each object in dynamic layer...
      calculate influence
      add influence to map
      convert map to discrete form
end
```

Calculating Influence

```
travelTime <- distance( agent, object ) / agent_maxSpeed
radius <- ( object_speed * travelTime ) + object_size
influence <- new circle( radius, object_center )
```

Converting Map (To Grid)

```
grid <- new matrix( totalColumns, totalRows )
cellWidth <- map_width / grid_width
cellHeight <- map_height / grid_height
for each row...
      for each column...
            x <- area_width * column
            y <- area_height * row
            sample <- new rectangle( x, y, cellWidth, cellHeight )
            grid[i,j] <- cost( sample )
      end
end
```

Calculating Cost (For Grid)

```
if impassable( sample )
      cost <- infinity
else
      for each static shape...
            if intersects shape
                  cost <- cost + percent( intersection )
            end if
      end
      for each dynamic shape...
            if intersects influence
                  cost <- cost + proximity( sample_center )
            end if
      end
end if
```

The general flow of the program is quite similar to that of polygonal space decomposition and is subject to the same limitations described above.

As for updating the map, there are several options available when converting the fields into a discrete form. One is to create a matrix of decimal costs based on the amount of space available within a sample square. Another would be to use the shape of the fields

as part of the Polygonal Space Decomposition technique.

As discussed, the size of the influence field is dependent upon three factors: the time needed for an agent to reach the object's current location, the object's greatest potential speed, and the distance between the two.

To convert the fields into a grid, the overall map is divided into cells, the size of which may be determined by a predefined number of rows or columns. For each of these cells, a sample area is created. The cost for the cell is based on two factors: the percentage of the cell that is impassable due to static objects and the proximity to any objects whose influence effects the sample area.

Once the map is in a discrete form, it can be searched using a basic tree search.

Example

The following image shows an example of the influence fields generated with Time Encoding. The rectangle represents the agent and the circles are mobile obstacles, the shaded area being their potential movement fields. The numbers represent the relative cost once the map is turned into a discrete grid.

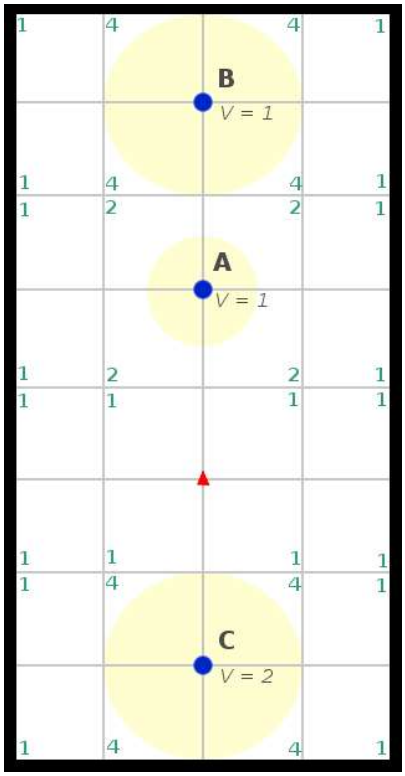


Figure 3.0 – Weights calculated by Time Encoding

Object A and B both have equal maximum velocities, but Object B is twice as far away, therefore its movement field is twice as large. Meanwhile, Object A and C are both the at the same distance, but Object C has a velocity twice that of Object A and thus has a

field as Object A.

The areas around Object A and C will have the greatest cost when the map is discretized as those cells are mostly filled by the movement filed. The cells around Object A are only partially filled and thus will have a lower cost.

2.7 Evolutionary Algorithms

The concept of computer learning through evolutionary strategies has been applied to path finding in numerous studies [5][19]. As with other problems, this research has aimed at evolving a more efficient path finding technique capable of handling dynamic environments. The potential benefit is that such an algorithm will react faster and run more efficiently than any developed directly.

There are several drawbacks when applying evolutionary tactics to path finding. There are many different types of map layouts as well as object movement patterns. An all-purpose path finder would require countless combinations for the learning process. A similar problem occurs with map representation. Path finders would have to be evolved for particular applications, which could become problematic. A robot on a distant planet might encounter unexpected terrain; game designers may make modifications to the data structures used for the game's maps. Though hopefully the algorithm would be able to handle these changes, it is possible that it will become sub-optimal and may even need to be retrained to respond.

Another drawback is the presence of human interaction. Path finding applications, such as computer games, where humans interact with the agents present another challenge for evolved path finders. Each user may behave in a novel fashion making it difficult to train the path finder when it must adapt to the user's actions. This is one reason many game developers are skeptical of evolutionary algorithms. Traditional structured algorithms can be designed and modified to anticipated likely exploits and provide reliable – if less clever – artificial intelligence. The same holds true with other applications; without understanding how the path agent works, it can be difficult to effectively employ.

3 Proposed Solution

3.1 Concept

Objective

The primary objective of developing Midpoint Seeking is to create a computationally simple algorithm that allows individual paths to be generated quickly. If the amount of calculation is kept to a minimum, paths can be updated more frequently, leading to more effective paths. Less waste and fewer conflicts are also desirable, as this will conserve system resources. Ideally, it will be capable of acting as an independent alternative. This will be the focus of this study; however, it may also serve as a supplement to other path finding techniques.

Inspiration

The inspiration for the Midpoint Seeking algorithm comes from considering the following question: How might a human being go about finding their way from one point to another when the terrain between is unknown?

The shortest line between two points is a straight line; such knowledge is intuitive. Indeed, when such a path were possible, a human being doesn't even need to think about how to reach their destination. This straight line is the preferred path and we humans attempt to follow it as closely as we can. However, people also tend to select paths that require minimal changes in direction. For example, if there is an sizable object in the way (one tall enough that is cannot be stepped over), most people will slowly step to the side rather than wait until it is close before curving around it.

Acknowledgments

These observations were used in the design of the original Midpoint Seeking algorithm. They also influenced the work of Bernard Moulin and Driss Kettani [18] who were interested in generating a realistic description of a path via computer.

Moulin and Kettani's system considers major obstacles (buildings) and standard path areas (walkways) when finding a path. The traversable areas are segmented by the influence of obstacles. For example, the portion of the sidewalk in front of a library would be a separate segment from the sidewalk to either side. These segments served as path nodes; during the path search, however, distance from the start and to the goal were not the only parameters used for selection. The relative orientation of nodes was also considered. Veering away from the goal was discouraged as was frequent changes in direction.

Since the focus was on realism, the algorithm was not optimized for fast computation. Nonetheless, some of the concepts presented proved useful in developing Midpoint Seeking. Chiefly, they were the idea of using an individual object's influence and the idea of using the difference in orientation as a parameter in a path node's cost. (Prior to their incorporation, Midpoint Seeking checked points at positions relative to the midpoint and was concerned solely with clearance.)

Another body of study related to Midpoint Seeking is the work by Rob Teather [24]. His algorithm also uses the midpoint between the start and end positions to define the path. Subsequent midpoints are checked for validity until a complete graph around obstacles is created. However, whereas Midpoint Seeking was refined to handle dynamic environments, Teather's work focuses on generating a viable set of way nodes. It is well suited for handling arbitrary masses, but is not designed specifically for handling dynamic environments.

3.2 Algorithm

Basic Procedure

To calculate paths in a highly dynamic environment with limited resources, I propose that an alternative methodology is needed – an algorithm that minimizes the calculation needed to find a new path thus minimizing "wasted" calculations and maximizing the frequency with which paths can be generated. To this end, I have developed an algorithm which I call "Midpoint Seeking."

The basic steps of the Midpoint Seeking algorithm are as follows:

- Find the midpoint between the current location and the goal.
- Check if this point is traversable. If not, examine points adjacent to the obstacle in question. Select the one with the least distance to goal and least difference in angle from the agent's current trajectory. (Alternatively, the distance from the agent to the point can be used; reaching the closer points typically requires a smaller change in direction.)
- Once a legal point is found, find the midpoint between the legal point and the start location.
- Recursively select more points until a predefined condition is met, such as a set number of iterations or a set distance between the agent's location and the point being checked.
- Connect these points to create a tentative path.
- At set time intervals, calculate a new path using the path finder's new location.

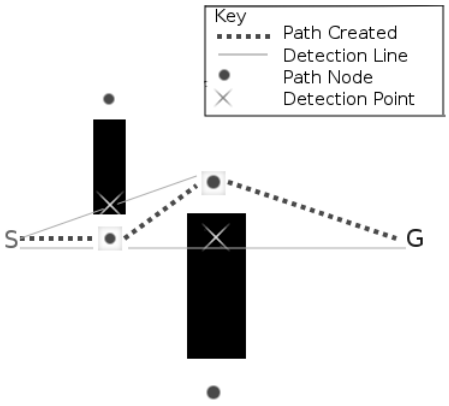


Figure 4.0 – Midpoint Seeking in Action

During initialization, all objects in the environment, regardless of potential movement, are analyzed and a series of "influence points" are calculated for each. These are the points used when selecting a path. Their positions and the associated orientation vectors (used for angle calculations) are relative to the associated object meaning they are implicitly updated as the object is moved. There is no need to recalculate the map whenever a path is needed, reducing the overall cost for a path.

As for the computational cost per individual cycle, it is presumed that the above procedure will be less expensive. It requires little more than simple arithmetic over a small (and possibly fixed) number of points. However, this will need to be affirmed through detailed analysis, proper implementation, and extensive experimentation.

Also, there may be additional considerations concerning the design of the algorithm that will become more apparent after studying Midpoint Seeking in action.

3.3 Potential Advantages

There are certain features of the Midpoint Seeking which will (in theory) permit it to be more adaptable or otherwise more desirable than previous solutions.

Quick Path Generation

This is the primary motivation of creating Midpoint Seeking. As previously discussed, Midpoint Seeking should produce each individual path in a much shorter time. This will allow more frequent path calculations, which in turn means the agent can respond to a greater number of map states. By reducing the influence of any one path, it is hoped that the emergent path will better represent the ideal path for the entirety of the goal seeking process. In addition, it will improve the responsiveness of the path finding.

Minimal Search Space

Previous solutions still rely on the traditional approach of searching available space rather than considering space which cannot be traversed. Though algorithms such as A* are designed to limit the area that is searched, the potential remains to explore excessive amounts of the search space. The load from path finding on a map with many moving objects could cause sudden performance degradation under certain conditions. For example, a bottleneck at a common passage point would force other agents to expand their search farther.

A similar situation can occur with Midpoint Seeking, but it is far less likely, particularly when influence areas can merge. This is because Midpoint seeking will need to consider additional obstacles, not additional space; in a typical situation this means less nodes to explore.

An excessive increase of nodes searched will only occur when the following conditions are met: 1.) the recursive cut-off is not constant 2.) the number of obstacles along the alternate path is significantly greater 3.) the influence area of the obstacles do not overlap 4.) the obstacles are arranged such that circumventing one forces the agent to avoid another. The first condition is a matter of implementation, while the other three require very specific events to occur. Contrast this with the condition needed to “spike” with other methods: alternate avenues that exceed a certain length.

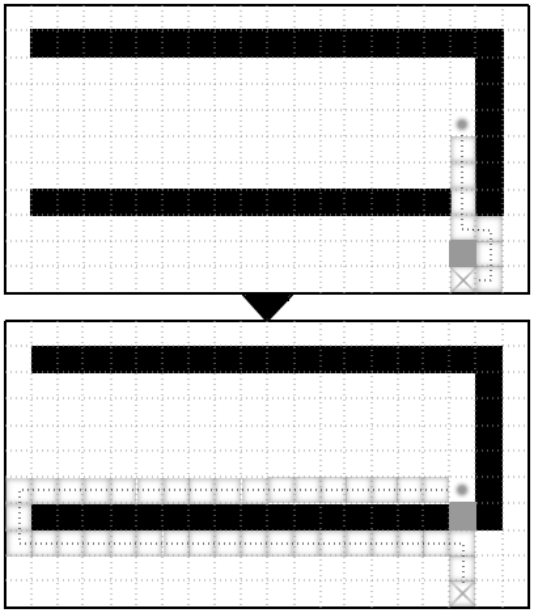


Figure 5.1 – A spike during a grid search

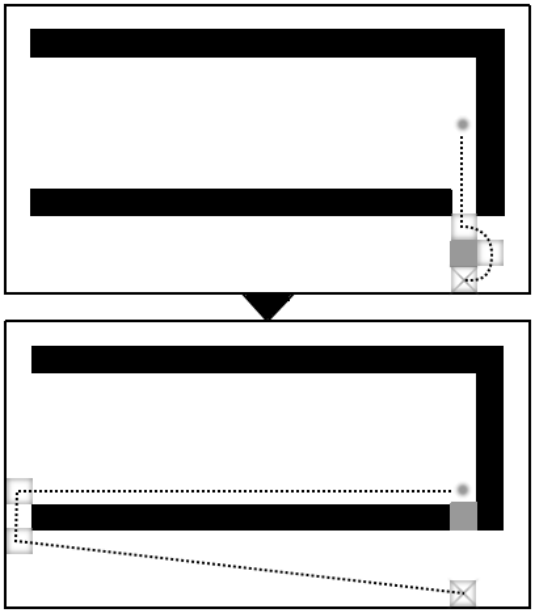


Figure 5.2 – Point Seeking in the same situation

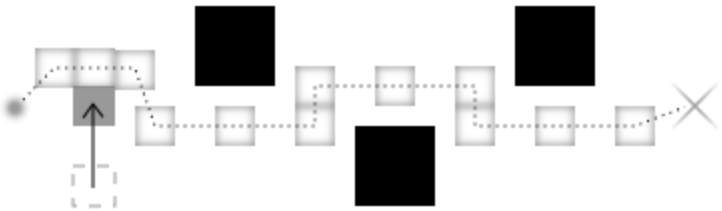


Figure 5.3 – A situation where Point Seeking can spike.

