THE APPLICATION OF ARTIFICIAL INTELLIGENCE TO SOLVE A PHYSICAL PUZZLE

Dana Cremer

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Master of Sciences
in the Department of Computer and Information Sciences
and the Department of Mathematical Sciences,
Indiana University South Bend
June 2007

Accepted by the Graduate Faculty, Indiana University South Bend, in partial fulfillment of the requirements for the degree of Master of Science.

Master's Thesis Committee

_____

Chairperson, Dana Vrajitoru, Ph.D.

_____

Morteza Shafii-Mousavi, Ph.D.

_____

James Wolfer, Ph.D.

Date of Oral Defense:

April 27, 2007

Dedication

This thesis is dedicated to my wonderful wife Cathy and son Jordan.

They are truly a blessing.

Acknowledgements

I would like to express my sincere appreciation for all the help I received from my advisor, Dr. Dana Vrajitoru. Her instruction and guidance made it possible to transform this project from a sketchy concept into a presentable thesis.

I would also like to thank Dr. Morteza Shafii-Mousavi and Dr. James Wolfer for all their help in reviewing this thesis and helping to polish it up.

Finally, I would like to thank my wife Cathy and son Jordan for all their support and encouragement. I am very fortunate to have a family with the understanding and willingness to take care of almost everything so I could spend all my time playing with my marbles.

Abstract

This thesis presents the design, development, and implementation of an intelligent agent capable of solving a physical puzzle. The puzzle is a three dimensional maze in which a marble must be moved from its starting point to a target cell in the opposite corner. The movement of the marble is strictly the result of movement of the maze itself, the marble's response to gravity, and collisions with the walls of the maze. The physical nature of the puzzle provides an interesting challenge for the intelligent agent attempting to solve it, since it does not have complete control over the effects of its actions, and is not able to predict with certainty what those effects will be.

A software framework is developed to integrate the artificial intelligence, physics simulation, and computer graphics required to solve the puzzle. A control scheme is designed to enable the agent to perform the physical moves to be simulated. Several solution algorithms are developed and implemented, incorporating varying levels of knowledge of the maze's geometry and the physics involved. In general, it is shown that by increasing the 'intelligence' of the agent, the performance was significantly improved.

This thesis is a unique integration of artificial intelligence, physics simulation, and computer graphics. The result is the graphical animation of the solution to a physical puzzle that could not be solved without each of the three technologies.

Table of Contents

1. Introduction

The system developed for this Master's thesis is an intelligent agent that uses elements of artificial intelligence to determine and execute the necessary moves required to solve a physical puzzle.

The specific puzzle is a random 2 or 3-dimensional maze, similar to those shown in Figures 1 and 2. The goal of the puzzle is to roll a marble from its starting location in one corner of the cube to a target cell at the opposite corner. The agent solves the puzzle by rotating the maze to move the ball, which then reacts in response to gravity.

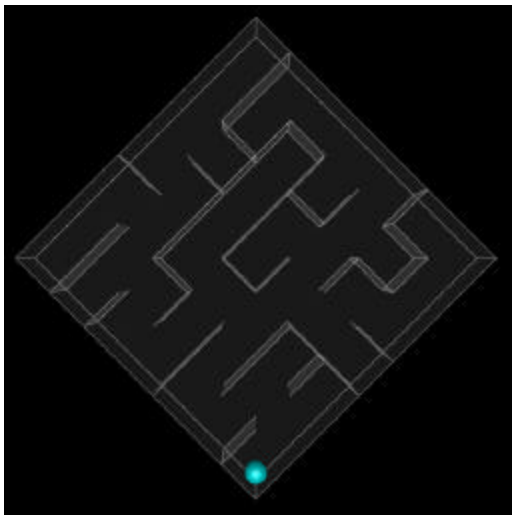

Figure 1: 2-D Maze

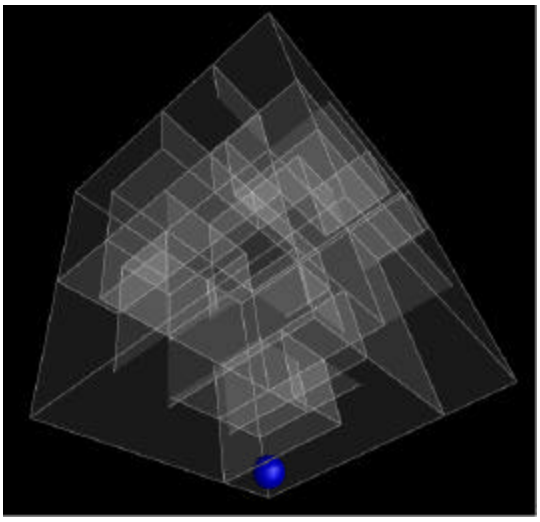

Figure 2: 3-D Maze

This thesis differs considerably from other related path-finding algorithms present in the literature. In most cases, it is assumed that the intelligent agent searching for a path to a desired location or state has complete control over the location to which the object is moved. *The Role of Simulation in Developing Game Playing Strategies* discusses two typical examples [Jones and Thuente, 1990]. In this paper, strategies are developed for playing the games Tic-tac-toe and ConnectThree. In both of these games, the state of the game immediately after a move is known with certainty before the move is made.

The only exception found in the literature is for blind searches, where there is the possibility of running into an obstruction that prevents the move, keeping the object in its current state. This situation is covered in *A Comparison of Fast Search Methods for Real-Time Situated Agents* [Sven Koenig 2004]. Unlike this thesis, the agents studied in that paper also acted on a graph that was undirected, and backtracking therefore was always possible.

One of the key elements of this thesis however, is the creation of an intelligent agent that can deal with events that are not directly initiated by the agent. These events, which are an unpredictable consequence of the agent's actions, can place the object in an unpredictable, potentially unrecoverable state. This added dependency on the physics simulation requires the agent to be able to learn the effects of other forces acting upon the object and be able to find a solution without complete control.

The graphical output is also a unique and critical component to the system, providing the real-time 3 dimensional animation of the problem solution that would be very difficult to visualize without it.

2. Literature Review

Modern advances in the processing power of personal computers have made the real-time accurate simulation of physics possible for an affordable price. Advances in graphics co-processors have made it possible to render the three dimensional results of that simulation in real time. Much of this advancement in technology has been driven by the mass market for computer games, many of which require sophisticated software or 'game engines' to handle the physics, artificial intelligence, and graphical rendering needed by the game.

Michael Lewis and Jeffrey Jacobson in the January 2002 Communications of the ACM noted that 'The most sophisticated, responsive interactive simulations are now found in the engines built to power games" [Lewis and Jacobson, 2002]. Lewis and Jacobson also observed that the modularity of game engines (of which the 'physics engine' is a component) allows the game code to be used for other scientific research. They give an overview of several research projects in the areas of robotics and artificial intelligence that are utilizing game engines originally developed for the games "Quake" and "Unreal Tournament".

Integration and Evaluation of Exploration-Based Learning in Games [Karpov et. al. 2006] describes how a system called the Test bed for Integrating and Evaluating Learning Techniques (TIELT) enables researchers to take advantage of video game environments to test new learning algorithms. They used (TIELT) to test the NeuroEvolution of Augmenting Topologies (NEAT), an artificial intelligence algorithm in the Unreal Tournament video game environment. The NEAT algorithm is used for path-finding in a game map. They also describe the high level architecture of their test environment, which much like this thesis, separates the 'Game Model' (simulation) from the 'Decision System' (knowledge engine) and interfaces the two.

A major component of many video game environments that is required by this thesis is the physics engine, which is responsible for performing the physical simulation. The wikipedia site: http://en.wikipedia.org/wiki/Physics_engine lists several open-source and

commercial physics engines that can be incorporated in computer science projects requiring real-time or even high precision physics simulation. One of the physics engines listed at this site, 'Newton Game Dynamics', is the engine used for this project [http://www.physicsengine.com].

In addition to the physical simulation, this thesis requires algorithms to determine the solution of the maze. This aspect of the project is very much related to path finding, or traversal of graphs. Much research has been done in the area of graph theory to develop algorithms for determining optimal paths to follow on a graph or game tree to lead to a desired state. For example, *Artificial Intelligence: A Modern Approach* [Russell and Norvig, 1995] gives a good description of the A* and other search algorithms, which can be used to find the least-cost (shortest) path between two nodes in a graph. It also covers the development of heuristic functions to choose between possible branches at a given node of the graph. These methods provide a basis for the algorithms developed as part of this thesis, but were modified to account for the fact that the results of each move in the physical puzzle are not completely known before the move is made. *Artificial Intelligence for Games* [Millington, 2006] describes the representation of a game environment, and how graph search algorithms can be applied to path finding in games.

Optimizations of Data Structures, Heuristics, and Algorithms for Path-Finding on Maps [Cazenave 2006] presents optimizations to the A* and IDA* algorithms as applied to path-finding on maps. Among the suggestions made in this paper are the use of an array of stacks rather than a priority queue for maintaining open nodes, the use of 'lazy cache initialization', and adding a constant to the next threshold of IDA*. The use of heuristics such as the ALT and ALTBestp is recommended for path-finding on maps, and it is suggested that they are especially appropriate when applied to 'game maps that are complex and are close to mazes'.

A comparison of Fast Search Methods for Real-Time Situated Agents [Sven Koenig 2004] describes how path-planning problems (in computer games) are different from traditional off-line search problems in other fields because autonomous agents in games

will initially have incomplete knowledge of the terrain which results in a large number of contingencies that makes planning difficult. The paper discusses how path planning for agents in games must interleave planning with movement, and compares real-time and incremental heuristic searches for path planning in mazes. It also discusses how heuristics such as the Manhattan distance can be very misleading when applied to mazes.

As part of this project, heuristics were developed to determine which is the best of several possible moves that can be made as part of the search for the physical solution. Rhys Pryce Jones and David Thuente in 1990 demonstrated the use of repeated simulations of a game to test possible heuristics for move selection when the complete game tree is intractable [Jones and Thuente, 1990].

In order to utilize incremental heuristic search algorithms, it is necessary to divide the search space into a finite number of discrete states, representing the marble's current location and the orientation of the maze. Mark Atkin and Paul Cohen suggest a method for using "critical points" for defining state boundaries to facilitate the use of state-based search algorithms in continuous dynamic search spaces. In their paper, they explain the need to distinguish between states only when the "consequences of being in state A differ from those of being in state B". Their algorithm for implementing critical states is demonstrated in the 2 dimensional graphical game simulation "Capture the Flag" [Atkin and Cohen, 2000].

Also required for this thesis is the ability to randomly generate the geometry of the maze. Mark Allen Weiss, in *Data Structures and Problem Solving Using C++* [Weiss, 2000] demonstrates how the 'Disjoint Set' class can be implemented to generate two dimensional mazes with a single unique solution.

The display and animation of the solution to this thesis requires the effective use of OpenGL computer graphics techniques, especially for the display of transparent objects (since everything that happens occurs inside the maze). *Advanced Graphics Programming Using OpenGL* [McReynolds and Blythe, 2005] describes the use of alpha

blending combined with the careful ordering of object rendering to handle the display of transparent objects. Another article found at: http://www.opengl.org/resources/faq/technical/transparency.htm contains information on using selective lighting and multiple rendering to achieve the appearance of clear 'glass' objects.

Finally, the various components of this system (physics simulation, artificial intelligence, graphical display) had to be interfaced to enable them to stay synchronized during the real-time simulation. http://www.gaffer.org/game-physics/fix-your-timestep is an excellent article on synchronizing a physics simulation with the graphical display, maintaining a smooth animation while satisfying the time-step requirements of the physics engine. *Artificial Intelligence for Games* [Millington, 2006] describes the use of state machines to control the behavior of the Artificial Intelligence in games.

3. Problem Description and General Settings

In this section we expand on the objectives of this thesis described in the introduction, and discuss theoretical and practical aspects of its solution development.

3.1 General Discussion

The mazes used in this thesis are a three dimensional extension of the common two dimensional labyrinth. A random two dimensional maze can easily be created by viewing it as rows and columns of square 'cells' separated by walls. The walls are then randomly chosen and removed until there is a single path connecting any two cells, including the starting and ending cells. The extension of this model to three dimensions is fairly straightforward. The 'cells' of the maze are cubes instead of squares, and each cell has 6 walls corresponding to the six directions that one can move to from that cell.

The methodology implemented for determining the physical solution to the maze, i.e. moving the marble from the starting cell to the target cell, is based on a control scheme in which the maze itself is physically rotated, and the marble moves in response to gravity. Each rotation of the maze will be referred to as a 'move', and is implemented as a rotation from one well-defined orientation of the maze to another. This control scheme enables the analysis of the system as a finite number of discrete states, each corresponding to a specific marble location and maze orientation. The states and the moves between them can then be represented as a graph or network and elements of graph theory can be applied when determining the physical solution.

Naturally, the algorithm we propose is not the only control scheme that could be implemented to solve this problem. For instance, a control scheme continuously analyzing and responding to the movement of the marble, and therefore not dependent on specific discrete states, could be developed. It is also possible that such a control scheme, or some other, could physically solve mazes that are unsolvable given the methods used

for this thesis. The subject of this thesis however is the analysis and solution of the system as a set of discrete states. Modeling the problem in this way facilitates the use of well-known Artificial Intelligence search algorithms to find the solution.

Describing this system in terms of states and moves that cause the transition between them has some interesting analogies to traditional game theory. The agent attempting to select the optimal moves can be considered as one 'player', and the physical simulation as an opposing player making a move in response to the agent's. Some of the algorithms implemented in this thesis do in fact involve the agent attempting to anticipate the responding 'move' that will be made by the physics engine when choosing its next move to make, similar in concept to the popular Minimax algorithm [Russell & Norvig, 1995].

The primary goal of this thesis is to develop a solution to a physical puzzle where the consequences of the moves made are complex enough that they can only be determined through simulation. This particular puzzle (the three dimensional maze) does in fact appear simple enough that a deterministic physical solution might be found without resorting to simulation. The geometry consists only of rectangular prisms (the walls) and a single sphere (the marble). Considering all the physical phenomena that can affect the outcome of a move in reality however, this is not as simple as it first may appear. Not only will the marble accelerate in response to gravity, but there are also the effects of static and kinetic friction to consider, the resulting angular momentum, and its effect on collisions, some of which can involve multiple walls. In fact, as will be discussed later in this document, some of the moves encountered in this system are not repeatable in the simulation itself.

It is likely that any intelligent agent attempting to completely determine on its own the results of its moves would essentially be performing a simulation, or at least doing many of the calculations involved in a reasonably accurate simulation. Even if the agent could accurately determine the results of its actions, the goal of this thesis is to model a system where it does not have complete knowledge, and must solve the puzzle without that information. The fact that the simulation itself is not always repeatable means that the

agent, which must contend with the results of that simulation, can never be completely certain what the results of its actions will be.

3.2 Maze Generation

The methodology for generating the geometry of the mazes begins with generating all possible walls, then randomly selecting and removing walls until there is a single unique path connecting any two cells. This methodology of generating mazes is analogous to Kruskal's algorithm for generating minimum spanning trees. Kruskal's algorithm starts with an empty set of edges (arcs) and adds edges to it in order of their cost, discarding those that create cycles, until the graph is connected [Weiss, 2000]. The paths between the cells of the maze can be viewed as arcs between the nodes of a graph, and the goal is to make sure that there is a single arc between any two nodes with no cycles. Unlike Kruskal's algorithm, we select the edge (the walls to remove) at random, instead of selecting the next edge of minimum cost.

A 'Collection of Disjoint Sets', also commonly known as the 'Union / Find" data structure, provides a convenient design pattern for creating the spanning tree. It is used to keep track of which cells are already in the same set (meaning that they are already directly or indirectly connected), and joining separate sets when a wall separating two sets of cells is randomly chosen to be removed. This methodology for generating the spanning tree guarantees that that there is one and only one path between any two cells [Weiss, 2000].

The way the Collection of Disjoint Sets is implemented in this thesis to generate a maze is as follows:

1. The numbers of the internal cells of the maze are identified and each is stored in a separate set (the numbering conventions for the maze cells and walls are covered in Section 6.2 '*Computer Model*').
2. The numbers of the internal walls that separate the internal cells (walls that can be removed) are stored in a list of remaining internal walls.
3. From the internal walls that have not yet been removed (the remaining walls), one is chosen at random.
4. If the two cells separated by the wall are currently in different sets (meaning that there is no path between them), their sets are joined, and the wall is removed from the list of remaining walls.
5. The process is repeated from step 3 until only one set of cells remains.