Object Oriented

The structure of Midpoint Seeking allows for an object-oriented paradigm in path finding. Other techniques divorce the physical presence of an object from its semantic nature. The objects become indistinguishable to the agent searching for a path. This need not be the case with Midpoint Seeking, however, as the influence points guiding the path are directly associated with an object.

How might an object oriented paradigm aid in path finding? One instance would be handling varying influence areas. Depending on the nature of the object, it may be necessary to discourage the path finding entity from approaching. For example, keeping a robot away from damaging heat sources.

Using an object oriented paradigm, several solutions are possible: 1.) Modify an object's influence area so that it is remains proportional to the object's current temperature. 2.) Maintain separate sets of influence points, one for each degree of

temperature-sensitivity. 3.) Upon recognizing a potential obstacle, identify its temperature and adjust accordingly.

With any of the above approaches, the association between the physical presence of an object with other data can streamline the process. Note, however, that the focus of the solutions are different. Option 1 makes the state of the influence area independent of the path finding agent whereas Option 3 requires the agent to adjust the influence as it sees fit. Option 2 allows for compromise, but at the cost of memory and increased computation.

To appreciate the advantage of object orientation, consider how a similar effect might be achieved with other algorithms. With Adaptive A*, path nodes surrounding the heat source could have their costs increased to deter selection of those points. However, some form of connection would have to be added in order to know when the node could revert to normal; otherwise, a process would have to be continually monitoring and updating the nodes.

With Polygonal Decomposition, Option 1 is viable. The object's shape could be enlarged; depending on the structure, this may not be an expensive procedure. However, it would not permit any distinction between path agents. Options 2 and 3 would require that the map division process be informed of the agent's sensitivity. In addition, it would require separate calculations of the static space, either multiplying the amount of map data to retain or negating the advantage of polygonal division by forcing a new spatial division for all objects.

Time Encoded searches would not have any significant problems with adopting a object-dependent solution; after all, the influence field calculations are dependent upon knowing both the agent's velocity and the velocity of the potential obstacle. Accounting for other factors would be a simple matter of adjusting a bias of some form. The only potential drawback is that areas considered impassable due to possible collision would be not distinguishable from those impassable due to heat.

These examples illustrate how comparatively simple it is for Midpoint Seeking to integrate object states into its path calculations. A higher level of abstraction can be maintained and the amount of additional computation and messaging needed is minimal.

Scaling Representation

As mentioned above, when depicting a map, there is often a trade-off with between accuracy and memory usage. For example, with a simple grid, a coarser grain will improve search speed but may cause the agent to either collide with or overact to a small obstacle. Meanwhile, a finer grain will improve path accuracy but increase the expense of finding a path. Similarly, a graph of way nodes will need to be denser to conform with more complex map contours. Map representations that can scale to handle objects of varying sizes inherently balance path speed and accuracy.

Spatial decomposition techniques have the property of scaling representation. However, they are difficult to optimize. Figure 6.0 below is an example of Binary Space Partitioning [26][9]. It demonstrates how the order of obstacle insertion determines the number of resulting nodes. Arranging the obstacles to be inserted in the best order can introduce significant overhead.

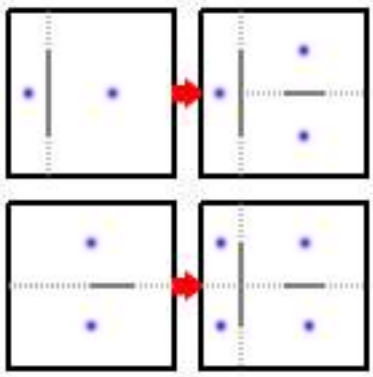


Figure 6.0 – Binary Space Partitioning with Different Object Orders

Midpoint Seeking has the advantage of scaling representation without having a search space that is dependent upon insertion order. Every obstacle is given a constant number of influence points. Linking of these points can be done in any arbitrary order and will lead to the same configuration.

The description of the physical environment is not the only aspect of the path finding that benefits from Midpoint Seeking's scaling representation. The paths generated also adapt to the varying scales within a map; they are only as complex as necessary to navigate the given map geometry.

This is, in part, because Midpoint Seeking forgoes the traditional space-oriented searching. There is no need to convert the traversable area into a discrete search space. It is not the focus of this algorithm; instead it is only the potential barriers that are considered, allowing the algorithm to handle the search space in a continuous manner.

Continuous State Analysis

Most path finding algorithms work with only the current status of the environment. The states they analyze are discrete and disjoint. This can cause particular events to over-influence the path. For example, a normally open doorway may be briefly obscured; if calculation occurs at this time with these “snapshot” techniques, the resulting paths will ignore a viable option. To properly avoid this, an algorithm needs to be aware of all possible states between the initial states and arrival at the destination.

Using prediction based models, such as Time Encoded Maps, can achieve the appropriate level of awareness. Midpoint Seeking does this to a lesser extent. Rather than using predictive models to consider what might occur, it uses “brute force” to capture as much of what actually does occur.

The difference between these two effects is important. A prediction model is like using grid paper to plot out a course across rugged terrain. The less desirable regions are crossed out, leaving the most clear path. The trade-off is some of the excluded terrain might not be as hazardous as you expect; over caution will delay arrival at the destination.

Meanwhile, Midpoint Seeking is akin to remotely driving a vehicle through an automated factory using a mounted camera. If the camera receives and transmits

information at a fast enough pace, you will get the appropriate sense of speed, location, and movement of all the machines. You'll be able to keep a safe distance so long as the machines don't make any unexpected movements. Delays may reduce your accuracy, but as long as there is no sudden suspension of the visual feed, navigation should still remain reliable. This system allows more freedom in choosing a path and thus improves the ability to find one that is more optimal.

The important aspect of this distinction is how the algorithms will respond with fewer updates. If there is a large lapse, Midpoint Seeking may not respond efficiently, but with other algorithms such as Time Encoding, paths will be generated using outdated, inaccurate maps.

Note that there is no reason Midpoint seeking could not effectively employ predictive models. A minor adjustment to the placement of influence points is all that would be needed. The speed of the algorithm should not be compromised noticeably if the prediction calculations are efficient.

4 Examining Current Methods

4.1 General

Objectives

Before designing and implementing Midpoint Seeking, it would prove useful to examine existing algorithms so as to understand their strengths and weaknesses. To this end, Polygonal Space Decomposition and Time Encoding were implemented and tested. These two algorithms were chosen as they share a common trait; all three focus on representation of the environment.

The first series of experiments conducted studied the performance of Polygonal Space Decomposition and Time Encoded Maps when key parameters, such as the frequency of path recalculation and the underlying tree search method used, differ. This is to provide a more robust appreciation of the algorithms. It will also help select what parameters are more suitable so they can be more appropriately compared with Midpoint Seeking.

Metrics

The following features will be considered in accessing the performance of the path algorithms:

- **Execution –** The average number of cycles and amount of time it takes for an entity to reach its goal when using the given algorithm. This will be influenced by the size of the search space, the number of entities running concurrently, and the objects present in the map.

- **Waste –** Any node or point along a path that does not directly influence the movement of the entity will be considered wasted. Though the actual "wastefulness" of an algorithm will vary with the amount of computation per node, considering the number of wasted nodes offers a clear comparison between algorithms.
- **Conflicts –** The number of times an entity collides with another object due to inaccurate pathfinding. The frequency with which the algorithm will stall, hesitate, or otherwise fail to continue on a proper path should also be considered.
- **Path Length –** The length of the resultant path. This will be used to approximate the optimality of the algorithm. In a static map, or a map with a minimal number of object states, the best path can be calculated; algorithms can be compared against an absolute measure. These experiments, however, will use more dynamic maps. The length will be used as a relative measure of how "reasonable" it is.

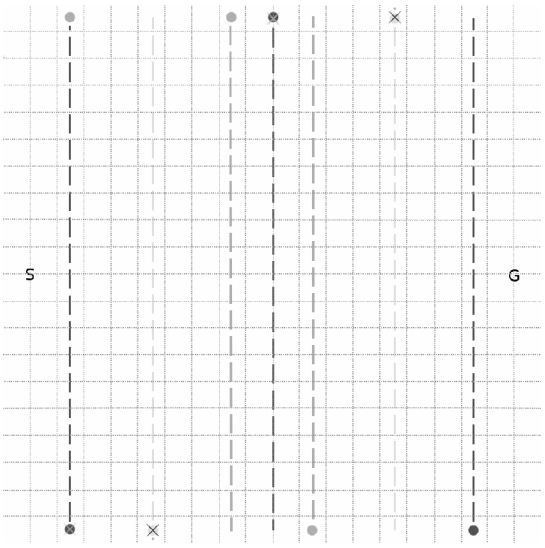
Maps

The following diagrams illustrate the maps used for the various experiments. Solid black objects represent impassable walls. The letters "S" and "G" mark the start and goal respectively. The dotted paths represent the paths of scripted objects; the shade represents which layout the object is present in. Light gray means the object is present in light density maps and greater. Medium gray symbolizes medium density and greater while dark gray is used for those objects present exclusively in heavy density maps. Note that with some maps, an object may be moved, eliminated, or reinserted in a greater density version. An "X" represents removal or modification and a small dot means reinsertion.

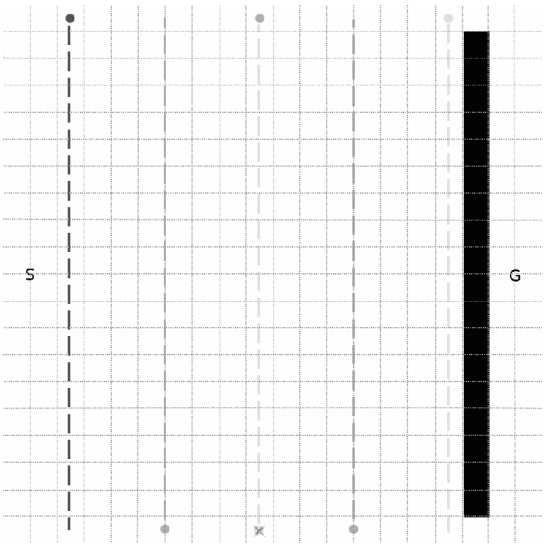
The latter maps contain additional objects. Simple dots without corresponding paths are wandering objects. Boxes marked with an "A" are adversarial objects that chase the path agent while rounded bars represent fading panels.

The coordinate system used for path finding is the same as that used for representing graphical objects; it is measured in arbitrary world units. The sizes of the maps range from 500 to 1500 world units for both dimensions. The dimensions of the walls are typical multiples of 50, but thinner walls are only 10 units thick. The path finding agent occupies an area of approximately 20 by 20 units.

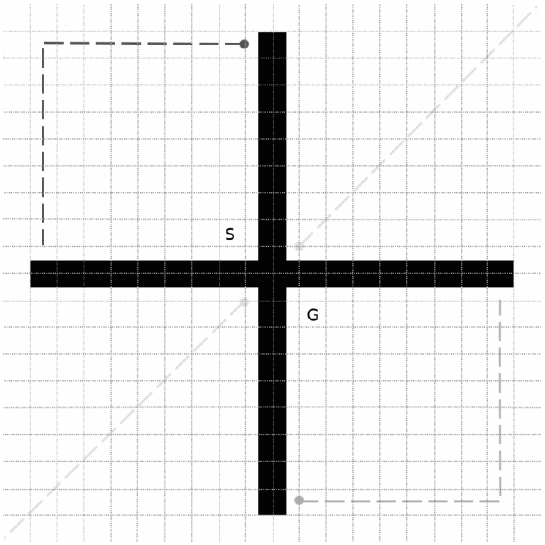
Map Set 1 – Initial Experiments



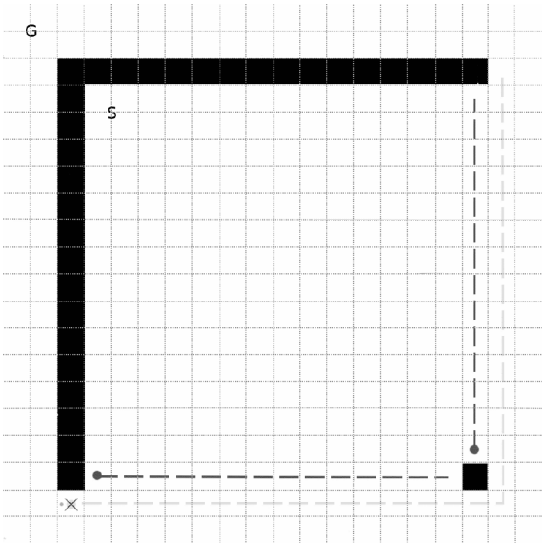
Map 1.0 Open



Map 1.1 Bar / Reverse Bar

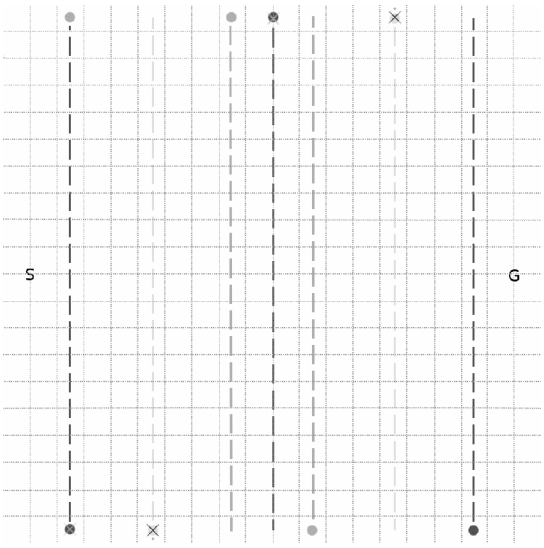


Map 1.2 Cross

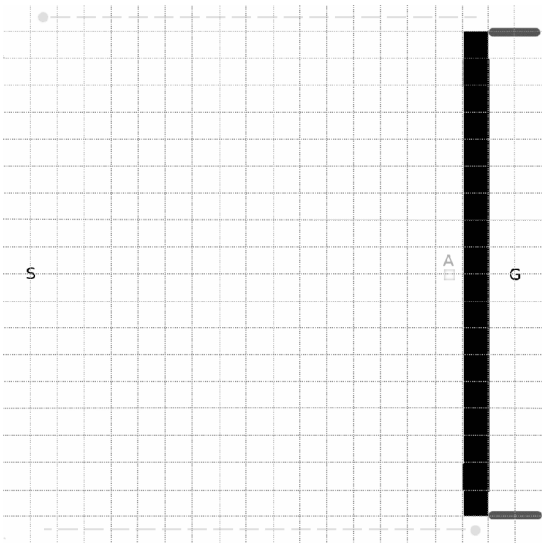


Map 1.3 Elbow

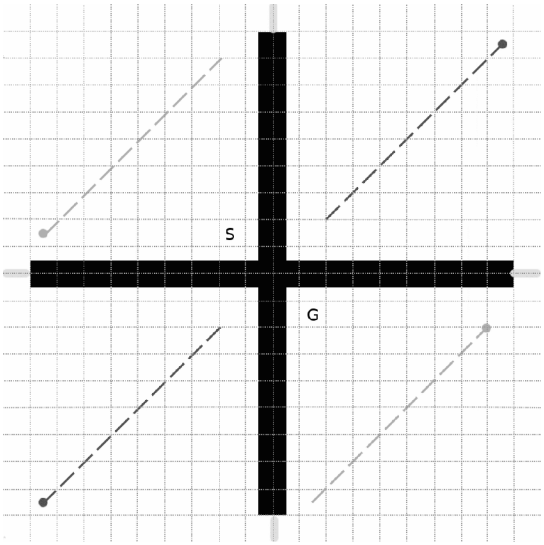
Map Set 2 – Later Experiments



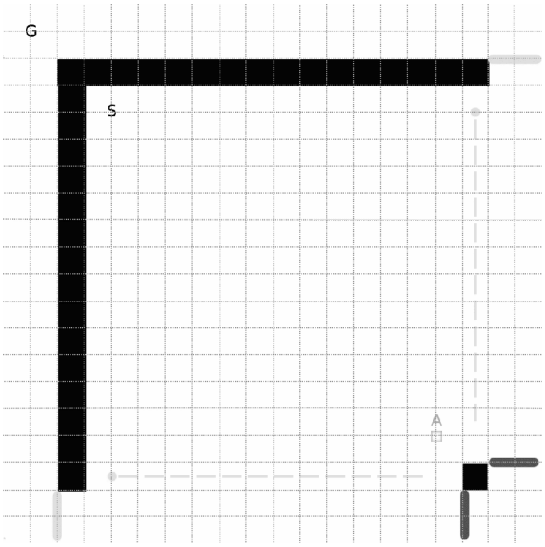
Map 2.0 Open



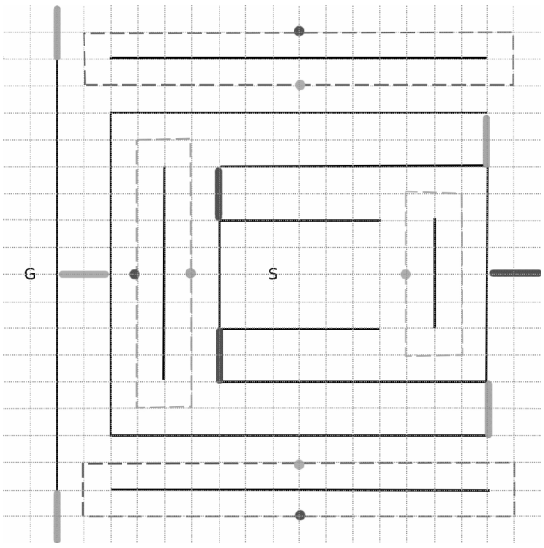
Map 2.1 Bar / Reverse Bar



Map 2.2 Cross



Map 2.3



Map 2.4 Prison

4.2 Implementation

Introduction

To conduct the experiments, the game engine developed by Dave Brackeen was used. The primary reason for choosing this particular engine is because of previous experience using it. All of the source code is ready available and easily modified as well. This reduced time and possible errors as learning a scripting language or a new system was not necessary. It also made the system more readily adapted to better serve the experiments. For example, the three dimensional collision detection using Binary Space Partitions was discarded in favor of a simpler, two dimensional collision detection system that worked more seamlessly with the two dimensional implementations of the various algorithms.

To gather data, pathfinding "bots" were created and launched in real time. While finding a path through the environment, most of the pertinent information was recorded and written to file at the conclusion of the session. Each session was limited to thirty seconds; if the agent could not reach the goal in that time, it was assumed to be caught in an intractable situation (as was most often the case.)

Different behaviors were considered for the bots. The simplest would be a "dumb" bot with no means of rectifying an erroneous path. This would require the path finding algorithm to do all of the work in evading obstacles. Though it would serve as a good test of the path finding algorithm's robustness, it would ultimately cause many trials to fail when only a minor course correction was needed. Thus "smart" bots capable of obstacle avoidance were considered. Such bots, however, would make it difficult to properly assess conflicts. If the agent didn't hit the wall but was forced to steer around it, should that be considered a conflict? Also, to what degree could avoidance of an obstacle be contributed to the algorithm and not the bot? Despite these concerns, "smart" bots were

used as this is a very common feature in current path finding applications.

The experiments were all conducting in real time so as to provide realistic data. Simulations would have allowed for faster trials (and thus more data) but may not have properly represented the load created by moving and rendering objects in a real-time environment. It also allowed for direct observation of the agent's behavior from which valuable insight could be gleamed.

Note that this game engine was used throughout the entirety of the experiments described by this paper.

Original System

The original engine was designed to illustrate path finding using a simple First Person Shooter game. The player can move their character with the keyboard arrows and look around with the mouse. They can also fire projectiles with the left mouse button and jump with the space key.

Pathfinding is illustrated by a series of agents controlled by a basic artificial intelligence system. Once placed in the system, they find a path to the player's character, adjusting as the player moves throughout the environment. This is achieved via the basic *PathBot* class. The engine periodically updates the path bot(s), passing the player object and the time since the last update. This information is used to determine if another path should be calculated. If so, the path bot passed information about itself and its current goal (the player object) to a path finder object. This path finder then calculates a path to follow.

The system provides a set of features for creating a virtual environment. It can parse a specifically formatted file to create a map. (See the *MapLoader* documentation for details.) This map is represented using Binary Space Partitioning; the resulting BPS tree is used for the graphics rendering, collision detection, and path finding. The system also provides classes to handle sound and input and utility functions for geometric measurements and thread management.

All of these subsystems are created and coordinated via the system's *GameCore*. To create a custom environment, users need only create a map file and – if they wish to use images other than the defaults – a texture list and the corresponding image files.

Basic Modifications

Several modifications were made to make the Brackeen Engine more extensible.

A means of specifying parameters via data files was added. The *ObjectPathList* class was used to locate resource files while the *SystemPropertyList* class helps specify any extra parameters needed for a particular scenario. The *ObjectPathList* was a necessity since the IDE used for development, Eclipse, did not execute the program in the directory expected by the original engine. It could not find the image and map files using the hard-coded relative paths.

Several features were added to the collision detection. One is the *CollisionDetector* class. It groups a set of objects with a collision detection system, providing a simple means of checking collision for a specific object or to check if a

straight line is interrupted by any obstacles.

Another addition is the *ImmaterialObject*. This interface causes the object to be ignored by the collision detection thus allowing custom objects to be inserted into the environment without colliding with other objects. Such an object can be used for tasks such as creating a visual marker. Related functionality was added to the *Polygon3D* class that allowed polygons to become invisible and thus not rendered.

The basic *Vector3D* class was enhanced with additional functionality. A vector can now calculate its distance from another and can judge whether it is "near" another. (The later is to judge equality of two vectors without being affected by differences due to floating point representation of values.) Conversions to two-dimensions is also provided, mostly to centralize changes needed to later extend the system.

Two-dimensional conversion was also provided for *PolygonGroup*. For each vertex, the x and z coordinates (which represent the horizontal plane) of each vertex were used to define a two-dimensional polygon. In essence, the object would flatten itself.

Additional functionality was added to the *MoreMath* class as well. *quickDistance (Point,Point)* does a quick estimate of distance as an alternative to the standard Euclidean method. *difference(Point,Point)*, and *relativeDifference(Point,Point)* provide ways to compare points while *bind(int,int,int)* is provided to clamp values within a given range. As with *Vector3D*, these changes were to extend the system, but also provide some general utility.

The *GameObjectManager* was changed to work with a "focus" object as opposed to the "player" object. Though this is a semantic difference, it helps to generalize the system; it is likely that an object other than the player object will be used as a goal for path finding.

The most important change is a small one with a profound impact. The signatures of the methods within the *PathFinder* interface were changed to have a different return type. (All implementing classes were changed accordingly.) The original path finders converted the path into a unmodifiable list of nodes and returned an iterator over this collection. To move, the path bots would extract points via the iterator.

This proved problematic later in development of particular algorithms. The path calculated by a path finder could not readily be recorded, modified, or analyzed by any outside classes. To make the calculated path more accessible, the step involving the unmodifiable collection and its iterator were removed; the new interface provides a list containing nodes which the calling class can use as it needs.

Extended System

The first step in extending the Brackeen engine into a test bed for path finding experiments is to change the perspective. A top down view is more appropriate for visually studying agents as they path through the environment. This is the primary role of the *BasicViewer*; it alters the existing *PathFindingTest* so the camera is controlled by the user panning and zooming freely. The player object is turned into a point of reference.

The *BasicViewer* uses a *DynamicMapLoader*. This loader allows the user to specify an initial position for the camera. It also offers options to help the map author to add dynamic objects. For example, the *DynamicMapLoader* recognizes the declaration of

a “movement” which defines a set of points and the time intervals at which a scripted object should reach those points.

There are a variety of viewers derived directly or indirectly from the *BasicViewer*, such as *MapViewer* which visualizes the data from the map, laying it over the view of the environment. These tend to be specialized viewers used to analyze particular scenarios.

The key viewer is *TestViewer*. It takes a series of command line arguments which specify the search method and map representation to use for path finding. Using this information, it can create the game objects as well as any path bots, path finders, and data maps needed to run tests. Its child, *TestRecorder*, takes various measurements (with the help of and outputs these to file.

Many viewers, including *TestViewer*, use a *DataMap* to represent the environment. The *DataMap* is a two-dimensional representation of the environment tailored for the path finding method in use. *DataMap* is necessary as many algorithms are designed to work in only two-dimensions; the 3D objects are flattened to simplify the search space. (*CollisionsDetectionWithBounds* is provided to do collision detection in this new space.) This conversion was also needed because many algorithms are incompatible with the BSP tree used by the Brackeen engine.

The *DataMap*, combined with a *Visualizer*, also provides a way to visual display the path finding process. The *DataMap* shows the objects in the map as they are represented by the current algorithm while the *Visualizer* is called by the path finding system to display important information, such as nodes selected, used to generate a path.

The remaining classes serve one of the following functions: a) runs a test routine, b) represents the environment in a fashion needed for a particular path finding algorithm, c) represents a special object (e.g., an object following a scripted path), or d) performs path finding calculations. These systems all pertain to the design, implementation, and testing of individual algorithms and have no bearing on the core test system.

Using the System for Other Applications

To design and test additional algorithms, following these steps:

1. Determine the fashion in which individual objects as well as the map as a whole will be represented. Extend DataMap if you need to create a new map representation.