A unique path from the starting cell to the target cell actually exists as soon as the starting and target cells are in the same set, but step 5 above guarantees that there is a unique path between any two cells, thus creating the numerous 'incorrect' paths that are an important characteristic of such mazes. It is in fact these incorrect paths that provide much of the challenge for the intelligent agent attempting to find a physical solution to the maze.

The list of remaining internal walls used in the above algorithm is saved and used by several of the software components of the system, as discussed later in Section 6 '*Implementation*'. It is used to build the 'collision mesh', which is the representation of the physical entity used by the Physics Engine. It is also used by the Knowledge Engine to determine the absence or presence of walls that might block the path of the marble. Finally, it is used by the Rendering Engine to identify those walls that must be displayed on the screen.

The method described previously for generating a three dimensional maze creates what will be referred to in this thesis as an 'all-layer' maze, meaning that every cell in the cubic maze is part of the actual maze generated, and possibly part of the theoretical or physical solution path. This type of maze is useful for some development and testing, but

has some serious disadvantages as well. The complexity of an all-layer maze is on the order of N^3, making the simulation of large mazes impractical. Perhaps a more serious drawback to all-layer mazes is that it is extremely difficult to interpret the visual simulation results. As can be seen in Figure 3, the numerous internal layers of maze walls make it virtually impossible to visualize the internal structure.

An alternative type of maze developed for testing the various algorithms used in this thesis will be referred to as an 'outer-layer' maze. This type of three dimensional maze is generated such that only the outer layer of cells is part of the maze. The inner $(N-2)^3$ cells of the cube are just empty space. The starting and ending cells are the same as for the all-layer maze, but the solution is constrained to the cells which are in contact with the outside walls. The outer layer of cells on all six sides of the cube are part of a single maze, so the unique solution to the maze can go through any of the sides, possibly visiting a side more than once. Figure 4 shows an outer-layer maze of the same size as the all-layer maze in Figure 3.

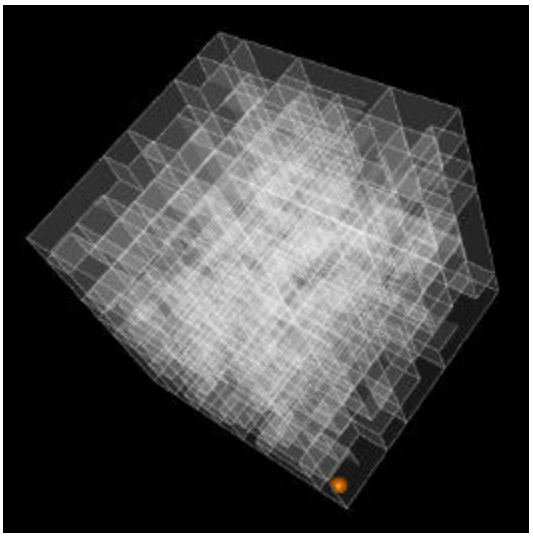

Figure 3: Sample All-Layer Maze

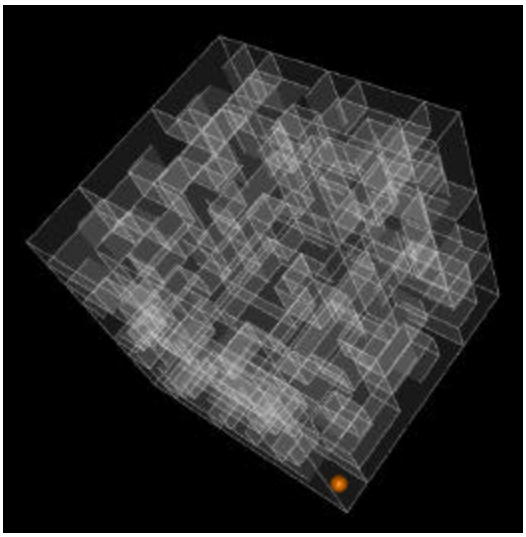

Figure 4: Sample Outer-Layer Maze

There are several advantages to using this type of maze for testing the solution algorithms. First, as can be seen in Figure 4, with the maze making up only the outer

layers of the cube, it is much easier to see what is happening in the graphical display. Not only does the marble remain close to the surface, but all the lines forming the walls at the 'core' of the maze have been eliminated, making the graphical representation much clearer. Secondly, building a maze with only the outer layers of the cube enables larger mazes to be implemented with the same memory and computing capability, since the complex collision geometry that must be handled by the physics engine is restricted to the walls of only the outer cells.

It is important to understand that the geometry of such outer-layer mazes does not reduce this problem to a two dimensional case. All six sides of the cube are part of a single maze with a unique solution. From a cell in the edges or corners of the maze, the marble can also potentially travel along all three axes, so the algorithms implemented cannot deal with this by ignoring one dimension. In fact, all of the solution methods developed for this thesis were tested, without modification, on both all-layer and outer-layer mazes, but much of the performance data was derived using outer-layer mazes, due to the excessive time required to test numerous all-layer cases.

Although we still have the benefit of testing and seeing a 'true' three-dimensional case, constraining the maze to the outer layer does in fact reduce the memory and time complexity to the order of N^2 instead of N^3. There are now $N^3-(N-2)^3 = 6N^2-12N+8$ cells in the maze, rather than N^3. This does not make a huge difference in size for the small mazes that are used in this thesis (since most of the cells are in the outer layer), but when coupled with other advantages such as improved visibility and three-dimensional geometry, they are well-suited as test cases for the agent attempting to solve the maze.

The algorithm used to generate the outer-layer mazes is fundamentally the same as that used to generate all-layer mazes. It still involves randomly selecting walls to be removed. Now however, only those walls positioned between the outer-layer cells of the maze can be removed. These walls must be identified before applying the maze-generation algorithm. In addition, since the cells making up the internal 'hollow' core of the maze are no longer part of the maze, the walls associated with those cells are not added. Some of

these walls are however needed to enclose the outer layer, and must therefore be added as an additional step of the maze generation.

3.3 Analysis of the Theoretical Solution

The mazes in this thesis can be analyzed as a graph or tree, where the nodes of the graph represent the cells of the maze and the connections between them represent an open path between the cells. For example, Figure 5 below represents a simple 2 dimensional maze and Figure 6 shows the undirected graph representing that maze.

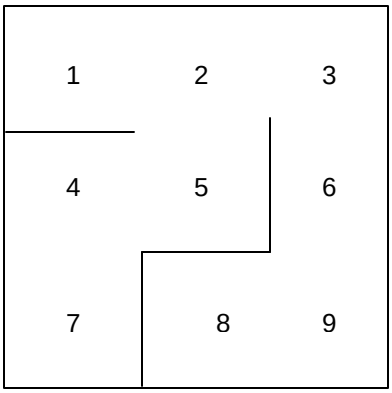

Figure 5: Example 2D Maze

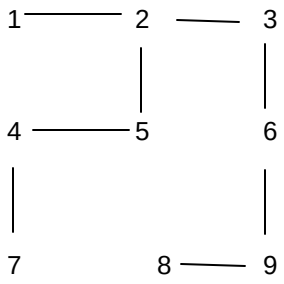

Figure 6: Graph Representing the maze

This graph, representing the unique path between each cell of the maze, will be referred to as the *'theoretical'* graph. This denomination is chosen to indicate that these are the paths that could be followed if there were no physical restrictions on potential moves from one cell to another, as there are for the *'physical'* solution which is developed for this thesis. Notice that the theoretical graph is actually a tree because of the way the maze is generated. The branching factor is equal to three, twice the number of dimensions minus one. This is due to the fact that any node in the tree will have branches corresponding to each of the two directions along each axis. The path will have come from one of these directions, so that direction is subtracted from the branching factor. However, when implementing this tree to perform a search from any node to any other, the root will have an additional branch, since that node will have no parent, and therefore

no 'from' direction. Figure 7 shows the general form for a two dimensional theoretical graph.

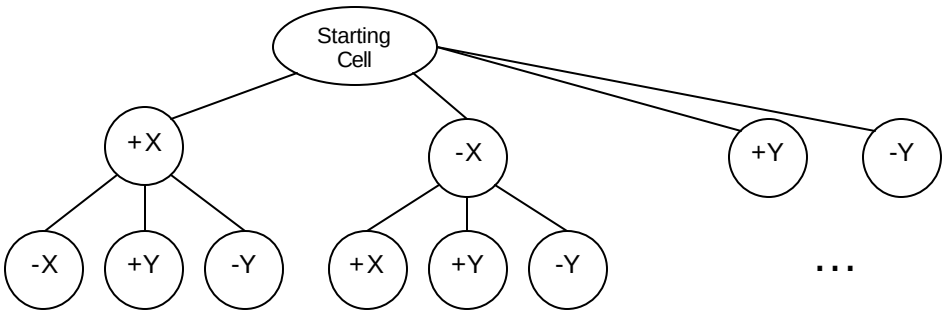

Figure 7: Theoretical Graph for Two-Dimensional Maze

For an actual instance of this model (a specific randomly generated maze), many of these nodes would be absent from the graph, since they would correspond to cells that are unreachable (due to a wall blocking the path) or to locations that are outside the maze (blocked by an outer wall).

For a three dimensional maze, the theoretical graph would look similar but have up to six branches from the starting cell and 5 for any other cell, since movement is possible in two additional directions along the Z axis. The cells and paths can be viewed as a tree with a branching factor of 5 (the six directions in space minus the direction that the cell was reached from). This representation of the three dimensional maze as a tree enables the two dimensional visualization of many aspects of this three dimensional problem, and it will assist in developing and representing many aspects of its solution.

Given the definition of a maze that is being implemented (with a single unique path between any two cells), the theoretical graph, in fact, forms a spanning tree, with no cycles. It is also undirected, since we are not considering any physical limitations that would prevent movement in either direction. Another relevant aspect of this method is that the complete tree, starting at any cell, will contain one node for every cell in the maze.

The similarity of this problem to traditional graph theory is a critical element of this thesis. The representation of a maze as a graph or tree provides a precise definition of the notion of 'maze' that facilitates the construction of mazes as described previously. It also gives us a mathematical model of the maze that can be used in the development and analysis of a solution to the maze.

The path in the tree from the any starting cell of the maze to any target cell without physical considerations will be referred to as the *'theoretical solution'*, as opposed to the *'physical solution'*, which involves physical simulation and is dependent upon rotation of the maze and the marble's response to it. Finding the theoretical solution is a fairly straightforward goal studied in Artificial Intelligence. A simple depth-first search algorithm with backtracking can be applied to the theoretical graph to find the path [Russell & Norvig, 1995]. This algorithm is guaranteed to be complete, and unlike many applications of a depth first search, the resulting solution will also be optimal (since there is only one solution path).

This theoretical solution is used by algorithms in this thesis to improve the search for the physical solution. The length of the theoretical solution line (expressed in the number of cells along the path) is used as a heuristic, indicating the shortest possible physical path from a given cell to the target cell.

3.4 Analysis of the Physical Solution

The physical solution to the maze is much more challenging. Unlike finding the theoretical solution, in the simulated world, the agent rotating the maze does not have definitive knowledge of what the outcome of its move will be. The complexity of the geometry combined with the interactions (collisions and responses) between the marble and the walls make it difficult if not impossible to predict with certainty which cell the ball will end up in. Comparing this to a graph, the effect is that we cannot determine which node of the graph an action (path along the physical solution) will lead to. In fact,

some nodes of the graph may represent states that are impossible to attain. This could occur due to missing walls that prevent the marble from resting in a given state.

Perhaps even more relevant is the fact that backtracking to a previous state will often not be possible. It could even be the case that finding a sequence of actions (rotations) that leads back to a previous state is as difficult, or more difficult, than finding a solution to the goal state. Using the example given earlier, Figures 8, 9, and 10 illustrate the difference between the 'theoretical' solution to the maze and the solution that is physically possible. For example, in Figure 10 we can see that if the maze were oriented such that a marble was resting in cell #3, a counter-clockwise rotation of the maze would result in the marble being located in cell # 9, rolling past cell #6.

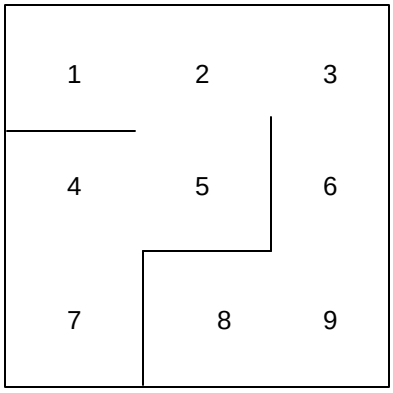

Figure 8: Example 2D Maze

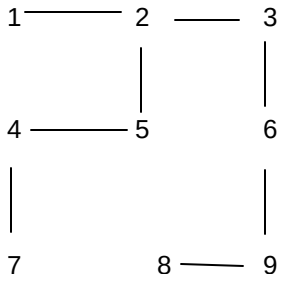

Figure 9: Theoretical Solution

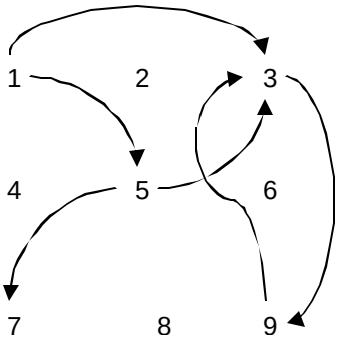

Figure 10: (partial) Physical Solution

Unlike the theoretical solution graph, the graph suggested by Figure 10 is in fact directed. This is due to the fact that some moves are not reversible. For instance, we can roll the marble from cell 5 to cell 3 in a single move, but we cannot move directly from cell 3 back to cell 5. The fact that this graph is directed has serious implications for the physical solution of the maze. It implies that deterministic backtracking cannot be used when searching for the physical solution. This graph also illustrates some other important differences from the theoretical graph that are directly related to the challenges of finding a physical solution. The arcs in the graph do not necessarily connect cells that are adjacent in the maze. They do not form a spanning tree, meaning that some cells are not

reachable by any means. The graph can also contain loops, which must be avoided in the physical solution.

There is a significant difference between this graph and the 'Physical State Graph' that is actually implemented in this thesis to describe the physical states and the moves between them. The physical state graph that is implemented (as described later) considers not only the maze locations, but the physical orientation of the maze as well. It does however share the same characteristics (directed, not spanning, contains loops) as the graph in Figure 10.

Figures 11 and 12 illustrate the difference between the theoretical and physical solutions to a maze. This particular case was one that was generated at random and solved as illustrated using one of the algorithms implemented in the software developed for this thesis. The cells of the maze are numbered starting in the lower-left corner at cell one and increasing from left to right, bottom to top. The first illustration shows the 'theoretical' path through the maze, which can be easily found using a depth-first search algorithm with backtracking. The second illustration shows the optimal physical path that could be taken by the marble given a control scheme in which we begin with the 'Start' cell at the bottom and rotating one-quarter revolution at a time about a central axis.

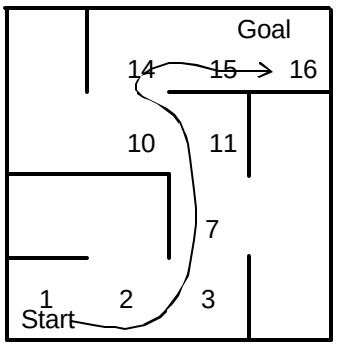

Figure 11: Theoretical Solution

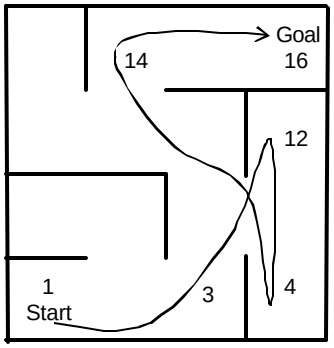

Figure 12: Physical Solution

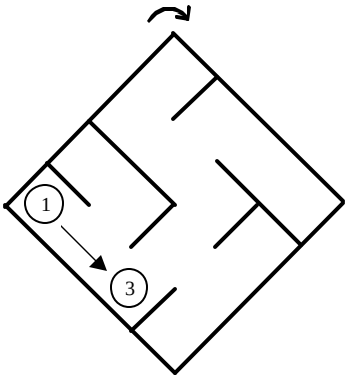

Figure 13: Move 1

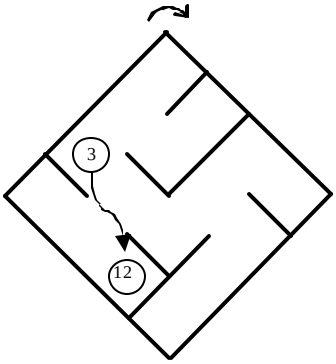

Figure 14: Move 2

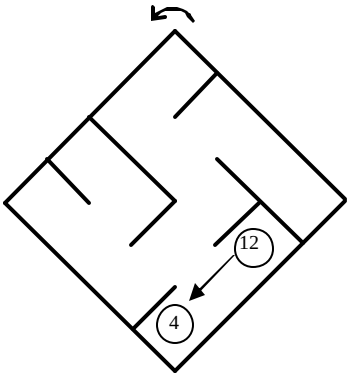

Figure 15: Move 3

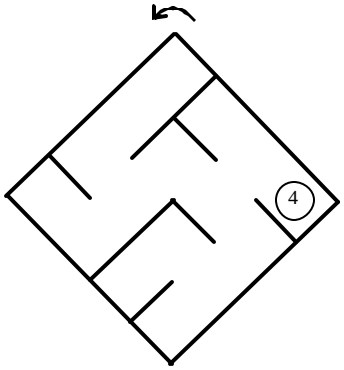

Figure 16: Move 4

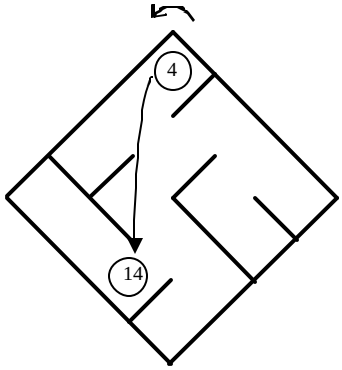

Figure 17: Move 5

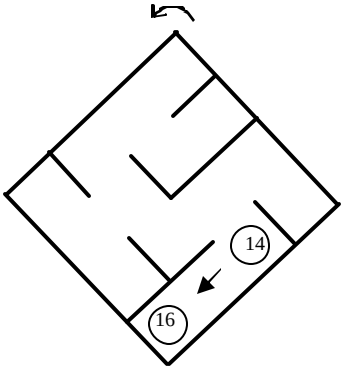

Figure 18: Move 6

Figures 13 to 18 show the progress made at each step in the solution presented in Figure 12. The Marble would start out at rest in cell 1. A clockwise rotation would cause the marble to move to cell 3. Another clockwise rotation would relocate the marble to cell 12. A counter-clockwise rotation now moves the ball to cell 4. The next move is another counter-clockwise rotation, which results in the marble remaining in cell 4. Another counter-clockwise rotation causes the marble to move from cell 4 and make its way to cell 14, and a final move then rolls the ball to the goal, cell 16.

Figure 19 shows the path followed by the marble during the actual simulation of this example.

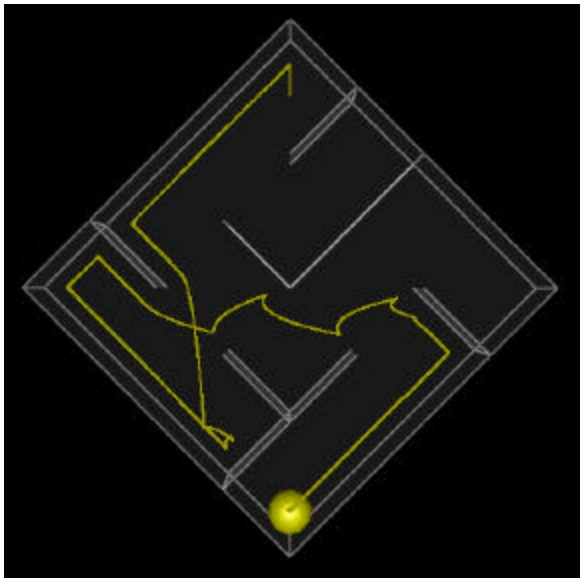

Figure 19: Simulation of Example

Notice that in Figures 15 and 16 the marble has remained in the same location – cell 4. For the purpose of the physical simulation however, this move was necessary to orient the maze such that the next move would result in the desired outcome: moving the marble to cell 14. In other words, it took 2 counter-clockwise rotations to move the marble from cell 4 to 14. The results of a given move are therefore dependent not only on the previous location of the marble, but also on the previous orientation of the maze.

This illustrates that for the purpose of finding and simulating the physical solution, each state (which will later be implemented as nodes on the physical state graph) must be defined not only by the location of the marble, as was the case with the theoretical solution, but also by the orientation of the maze. I have therefore adopted the convention of numbering each distinct state such that the State = (Marble Location -1) * (Number of Possible Orientations) + Orientation. This equation is easily solvable for the value of the State, Orientation, or Location, given the number of possible orientations and the other two values. For the two-dimensional maze used as an example above, the 'Number or Possible Orientations' is equal to 4, since the rotations (moves) were each of 90 degrees. For the three dimensional mazes described later in this thesis, more orientations are required.

The algorithms developed for this thesis are all fundamentally based on building and analyzing a 'Physical State Graph' defined by the physical states (marble location and maze orientation) and the moves (rotations of the maze) that transition the system from one to another.

Figure 20 shows the general form for the physical state graph in which we have two possible moves that can be made at any time, as is the case in the example in Figures 13 to 18 above. The states are represented as the nodes of the graph. The arcs connecting the nodes represent the moves, or rotations that can be applied to the maze to change the state of the physical system. The branching factor, or number of connected nodes, is equal to the number of moves that can be made from a given state. This in turn is dependent upon the possible orientations of the maze. In the example given previously in Figures 13 to 18, the branching factor would be 2, one for a 90 degree counter-clockwise rotation and one for a 90 degree clockwise rotation.

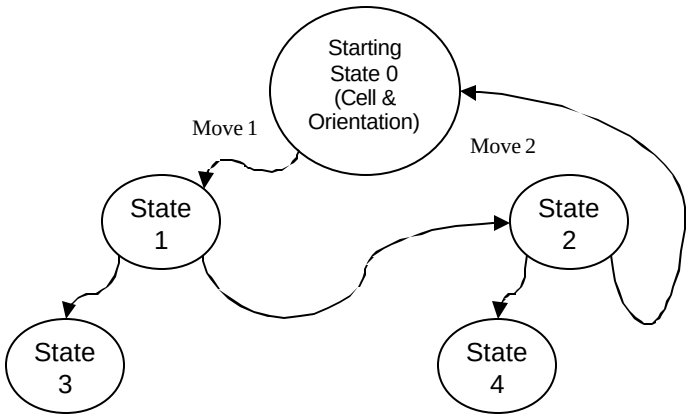

Figure 20: Physical State Graph for Two-Move Control Scheme

There are some important differences between the physical state graph depicted in Figure 20 and the theoretical graph discussed earlier and shown in Figure 9. Like the Physical Graph shown in Figure 10, and by the same reasoning, this Physical State Graph is directed, non-spanning, and contains loops. Other important differences exist between this and the theoretical graph. In this graph, multiple nodes can correspond to a given cell

in the maze. This is of critical importance when the agent needs to determine if a desired cell has been reached, or when deciding if one node is 'closer' to the target cell than another.

The branches in the theoretical graph correspond to adjacent cells in each direction. Since theoretical movement in a given direction may not be possible (due to the presence of walls), there will be many branches that are absent in the theoretical graph. In the physical state graph however, the branches correspond to the physical moves rotations of the maze that can be made. Since all defined rotations of the maze can be made at any time, all branches in the physical state graph are defined. This does not however imply that all defined states will be present in the maze. Many if not most of the defined states will simply not be reached by any branches in the graph, either because the cells corresponding to these are not reachable, or more likely, because the physics do not allow the marble to stabilize in that state (one or more of the walls required to stop it have been removed). It is also relevant that unlike the theoretical graph, the nodes in the physical state graph that are directly connected do not necessarily correspond to adjacent cells in the maze. This is due to the fact that a single move can result in the marble staying in the same cell (in a different orientation) or moving many cells away.

When the simulation begins, the system is initialized with only one node in the physical state graph. This initial node represents the marble's starting location and the initial orientation of the maze. From this initial node, there will be branches or arcs to other states that are currently unknown (since we are assuming that the marble location resulting from a move can only be determined by the physics simulation). After a move is made, and the marble has come to rest in a 'stable' state, the system can obtain the resulting marble location from the physics engine. The physical state graph is then updated with the resulting state for that branch.

Figure 21 shows the physical state graph corresponding to the first move depicted in Figure 13 from the previous example (duplicated here for convenience). The '0' state represents the initial starting location of the marble (cell 1) and the initial orientation

(orientation 0). The first move rotates the maze to orientation 1 and moves the marble to cell 3, corresponding to state 9 ((Marble Location -1) * (Number of Possible Orientations) + Orientation).

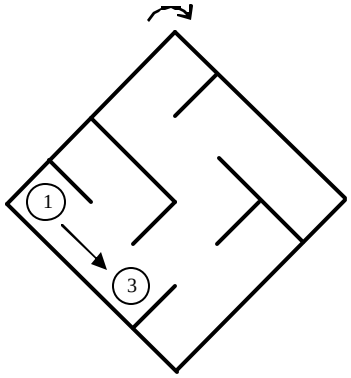

Figure 13: Move 1

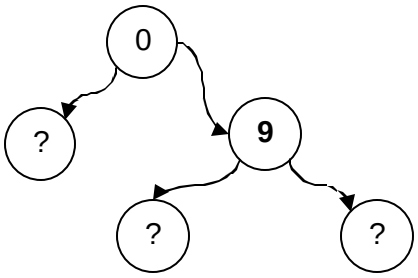

Figure 21: Physical State Graph after First Move

Here we can see one of the primary characteristics that distinguish this thesis from many other search applications found in the study of artificial intelligence. For our purposes, after the first move is made, the marble is now physically located in state 9. Although we could in this case rotate the maze in the opposite direction to get the marble back to state 0, this is not generally the case, and therefore cannot be assumed. For example, referring again back to the previous example, Figure 17 (duplicated below) shows a counter-clockwise rotation moving the marble from cell 4 to cell 14. It is easy to see that no single move will move the marble back to cell 4. In fact it is entirely possible in some mazes that there would be no move or sequence of moves that will return the system to a previous state. What this means is that this is a directed graph and backtracking, which could be used in finding the theoretical solution to the maze, cannot be utilized in finding the physical solution.

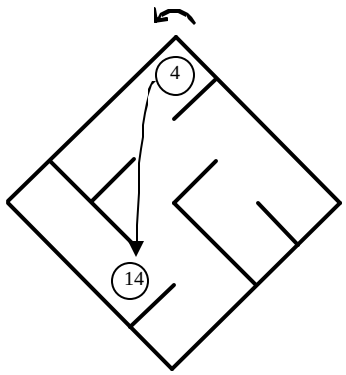

Figure 17: Move 5

This has some significant consequences for this system. Looking back at Figure 21, this implies that in order to evaluate the branch off the root that was not initially chosen (the unknown state to the left of '0' that corresponds to a counter-clockwise rotation from the initial state), we must somehow find another path to it or a path back to the initial state. It is even possible that a path not initially chosen in fact leads directly to the goal state, and choosing the wrong move will prevent us from ever getting back to it.

3.5 Simulation of the Physical Solution

In order to test and demonstrate the solution algorithms developed for this thesis, the Newton Game Dynamics physics engine was used to model the maze and marble, and simulate the actions performed on the system. For the simulation to be performed, a control scheme was developed to define the possible orientations of the maze and the actions that can be performed on it.

The possible orientations of the maze were chosen such that in each orientation, one vertex of the maze is pointing straight upward. This choice of orientations ensures that after a move, the marble will come to rest in the corner of a cell of the maze, in a specific state. The moves are therefore simple rotations of the maze from one orientation to another. The axis of the rotation can easily be computed as the cross-product of the vectors representing the starting and ending orientations.

Figure 22 illustrates how this works. The green line is a vector representing the starting alignment. It points from the center of the cube to the vertex at the top. The red vector points from the center of the cube to the vertex that is to point straight upward after the move. The blue vector is the cross product of the green and red vectors. Since the cross product is necessarily perpendicular to both the starting and ending orientation vectors, it can be used as the axis of rotation. Also, since for every possible move, the starting and ending orientation vectors will always point from the center of the cube to different vertices, they cannot be parallel and the cross product will therefore always exist.

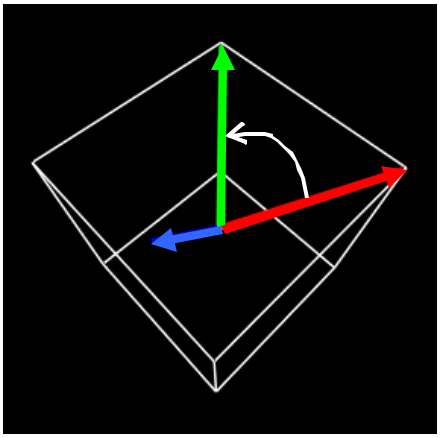

Figure 22: Orientations and Axis of Rotation

The simulation of a move is performed by applying a sufficient amount of torque about the axis of rotation, which is modeled as a '*Hinge*' in the Newton physics engine. Essentially, this is very similar to the way a human would solve the puzzle by rotating the maze with his or her hands and wrists to a new orientation to get the marble to roll in the desired direction. Section 6, *'Implementation'*, discusses this control scheme and the use of the Newton physics engine in greater detail.

3.6 Additional Challenges

The physical aspects of this problem provide several interesting challenges for this thesis. These challenges are described in this section, and their solutions are presented in subsequent sections.

Unsolvable Mazes

In a previous section, a fairly straightforward method for building a random maze was described. One of the challenges of this thesis is created by the fact that not all mazes built in such a way are physically solvable. The simple 2-dimensional example in Figure 23 below shows how this can happen. Although this is a perfectly legitimate maze (all cells are connected by a unique path), we can see that any simple rotation, in either direction, would cause the object in the maze to fall or roll between cells A and B, never reaching the goal.

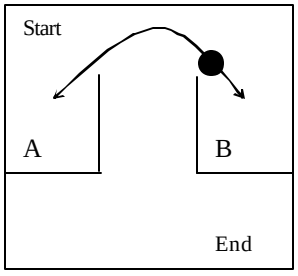

Figure 23: Physically Unsolvable Maze

This problem could be solved in a number of ways. One would be to attempt to quickly shift the direction of rotation while the object is falling to get it to go through the opening leading to the end. This would be very difficult for an intelligent agent, and might not be physically possible in some similar situations, given the momentum of the object and the mass of the maze itself which would prevent instantaneous changes in rotation. Another solution would be to rotate the Z axis upward (so that we would be looking down on the maze), giving us a flat surface on which to roll the object through the opening. This would require a much different solution methodology and control scheme than the one

developed for this thesis. One of the requirements for this system is therefore that the agent recognize when it is incapable of solving the maze that has been generated, so that it does not go into an endless loop of failed attempts.

Problems similar to these, which make some mazes physically unsolvable, also preclude the use of AI search algorithms that rely on backtracking. The same situations that make the goal point unreachable could make any previous state unreachable as well.

Non-Repeatable Moves

The algorithms developed for this thesis are highly dependent on learning from past experience, and therefore repeatability is very important. When repeating the same action (move) from the same starting state, the agent should be able to expect the same results. In this manner, it is able to learn from experience, using past results as a basis for decision making. It may seem that repeatability could always be expected, given the same physical simulation of the same event, but this is not always the case. The simple two dimensional example in Figure 24 below shows why some results are not always repeatable. In this example, the marble, traveling with a velocity indicated by the arrow, may fall toward cell 7 (to the left) or proceed to cell 12 (to the right) with nearly equal probability. The slightest round off errors, possibly caused by the residual effects of prior events, could result in the marble ending up in cell 12 some times and cell 7 other times.