2. Decide how nodes will be generated and represented. If need be, create a sub-class of SearchNode.
3. If applicable, select or create a tree search method.
4. Design the path finding routine. The class that creates the path needs to conform to the PathFinder interface.
5. Modify the createMap(), createPathFinder(), and createMethod() functions in TestViewer so they create the objects needed for the new algorithm when the appropriate parameters are passed into the system. Alternatively, the TestViewer or TestRecorder can be extended and new versions of these methods written.

Class Diagrams

The following diagrams outline a few of the key classes.

This first set of classes are descendants of the GameCore designed to view and record experiments. They coordinate the map construction, object management, environment rendering, and the artificial intelligence.

The second set of classes represent objects within the environment, distinguishing mobile objects from static one. The path bot, originally a fairly minimal class, was refactored and augmented. Not only is now more capable of running any arbitrary path finding algorithm, it is also now counted as a moving object within the environment, allowing multiple path finding agents to recognize and path around each other.

The last set of classes are the basic interfaces a new algorithm will need to implement in order to be integrated into the current system.

Diagram 1: General Viewer Classes

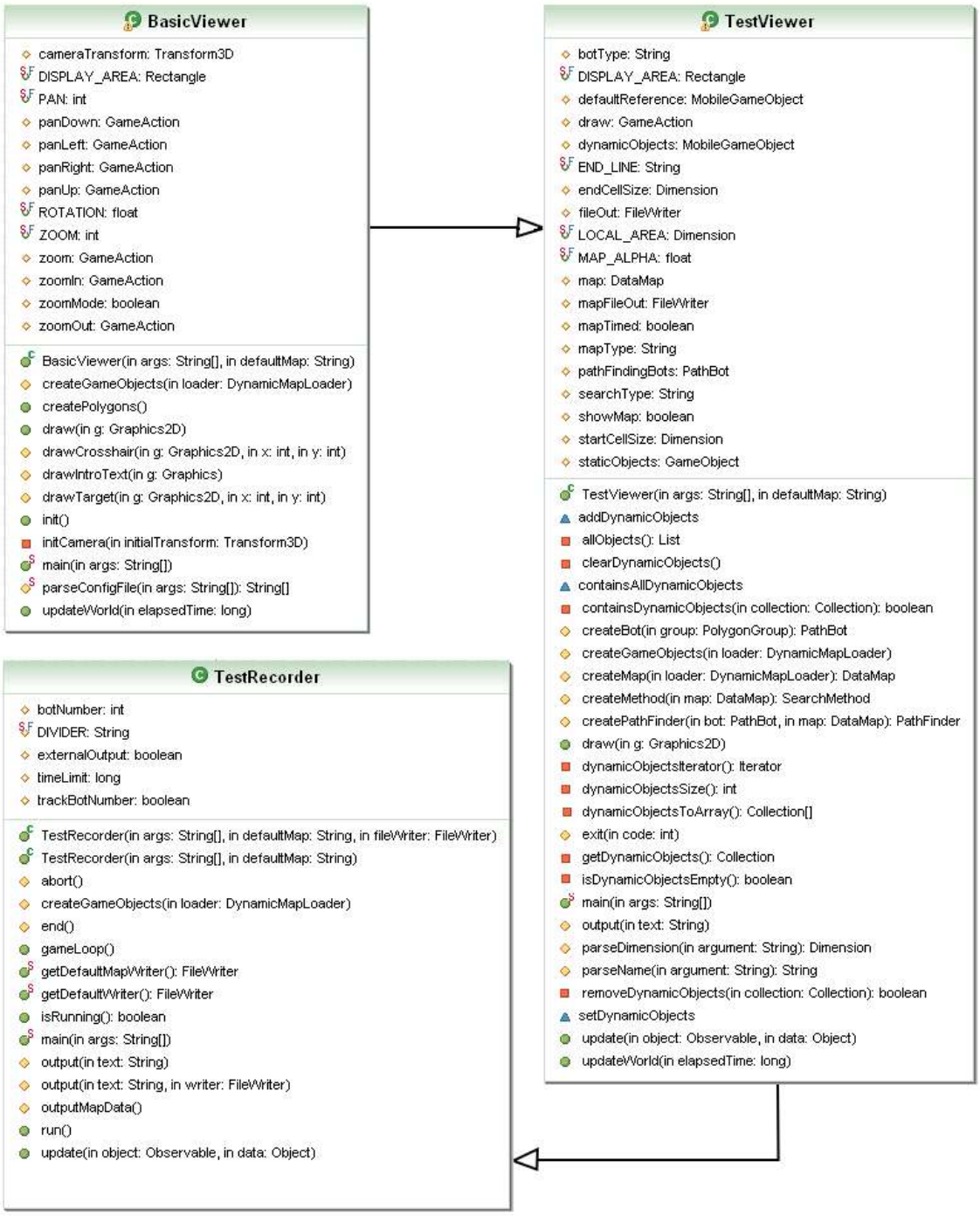


Diagram 2: Various Game Objects

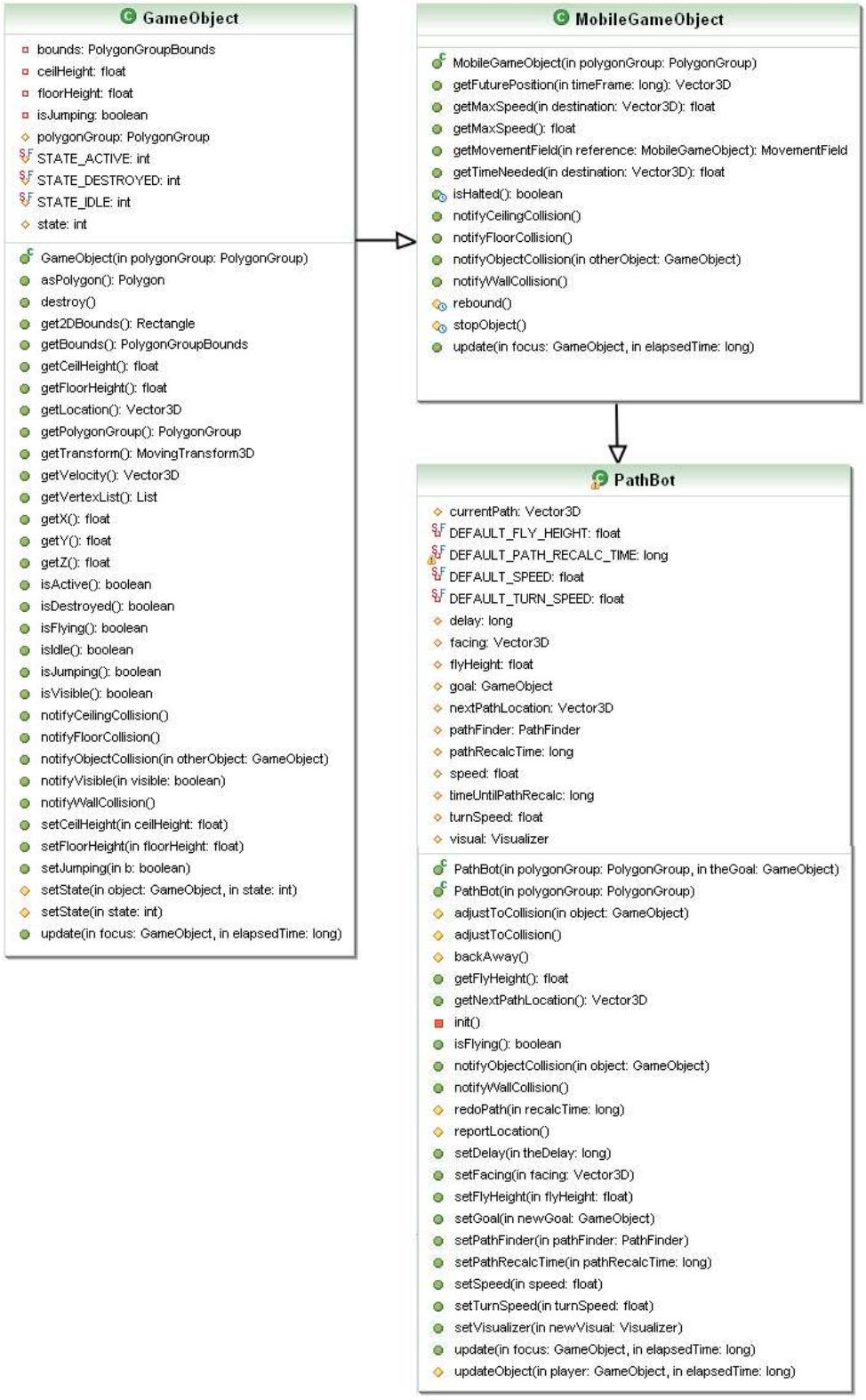
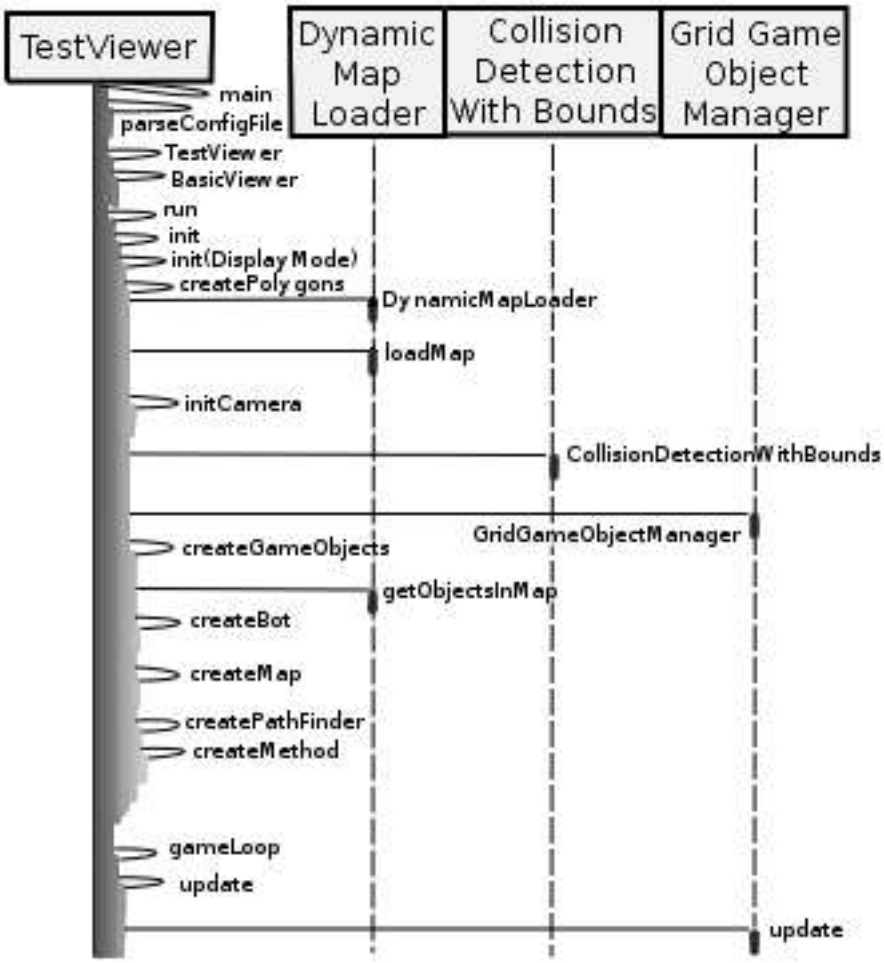


Diagram 3: Interfaces for New Algorithms

Diagram 4: Sequence Diagram

The following illustrates the basic sequence in which experiments are conducted.



The TestViewer class begins by parsing the configuration file which can be specified by the command line. This is used to, among other things, create the viewer with the appropriate settings and determine the algorithms to use. Once this viewer is run, it will initialize itself, parsing the map file to define the world geometry, camera position, game objects, and the corresponding data representations.

Once this setup is complete, the viewer begins the game loop. This loop calls the graphics rendering, collision detection, and artificial intelligence systems. It continues until terminated by the user or the path agents all arrive at their destinations.

Additional Documentation

The Javadocs provided with this summary offer more information on specific classes and functions. Note that the Brackeen engine is not fully documented as much of the information was not provided by the original author.

The accompanying UML diagrams outline each class, showing the various relationship within a package. There is also a sequence diagram outlining the program flow of an experiment conducted with the TestViewer.

4.3 Using Polygonal Space Partitioning

Objectives

The objective of this experiment was to determine suitable parameters for Polygonal Space Partitioning when operating within the Basic Maps described in Appendix C. The two primary parameters are the frequency of path updates and the tree search method (Depth First, Best First, A*, or IDA*) most suitable for the task. For A* and IDA*, the Euclidean Distance was used as the heuristic.

Results

Table 1.0 summarizes the results of this experiment. Table 1.1 shows how the algorithm performed with maps of varying numbers of obstacles.

Table 1.0 Polygonal Space Decomposition

Method	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
			World Units*	Milliseconds	Milliseconds	
Depth	1.11	64.53	1709.78	18.8072	1.1519	31.7%
Best	2.64	55.68	2815.23	17.7729	0.9473	38.3%
A*	1.77	25.79	2141.64	6.3175	0.7712	78.3%
IDA*	1.25	1.14	1556.33	7.0502	3.0062	75.0%

Table 1.1 Polygonal Decomposition (Using A*) with Vary Obstacle Densities

Density	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
			World Units*	Milliseconds	Milliseconds	
None	1.11	12.83	1981.64	6.3021	1.1400	80.0%
Light	1.26	24.48	1786.64	5.6294	0.7682	68.9%
Medium	1.8	20.33	2023.25	14.4226	1.3001	66.7%
Heavy	2.32	24.23	1918.75	9.2034	1.4536	62.2%

* - Lengths are in World Units, map range from 500 to 1500 World Units in height and width.