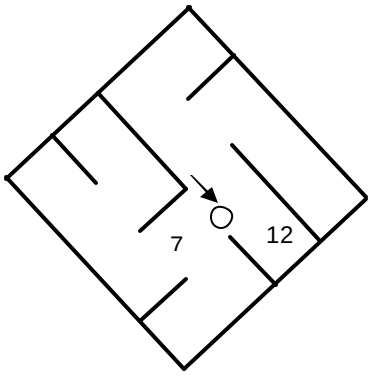

Figure 24: Unpredictable Results in 2-D

The opportunities for non-repeatability are even greater when three dimensional mazes are modeled. Figure 25 shows why this is the case. Here we can see that the marble, which is rolling or sliding down the valley between walls 'A' and 'B', will reach a point where it will fall to either the right or the left with equal probability. There is no way to reliably predict which way it will turn, and past history will not be a good indicator of future results.

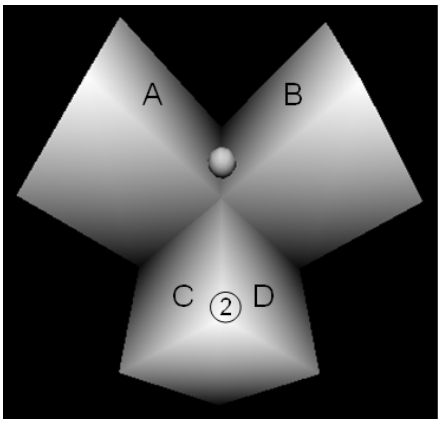

Figure 25: Unpredictable Results in 3-D

Frozen States

Another difficulty described by Figure 25 is that the marble may not fall to either the right or left, but roll down the edge between walls 'C' and 'D', stopping when it reaches another wall (not shown) at point '2' and remaining balanced there. This does not result in the marble resting in one of the states that we have defined. In fact, looking at the projection in Figure 26, it is clear that the marble rests partially in 3 different cells. This would be a nearly impossible condition if this were an actual maze with a human controller. It would require perfect balance and control. On a computer however, this can be easily simulated with the appropriate control scheme and choice of orientations, and in fact does occur fairly often with some of the algorithms developed for this thesis.

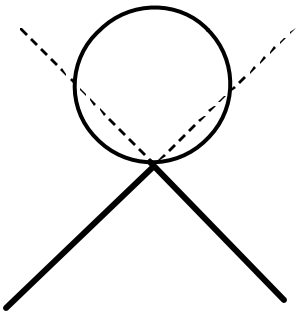

Figure 26: Marble Balanced in Frozen State

In this thesis, this condition will be referred to as a '*Frozen*' state, since the marble has come to rest in a state that is not well-defined for our purposes, and the physical solution can therefore not proceed. It should be understood that this does not represent any instability as far as the physics simulation is concerned. The maze could in fact be tipped to one side or another to cause the marble to fall, or even rotated carefully to get to roll back down the edge between walls 'C' and 'D' in the opposite direction from which it came.

Fortunately, before assuming that the marble has settled in a well-defined state and updating the physical state graph accordingly, we can verify that it has indeed settled in an acceptable state. The way that this is done is to project the vector representing gravity onto the three axes. The possible alignments for the maze have been chosen such that the gravity vector will have a non-zero projection on each of the three axes. Since the cells of the maze are aligned with the axes, these projections represent directions in which there must be a wall blocking movement. We can check for the presence of the required walls, and if any are absent, we know that we are not in a well defined state.

In the first few algorithms that are developed, those mazes in which a frozen state such as this occurs are treated as unsolvable. Later in this thesis, a method is discussed to nearly eliminate the possibility of this situation occurring.

4. Solution Algorithms

In this section we describe six algorithms implemented to physically solve the puzzle, and discuss their performance.

4.1 General Solution Methodology

The algorithms developed for this thesis are all fundamentally based on analyzing a 'physical state graph' that is built based on the physical states of the maze and marble, and the moves that transition the system from one to another. Figure 27 represents an example of a physical state graph after several moves have been made. The nodes represent states, which are combinations of the location of the marble and the orientation of the maze. The directed arcs correspond to the moves (rotations) of the maze that were made to transition from one state to the other. The nodes identified by a question mark indicate unknown states that can be reached by a single move from an already known state. Since these are moves that have not already been made, it is unknown whether these states represent states that have been previously visited, states that have not yet been discovered, or even states corresponding to the target cell.

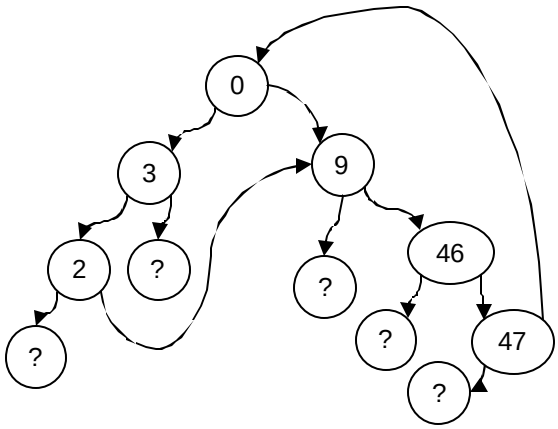

Figure 27: The Physical State Graph

Implementing this graph to find the physical solution differs in several significant ways from most search algorithms typically utilized in artificial intelligence.

- Whenever the next move must be chosen, the agent does not have knowledge of the entire graph; only of the part that has been developed based on previous actions. It therefore cannot use a complete search algorithm to find a physical path leading to the target cell.

- Search algorithms typically involve selecting a node to expand, and then evaluating all the children of that node. Different search algorithms use different criteria for determining the order in which to evaluate the child nodes [Russell and Norvig, 1995]. For this system, evaluating a child node involves making a physical move to the new state. Since backtracking is not feasible, once a child is chosen, the option to evaluate its siblings no longer exists.

- There is nothing known about each unknown state. It is a function of the parent state, the geometry of the maze, and the physics involved, and can only be determined after the move is made. We therefore have no immediate way of determining which may be the best branch to take.

Although physical backtracking is not considered possible, the system can search for the goal by choosing from the available branches off the current state. In general, a move to an unknown state will have one of 2 possible outcomes: it can lead to a previously undiscovered state, which will be added to the physical state graph, or it can lead to a state that has already been encountered, and is therefore already in the physical state graph. In the later case, the arc corresponding to the move that was made is connected to the node already representing that state. In this system, branches cannot lead back to the same state. Although after a move, the marble will often remain in the same location, but the orientation after the move, and therefore the state, will be different. Proceeding in this manner, the agent can 'explore' the maze, building a graph of the results as it proceeds.

When searching this physical state graph for a desired state from the current state, backtracking back to the node representing the current state can be used. The reason is that this graph represents actions and resulting states that have already been discovered.

We can safely assume that the same actions taken from the same states will again lead to the results indicated in the graph (it turns out that this is not always the case, but the repeatability is reliable enough that we can base decisions on it with reasonable confidence). To help clarify this, assume that the right branch from state 47 in Figure 27 is taken and found to lead back to state 0. If we now want to find a path back to state 46, we can use any traditional search algorithm employing backtracking (e.g., depth-first, breadth-first), to find the path (set of moves) 0--> 9 --> 46.

We can implement a traditional search algorithm, much like the depth-first search algorithm used to find the 'theoretical solution' discussed earlier, since in this case we are not actually making any physical moves, simply evaluating paths that are already known. Once the search is complete, we can then perform the physical moves leading to the desired state found. In this manner, a 'theoretical' search algorithm is implemented to guide the decision-making for the 'physical' search.

As part of this thesis, six algorithms were developed to find the physical solution to the puzzle. The algorithms differ in the level at which they incorporate additional knowledge of the maze geometry and physics. Essentially, they improve the performance of the agent solving the maze by improving its intelligence.

The algorithms are listed here, and their description and performance is discussed in the following sections:

- Blind Search Methods
    - Depth-First
    - Breadth-First
- Informed Search Methods
    - Heuristic Search
    - Improved Heuristic Search
    - Single Move Prediction
    - Complete Solution Prediction

4.2 Blind Search Methods

The first two algorithms developed to find a physical solution use no additional knowledge of the maze other than the physical state graph that has been developed based on moves already made. Search algorithms that employ no knowledge of the domain when choosing which path of a tree to expand are commonly referred to as 'blind' or 'uninformed' searches [Russell and Norvig, 1995]. These methods can therefore be summarized as blind searches in a directed graph with no backtracking.

Figure 27, discussed earlier, shows the physical state graph that describes the known states and the moves that have been shown to lead between them. At the time a move needs to be chosen, the graph only reflects what has already been discovered, so the agent cannot find the complete path to the target cell. The best it can do with no further knowledge is to find a path from its current state to one of the unknown states in the physical state graph. By such experimentation, the physical state graph is expanded until a state corresponding to the target cell is found.

This exploratory methodology differs considerably from the way that search algorithms are generally employed. Whereas depth-first and breadth-first algorithms are generally used to find a desired state in known data, here they are being employed to intentionally find unknown states.

The use of blind search algorithms in this way to find the physical solution to the maze is not likely to be efficient, but could be considered 'complete' under three conditions:

- The algorithm is implemented in a manner that avoids loops. Figure 27 above shows that cycles can exist in the physical state graph, and these must be detected to avoid the possibility of infinitely long sequences of physical moves.

- A solution must be possible given the control scheme (the physical moves made to the maze and the simulated results). It is possible that this methodology will not

be able to physically solve a maze that could be solved another way – perhaps using different orientations or a simulation that implemented a continuous control scheme rather than finite states.

- No set of states are encountered that the control scheme is not capable of moving out of. For the same reasons that it may not be possible to reach a target state, the inability to physically backtrack means that the marble could end up in a set of cells from which it cannot escape. It is also possible that another algorithm, even with the same control scheme, could have avoided this 'hole' by choosing a different move at some point and found a path to the target state.

If these conditions are met, the physical solution to the maze will be found. The system also detects when it cannot find a solution, since it is no longer able to find any unknown state from its current state when searching the physical state graph, implying that it has already been everywhere it is capable of going.

The three conditions listed above are in fact not always met given the mazes that are generated for this thesis. For the results presented in this subsection, the cases that failed were simply discarded, and only the data for solvable mazes was used to measure the algorithm's performance. Later sections of this thesis cover methods to improve the reliability of the agent solving the maze.

In Table 1 below, the performance results are summarized for the depth-first and breadth-first algorithms finding a the physical solution by searching for unknown states in the physical state graph. For these test cases, 100 three-dimensional all-layer mazes measuring 3 cells on each side were used. The same mazes were used for the breadth-first and depth-first tests. The control scheme utilized 8 possible orientations, each corresponding to one corner of the maze being oriented upward. The choice of fairly small mazes was driven by the inefficiency of the blind search algorithms, and the extensive time therefore required to run all 100 cases (the depth-first cases took almost 10 hours to run).

The measure of performance used here is the total number of physical moves of the maze that must be simulated, including repeated moves, to get the marble to the target cell. This is related to another aspect of this thesis that differs considerably from other applications of search algorithms. Here the time to perform the search of the physical state graph is not even considered to be important. The reason is that the size of the graph (for mazes that are small enough to perform a reasonable simulation) is very manageable, and the search algorithm is polynomial. Even the largest mazes tested had graphs with only a few thousand nodes. This means that a search of the graph using any reasonable algorithm can be performed in a fraction of a second on a modern personal computer, whereas the time required to simulate the move chosen can take several seconds. In fact, algorithms discussed later in this thesis which are designed to make better decisions on which physical moves to make actually spend a lot more time searching the physical state graph.

100 3x3x3 All-Layer Mazes *Average Length of Theoretical Solution* *from Starting Cell to Target Cell = 8.5 Cells*					
Search Algorithm Employed	Avg. Num. Moves	% Frozen	% Unsolvable	Num. Repeated Moves	Num. Unrepeatable Moves
Depth-First	96.8	33	1	4013	17
Breadth-First	52	11	2	610	4

Table 1: Blind Search Results for 3x3x3 All-Layer Mazes Using 8 Orientations

The data shows that even for small 3 dimensional mazes, this 'exploratory' method of finding a physical solution is very inefficient. The depth-first algorithm is particularly bad, requiring nearly twice as many physical moves to solve the maze. The reason can be seen in the physical state graph in Figure 28 and the corresponding tree in Figure 29 representing the paths followed by the blind search algorithms (for simplicity, the example shown has a branching factor of 2, such as would be employed for a two-dimensional maze, but the principle is the same for the three dimensional test cases used).

If the current state is 0 (the root of the tree), and the agent is searching for an unknown state to 'explore', a simple depth-first search expanding the left branches first would find the path 0 > 3 > 2> 9 > ?. The path would require 4 physical moves to reach the unknown state. The breadth-first search algorithm will find the unknown state that is the fewest moves away, in this case 0 > 3 > ?. In this way, the breadth-first algorithm expands its knowledge of the maze (by building the physical state graph) much more quickly, thus finding the target cell with fewer moves.

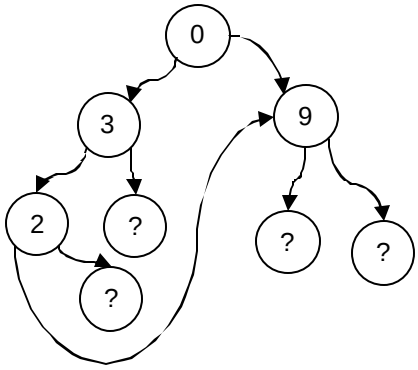

Figure 28: Physical State Graph

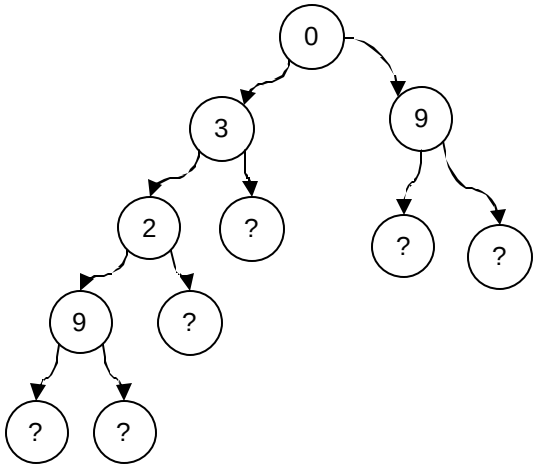

Figure 29: Corresponding Search Tree

When observing the animation of the solution using these algorithms, it can be seen that the use of the breadth-first algorithm results in fewer repeated or seemingly senseless moves. Although fewer paths are re-traced, the breadth-first algorithm still appears to be making many un-intelligent moves. For instance, it will often move the marble from one corner of a cell to another, and then roll it right back. From the point of view of an intelligent observer, this appears to make no sense. This does however make sense to an agent that has no knowledge of the geometry of the maze. It may have learned from experience that a given move from state 'A' results in state 'B' with the marble located in the same cell, but since the physical state graph is directed, it cannot predict the inverse move will change the state from 'B' back to 'A'. In fact, with this algorithm, it has no way to even determine what the 'inverse move' would be.

The test results in Table 1 also show the number of repeated moves and the number of unrepeatable moves. The number of repeated moves is the number of physical moves that have to be made more than once during the entire simulation. These repeated moves correspond to arcs that already connect two states in the physical state graph, and therefore do not lead to any increased knowledge of the maze. Obviously, for an algorithm such as this, these moves are undesirable.

In addition to the performance penalty, the other disadvantage to repeating the same move is that it may not always result in the same state. These 'unrepeatable' moves do not necessarily cause the algorithm to fail. It will simply update the physical state graph with the latest results for the move and continue searching for unknown states. This does however severely impact performance by effectively nullifying knowledge that has already been developed. Fortunately, the repeatability is over 99%, but increasing the number of repeated moves, as the depth-first algorithm does, is going to increase the number of unrepeated moves proportionately.

Table 1 also reflects the danger of implementing an inefficient algorithm that results in an excessive number of moves. As explained previously, the control scheme used here, with 8 orientations, makes it possible for the marble to come to rest balancing on an edge of the maze, in an undefined ‘frozen’ state. For these algorithms, continuing from a frozen state is not feasible, and the test case therefore must be terminated. Later, a methodology for eliminating this possibility is developed, but for the results shown above, these frozen test cases were treated as unsolvable, and not factored into the performance statistics.

As we can see from Table 1, the number of moves required by the depth-first search significantly increases the chances of encountering a 'frozen' state. It resulted in the failure of 33 test cases, even thought the breadth-first algorithm was able to solve 21 of those same mazes. In fact, only 2 of the 100 mazes were determined using either algorithm to be physically unsolvable, meaning that the agent moved the marble to every state it was physically able to, without ever reaching the target cell.

4.3 Heuristic Search

In order to find a more efficient physical solution to the maze, knowledge of the maze itself can be utilized to evaluate the possible unknown states that can be investigated. Heuristic searches [Russell and Norvig, 1995] employ heuristic functions to improve the efficiency of the search. At every step, they make a decision about which node or state to be expanded next based on the heuristic estimation of how likely it is that that state will lead to the solution faster. Unfortunately, due to the random nature of the maze, simple heuristics such as the Manhattan distance have been shown to be very misleading, and therefore not of much value [Sven Koenig 2004]. This algorithm uses the length of the theoretical path from the marble's location to the target cell as a heuristic to decide which unknown state to explore next.

Unlike the Manhattan distance, the theoretical solution path from a given cell to the target cell does take into account the actual physical nature of the maze, and therefore the length of the theoretical path has the potential to serve as a reasonable heuristic. The method by which the maze is generated (as a spanning tree) guarantees that there is a unique theoretical path from each cell to the target cell. The path itself is found using a depth-first search of the theoretical solution tree, as discussed previously in Section 3.3 '*Analysis of the Theoretical Solution*'. Since that theoretical path can be quickly determined for each unknown state (given the reasonably small size of the theoretical graph), it is reasonable to use this to evaluate the unknown states that are available.

This heuristic search algorithm involves searching the physical state graph to find all the unknown states that are reachable from the current state, and are within one move from a known state. These are evaluated based on a heuristic value equal to the length of the theoretical path (the length being defined as the number of maze cells along the path) connecting the known parent state to the target cell. We cannot actually evaluate the unknown state itself, since nothing is known about it (the move to the unknown state could result in the marble resting in the same cell, or many cells away). We can however use the hypothesis that there is a reasonable correlation between the desirability of this

unknown state and the heuristic value of its parent state. In other words, the agent will act on the assumption that if a state is close to the goal, a single move from that state is likely to get the marble close to the goal.

The algorithm can be summarized as follows:

1. For each unknown state in the physical state graph:
    1.1. Find a path from the marble's current state to the node corresponding to the unknown state using the same breadth-first search implemented in the previous solution algorithm.
    1.2. Determine the known parent cell for that unknown state
    1.3. Set the heuristic value equal to the theoretical path length from the known parent cell to the target cell.
    1.4. If this heuristic value is better than the heuristic value for any other unknown state previously evaluated, designate this as the 'best unknown state' and store the path leading to it.
2. Perform the physical moves corresponding to the path to the 'best unknown state'.

This algorithm is very similar to the 'Best-First-Search' algorithm defined on page 93 of Artificial Intelligence [Russell and Norvig, 1995], with the theoretical path length serving as the evaluation function 'Eval-Fn'. Unlike the general 'Best-First-Search' algorithm however, the physical constraints in this system mean that only one branch of the tree can be expanded, i.e. physically simulated. Many of the known states in the physical state graph have multiple unknown children. Since it is the theoretical path from the parent's cell to the goal that is used as a heuristic, each of the children will be given the same value.

Since we can only evaluate one of the children (physical moves are not reversible) we must choose between them. For the algorithm in this section, the choice is made arbitrarily (the first child found ends up being selected). This arbitrary decision is reasonable. If a bad choice is made, the same child state will not be chosen again

(assuming that the marble finds its way back to the parent state) since that child is no longer an unknown state. The next algorithm in this thesis discusses a more intelligent way to chose between the unknown child states.

To find the unknown states to be evaluated, the physical state graph must be searched, starting in the current state. For this algorithm, a breadth-first search is implemented to find each unknown state. This will necessarily find the path with the fewest physical moves to reach the unknown state that is chosen to be investigated. The agent's performance will be improved by minimizing the number of repeated moves, and reliability will also be improved by reducing the number of opportunities for non-repeatable moves or frozen states.

There is one major difference between this and many other artificial intelligence search algorithms. Here the objective is definitely not to reduce the time searching for a node in the graph. In fact, the time spent on searching is being dramatically increased. In the blind search algorithms discussed previously, we were searching for a single unknown state in the physical state graph to be investigated. With this algorithm, all the unknown states must be searched for. Then, for each of those unknown states, a complete depth-first search of the theoretical graph must be performed to determine the value of the heuristic. The number of repeated searches can be reduced by calculating and storing the heuristic value for each state when it is first encountered, but all of the unknown states will still have to be found at each decision point.

The reason the increased time for searching is justified is that the time cost of searching the physical state and theoretical graphs with any reasonably efficient algorithm is orders of magnitude less than the cost of actually simulating the moves. It is therefore worth the extra time to do additional searching to minimize the number of physical moves that must be simulated. In addition to saving time, reducing the number of physical moves to be simulated also improves reliability as discussed earlier.

Table 2 below summarizes the results of testing this heuristic search on the same 100 3x3x3 all-layer mazes used to test the blind search algorithms.

100 3x3x3 All-Layer Mazes *Average Length of Theoretical Solution* *from Starting Cell to Target Cell = 8.5 Cells*					
Search Algorithm Employed	Avg. Num. Moves	% Frozen	% Unsolvable	Num. Repeated Moves	Num. Unrepeatable Moves
Depth-First	96.8	33	1	4013	17
Breadth-First	52	11	2	610	4
Heuristic	31.5	9	2	509	5

Table 2: Heuristic Search Results for 3x3x3 All-Layer Mazes Using 8 Orientations

The data shows that the use of a heuristic to estimate the desirability of an unknown state can significantly improve the performance of the physical search. The use of the theoretical path length as a heuristic is effective, even when basing it on the parent state. The number of moves required to physically solve a maze, was reduced significantly, showing that there is indeed a positive relationship between the distance from the parent state to the target, and the distance from the child state to the target. The next algorithm presented provides a method for evaluating the desirability of the unknown states themselves.

4.4 Improved Heuristic Search

In the previous heuristic algorithm, it was assumed that nothing was known about each unknown state in the physical state graph, and its desirability could therefore not be determined by evaluating it directly. The theoretical distance from the parent state's cell to the target cell was therefore used as a heuristic for each of the parent states unknown children. In this section, a method is developed to calculate a heuristic for the unknown (child) states themselves. It does this by utilizing knowledge of the maze geometry, the nature of the moves that can be made from a given state, and the physics involved.

The choice of possible orientations for the maze (which consists of one corner or vertex of the maze pointing directly upward) enables the algorithm to determine with certainty the cell of the maze that the marble will move to next in response to a specific move. For the two dimensional case, it is easy to see that a rotation of the maze will result in the marble rolling away from the corner it is currently resting in, and either being stopped in the current cell by another wall or entering the adjacent cell. The direction of travel, and therefore which adjacent cell will be entered, can be determined by projecting the vector representing gravity onto the two axes. One of these projections will be in the direction of a wall that the marble is resting on, and the other will represent the direction of travel.

For the three dimensional case, the nature of the geometry and the possible orientations guarantees that any time the system is stable, the marble will be resting in a corner where three mutually perpendicular walls intersect. We can picture the vector representing gravity pointing from the center of the maze cell where the marble is located to the corner where the marble is resting. In the next orientation, after any move, the gravity vector will point from the center of the cell to one of the adjacent corners. This new ‘downward’ corner will necessarily have two walls in common with the original orientation, and movement in those directions will therefore not be possible. The only possible movement will be in the direction of the new third wall, which can be present or absent in the maze. If the wall is present, the location of the marble in the new orientation will be the same as the old (it will simply roll to and stop in a new corner of the same cell). If the new wall in

the new orientation has been removed (as part of the maze generation), the next location of the marble will be in the cell corresponding to the other side of that missing wall.

Figures 30 and 31 illustrate this. In Figure 30, we can see that the marble is initially at rest at the intersection of walls 1 (bottom left), 2 (front), and 3 (bottom right). After a rotation to an adjacent vertex of the cube, Figure 31 shows that the marble will roll down the edge between walls 2 and 3. Walls 2 and 3 will still be preventing any movement in those directions, so the only possible progress will be in the direction of wall 4. If wall 4 is present, the marble will come to rest in the new corner. Otherwise it will continue to the cell on the other side of wall 4.

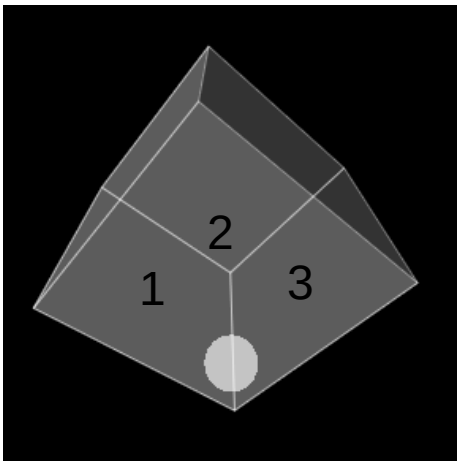

Figure 30: Marble Resting Before Rotation

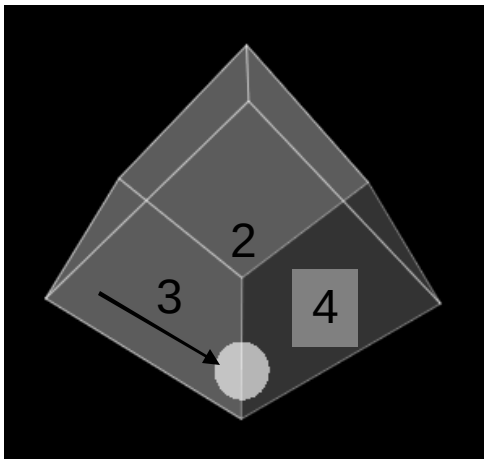

Figure 31: After Rotation

It is important to understand that if the marble does relocate to a new location as the result of a move, this adjacent location is not necessarily the location it will end up in. It may very well be the first of many locations that will be 'visited' while the marble rolls or falls through the maze in response to gravity and collisions with other walls. However, we are now assuming that the entire path that the marble will take is too complex to determine without simulation, so the next adjacent cell is the best estimate for the unknown state occurring after the move and will be used as a basis for the desirability of the move being evaluated.

Using this next adjacent cell as an estimate of the cell corresponding to the unknown state, we can use the length of the theoretical path from that cell to the target cell as a

heuristic to determine how desirable the unknown state is. As with the previous solution algorithm, the theoretical path length to the target cell is determined using a depth-first search of the theoretical solution tree. In this case however, it is the length of the theoretical path from the adjacent cell (which is itself an estimate of the cell corresponding to the unknown state) to the target cell that will be used. In this way, the algorithm will estimate the desirability of each unknown state, which will also enable it to choose between multiple unknown states with the same parent.

This 'Improved Heuristic Search' algorithm can be summarized as follows:

1. For each unknown state in the physical state graph:
    1.1. Find a path from the marble's current state to the node corresponding to the unknown state using a breadth-first search.
    1.2. Determine the known parent cell for that unknown state.
    1.3. Determine the next adjacent cell that the marble will occupy as a result of the move corresponding to the branch from the parent state to the unknown state.
    1.4. Set the heuristic value equal to the theoretical path length from this next adjacent cell to the target cell.
    1.5. If this heuristic value is better than the heuristic value for any other unknown state previously evaluated, designate this as the 'best unknown state' and store the path leading to it.
2. Perform the physical moves corresponding to the path to the 'best unknown state'.

In Table 3 below, the performance results are summarized for the simple heuristic algorithm and this 'improved' heuristic algorithm. Due to the improved performance of these informed searches over the blind searches tested earlier, more complex mazes can be used. For these test cases, 100 three-dimensional outer-layer mazes measuring 5 cells on each side were used. The same mazes were used for both algorithms. Again, the control scheme utilized 8 possible orientations, each corresponding to one corner of the maze being oriented upward.

100 5x5x5 Outer-Layer Mazes *Average Length of Theoretical Solution* *from Starting Cell to Target Cell = 18.4 Cells*					
Search Algorithm Employed	Avg. Num. Moves	% Frozen	% Unsolvable	Num. Repeated Moves	Num. Unrepeatable Moves
Simple Heuristic	74.3	30	9	1037	41
Improved Heuristic	20.3	26	10	221	7

Table 3: Heuristic Search Algorithm Comparison for 5x5x5 Outer-Layer Mazes with 8 Orientations

As we can see, incorporating knowledge of the next cell that will be visited as a result of a given move dramatically improved the performance of the algorithm. The average number of moves required to solve the maze was less than a third of the number required using the simpler heuristic. The number of repeated moves was reduced to less than a forth the number previously required.

At first it seems surprising that simply knowing the next cell that the marble will occupy, with no knowledge of where it will eventually stop, could improve performance this much. Perhaps a more intuitive explanation for this improvement comes from realizing that the next adjacent cell visited not only gives a (possibly temporary) location for the marble, but a direction as well.

This heuristic algorithm also improves the performance of the agent over the blind search algorithms in another more subtle way as well. Recall that the blind search methods evaluated earlier expand the agent's knowledge by finding and exploring unknown states in the physical state graph. Remember that a ‘state’ in this system is a combination of the marble’s location and orientation. As previously explained, it is necessary to distinguish between states where the location is the same but the orientation is different, since the results of a given move will be different if performed from a different orientation.

When searching for the goal by expanding our knowledge of the maze, however, we would intuitively want to explore as many new cells as possible. We would therefore

want to give priority to a state corresponding to a new cell over a state corresponding to the same cell in a different orientation. The heuristic employed in this algorithm is a measurement of the desirability of the state's cell (the marble location), and will therefore favor moves that physically relocate the marble closer to the target cell.

4.5 Single Move Prediction Search:

The next algorithm developed to improve the 'physical search' for the maze solution involves predicting the path followed by the marble in response to a move. Using the last algorithm discussed, the agent was able to improve its performance over a simple heuristic search by determining which adjacent cell the marble would pass through or come to rest in after a move. That cell can be conclusively determined due to the nature of the geometry, but we cannot be sure where the marble will travel after that next cell. We can however make a prediction of the entire path that the marble will travel, and use the theoretical distance from the predicted end cell to the target cell as a heuristic to judge the value of that move. This prediction will be made for every reachable unknown state in the physical state graph, and the move believed to be the best will be pursued. As with the previous method, a breadth-first algorithm is employed to find the shortest path (the fewest physical moves) to reach each unknown state.

To make the prediction of the entire path that will be followed by the marble from a given state in response to a specific move (rotation to another orientation), a recursive algorithm was developed. For simplicity, the algorithm is described here as it applies to two dimensional mazes, with the required adjustments for three dimensional mazes discussed later.

As discussed previously, we can determine the first adjacent cell that the marble will move to in response to a given move. From this cell one of two events can occur. The marble can continue moving in the direction of its current velocity, referred to here as the 'velocity-dominate' direction, or it can fall in the other 'gravity-dominate' direction. The vector representing the direction of gravity after the next move is projected onto the axes of the maze's coordinated system, and the component that is parallel to the velocity vector is eliminated to determine the gravity-dominate direction. We can then query the geometry of the maze to determine if movement in either or both of the velocity-dominate and gravity-dominate directions is possible, given the walls that are either present or absent.

If movement is not possible in either direction, we can safely assume that the marble will come to rest in this cell. If movement in only one of the two directions is possible, we will assume that it will travel in that direction. If however, movement in both directions is possible, we will use an estimate of the marble's speed to determine which direction will be taken. If the estimated speed of the marble is above a specified threshold value, we will assume that the marble will continue in the velocity-dominate direction. If the speed is below that threshold, we will assume that the marble will 'fall' in the gravity-dominate direction. Intuitively, we would expect this to make sense. The faster the marble is traveling, the more likely it is that it would pass over 'holes' without falling in.