Analysis

The first important observation is the success rate of the various methods. Depth First timed out quite often, particularly with frequent updates. Best First was not much more successful in this regard. A* and IDA* have significantly better rates, seldom failing in less strenuous situations. (See Table 1.1.)

Failure can be attributed to many different factors, but two causes pertain to the particular nature of the search methods. Depth First and Best First both descend straight down the search tree. The first consequence of this is a jittering effect that occurs when near the border of two polygons. If the obstacle that generated the polygons is moving in a similar direction as the agent, the agent will continuously play “catch-up” with the node directly in front of it. The agent is fixated on this node because it is either the first it detects (Depth First) or it is the closest and least costly (Best First). The second consequence of the straight tree descent is oscillation between two points. This was especially pronounced with frequent updates of Depth First. This problem arose because the dynamic layer was partitioned in no particular order; Depth First would switch between the two nearest points while Best First would do the same if the costs of the two were equal.

Other conflicts, which were likely the cause of failure for A* and IDA*, include extreme angles between nodes and failure to respond quickly. Extreme angles (either near zero or near infinite slopes for the line connecting the nodes) caused issues when the agent approached the later node. Even with the bot's reactivity, the agent was not always capable of moving around the object when approaching at such an angle. As for response time, if the bot was “blind-sided” by a moving block, it was sometimes forced into a wall and trapped there. This prevented it from moving when it did eventually calculate a new path. (For later experiments, the maps were adjusted to prevent this from occurring.)

A* and IDA* proved to have different advantages. A* was certainly faster in executing. It also has a better success rate. However, there is greater variance in the performance. More importantly, there was frequent conflicts. IDA* may have been slower on average but it had shorter paths with fewer conflicts and greater consistency.

A plausible explanation is the general direction the two algorithms prefer. A* searches have no limits as to how deep they go before expanding breadth-wise. As a result, the path it generates tend to head straight at obstacles, turning at the last moment to get around them. (This is most apparent in a grid based map.) IDA*, however, expands depth and breadth equally. The resulting path, though equal in length to A*, will change more gradually as it improves the optimality of the path with each level instead of all at once.

This difference proves substantial in dynamic environments. Consider the Bar map layout (Map 1.2): a single wall near the end of the map with several moving blocks moving parallel to it. A* will head toward the wall, thus going straight through the center of the mobile obstacles. Here, the agent is more likely to collide with obstacle than with IDA* which moves to the side of the map.

Selecting one method over the other is difficult. A* is more reliable for generating a path, but when IDA* does make a path, it will be of a better quality. Thus both algorithms will be used as seems fit for the given experiment.

As for the update times, 1000 millisecond and 5000 millisecond intervals both proved effective for IDA*. With standard A*, an update of 500 milliseconds allowed the agent to adapt properly without straining the system.

4.4 Using Time Encoded Maps

Objectives

The objective of this experiment was to determine suitable parameters for Time Encoded Maps when operating within the standard maps described above. The two primary parameters are the frequency of path updates and the tree search method (Depth First, Best First, A*, or IDA*) most suitable for the task.

After some informal experiments, the chosen sizes of the cells in the grid hierarchy are 40x40, 20x20, and 10x10.

Results

Table 2.0 summarizes the results of this experiment using a hierarchical system. Table 2.1 summarizes the results of restraining the scope to the local area.

Table 2.0 Time Encoding Using Hierarchy

Method	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
			World Units	Milliseconds	Milliseconds	
Depth	1.31	175.81	1364.67	1080.0177	15.8669	10.0%
Best	4.98	221.14	1689.89	2963.6870	18.2723	25.6%
A*	5.38	94.88	1355.99	1351.6299	22.5198	60.6%
IDA*	0.54	8.51	1496.26	523.5652	203.9671	18.9%

Table 2.1 Time Encoding Using Local Scope

Method	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
			World Units	Milliseconds	Milliseconds	
Depth	6.5	388.22	2402.1	14.1178	0.4848	59.17%
Best	8.02	351.38	2633.78	15.3579	0.2728	48.89%
A*	3.99	211.47	1756.25	45.2354	0.4969	91.67%
IDA*	0.61	20.6	1910.61	17.2152	4.4418	88.0%

Analysis

The first set of data suggests that Time Encoding is too calculation intensive; many of the trials timed out. However, upon further study, it was concluded that the cause was the hierarchical system. Three separate grids at fine detail required too much active memory during calculation. For the later experiments, the hierarchy system was replaced with the local scope mechanism while the size of the grid cells was increased to 50 x 50 units. This proved to be far more successful, as illustrated by Table 1.2.

The improvement is significant, though there are still incidents of failure; these tend to be due to the same problems experienced with the Polygonal Space Decomposition.

When compared with Polygonal Space Decomposition, paths are shorter and take less time to calculate with Time Encoding. The trade-off is a minor increase in conflicts and major increase in the amount of waste. Time Encoding using a grid which typically partitions the space into more sections. This leads to each path having significantly more nodes, thus if a path needs to be changed, more nodes will be lost. The increase in

conflicts can be attributed to the greater degree of freedom afforded the agent; it can select paths that come closer to the objects, thus putting them at greater risk of collision.

4.5 Combining Time Encoding with Polygonal Decomposition

Objectives

The objective of this experiment was to analyze how Time Encoding would function in conjunction with Polygonal Decomposition (and vice versa). The two techniques can be easily combined by using an object's potential movement field instead of its physical bounds to define the polygons in the map.

Results

Table 3.0 summarizes the results of this experiment.

Table 3.0 Combining Polygonal Decomposition and Time Encoding

Search	Method	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
				World Units	**Milliseconds**	**Milliseconds**	
Overall	Polygon	1.32	1.37	1613.17	5.1169	2.1373	63.33%
	Time Encoded	4.78	242.92	2175.69	22.9816	1.4241	71.93%
	Combined	20.85	0	1743.05	10.7371	0.8341	96.67%
A*	Polygon	1.5	16.51	2005.61	7.9945	1.1835	82.2%
	Time Encoded	3.99	211.47	1756.25	45.2354	0.4969	91.67%
	Combined	26.06	0	1847.37	14.7432	0.5944	95.0%
IDA*	Polygon	1.14	1.37	1613.17	5.1169	2.1373	63.3%
	Time Encoded	0.61	20.6	1910.61	17.2152	4.4418	88.0%
	Combined	15.63	0	1743.05	10.7371	0.8341	96.7%

Analysis

The data from this experiment suggests that combining the two techniques will result in faster and more successful path generation. Greater success was anticipated. For Time Encoding, the polygonal maps reduced the complexity of the search space; for Polygonal Decomposition, the predictive models kept the agent clear of moving obstacles. Together these factors would lead to more successful trials.

Why faster paths? In the case of polygon-based Time Encoding versus grid-base Time Encoding, the cause is quite apparent. Fewer nodes were generated. This may also be the reason for the distinction between standard polygonal division and the combined technique. The object bounds enlarged by the predictions would be more apt to border an edge, removing nodes that would otherwise have been to the side of the obstacle.

Interestingly, the combined technique produced no waste but caused more conflicts. At first this would seem at odds. If the agent is constantly colliding with obstacles, wouldn't it be losing most of the path information it generates? A plausible explanation would be that the agent would, in some cases, become trapped in “limbo.” If the movement field of an obstacle enveloped the agent, it would no longer be positioned with the free space; without any nodes to use for path finding, the agent would halt, leaving it susceptible to constant collisions.

5 Developing Midpoint Seeking

5.1 General

Objectives

Midpoint Seeking at its core is rather simple. There are many possible additions or alterations that can be made that may improve the quality of the paths generated. However, these will all come at some cost of computation, therefore it is important to study the impact of each feature to assess its effects.

Considerations

The procedure described previously is only the core of the algorithm. Having only been recently devised, there are details of Midpoint Seeking still under consideration. The following are questions considered in improving the algorithm.

How should path points be stored? Should they be kept as a list and replaced with each iteration of the algorithm? Or should a stack be used with the top being the next destination that is popped when it is reached or passed? The later would give the algorithm a persistent memory making it less likely to "flinch" but also making it more likely to fixate on an undesirable path point.

Another consideration is how the target is selected. Would it be better to ray-cast out to the nearest obstacle and use that for path selection (thus making this "Nearpoint Seeking")? This would involve tracing the path of a line, a more expensive procedure than simply checking a point. It is also uncertain as to how such an approach would affected the paths generated. Would they become too heavily influenced by local optima and nearby obstacles?

Perhaps most importantly is how to deal with adjacent obstacles. If a large series of individual obstacles all lay next to each other, it could have detrimental affects on the path finder. The influence areas would overlap. If the terminal condition is a set number of iterations, then the algorithm may constantly cut-off before finding a traversable point that lies to one side of the group of obstacles. If the terminal condition is a set distance, then the algorithm may cycle through an extensive number of recursive steps before finally returning an answer.

One possible solution for this is to link influence points. Whenever an object moves, the points are checked for potential collision with other objects. If one occurs, the influence point on the opposing side of the detected object is selected. The probing point is linked to it, its offset being increased so as to refer to the same location. (The point is subsequently reset when the new object no longer collides with the original point.) In effect, an object's influence zone expands to include those objects within its proximity.

This solution does add overhead as collision detection will need to be done for each influence point every time its parent object moves. However, this is still considerably less computation than calculating the overlapping influence between objects from scratch each time a new path is needed.

Metrics

The same metrics used analyzing existing algorithms will be used to judge the improvements made to Midpoint Seeking.

5.2 Effects of Linking and Ray-Casting

Objective

Two primary modifications to the basic Midpoint Seeking algorithm described above were developed first: Near Point Seeking and linked influence points. The four combinations of these two options were tested in the same conditions used for the previous experiments. The sole exception was that pathing intervals of only 100 milliseconds were also recorded; this was to examine how the algorithm might handle more frequent updates.

Near Point

Near Point Seeking is essentially Midpoint Seeking with one key distinction. With Near Point Seeking, a ray is cast from the starting point to the goal. The first object the ray collides with is the object used for path finding. Near Point is predicted to be more computationally intensive, as it does not simply calculate a single midpoint but must instead determine if a line intersects any of the objects in the map.

Linked Influence Points

Influence points are, by default, static. They are always at the same offset from the object's center regardless of the environment around the object. This means more iterations of Point Seeking are needed if a chosen point is within another object.

To avoid this conflict, points can be linked. Whenever an object moves, the influence points are checked for collision. If one occurs, the point is linked to a corresponding point of the object with which the initial point collided. It allows for adapting influence fields, but requires that points be updated explicitly, adding overhead to map updates that previous did not exist with Point Seeking.

Results

Table 4.0 summarizes the results of this experiment. Table 4.1 is illustrates the basic operation cost of the algorithms. Table 4.2 shows the algorithm unfavorable conditions.

Table 4.0 Comparing Linked Maps and Ray-Casting

Selection	Linking	Conflicts	Waste	Path Length World Units	Total Path Time Milliseconds	Average Path Time Milliseconds	Success Rate
Midpoint	Off	1.69	21.46	999.56	1.4641	0.1604	19.6%
	On	0.48	21.21	966.10	1.1101	0.1870	20.0%
Near Point	Off	18.27	20.03	1630.45	6.9794	0.3232	33.3%
	On	15.05	24.01	1423.14	5.7088	0.2039	39.2%

Table 4.1 Midpoint Variants on Open Map with 1000 ms Intervals

Selection	Linking	Conflicts	Waste	Path Length World Units	Total Path Time Milliseconds	Average Path Time Milliseconds
Midpoint	Off	0	9	1447.00	0.2886	0.0480
	On	0	9	1447.00	0.2651	0.0429
Near Point	Off	0	10	1445.00	0.4330	0.0589
	On	0	10	1445.00	0.2995	0.0492

Table 4.2 Midpoint Variants on Cross Map

Selection	Linking	Conflicts	Waste	Path Length World Units	Total Path Time Milliseconds	Average Path Time Milliseconds	Success Rate
Midpoint	Off	N/A	N/A	N/A	N/A	N/A	0.0%
	On	N/A	N/A	N/A	N/A	N/A	0.0%
Near Point	Off	8.47	1.33	1595.14	4.6831	0.4968	22.9%
	On	18.83	44.56	1741.38	9.6134	0.1617	16.7%

Analysis

At first, either option appears to offer only a minimal improvement. The success rate is doubled but the total path time increases by a factor of 6 and the amount of conflicts increases by a factor greater than 10.

However, the data for the base Midpoint Seeking is deceiving. The 20% of the cases that it did succeed in are all the instances in which the map had no large, permanent barriers.

The cause is quite apparent; selecting the midpoint to do path finding is rather arbitrary and ineffective. In any non-trivial case, the agent is likely to encounter a situation in which a barrier lies between the agent and the midpoints the agent is checking. The agent will completely ignore the obstacle directly in front of it. For this reason alone, Near Point is a preferable approach.