After determining which cell the marble will move to next, we adjust the speed and direction of the marble and apply the procedure recursively until we reach a cell where no further progress is possible, which is the predicted final location for the marble in response to the move. At each step of the algorithm, the new direction is simply a vector connecting the current cell to the next predicted cell. The predicted speed is adjusted as follows: if the direction of the marble has changed (from the velocity-dominate to gravity-dominate direction, the speed is reset to zero, assuming that a collision with a wall has probably occurred impeding the forward progress. If however the marble continues in the velocity-dominate direction, the predicted speed is increased by an amount proportional to the magnitude of the projection of the gravity vector onto the velocity vector. This will increase the speed more when it is rolling down steeper hills. This new speed is then used in the next iteration of the algorithm (for the next cell), being compared to the set threshold value to determine how likely it is that the marble will continue in the velocity-dominate direction.

This recursive path prediction method can be summarized as follows:

getPredictedPath(Cell, Speed, Velocity Vector, Orientation, Path):

1. Determine the gravity-dominate and velocity-dominate directions based on the Orientation and Velocity Vector.
2. If movement in only one of these directions is possible,
   2.1. Add the next Cell in that direction to the Path
   2.2. Adjust the Velocity Vector to point in the direction of the next Cell
   2.3. Increase the 'Speed' if the 'Velocity Vector' did not change
   2.4. call getPredictedPath() with the new Cell, Speed, and Velocity Vector to add any further progress to the Path
3. If movement in both directions is possible
   3.1. Compare the speed against the threshold value
      3.1.1. if Speed >= threshold value, the next Cell will be in the velocity-dominate direction
      3.1.2. if Speed < threshold value, the next Cell will be in the gravity-dominate direction
   3.2. Adjust the Velocity Vector to point in the direction of the next Cell
   3.3. Increase the Speed if the Velocity Vector did not change
   3.4. call getPredictedPath() with the new Cell, Speed, and Velocity Vector to add any further progress to the Path
4. If movement in neither the gravity-dominate or velocity-dominate directions is possible, return the predicted path and final location (cell) of marble.

The amount by which the speed is increased in this method when the velocity vector remains the same is essentially arbitrary when the walls are all at the same angle with respect to gravity, as is the case for the test cases discusses so far in this thesis. Instead of using the projection of the gravity vector onto the velocity vector, we could simply increase the speed by a set amount, and adjust the threshold value (used to determine if the velocity-dominate direction is chosen) to a value that works most of the time. For later tests however, alignments are chosen such that the walls are not all at the same

angle, and thus we must account for the 'steepness' of the walls involved. The projection of the gravity vector onto the velocity vector is straightforward to compute and handles all cases.

Figures 32 and 33 are used to illustrate how this algorithm works in the two-dimensional case.

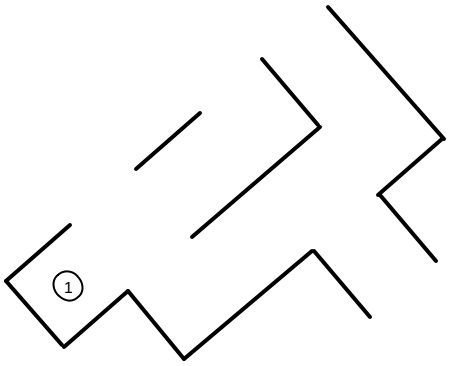

Figure 32: Initial State

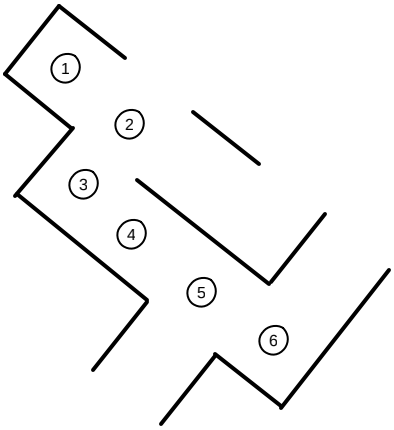

Figure 33: Predicted Path

The marble is initially at rest in position 1 as shown in Figure 32. The move resulting in the orientation shown in Figure 33 initially causes the marble to move to position 2. At this time, the speed is not sufficient to continue moving in the velocity dominate direction, so the marble 'falls' in the gravity-dominate direction to position 3. Movement from position 3 in the new velocity-dominate direction is blocked by a wall, so the marble moves in the gravity-dominate direction to position 4. From position 4, the marble can only continue its current velocity-dominate direction to position 5, increasing its speed as it does so. From position 5, the speed of the marble is sufficient (above the threshold value) to continue in the velocity-dominate direction to position 6. At position 6, the marble will come to rest in a well defined state, since further progress is not possible in either the velocity or gravity-dominate directions.

Figure 34 below shows the path followed in the actual simulation of the example used in Section 3.4 '*Analysis of the Physical Solution*'. It can be easily seen that a collision occurred at most locations where the marble's path changed. Looking at the path that is followed when one of these collisions occurs, we can see that the marble changes directions. Intuitively, we can assume that the marble's speed would be reduced when one of these collisions and resulting change of direction occurs. The getPredictedPath() path prediction method (which was able to predict this entire solution accurately) simplifies the response to these collisions by merely changing the velocity vector to point to the next cell and re-setting the speed to zero.

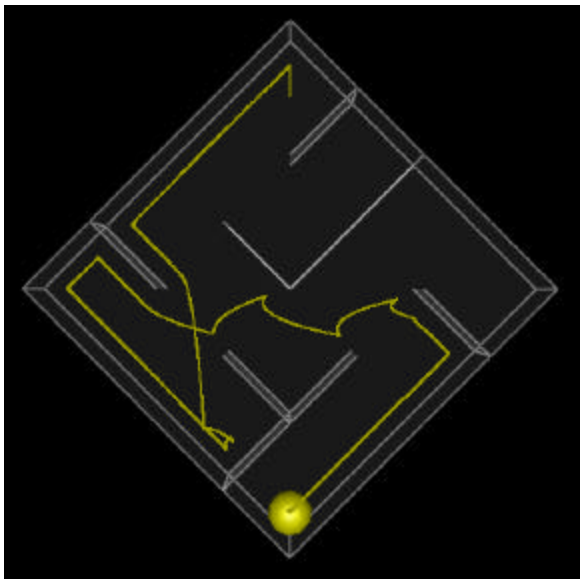

Figure 34: Example of Simulated Solution Path

The three-dimensional case is similar, with the added complexity of three possible directions of movement at each step, as determined by the projection of the gravity vector onto the three axes. One of these vectors will be the velocity-dominate direction, and the two others will be gravity-dominate. There is one particularly interesting scenario in the three-dimensional case. This occurs when it is determined that progress in the velocity-dominate direction is not the predicted outcome, and movement in both gravity-dominate directions is possible and each is determined to be equally likely. This is the case that is pictured in Figure 25 in Section 3.6 '*Additional Challenges*'. For now, one of the gravity-

dominate directions is chosen arbitrarily, but later in this thesis a method for avoiding this problem is developed.

Notice that this algorithm is designed to provide an extremely fast (instantaneous for the purposes of this system) prediction of the response of the marble to a given move, without having to rely on simulation. As such, it does not attempt to model the physics with a high degree of accuracy. As stated earlier, one of the premises on which this thesis is based is that we cannot be certain of the results of a move until the move is 'physically' carried out (simulated). This predictive algorithm does however involve (on an extremely simplified level) some of the fundamental components of a physics simulation. Collision detection takes the form of determining which directions of movement are possible, given the walls that are present. The collision response is simplified into modifying the speed and direction of travel. Acceleration due to the force of gravity is also estimated when adjusting the speed at each step.

The similarity to a physics simulation is not however to simulate reality, but is merely the result of attempting to make the prediction feasible. The primary purpose is to quickly provide a prediction of the state resulting from a move, in the hope that it will be close enough to serve as a reasonable basis for the decision making. This extreme simplification of reality is in fact shown to be over 97% accurate in the test cases used for this thesis. Naturally, more realism could be incorporated into the prediction, but at some point it would become too complex to provide the 'instantaneous' predictions that are desirable for this algorithm. Consideration must also be given to the fact that the simulation itself is not always repeatable, and therefore no prediction algorithm can be guaranteed to be 100% accurate.

It is interesting that the objective of this algorithm is essentially to provide the intelligent agent with the ability to 'guess' what the outcome of a move will be before making it. This is similar to the process a human agent would use: evaluating his or her options by predicting what the outcomes are likely to be.

After the recursive path prediction method is used to estimate the cell corresponding to an unknown state in the physical state graph, the length of the theoretical path from that cell to the target cell is used as a heuristic to determine the desirability of the unknown state.

This 'Single Move Prediction Search' algorithm, which incorporates this recursive path prediction methodology, can be summarized as follows:

1. For each unknown state in the physical state graph:
    1.1. Find a path from the marble's current state to the node corresponding to the unknown state using a breadth-first search.
    1.2. Determine the move corresponding to the branch from the parent state to the unknown state.
    1.3. Use the recursive path prediction algorithm discussed previously to estimate the cell the marble will occupy as a result of the move to this unknown state.
    1.4. Set the heuristic value equal to the theoretical path length from this predicted cell to the target cell.
    1.5. If this heuristic value is better than the heuristic value for any other unknown state previously evaluated, designate this as the 'best unknown state' and store the path leading to it.
2. Perform the physical moves corresponding to the path to the 'best unknown state'.

In Table 4 below, the performance of the Single Move Prediction algorithm is compared to the results of the previous heuristic algorithms. The same 100 5x5x5 outer-layer mazes were used for the testing, with a control scheme utilizing the same 8 orientations.

100 5x5x5 Outer-Layer Mazes *Average Length of Theoretical Solution* *from Starting Cell to Target Cell = 18.4 Cells*					
Search Algorithm Employed	Avg. Num. Moves	% Frozen	% Unsolvable	Num. Repeated Moves	Num. Unrepeatable Moves
Simple Heuristic	74.3	30	9	1037	41
Improved Heuristic	20.3	26	10	221	7
Single Move Prediction	18.6	27	11	117	0

Table 4: Single Move Prediction Results for 5x5x5 Outer-Layer Mazes Using 8 Orientations

The data in Table 4 shows that there is a slight improvement over the 'Improved Heuristic' covered in the previous section, but perhaps not as significant as we would expect. One possible explanation for this lack of improvement is that the path prediction is not accurate enough. The data generated for these test cases shows however that 97.7% of the 1152 moves that were required to solve the 62 solvable mazes were predicted accurately. It is reasonable however to assume that any prediction error could result in the marble deviating far from the intended state, and that such an error could be difficult to recover from. It is also obvious that given the large number of moves required to run all cases, even with a prediction algorithm that is nearly 98% accurate, several of the test cases will be affected by prediction errors. The prediction errors that did occur affected only 20 of the mazes solved. The average number of moves required to solve the 42 mazes that did not incur prediction errors was only 14.5 moves.

4.6 Complete Solution Prediction Search

In the 'improved heuristic' algorithm, a computationally efficient method was developed to determine the next cell that the marble will enter in response to a move of the maze. The 'single move prediction' algorithm was able to determine a more optimal physical solution by extending this prediction to estimate the entire path that the marble will travel as part of the next move. The algorithm described in this section attempts to improve the performance of the agent solving this puzzle by predicting a complete physical solution path to the target cell.

One method for determining the complete path that would be followed by the marble is to actually run the simulation in the background before each move is made, without changing the actual state of the system until the results of the simulation are complete, and we've decided to accept the results. This would turn the problem into a more traditional artificial intelligence search problem. The ability to effectively backtrack (since no physical move is actually being made) would enable us to expand branches of the search tree as desired, and prioritize the resulting nodes using one of the more common search algorithms.

In effect, this methodology would be the equivalent to the depth-first search algorithm used to determine the theoretical solution to the maze. For that solution, a 'behind the scenes' depth-first search was employed to find the path, with the assumption that at each node, any direction could be followed that does not penetrate any walls. Here, any common search algorithm could be used to find the physical solution, with the assumption that at each node, the direction indicated by the simulation would be followed.

This methodology is not entirely impractical. With the proper configuration of the Newton physics simulation, and without slowing the system down to update the graphics, we can run the simulation faster than real-time. However, it would still not be nearly fast enough to provide seemingly 'instantaneous' decisions, especially when we consider that

it would involve simulating numerous moves, many if not most of which would not be on the path to the solution, to search the state space for the solution.

This may be a more practical solution in the future, as computing power continues to grow exponentially and enables the entire simulation to be performed quickly enough. It is also possible that new advances in computer hardware, especially physics co-processors, will do for simulation what GPUs have done for graphics, and make behind-the-scenes simulations like this one a potential component of AI algorithms. This however is not the focus of the algorithms developed for this thesis. Here the simulation is being treated as a real-time representation of an actual physical domain, and the intelligent agent developed must deal with it as if it were reality.

The 'Complete Solution Prediction' algorithm developed here is based on repeated use of the 'Single Move Prediction' algorithm covered in the previous section. That method is applied to predict the resultant state of each potential move. Each prediction is assumed to be correct, and a new prediction can be made for each potential move from the new state. Since all possible moves can be analyzed, and backtracking can be employed (since no actual physical moves are being performed), a traditional search algorithm can be employed to find a sequence of moves that should lead all the way to the target cell.

This method works by building and searching a different graph, the 'Predicted State Graph' for a state corresponding to the target cell. The theory behind this method is nearly identical to the Blind Search Algorithms employed on the physical state graph. The predicted state graph has the same form and properties as the physical state graph. The only difference is that the Complete Solution Prediction algorithm expands the graph with the predicted results for potential moves, rather than the results of simulating the physical moves. This algorithm can therefore build and search the entire graph, employing backtracking as required, until an entire path to the goal is found.

Essentially, this algorithm is doing the same thing as the background simulation theorized above. This method is more practical however, because unlike a complete off-line

simulation, the Complete Solution Prediction algorithm is fast enough that it can be applied numerous times to develop a complete solution before a single move is decided upon.

The search algorithm employed to search the predicted state graph for a solution is the breadth-first search, again modified to eliminate cycles (which will appear in the predicted state graph, just as they do in the physical state graph). The breadth-first search will find a predicted path with the fewest possible number of physical moves. This is important, not only because it will improve the performance of our agent, but because it will improve reliability as well. Since each move is based on predicted results, each is subject to inaccuracies in the prediction algorithm. Minimizing the number of moves required will reduce the chances of the marble deviating from the predicted path.

This is actually the first of the algorithms developed for this thesis that attempts to find an optimal physical solution. If the results of every possible move are predicted accurately, and the path with the fewest number of moves is chosen, this should result in the optimal path being followed. It turns out that complete optimality is not possible (discussed below), but the accuracy should be good enough to improve performance.

Unlike the previous algorithms developed, which are based on finding and 'exploring' states that were previously unknown in the physical state graph, this algorithm is based on predicting and following a complete path from the marble's current location to the target cell. This fundamental difference has three significant implications that must be dealt with in the implementation.

1. The algorithm may not be able to find a complete solution to a target state.

   For the cases in which a predicted path leading from the current state to a target state cannot be found, there are two choices: the maze can be dismissed as unsolvable, or we can employ a different algorithm to search for the target. Although the Single Move Prediction algorithm developed for this thesis has

proven to be over 97% accurate when applied to the test cases used, there is still the possibility of prediction errors occurring. The number of moves required to find a complete path to the solution increases the chances of an error occurring. This can and does result in predicted solutions not being found for many mazes that are in fact solvable.

The solution implemented in this system is to utilize the Single Move Prediction algorithm to find and explore the best unknown state in the Physical State graph whenever a complete solution cannot be found in the predicted state graph. For this reason and others (discussed below) the physical state graph is built and maintained with the results of the simulated moves, just as it is with the previous algorithms. After making any move based on the single move prediction, the agent will always attempt to find a complete predicted path to the solution before resorting to the single move prediction algorithm again.

2. The algorithm finds a predicted solution that has inaccuracies, and therefore does not lead to the target.

   Each move in the predicted solution path has the possibility of leading to an unpredicted state. After such a deviation, it is unlikely that the subsequent predicted moves will lead to the target. The way this is dealt with is for the agent to search for a new complete path to the solution whenever it deviates from the predicted path. If a complete predicted path from the new state cannot be found, the agent will use the single move prediction algorithm as described above.

3. The predicted path leads back to the current state or a previous state.

   If an error in a prediction results in the system returning to a previous state, the same predictions will be made again, most likely ending up with the same results. This would create an infinite loop in the physical simulation. The solution is to build and maintain the physical state graph with the results from the simulated

moves. Whenever the results of a move are to be predicted, the physical state graph is checked first and the previous results of the move are used if available. In this way, the agent effectively learns from its mistakes and past experience.