What would the trade-off of using Near Point be though? And what are the effects of linking influence points? To answer this question, a single trial case was considered: a large open map without any obstacles. This information, shown in Table 4.1, is the minimal "operating cost" of the various methods. As can be seen, there is no appreciable overhead with adding either option.

Unfortunately, even with several enhancements, Midpoint Seeking (as initially implemented) never achieved a success rate of greater than 45%. In some cases, such as the cross map (Map 1.3, Table 4.2), the improvements only produced success in no more than 23% of the trials. Clearly, these problems warrant further investigation.

5.3 Improving Near Point Seeking With Stacks

Objective

After reviewing the results of the previous experiment, it was decided that points selected should not be directly to the side of the obstacle, but rather at its near corners. This change, however, was not enough; something more than a simple adjustment was needed to make the algorithm successful. An entire map layout (the Cross Map depicted above) independent of size or path timing, was causing Near Point Seeking to fail. In attempt to resolve this problem, a stack was introduced to the path finding.

Essentially, each time a path was generated, the points were placed on a stack. If the agent reaches a particular point in the path, it'll pop off that node. If, however, it calculates a new path before reaching the next node of the previous, the new path will be pushed on top of the old one. In any case, when a path is generated, the goal is the point at the top of the stack.

The purpose of this stack is to retain information from a previously generated path. It prevents the oscillating that occurs when the agent attempts to follow conflicting paths at each interval by encouraging the agent to head along one route.

Results

Table 5.0 summarizes the results of this experiment.

Table 5.0 Effect of Using Stack on Near Point Seeking

Interval	Version	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
				World Units	Milliseconds	Milliseconds	
Overall	Standard	15.05	24.01	1423.14	5.7088	0.2039	39.2%
	Stacked	4.88	13.06	1354.4	5.9592	0.3309	50.4%
1000	Standard	4.28	2.54	1196.88	2.2285	0.2377	45.00%
	Stacked	6.06	4.24	1493.14	4.9668	0.3763	56.67%

Analysis

In general, the addition of a stack provided excellent benefits. The success rate increased by over 10%, the number of conflicts was reduced by a factor of 3, and the waste cut in half; the cost for this being a moderate increase in path time. However, compared to standard Near Point at 1000 ms intervals, the improved stack version actually produced slightly more conflicts and waste.

A success rate of 57% is decent, but given that under the right conditions, other methods have a success rate greater than 90%, this is still not acceptable. What may have been missed or introduced that is causing the agent to time out? With this experiment, the time outs happened primarily on maps where the agent started close to a wall that rested between it and the goal.

Further study revealed that the current system could continue to drive the agent into the same obstacle if the choice points were to the side. If the agent could not properly steer around the corner, it would become stuck.

To resolve this, the algorithm could select points that were on the same side as the agent but clear of the corners. (In other words, the two of the points at 45 degree angles from the obstacle's horizontal or vertical intercepts that were closest to the agent.) However, this was the technique used in Experiment 2.0 and may ultimately be responsible for its failure on the cross map. These closer points may be more readily accessible to the agent in most cases, but it does little to encourage the agent to actually go around the obstacle.

5.4 Side Point Seeking

Overview

Of the above features, using the nearest obstacle, linking influence points, and implementing a stack to remember paths were all explored as potential additions. (See Section 5 below for details). Together, they were used to create a fuller, more robust version of Midpoint Seeking known as Side Point Seeking. It is an attempt to improve the success rate of Midpoint Seeking while retaining much of its speed and flexibility.

Procedure

The following describes the basic procedure of finding a path with Side Point Seeking:

```
General Program Flow
for each object...
      create influence points
end

agent_goal <- finalGoal
push( agent_stack, finalGoal )

for each cycle...
      for each object...
            if object is path agent
                  if agent_position = agent_goal
                        pop( agent_stack )
                        agent_goal <- top of agent_stack
                  end if
                  find path( agent_goal )
            end if
            update object
            update influence points
      end
end
```

Creating Influence Points (Square)

```
xOffset <- { -1, 0, 1 }
yOffset <- { -1, 0, 1 }
padding <- min( object_width, object_height )
for each xOffset...
      for each yOffset...
            if not xOffset = 0 and yOffset = 0
                  x <- ( object_width / 2 + padding ) * xOffset
                  y <- ( object_height / 2 + padding ) * yOffset
                  add new point( x, y )
            end if
      end
end
```

Updating Linked Influence Points

```
for each point...
      if point is linked
            if point does not intersect other object's influence
                  unlink point
            end if
      else
            if point intersects other object's influence
                  link point
            end if
      end if
end
```

Find Path

```
object <- getFirstObjectBetween( start, goal )
decisionPoints <- getDecisionPoints( object_influence, agent )
bestPoint <- chooseBest( decisionPoints )
pointSet <- getPointSet( object, bestPoint )
add agent_position to pointSet
add goal to pointSet

if not past limit
      for each point in pointSet...
            find path( point, nextPoint )
      end
end if
```

Choose Best Point

```
shortestDist <- infinity
for each point in pointSet...
      distanceTo <- distance( start, point )
      distanceFrom <- distance( point, goal )
      totalDistance <- distanceTo + distanceFrom
      if totalDistance < shortest
            short <- totalDistance
            bestPoint <- point
      end if
end
```

In addition to the map initialization present in the previous algorithms, there is also a brief routine that sets the goal of the path agent. Combined, these should be no more costly than the initialization phase of Polygonal Space Decomposition or Time

Encoding.

Note that in each cycle, influence points are updated with each object update. This adds overhead to each object. However, the process is largely a series of different checks; little to no calculation is required.

The influence points can be defined in a number of ways. The method described above creates an eight-point square, where the distance of the points from the center of the object is equal to the width and/or height of the object. Though suitable for square objects, more complex objects will require a more complex definition of points.

"Past limit" is a vague statement. What determines the limit will be a matter of implementation. The limit used for purposes of experimentation was a limit on recursive calls.

When choosing the best point, distance to and from the point are the only measurement used. The check for the minimal angle is done implicitly. The farther away the point is, the greater the angle change needed to steer to it. This is assuming that the points lie perpendicular to the casted ray. If this is not the case, a different means of evaluation will be necessary, one that explicitly judges the change in angle.

Example

The following diagram illustrates Side Point Seeking in operation. The narrow triangle is the agent and the small square is the end goal. The selected path is marked with bold lines. The light dots are the nodes selected. The dark dots were considered nodes while the gray are nodes that were never involved in the search. The broad triangles point to the pivotal nodes used in deciding the path. The thin lines represent the rays cast to find obstacles.

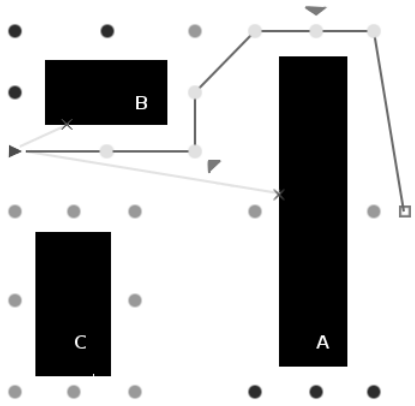


Figure 7.0 – First Step of Polygonal Decomposition

The path finding begins by ray casting towards the goal. Object A is found to be in the way. The nodes directly above and below Object A are used as decision points. The upper node is clearly the closer one (and the one that would take the least change in angle to reach) and thus is selected. It and the corner nodes next to it are used for the next iteration.

Three of the four sub-paths are clear. However, the sub-path from the start to the near corner is not. The ray strikes Object B. Because of the angle at which the object was struck, the lower right and upper left nodes are used for path selection. The upper left is closer, but the overall distance if it is selected is slightly greater. For this reason, the lower right is chosen.

Further iterations are all trivial as there are no more obstacles in the way. All of the given points are pushed onto the stack. Future paths will use the process, with the exception that the goal will be the current node on the top of the stack.

Results

Table 6.0 summarizes the results of this experiment. Table 6.1 serves as a reference so as to compare Side Point Seeking against other studied algorithms.

Table 6.0 Side Point Seeking

Interval	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
			World Units	Milliseconds	Milliseconds	
Overall	28.75	0	1556.98	5.9705	0.1542	94.6%
100	75.8	0	1613.63	12.9610	0.1194	95.0%
500	21.67	0	1623.26	5.4129	0.1455	98.33%
1000	11.04	0	1507.58	3.4747	0.1736	98.33%
5000	6.52	0	1483.45	2.0333	0.1785	86.67%

Table 6.1 Performance of Select Techniques

Method	**Conflicts**	**Waste**	**Path Length**	**Total Path Time**	**Average Path Time**	**Success Rate**
			World Units	**Milliseconds**	**Milliseconds**	
Side Point Seeking	11.04	0	1507.58	3.4747	0.1736	98.33%
Stacked Near-Point Seeking	6.06	4.24	1493.14	4.9668	0.3763	56.67%
Polygonal Decomposition	1.87	19.62	1660.14	14.2269	1.6966	90.0%
Time Encoding	3.99	211.47	1756.25	45.2354	0.4969	91.7%

Analysis

The initial studies of Side Point Seeking appear quite favorable. Most notable is the significant increase in the success rate. Side Point does a comprehensive search along the approach, evasion, and departure vectors for each obstacle, whereas previous versions did only a minimal analysis of the approach and evasion. Thus, a better success rate was expected; the decrease in path time was not.

The probable cause of this decrease is the refinement of the path. With each algorithm, the nearest portion of the path is continuously updated. Near-Point splits the path in half whereas Full Point splits it into quarters. Therefore, even if the initial path is more expensive, the subsequent paths can be twice as cheap.

Side Point Seeking also lacks waste, but this is a deceptive figure. It was achieved by retaining most of the initial path. It will not be discarded with future calculations. The trade-off with this steady path is, naturally, adaptability. Changes the occur at greater distances will not affect the path finding because the later portion of the path remains unyielding. The consequence is that if a persistent change occurs remedial action cannot be made until the agent actually approaches the obstacle and becomes aware of its presence. As for a favorable condition, such as a new path opening up, a full point seeker

may never take advantage of it. Unfortunately, this was the type of behavior Midpoint Seeking was aimed at reducing.

Also note the increase in conflicts. In particular, how the conflicts are inversely proportional to the path interval. More frequent path calculations actually led to more conflicts. Why would this be? Point Seeking was designed to allow more frequent updates so as to reduce conflicts. A likely cause is the lack of prediction. With the moving obstacles, the agent may be selecting the points in front of the object, thus steering into the obstacle's path rather than steering around it or behind it. Such an error is more likely to happen the more likely the agent is to detect an object in front of it.

Side Point Seeking is still successful, despite these flaws. When compared with Polygonal Partitioning with A*, it had a greater success rate with significantly less time spent on generating paths. Nonetheless, a few final adjustments were considered to overcome Side Point Seeking's "stubbornness."

5.5 Filtered Side Point and Biased Side Point

Two potential solutions were considered, implemented, and studied. The first of these is a filtering process. Rather than simply seeking the first point on the stack, points are analyzed for their utility. The first point is popped off the stack. The following point is ray-cast to. If this second point can be reached directly by the agent, the first is discard. This process is repeated until a collision is found, at which point, the select point is placed back onto the stack and the path finding proceeds as before.

This solution eliminates the doubling-back that standard Side Point Seeking occasionally produces. The cost is additional overhead, especially if the ray-casting employed is not efficient. The potential to analyze the entire stack also exists, however, this is minimal as the influence points should be positioned such that only a few can be "seen" by the agent from any one position. The major drawback of this solution is that it is still subject to the original path generated.

The other solution considered was removal of the stack. In its place a simple bias point is stored. This point is the first node of the previous path generated. For the initial selection of the current path, the distance between a potential node and the bias point is calculated and added to that node's cost. This causes the Point Seeking to prefer a path that starts in a similar direction as the previous, as more deviate paths will often have a greater cost. The remainder of the search is conducted as before.

Biasing Side Point Seeking allows the path to be more flexible. It also removes the memory requirements of maintain the path in a stack. Unfortunately, the bias is not as solid a means of path persistence, meaning some "indecision" can occur in which the agent wavers between potential paths.

5.6 Defining Influence Points

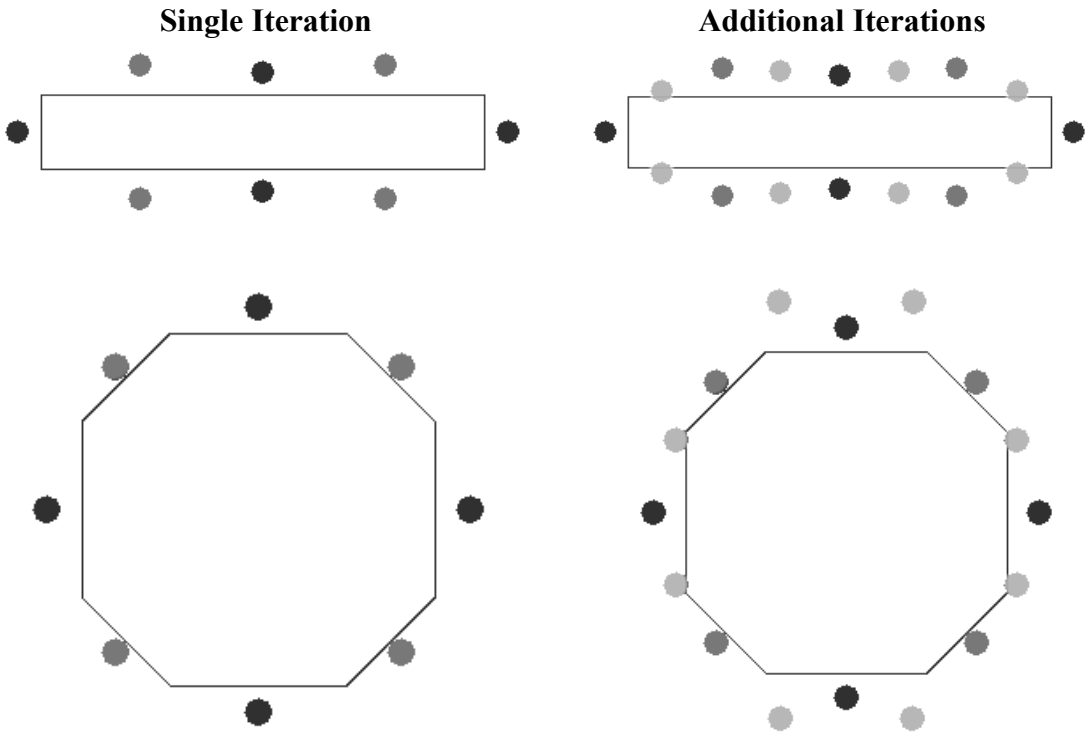
For the basic geometry present in the test maps (as illustrated in Appendix C), the influence points around each object can be defined in a simple fashion. The most basic is to place eight points along the perimeter of the object's extended bounding box to form a square or circle. These are sufficient in guiding the agent around the obstacle.

A slightly more representative option would be to defined influence points based on the vertecies of the object. A point would be placed a short distance outside the shape along the rays from the center point to each vertex. This approach is good for an convex polygon, but becomes problematic when dealing with concave shapes.

The algorithm developed by Rob Teather[24] can be adapted for this situation. His algorithm starts with the midpoint between the start and goal. Two nodes along the perpendicular to the start-goal axis are placed just outside of any immediate obstructions. This process is repeated along the lines from the first and second points. After this iteration, a node is found between each possible pair of nodes. The result is a mesh of nodes around any of the masses in the map.

Several alterations were made to apply this technique to individual obstacles in the map. The first was the axis. Instead of being a line between the start and goal, the vertical or horizontal axis of the extended bounding box was selected depending on which was longer. If the midpoint lay within the shape of the object in question, the original algorithm was followed for a limited number of iterations, without the final connection between all points. If no obstruction existed at the midpoint, the original axis was split in two and these were used instead of the original axis. Another alteration was that nodes would only be added if they extended away from the center. In addition, nodes were clamped to remain within a initial bounds.

In general, this process produces a suitable set of influence points around the object. Below are a couple of examples:



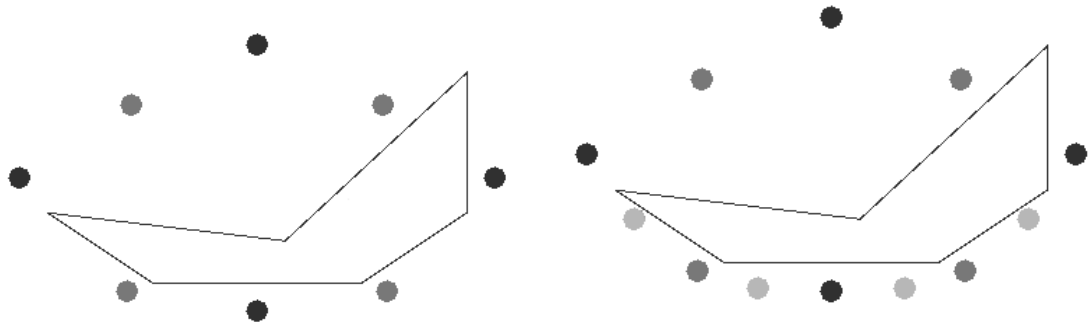


Figure 8.0 – Points generated by the modified Teather algorithm

As demonstrated by the later two diagrams, the modified Teather algorithm will produce an efficient set of points evenly distributed around any moderately complex polygon. The distribution is beneficial to Side Point Seeking because it prevents the approach of the agent from influencing the accuracy of the route generated. The size of the set is useful as it keeps the number of "nodes" to a minimum.

The maps used for experimentation in this paper utilize only simple rectangles. For this reason, the Teather point definition was deemed unnecessary. The results from a single iteration are similar to that of using simple offsets, but require slightly more computation. Additional iterations only serve to add extraneous nodes; it is unlikely that these points would lend additional accuracy but it is certain to add to the overall computation of Point Seeking. However, this assertion remains unproven.

6 Comparing Algorithms

6.1 General

Objectives

Having refined Point Seeking to a satisfactory degree as well as establishing the desirable parameters for existing algorithms, it is time to compare the techniques against each other. Data from a variety of scenarios will be collected.

The exact algorithms under consideration are: Line Splicing, Time Encoding, Polygonal Decomposition, Filtered Side Point Seeking, and Biased Side Point Seeking. A basic static search was employed for comparison.

Each of these will be recalculated at 500 millisecond intervals and, where applicable, utilize A*. The sole exception is Polygonal Decomposition. This is based on the initial experiments which suggest that IDA* will reduce conflicts while intervals of 1000 milliseconds will reduce indecision without compromising responsiveness.

6.2 Additional Object Types

Overview

The first scenario used in prior experiments was mobile objects that moved back and forth on a set path. This path typically ran parallel to the optimal path for the agent. This provided sufficient opportunity for conflicts, but it also meant that clearing the obstacles was largely a matter of timing.

However, not all obstacles in the environment will exhibit this same behavior. For this reason, additional object types were created. The initial maps were populated with these new objects as part of the second stage of experiments.

Description

The following summarizes the type of objects added.

Fading Panel	These panels can appear and disappear. Placed in strategic areas, they serve as doors. They are intended to test how responsive algorithms are to sudden obstructions (or removal thereof).
Wanderer	These objects float about the map in a mostly random fashion. (It is weighed to prefer straight lines, however, to avoid constant changes.) Used in many map designs and intermediate studies, none were incorporated into the final set of maps.
Adversarial Objects	These objects aggressively seek out the agent, heading in a straight line towards it. They serve to test the agents awareness of on-coming obstacles as well as to increase the difficulty of finding a path.

6.3 Experiment Summary

Tables 7.0, 7.1, and 7.2 show the performance of various algorithms in environments where the standard scripted objects were replaced with one of the object types listed above.

Table 7.0 Performance with Wandering Objects

Method	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
Polygon, A* @ 500 ms	1.3	52.06	1694.99	14.4829	0.8298	82.22%
Polygon, IDA* @ 3000 ms	1.36	3.85	1769.04	5.3539	1.6378	88.89%
Prediction, IDA* @ 1000 ms	1.19	30.16	1772.27	8.6679	1.1139	88.89%
Near-Point @ 1000 ms	8.45	4.01	1338.69	6.4741	0.4589	37.78%
Side Point @ 1000 ms	0.35	9.57	1804.68	3.8806	0.1667	88.89%

Table 7.1 Performance with Adversarial Objects

Method	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
Polygon, A* @ 500 ms	1.96	70.86	1938.74	9.4657	0.6421	35.56%
Polygon, IDA* @ 3000 ms	1.11	4.29	1715.87	4.0734	1.4818	60.00%
Prediction, IDA* @ 1000 ms	1.7	23.1	1976.88	7.9154	0.9739	42.22%
Near-Point @ 1000 ms	6.44	3.67	1236.21	6.1594	0.3592	37.78%
Side Point @ 1000 ms	1.5	8.74	1988.41	3.7908	0.1803	53.33%

Table 7.2 Performance with Fading Panels

Method	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
Polygon, A* @ 500 ms	1.71	98.56	2032.54	20.9413	0.7704	63.33%
Polygon, IDA* @ 3000 ms	1.12	3.57	1557.76	6.4246	1.5147	71.67%
Prediction, IDA* @ 1000 ms	1.44	27.16	1860.24	11.4522	1.0667	71.67%
Near-Point @ 1000 ms	5.03	3.17	1112.56	4.8436	0.6284	28.33%
Side Point @ 1000 ms	0.95	11.38	1990.62	10.7559	0.2434	63.33%

Table 8.0 shows the overall performance of the selected algorithms when run under the second set of testing maps (which contain a mixture of the object types). Table 8.1 summarizes the results when using the geometrically simple set of maps while Table 8.2 summarizes the results of the more complex "Prison" map.

Note that "Basic" refers to the results of simply recalculating a grid-based map and performing the core tree search (i.e. A*) on the new map. A local scope was applied to limit the amount of recalculation.

Table 8.0 *Overall Performance*

Method	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
			World Units	**Milliseconds**	**Milliseconds**	
Basic	11.12	1466.82	2086.49	234.3890	0.0607	91.67%
Line Splice	14.5	1854.67	2491.42	8.0507	0.0724	63.89%
Time Encoded	7.88	1100.01	1989.17	217.1759	0.2818	94.44%
Polygon	3.54	1184.06	3100.21	18.4710	0.5713	70.83%
Side Point Bias	14.16	740.33	1636.91	278.1068	0.1101	66.67%
Side Point Filtered	7.52	662.52	1713.85	27.4875	0.1932	73.61%

Table 8.1 *Basic Maps*

Method	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
			World Units	**Milliseconds**	**Milliseconds**	
Basic	8.76	1073.13	1783.06	190.1813	0.0546	90.00%
Line Splice	13.57	1610.75	2349.16	7.5019	0.0756	58.33%
Time Encoded	6.53	1054.59	1989.17	217.1759	0.2818	93.33%
Polygon	3.54	113.68	3100.21	18.4710	0.5713	85.00%
Side Point Bias	14.16	172.08	1636.91	278.1068	0.1101	80.00%
Side Point Filtered	6.93	82.29	2301.14	41.1753	0.2002	86.67%

Table 8.2 *Prison Map*

Method	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
			World Units	**Milliseconds**	**Milliseconds**	
Basic	22.92	3435.25	3603.64	455.4274	0.0907	100.00%
Line Splice	19.18	3074.27	3202.73	10.7945	0.0565	91.67%
Time Encoded	14.67	3235.17	3200.93	486.8419	0.3907	100.00%
Polygon	N/A	N/A	N/A	N/A	N/A	0.00%
Side Point Bias	N/A	N/A	N/A	N/A	N/A	0.00%
Side Point Filtered	4	206	5237.61	109.6145	0.2351	8.33%

7 Conclusions

7.1 General Remarks

A few general observations regarding the problem of dynamic path finding can be made from the experimental scenarios described above.

The nature of obstacle movement can alter the performance of the algorithm in use. Some algorithms are suited for gradual changes while others can do better with sudden alterations. Some deal better with patterns and others with arbitrary movement.

In general, it is best to maintain as much distance as possible from all obstacles. Being to hesitant to approach an object, however, can discourage the use of viable avenues.

Some situations are difficult to resolve with any algorithm. An example of this is a passage that is being repeatedly opened and blocked. Being able to get a path at one moment then not at the next will stall out the agent.

Having an effective means of avoiding or reacting to collision is crucial. Though obstacle avoidance was used for the later experiments,it was not sufficient to guarantee suitable adjustments in all situations. There were still cases in which a poor path lead to a collision which the obstacle avoidance could not resolve.

7.2 Analysis of Algorithms

Line Splicing

Compared to the standard A*, Line Splicing with A* performed slightly worse. This is fairly logical as Line Splicing adds overhead that does not directly aid in making the path finding more efficient; it is simply designed to make the agent more responsive.

The only thing of note about Line Splicing is its low success rate. This can be accounted for however. The splice to the selected path node does not take into consideration the particular angle of approach nor the velocity. Under the right circumstances, the agent may not turn onto the path properly. Instead, it will collide with a nearby obstacle. The obstacle avoidance may not be able to help it recover in this close proximity, thus the agent will become impeded indefinitely.

Judging from these experiments, Line Splicing does not provide any significant benefits beyond its initial responsiveness. It is best suited for relatively open environments (due to the need to splice to a path node) in a system where sudden

overloads may occasionally impeded reaction times. It is not well suited for a system where high loads are consistent or where overall computational time is more important than responsiveness.

Time Encoded Maps

Of the algorithms examined in detail, it is clearly the most successful. In fact, in the experiments, it was the only one to have a greater success rate than the standard A*. Time Encoding used the same grid map structure as the basic path finder meaning it had the same level of detail in the paths it generated. The addition of predictive capabilities allowed the path finder to better evade potential blocks.

Due to its fewer conflicts and failures, the performance of Time Encoding is slightly greater than the basic path finder in most respects. The key exception is the time per path, which is over 4 times greater. Given that the improvements are all marginal, this is a large price to pay when time is at a premium.