In Table 5 below, the performance results are summarized for the Single Move Prediction and Complete Solution Prediction algorithms. For these test cases, the same 100 5x5x5 outer-layer mazes were used to test both algorithms. The control scheme was the same as well, utilizing 8 possible orientations, each corresponding to one corner of the maze being oriented upward.

100 5x5x5 Outer-Layer Mazes *Average Length of Theoretical Solution from Starting Cell to Target Cell = 18.4 Cells*					
Search Algorithm Employed	Avg. Num. Moves	% Frozen	% Unsolvable	Num. Repeated Moves	Num. Unrepeatable Moves
Single Move Prediction	18.6	27	11	117	0
Complete Solution Prediction	21	25	7	236	6

Table 5: Comparison of Prediction-Based Algorithms for 5x5x5 Outer-Layer Mazes with 8 Orientations

Here we see that the result of attempting to predict an entire path to the target cell is to actually diminish performance (though not significantly). There are two primary effects that contribute to the lack of improvement. Firstly, the number of predicted paths which are part of the complete solution increases the odds of an inaccurate prediction being made. This has the effect of reducing the algorithm to multiple single move predictions. Secondly, for many of the mazes, a complete predicted solution from the start state to the goal state could not be found, so much of the physical solution was actually found using the single move prediction algorithm.

For the 39 mazes in which complete paths to the target cell were successfully predicted, the average number of moves was only 8.9. It is tempting to assume that an optimal

solution for these mazes (given this control scheme) was indeed found, since the algorithm found the fewest number of predicted moves to reach the target cell, and all of the predictions were correct. This however is not necessarily the case, since there is always the possibility that another path, which was predicted incorrectly by this algorithm, could have led to the target with fewer moves.

The above argument has some interesting implications. Since the simulation of this problem is not always repeatable, no path prediction can be guaranteed to be 100% accurate. By this argument, no predictive algorithm can be guaranteed to find the optimal physical solution. In essence, unlike the search of a graph of known states, the dependence of the physical solution on the simulation has eliminated the possibility of guaranteed optimality.

4.7 Improved Control Scheme

As previously explained, the tests performed thus far on three-dimensional mazes have been using a control scheme with 8 possible alignments of the cubic maze, each with one vertex of the maze pointing straight up. This has several advantages that are discussed later in Section 6.5 *'Simulation Controller'*, but it has a few significant disadvantages as well.

Since all the walls of the maze in these alignments will be at the same angle with respect to the gravitational vector, there are many occasions when the marble can take more than one path with equal probability. Figure 25 in Section 3.6 '*Additional Challenges*' showed an example of how this can happen. This causes three major problems for this system.

- The marble could take one path as a response to a move, and take another path (due to minor roundoff errors and other effects) the next time the same move is made. This increases the chances of non-repeatable moves occurring.

- The equal likelihood of more than one path being taken severely impacts the ability of the agent to predict which path will be followed.

- There exists the possibility that the marble will roll down balanced on the edge between the two walls, and come to rest in an undefined state.

To deal with these issues, a more complex set of possible alignments of the maze was developed. The alignments were carefully chosen such that the projection of the gravitational vector onto each of the three axes would each have a different magnitude. This should generally result in the marble being more likely to travel in one direction than in the others.

The strategy for choosing additional the possible orientations was to move the previous points corresponding to the orientations (the vertices of the maze) slightly away from

each vertex, a different distance along each axis. Figure 35 shows a 2 dimensional view of the maze in one of the previous alignments, with the vector indicating that the vertex is aligned at the top. Figure 36 shows the new points corresponding to the new alignments. Each of these points represents a point that will be straight up in one of the possible orientations, as shown by the vector in the Figure.

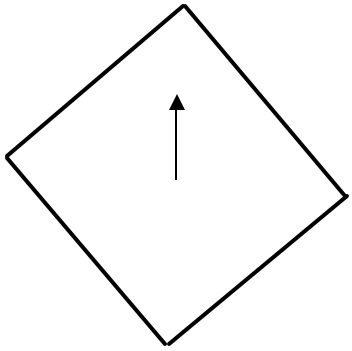

Figure 35: Old Alignments on Vertices

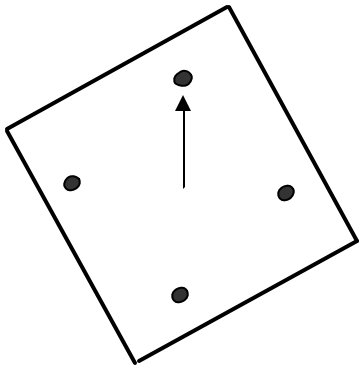

Figure 36: New Alignment Points

When this procedure is applied to the six sides of a cubic maze, there are 4 points on each side of the cube corresponding to possible alignments, for a total of 24 alignments. Figure 37 shows the cube (outlined in blue) and the new alignment points (the intersections of the white lines). The lines connecting the alignment points indicate the possible moves that can be made by rotating the maze from one alignment to another. The close-up view in Figure 38 shows that for each alignment, there are now 4 adjacent alignments, corresponding to 4 moves that can be made. Two of the moves will be to the other two points that are nearest to it, and 2 will be to the closest points near adjacent corners of the maze.

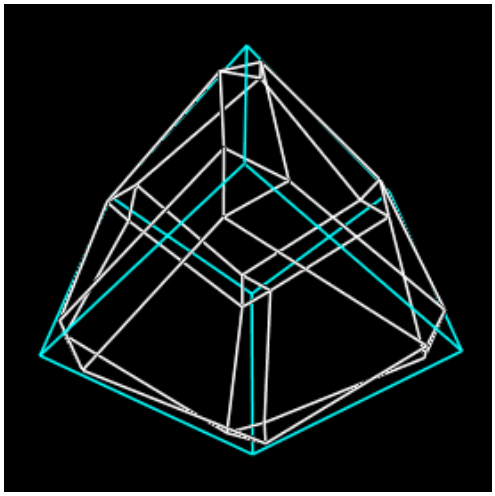

Figure 37: New Alignments and Moves, Cube Outlined in Blue

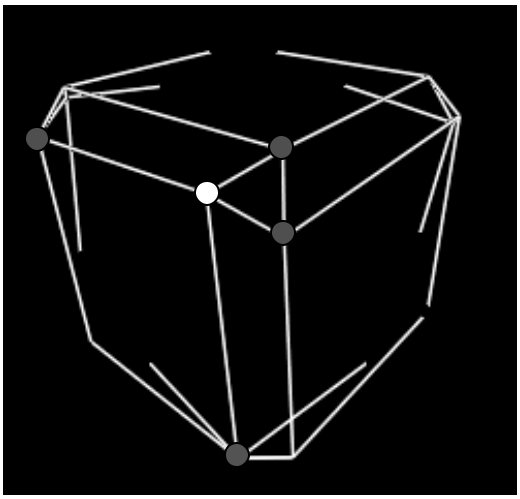

Figure 38: Adjacent Alignment Points

Since each alignment now has four adjacent alignments, there are now 4 moves that can be performed by the agent at any time. This means that the physical state graph now has a branching factor of 4 instead of 3. Also, since the states themselves consist of the marble location and maze alignment, we now have 3 times as many possible states. The hope is that the additional level of control and predictability of the movement will outweigh the additional complexity of the physical state graph.

One of the drawbacks to this scheme is that there are now three times as many states to be learned, i.e. added to the physical state graph. This could have a significant negative impact on the performance of the agent. This negative impact can be reduced somewhat by taking advantage of the geometry of the new alignment scheme. Looking at Figure 39, we can see that each vertex of the cubic maze has 3 alignments in close proximity to it. If the marble is at rest in one of these three alignments, it is reasonable to assume it will be at rest in the same cell of the maze when a move is made to either of the other two. Technically, this is due the fact that the projections of the gravitational vector onto the coordinate axes in each of these three alignments will be in the same directions.

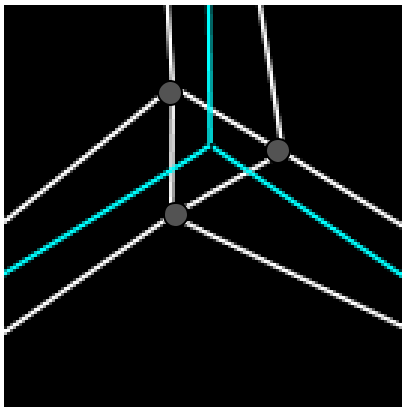

Figure 39: New Alignments Near Maze Vertex,
Cube Outlined in Blue

This means that whenever a state is encountered in one of these three ‘close’ alignments, we can predict with certainty which state will result from a move to the other two alignments in that corner of the maze, and can therefore update the physical state graph with that information without having to simulate the physical move. Essentially, we are giving the agent the ability to see and assume the results of ‘obvious’ moves, without having to waste time trying them out. These additional states will still have an impact on performance, since they are states that will likely have to be physically moved through to get to other states. But no time will be expended learning them initially.

In Table 6 below, the performance results are summarized for the Complete Solution Prediction Algorithm using both the previous 8 orientations and the new control scheme with 24 orientations. Again, the same 100 5x5x5 outer-layer mazes were used to test both algorithms.

100 5x5x5 Outer-Layer Mazes *Average Length of Theoretical Solution* *from Starting Cell to Target Cell = 18.4 Cells*					
Search Algorithm Employed	Avg. Num. Moves	% Frozen	% Unsolvable	Num. Repeated Moves	Num. Unrepeatable Moves
Complete Solution Prediction with 8 Orientations	21	25	7	236	6
Complete Solution Prediction with 24 Orientations	57	7	4	2354	30

Table 6: Solution Prediction Algorithm for 5x5x5 Outer-Layer Mazes Using 8 and 24 Orientations

As we can see from the data, the number of moves required to solve each maze actually increased. This is due to the increased number of possible orientations, which equate to states that must be moved through on the path to the solution. The reliability of the algorithm however was improved significantly. Even though the additional moves create increased opportunities for 'frozen' states to occur, the number of such states was reduced to about a fourth of what was typically experienced using only eight orientations. The number of unsolvable mazes was also reduced to 4, perhaps due the additional control provided by the additional orientations. The reductions in frozen states and unsolvable mazes resulted in 89 of the 100 mazes being solved, as opposed to the 68 which could be solved using 8 orientations.

Unfortunately, the accuracy of the path prediction algorithm was not improved with the additional orientations, and the additional opportunities for errors (from the increased number of moves required) resulted in only 17 of the mazes being completely solved with no prediction errors. These 17 mazes were however solved with an average of only 9.3 moves.

One significant disadvantage to this control scheme occurs when the maze is not physically solvable with any sequence of moves. The algorithms developed for this thesis do not ‘give up’ on finding a solution as long as there are unknown states in the physical state graph. This control scheme therefore takes significantly longer to detect that a maze is unsolvable, since there are now three times as many possible states. Furthermore, since there are now four times as many orientations, it can take many more physical moves to get to each unknown state to be explored. Although the results of many of these moves can be predicted, they still represent states that the marble must move through to get to the unknown states. In the worst case, the agent took almost one hour to perform over 1400 moves in order to determine that the maze was unsolvable.

5. Final Analysis and Comparison of Results

In order to summarize the results, and evaluate all the methods used in this thesis, each of the solution algorithms was used to solve the same mazes. Table 7 below shows the results of testing the methods on the same 100 outer-layer mazes measuring 5 cells along each edge.

100 5x5x5 Outer-Layer Mazes *Average Length of Theoretical Solution from Starting Cell to Target Cell = 18.4 Cells*						
Search Algorithm Employed	Avg. Num. Moves	% Solved	% Unsolvable	% Frozen	Num. Repeated Moves	Num. Unrepeatable Moves
Depth-First	181.1	20	4	76	3194	48
Breadth-First	114.6	34	15	51	687	10
Simple Heuristic	74.3	61	9	30	1037	41
Improved Heuristic	20.3	64	10	26	221	7
Single Move Prediction	18.6	62	11	27	117	0
Complete Solution Prediction – 8 Orientations	21	68	7	25	236	6
Complete Solution Prediction – 24 Orientations	57	89	4	7	2354	30

Table 7: Search Algorithm Comparison for 5X5X5 Outer-Layer Mazes

As one would expect, the blind search methods performed poorly. The depth-first search algorithm took over 24 hours to run on all 100 mazes, and could solve only 20% of them (100 total mazes minus 4 unsolvable and 76 frozen). The Single-Move prediction algorithm was able to solve mazes with the fewest moves, but the Complete Solution Prediction algorithm using 24 orientations was the most reliable, solving 89% of the mazes tested (100 total mazes minus 4 unsolvable and 7 frozen).

To compare the performance of the algorithms on less challenging mazes, each method was applied to the same 100 3x3x3 all-layer mazes that were used to initially test the blind search algorithms. The results are summarized in Table 8 below.

100 3x3x3 All-Layer Mazes *Average Length of Theoretical Solution* *from Starting Cell to Target Cell = 8.5 Cells*						
Search Algorithm Employed	Avg. Num. Moves	% Solved	% Unsolvable	% Frozen	Num. Repeated Moves	Num. Unrepeatable Moves
Depth-First	96.8	66	1	33	4013	17
Breadth-First	52	87	2	11	610	4
Simple Heuristic	31.5	89	2	9	509	5
Improved Heuristic	10.5	88	1	11	117	1
Single Move Prediction	8.8	93	3	4	43	0
Complete Solution Prediction – 8 Orientations	6.8	95	3	2	40	2
Complete Solution Prediction – 24 Orientations	8.9	98	1	1	231	0

Table 8: Search Algorithm Comparison for 3X3X3 All-Layer Mazes

Notice that unlike the case for the larger mazes, for these smaller mazes, the algorithms utilizing path prediction all perform better than the 'Improved Heuristic' method. This is due in part to the shorter paths involved in smaller mazes, leaving less opportunity for the marble to stray from the predicted path. Comparing the final algorithm developed (Complete Solution Prediction with 24 Orientations) to the first (Blind Depth-First), we see that the additional intelligence incorporated into the Knowledge Engine was able to improve its performance dramatically. The average number of moves required to solve a maze was reduced by over 90%. The reliability, as measured by the percentage of mazes

that could be physically solved, was increased from 66% to 98%. Repeatability, the percentage of repeated moves that ended with the same results was improved to 100%.

6. Implementation

In this section we discuss the software architecture developed to create the maze, perform the simulation, and test the algorithms developed for this thesis.

6.1 Software Components

There are six primary software components implementing the ideas presented in this thesis. They function together to find and simulate the physical solution to the maze. They are summarized here and described in detail in later sections.

- **Computer Model:** the data structures, naming conventions, and methods used to store and access the physical definition of the components (primarily the maze and marble) used in this simulation.

- **Maze Generator:** the logic and code necessary to generate the geometry of the 3 dimensional maze, through which the marble is to be moved from the starting cell to the target cell. Section 3.2 '*Maze Generation*' discusses the algorithm implemented by the Maze Generator.

- **Physics Engine:** the models of the maze and marble geometry used by the physics engine, as well as the simulation of physical moves of the maze and the responses to those moves.

- **Knowledge Engine:** the artificially intelligent agent responsible for determining a physical solution to the maze given the geometry of the maze and feedback from the Physics Engine.

- **Simulation Controller:** the component responsible for coordinating the simulation and interfacing the Physics Engine with the Knowledge Engine.

- **Rendering Engine:** the logic and code responsible for providing the three dimensional graphical display and animation of the results of the simulation.

The first of these components, the Computer Model, defines how the physical objects of the system will be represented, and therefore determines the protocol by which all the other elements of the system will access the geometry. The second component, the Maze Generator, is used only once at the beginning of each run of the system to define the random three dimensional maze to be solved.

The geometry defined by the Maze Generator and stored in the Computer Model is provided to the Physics Engine in a format it can interface with. The Physics Engine performs the physical simulation of actions (rotations of the maze) to determine the response to those actions.

The Knowledge Engine represents the artificial intelligence of the system, which encapsulates the methodology for solving the maze. Given the geometry of the maze itself and the results of past actions, it determines the next actions to be performed in order to move the marble from its current location to the target cell.