Polygonal Space Decomposition

Polygonal Space Decomposition produced mixed results with the environments used in the experiments. Paths were longer both in distance and in time. The average path time was greater than the standard path finder by almost a factor of 10. The success rate was lower by 20%. However, the number of conflicts was fewer and accordingly the amount of waste was significantly less. This means, though slower, it was also more efficient.

One cause of failure with Polygonal Space Decomposition can be attributed to the algorithm “twitching.” Since nodes are placed relative to objects, the agent's future paths can be altered without having an object directly obscuring it. For example, if the agent is moving parallel with an object, the nodes in front of the agent will gradually move ahead of it. If the agent does not travel faster than object, it may begin chasing the node, only reaching it after the object changes velocity. This produces a distinct “twitching” which is enough to impede agent so it cannot travel its path in a timely fashion. The path interval and tree search were changed to help alleviate this particular problem, but it was not enough to keep the algorithm on par with the others.

A related issue is the angles that can be produced between nodes. If a wide area is partitioned near the left or right edge, it will create a node that lies at an extremely shallow angle to the center of the partition. At such an angle, the obstacle avoidance may not detect the nearby wall.

It is important to consider the difference between the test environment for these experiments and the original use of Polygonal Space Decomposition. Polygon Decomposition was proposed as a means to handle changes in a Real Time Strategy game. The maps in these games typically have static geometry and are significantly larger in proportion to any potential obstructions (i.e. the units and buildings added to the map). This means the agent is likely to have more room to maneuver. These obstacles often move slower than the objects used in the experiments. Also, due to the organic nature of the maps, units are not likely to be moving in precisely parallel or perpendicular

directions as they transition between points. This means the “twitching” is less likely to occur.

In short, Polygonal Space Decomposition performed poorly in these experiments, but most likely due to it being applied to an inappropriate situation.

Side Point Seeking

When compared to the standard static tree search, Side Point Seeking doesn't appear to offer much. The average path speed is twice as long and the success rate is lower by as much as 25%. Paths are slightly shorter, but the only key improvement is the waste which is vastly less on the basic maps. Based on the data from the second set of experiments, there is little reason to use Side Point Seeking.

This is in sharp contrast to the initial data which suggests that Side Point Seeking was comparable to other techniques and indeed preferable in many cases. It had the highest success rate with the lowest path times.

What is the major factor is such disparate results? The initial experiments used only scripted, predictable objects. Subsequent experiments did test the algorithms against objects with varying types of behaviors. The results for Side Point Seeking were as follows:

Table 8.0 Performance of Side Point Seeking in Various Environments

Obstacles	Conflicts	Waste	Path Length	Total Path Time	Average Path Time	Success Rate
			World Units	**Milliseconds**	**Milliseconds**	
Scripted	11.04	0	1507.58	3.4747	0.1736	98.33%
Wanderers	0.35	9.57	1804.68	3.8806	0.1667	88.89%
Adversarial	1.5	8.74	1988.41	3.7908	0.1803	53.33%
Fading Panels	0.95	11.38	1990.62	10.7559	0.2434	63.33%

As can be seen, Side Point Seeking degraded severely when encountering fading panels and adversarial objects, both of which were integrated into the newer experiments. This exposes a major weakness of the Point Seeking technique: the sparse nodes tend to lie within proximity of objects. If that object is actively attempting to block the agent, the agent is apt to not steer away well enough to avoid conflict. Also, any sudden interruptions between nodes will invalidate a major section of the agent's path and force a costly adjustment. Essentially, Point Seeking can work well with objects of most sizes and shapes, but not of certain behaviors.

There are many factors not directly accounted for by this experimental data, however. First is the use of memory. The focus of these studies was processing time. As a result, the memory consumption of each algorithm was not explored. Presumably, Point Seeking uses less. The standard grid based map (if kept with the same level of detail) will require more memory as additional space is added; the memory needed only increases with Point Seeking when a new object is added. Many common maps have large amounts of space compared to the number of dynamic, non-trivial obstacles. The grid has a significant potential to grow in memory requirements while Point Seeking is kept to a minimum.

Another consideration is the complexity of the given algorithms. Point Seeking has a linear growth that is influenced solely upon the number of objects. With other techniques, the complexity is based in part upon the underlying tree search employed. Typically, this is greater growth than O(n). Even if the growth were linear, the additional processing of the map is more expensive. Thus, with large enough environments, Point Seeking, should, in theory, exhibit far better performance.

Biased versus Filtered Side Point Seeking

Using a bias versus a filtered stack has its trade-offs. The filtered stack version had longer path times (and greater memory requirements) but fewer conflicts and failures. The bias did mitigate indecision, but the stack still proved necessary to minimize collisions and non-optimal paths. Neither proved more advantageous than the other, however.

7.3 Final Comments

The objective of this work was to create a computationally simple algorithm that could be used to create paths frequently without producing a burden on the system. It was hoped that this would permit an agent to be more adaptable to change.

The original Midpoint Seeking achieved this, but the accuracy and reliability of these paths were exceptionally poor. In an attempt to overcome this, many features were adopted: an object oriented approach to point selection, ray-casting to the nearest obstacle, linking the influence areas of obstacles, ray-casting around an object, and keeping some form of memory of previous paths. The earlier features added were simple and made vast improvements; later ones increased the success rate of the algorithm but noticeably reduced its speed and flexibility.

Currently, Side Point Seeking is inadequate as a general purpose, stand alone algorithm despite the efforts to make it so. However, this research suggests that it does have specific application domains in which it would prove advantageous. Tight quarters and/or winding corridors overwhelm the path finder, but it works effectively in more open areas. Suddenly appearing and disappearing objects as well as aggressive obstacles can greatly compromise the agent's progress, but other types of objects can be handled effectively. Thus, Side Point Seeking – or another variant of the Point Seeking method - can potentially serve as a special purpose path finding algorithm that handles large, open areas with many free moving bodies.

There is also the potential of applying Point Seeking as an intermediary. One option is to use it as a quick path generator. The work with Line Splicing does suggest that something other than a straight line path would help select a reasonable path to start along. The preliminary versions of Point Seeking required minimal calculations, making them viable candidates for this role.

In summary, Midpoint Seeking and the current algorithms built upon it are limited. The general concept of Point Seeking still shows promise and is worth further investigation.

8 Future Work

The experiments described in this paper was only the initial phase of study. Much work is needed to improve Side Point Seeking and to verify its effectiveness. The following are potential next steps:

Devise Alternate Path Persistence

The current stack-based memory is perhaps the greatest flaw of Side Point Seeking; overcoming it will allow the algorithm to enjoy the adaptability it was intended to have. Some means of filtering or replacing outdated nodes is needed. Whatever solution is devised, it needs to prevent the agent from constantly selecting radically different paths without forcing it to remain along a particular path.

Test with a More Complex Environment

The maps used were relatively simply; objects and obstacles were all roughly of the same size and were all rectangular. This was to allow for a fair proof-of-concept and easier analysis. It did not, however, stretch Side Point Seeking to its limits. Having objects are many varying sizes and shapes is the means of properly determining if Side Point Seeking can represent all objects equally. (This would be a good situation to test the value of the Teather algorithm mentioned previously.) It would also allow for a more rigorous comparison as many other algorithms are known to be limited in their ability to handle irregular shapes.

Also, the size and layout of the maps as a whole were fairly simplistic. The "Prison" map illustrated clearly that Side Point Seeking has severe limitations. If any derivation of Point Seeking is to made into a viable stand-alone algorithm, it will need to pass this test as well as other more complex maps.

In a related issue, the game engine and its mechanics were fairly simple. Placing the agent in a resource intensive system can give a better idea of just how much Point Seeking can save in terms of calculation. Using a more robust version of collision detection (as opposed to the basic bounding boxes used for these experiments) will facilitate the development of maps with more elaborate shapes.

Expand to Three Dimensions

Translating a path finding algorithm into three dimensions is often more difficult than it initially seems. A major obstacle is how to deal with the vertical component when the agent is not capable of ascending or descending freely. One suggestion would be to simply ignore any influence point that exceeds the height which the agent can reach.

Another obstacle is the increased computation due to the third dimension. This is an opportunity for Side Point Seeking to shine; finding and selecting a suitable node simply means adding a third coordinate into the calculation. The number of additional nodes needed should be minimal. Other algorithms do not have this advantage. The third

dimension tends to increase the search space dramatically. For example, a simple 10 x 10 boolean grid has 100 path nodes. Extending this area equally into the third dimension increases the number of nodes to 1000.

Use in Conjunction with Other Algorithms

As stated before, the scope of this paper was to develop Midpoint Seeking into a viable stand-alone algorithm. However, there is no reason why it cannot be explored as a potential supplement. Point Seeking could be used to generate a tentative path, either to allow path calculation in the background or to provide a response when the system is burdened.

Point Seeking is far from being perfected, but it shows potential. It may indeed become a viable alternative or supplement to the current solutions used to handle dynamic path finding.

9 References

9.1 Works Cited

[1] Mark Brokington. Reimplementing A*. *Game Developer's Conference Proceedings,* 2000.
[2] Timothy Cain. Practical Optimizations for A* Path Generation. *AI Game Programming Wisdom*, 146-152, 2002.
[3] Tim Emonds, Antony Rowstron, Andy Hopper. Using Time Encoded Terrain Maps for Cooperation Planning. *Advanced Robotics*, vol. 13, 779-792, 2000.
[4] T Frohlich, D Kullmann. Automonous and Robust Navigation for Simulated Humanoid Characters in Virtual Environments. *Cyber Worlds*, 2002.
[5] V. Scott Gordon, Zach Matley. Evolving Sparse Direction Maps for Maze Pathfinding. *IEEE Congress on Evolutionary Computation*, 2004.
[6] Ross Graham, Hugh McCabe, and Stephen Sheridan. Pathfinding in Computer Games. *Institute of Technology Blanchardstown Journal*, 2003.
[7] Green, D.N., Sasiadek, J.Z., Vukovich, G.S. Path tracking, obstacle avoidance and position estimation by an autonomous, wheeled planetary rover. *Robots and Automation*, 1994.
[8] Dan Higgins. Pathfinding Design Architecture. *AI Game Programming Wisdom*, 123-132, 2002.
[9] Huibo Li, Wen Tang, Simpson, D. Behaviour Based Motion Simulation for Fire Evacuation Procedures. *Theory and Practice of Computer Graphics*, 2004.
[10] Yong K. Hwang, Narendra Ahuja, Gross Motion Planning – A Survey. *ACM Computing Surveys (CSUR)*, vol 24, issue 3, 1992.
[11] Sven Koenig. A Comparison of Fast Search Methods for Real-Time Situated Agents. *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, vol 2, 2004.
[12] Patrick Lester. A* Pathfinding for Beginners.

http://www.policyalmanac.org/games/aStarTutorial.htm, 2003.
[13] Kelly Manley. Pathfinding: From A* to LPA. *University of Minnesota*, 2003.
[14] James Matthews. Basic A* Pathfinding Made Simple. *AI Game Programming Wisdom*, 105-113, 2002.
[15] Sven Koenig, Maxim Likhachev. Adaptive A*. *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, 2005.
[16] Richard E. Korf, Micheal Reid, Stefan Edelkamp. Time Complexity of Iterative-Deepening-A*. *Artificial Intelligence,* vol. 129, p. 199-218, 2001.
[17] Mike Mika, Chris Charla. Simple, Cheap Pathfinding. *AI Game Programming Wisdom*, 155-160, 2002.
[18] Bernard Moulin, Driss Kettani. Route generation and description using the notions of object's influence area and spatial conceptual map. *Spatial Cognition and Computation*, vol. 1, issue 3, 227-259, 1999.
[19] Jared D. Mowery. A Genetic Algorithm Using Changing Environments for Pathfinding in Continous Domains. *Standford*, 2002.
[20] Amit J. Patel, Dealing with moving obstacles. *Amit's Game Programming Site,* Feb, 2006.
[21] Samuel Ranta-Eskola. BSP Trees: Theory and Implementation *DevMaster.net*, October 9, 2003.
[22] Craig Reynolds. Steering Behavior for Autonomous Characters. *Game Developers Conference*, 1999.
[23] Nathan Sturtevant, Michael Buro. Partial Pathfinding Using Map Abstraction and Refinement. *University of Alberta*, 2004.
[24] Rob Teather. Path Finding. *York University*, 2003.
[25] Guowei Wu. Polygonal Pathfinding in ORTS game. *University of Alberta*, 2000.
[26] J.M.P. van Waveren, L.J.M. Rothkrantz. Automated Path and Route Finding through Arbitrary Complex 3D Polygonal Worlds. *Robotics and Autonomous Systems, Volume 54, Issue 6, 30 June 2006.*
[27] Bjornsson Yngvi, Markus Enzenberger, Robert Holte, Jonathan Schaeffer and Peter Yap. Comparison of Different Grid Abstractions for Pathfinding on Maps. *University of Alberta*, 2003.

9.2 Additional Resources

[28] Lieutenant Colonel Rene G. Burgess, Christian J. Darken. Realistic Human Path Planning using Fluid Simulation. *Naval Postgraduate School*, 2003.
[29] Robert Niewiadomski, Jose Nelson Amaral, Robert C. Holte. A Performance Study of Data Layout Techniques for Improving Data Locality in Refinement-Based Pathfinding. *Journal of Experimental Algorithmics*, vol 9, 2005.