The diagram in Figure 40 below shows a high level view of the last 4 components and how they must interface with each other to perform the simulation once the geometry is defined. The key component integrating the knowledge engine and the physics engine is the Simulation Controller. The Simulation Controller interfaces with the Physics Engine to request the required actions (moves), and to receive the results of those actions from it. The Simulation Controller then provides those results to the Knowledge Engine, which uses them as input to determine future actions. Finally, The Rendering Engine continuously integrates the results from the physics simulation with the definition of the geometry and displays it in a format that can be understood by the viewer.

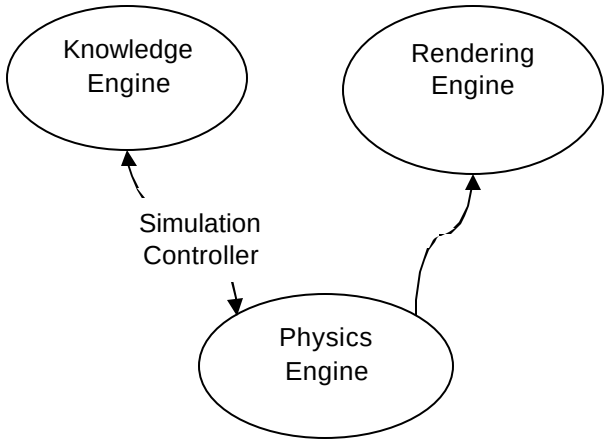

Figure 40: High-Level System Architecture

6.2 Computer Model

In order for the software components to efficiently store and access the description of the randomly generated maze, consistent and effective conventions for referencing the maze's individual components (cells and walls) were developed. The following are the conventions adopted for this system:

- **Maze Coordinates**:
  The coordinate system for the maze itself starts at the origin and is scaled at one unit for the width of each cell. A three dimensional cubic maze of size N will therefore occupy the space (in the maze coordinate system) from (0,0,0) to (N,N,N). This scaling enables the simple computation of the marble's location in the maze's coordinate system: `Cell Number = (((z-1)*(mazeWidth*mazeHeight)) + ((y-1) * mazeWidth) + x) + 1.`

- **Cell Numbering**:
  The cells are numbered starting with cell 1 at the origin and increasing by increments of one per cell along the positive X, Y, and Z axes, in that order. The decision to make the cell numbering one-based instead of zero-based was made in order to make debugging and development more intuitive, but the methods and

data structures affected could easily be changed to a zero-based convention. Cell number j of cubic maze of size N would therefore be located (with its lower left back corner) at x = ((j-1) % N^2) % N, y = (((j-1) % N^2) / N), and z = (j-1) / N^2. (Here '%' is the modulus operator)

- **Wall Numbering**:

  The walls of the maze are numbered starting at cell 1; the wall in the positive x direction is numbered 0; the wall in the positive y direction is numbered 1; the wall in the positive z direction is wall number 2. The count continues in the next cell (cell number 2) with walls 3, 4, and 5 in the positive x, y, and z directions respectively. In general, the walls for cell number j in the positive x, y, and z directions will be (3*j-2), (3*j-1), and (3*j). The walls in the negative x, y, and z directions are easily handled recursively by obtaining the wall in the positive x, y, or z direction for the cell on the other side of the wall (assuming there is a cell in the maze on the other side of the wall).

  This convention does number the outer-most walls in the positive x, y, and z directions, but does not assign numbers to the outer-most walls in the negative x, y, and z directions. This does not create any problems. The primary reason for numbering the walls is to keep track of which internal walls have been removed (as part of the maze generation), and the outer walls are always assumed to be present. This also means that the outer walls in the positive x, y, and z directions are being numbered unnecessarily, but doing so enables the use of the above numbering conventions, and facilitates the implementation of simple formulas to calculate the wall numbers for a given cell of the maze, and the cells on each side of a given wall.

Given these feature numbering conventions, functions were developed to enable the software components of this system to easily access the information they need. A few of the critical examples include:

- The component that creates the random maze (Maze Generator) will need to know which walls are internal walls that can be removed. It will also need to know which cells of the maze are separated by each wall.

- The agent attempting to find a physical solution to the maze (moving the marble to the target cell) often needs to determine if there is a wall in a specific direction from a given cell.

- The Rendering Engine will need to know the coordinates of the vertices of each wall of the maze in order to display it in the proper location.

6.3 Physics Engine

Given the importance of the physics simulation to this thesis, and to enable the focus of this thesis to be on the solution methodology, we chose to use a third party physics engine to perform the actual simulation. The physics engine chosen was Newton Game Dynamics. 'Newton' is a no-cost physics engine that is capable of modeling the physical phenomena that is relevant to this system (material properties, linear and angular momentum, static and kinetic friction, accurate collision detection and response). It is also a 'real-time' physics engine which is also important to this project. Unlike 'high-precision' physics engines, which sacrifice speed in favor of maximum accuracy, Newton is capable of performing the real-time simulation required for the graphical display. The speed of the simulation is also important when solving numerous test cases to determine the effectiveness of a given algorithm. Even running in real-time, the algorithms in this thesis required hours to test on a reasonable number of mazes.

In order for the agent to make intelligent decisions based on the outcome of each of its actions, it is imperative that the physical simulation is capable of providing reasonably accurate results and feeding them back to the agent. To ensure a visually believable simulation, and to serve as a fair basis for the agent to act on, the physics simulation must

at the very least implement accurate collision detection and response, acceleration due to gravity, and friction with the walls of the maze. It is not however necessary to implement a physics model that attempts to be extremely precise. Many details that would be considered important if we were performing a strict physics simulation can be reasonably ignored. Examples of such details would include wind resistance, the rate at which impulses travel through objects during a collision, etc.

What is important is that the simulation be repeatable. Performing a given action, such as a rotation of the maze, from a given state (the current location of the marble) should lead to the same results (the new location of the marble) each time the action is performed. If this were not the case, the agent would not be able to learn from its experience, which is a critical component of the solution methodology. Repeatability might at first seem like a given, especially on a digital computer, but slight differences in the state that may be visually imperceptible can lead to a different response to a given action that would result in a different state. Another critical requirement is that the physics model 'follow the rules' of the game. Common errors that occur in physics simulation, such as undetected collisions enabling a solid object to pass or "tunnel" through another solid object would prevent this system from functioning as desired or possibly from functioning at all.

There are several unique aspects to this system that present a challenge when implementing a general-purpose physics engine. First of all, the geometry of the maze itself is non-trivial, with numerous concave as well as convex surfaces. Each wall cannot be modeled separately, since they are all part of a single solid object and cannot move independently. This requires a single 'collision mesh' to be used. Non-trivial meshes such as this are common in other simulations, but generally for surfaces that do not move (such as terrain or buildings). These immobile objects are generally treated as having infinite mass in the physics simulation [Jerez, 2004]. In the case of this project, the maze must be movable (so it can be rotated), and therefore must have finite mass. It must also be constrained on a system of 'hinges' that prevent it from falling.

Newton Implementation

In order to implement the Newton physics engine, the system (maze, marble, and control scheme) must be modeled in a Newton-compatible format. The maze and marble are modeled as 'NewtonBody' objects, as defined by Newton's interface. There are primarily two important properties of the NewtonBody objects that are needed by this system: a 'NewtonCollision', which is the collision mesh used by Newton to model the interactions between objects, and a transformation matrix which Newton uses to store the current orientation and location of the object.

The collision mesh for the marble is easily created using the NewtonCreateSphere() method provided by Newton. The geometry of the maze is however far more complex. Recall that the maze is defined by numerous individual walls. These walls must be given a positive thickness for the collision detection to function properly. The walls cannot however be modeled as individual rectangular prisms, since they must be connected as part of the same collision mesh. In other words, the walls must all move together as one single entity. Fortunately, Newton provides a way to create what is called a 'Compound Collision'. This enables the creation of a maze with multiple rectangular prisms, and joining them in the proper locations as part of a single collision mesh. Functionality in Newton creates the necessary constraints to force the walls to act together as a single geometric body.

The code developed for this system defines the individual walls such that they have the required thickness and are positioned in the correct locations. The walls have to be carefully overlapped to create a continuous surface while avoiding 'pits' between them. Figure 41 below shows the complete collision mesh for a sample maze, with the wall thickness exaggerated for visibility.

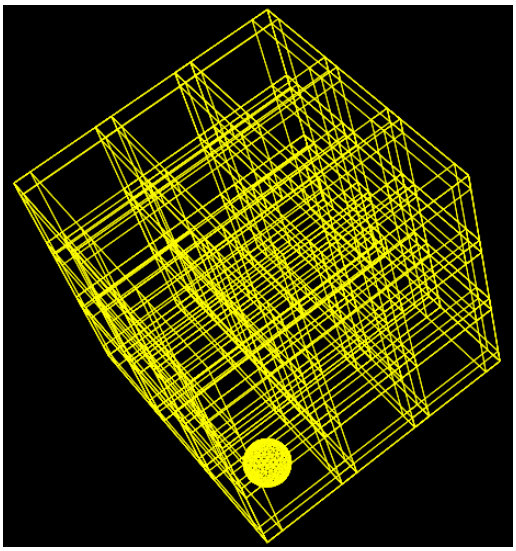

Figure 41: The Collision Mesh

Once the NewtonBody objects are specified to the Newton physics engine, the function NewtonUpdate() is called to request that the physics engine advance the simulation by a specified amount of time. The Newton engine then invokes callback methods in which this system can specify the forces that are applied to the objects. The callback function invoked for the marble simply applies a force to simulate gravity. The callback function invoked for the maze is much more complex. This is where the Simulation Controller is employed to obtain the desired moves from the Knowledge Engine and to carry out those moves by creating the proper constraints on the simulation model and applying the appropriate amount of torque to generate the desired rotation.

Newton feeds the results of the simulation step back to the calling program by invoking another function with a resultant transformation matrix for each object. This transformation matrix represents the new location and orientation of the maze and marble in Newton's global coordinate system. Our intelligent agent however requires the location of the marble to be in coordinates relative to the maze, in order to determine which cell of the maze it is in. To determine the location of the marble in maze coordinates, we need to multiply its position in global coordinates by the inverse of the maze's transformation matrix. Fortunately, the inverse of the transformation matrices used can be quickly and easily calculated [Lengyel, 2002].

Another implementation issue addressed was the compatibility of Newton transformation matrices which are in row-major order, and OpenGL (used for the graphical display of this system) which assumes column-major order. For this reason, a C++ matrix class was developed to perform the transposition as necessary.

6.4 Knowledge Engine

The purpose of the knowledge engine is to enqueue the next physical move or moves to be made in order to move the marble to the target cell. Different algorithms are used that determine the next move or moves to be made, as described in Section 4 '*Solution Algorithms*'. The input that is used for this decision is the historic knowledge of the simulation performed (in the form of the physical state graph as described previously) and varying amounts of knowledge of the maze itself and the physics involved.

As part of this thesis, several algorithms that can be invoked by the Knowledge Engine to determine the next move(s) were developed and tested. The algorithms vary in the amount of knowledge of the system that they use. The specific algorithms, and the performance resulting from the use of those algorithms have been covered in Sections 4 and 5.

6.5 Simulation Controller

The Simulation Controller serves two critical functions: It serves as the physical control mechanism, creating the required constraints and applying the appropriate forces to perform the actions requested by the Knowledge Engine, and it coordinates the activities of the Knowledge Engine and the Physics Engine to carry out the simulation.

Control of Physical Moves

The Control Scheme defines the possible actions (moves) that can be taken and the method by which they are carried out. As described previously, the moves are specifically designed to result in one of a finite set of states. These states are defined by the orientation of the maze and location of the marble in the maze. The choice of possible orientations is critical for this to work.

The orientations are chosen such that after any move, the marble will necessarily come to rest at the intersection of three walls (2 walls in two dimensional mazes), and therefore be located in a single cell of the maze which can be determined from the Physics Engine. This can be accomplished by choosing the orientations for which the projection of gravity onto the three axes is non-zero. Since the maze walls are aligned with the coordinate axes, this equates to the walls all being at an angle with respect to gravity, in which case the marble will naturally roll down the walls until it is stopped in all possible directions by walls of the maze.

For the two dimensional case the obvious choice that was implemented was to select four possible orientations in which each of the four corners is pointing straight up. This puts the walls at the steepest possible angle, resulting in the marble coming to rest in its final location as soon as possible. The analogous choice of orientations for three dimensional mazes is to have each of the eight vertices of the cubic maze pointing straight up. This choice has several advantages:

- It positions the walls such that they are at the steepest possible angle, thus leading to the marble coming to rest in its final location as quickly as possible after a move is made.

- There are only eight such orientations, the minimum possible to locate the marble in all potential resting locations.

- With a move defined in terms of rotating from a starting alignment to an ending alignment corresponding to an adjacent corner, there are only three possible moves, corresponding to a branching factor of three in the physical state graph. Figure 42 shows a vertex on the cubic maze corresponding to one possible alignment, and the three adjacent vertices that can be moved to.

- It is easy to find orientations that are adjacent to (one move away from) a given orientation. With the cubic maze centered at the origin, and the orientations defined by the Cartesian coordinates of the cube's vertices, the three adjacent vertices (and thus the three alignments that can be moved to) can be found by simply changing the sign of the X, Y, or Z coordinate.

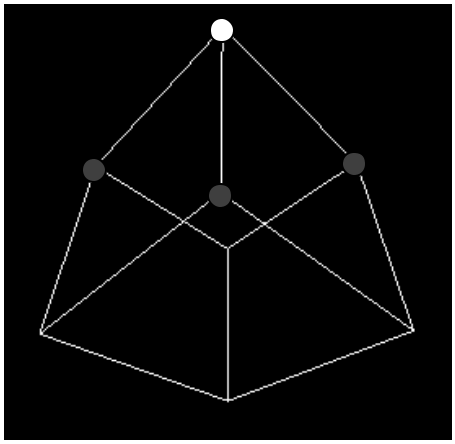

Figure 42: Adjacent Alignments

For these reasons, this choice of orientations was chosen for much of the testing performed for this thesis. There are however major drawbacks to this choice which were discussed and dealt with in Section 4.7 '*Improved Control Scheme*'.

A natural way to define and store these orientations in the implementation is as a unit vector pointing from the center of the maze to the corner of the maze that will be pointing straight up in that orientation. The orientation vectors are therefore simply the negative of

the vector representing gravity in that orientation. This provides a simple definition and implementation of the moves, or actions that can be performed on the maze by the intelligent agent. A move is simply a rotation about a single axis from one alignment to another. The axis of rotation can easily be determined by computing the cross-product of the vector representing the current alignment with the vector representing the next alignment, as covered in Section 3.5 '*Simulation of the Physical Solution*'. The required angle of rotation can also be quickly determined using the arccosine of the dot product of the two orientation vectors.

Once the axis of rotation has been determined, it can be modeled in the physics simulation using a constraint in the Newton engine referred to as a 'Hinge'. Each time the 'ForceAndTorque' callback function is invoked by Newton, an appropriate amount of torque about the axis of rotation is applied to accelerate or decelerate the rotation based on the current angle. The amount of torque required to attain the desired angular velocity is not easily calculated. The size of the maze and its geometry, which varies randomly, will affect its mass and moment of inertia [Eberly, 2004]. Since the marble itself has mass, its location in the maze will also affect the moment of inertia of the system, thus affecting the amount of torque required. If the marble is in motion, things are further complicated, especially when collisions occur between the marble and the walls of the maze. The maze's response to these collisions will often induce other rotational forces about the axis.

Fortunately, the amount of torque does not have to be computed exactly. What is important is that the torque is not so large that it 'pushes' the marble with excessive force, thus throwing off the resulting states which we would like to be primarily the result of gravity, not of other external forces. We do however need enough torque to rotate the system by the required angle, without moving too quickly when it reaches the desired alignment. The solution implemented was to make the mass of the maze walls large in comparison to the mass of the marble, so the effect of the marble on the rotation is negligible. The magnitude for the torque required to achieve the desired acceleration and deceleration was then determined experimentally for each maze size used.

After accelerating, a deceleration phase is implemented. The deceleration not only makes the animation of the move appear ‘smooth’, but also minimizes the momentum of the marble when the maze stops. If the marble was experiencing excessive motion from the movement of the maze (i.e. being pushed) when the maze was stopped, this too could have an unacceptable effect on the motion of the marble and its final location. It would also be a very unrealistic simulation if the maze, with its relatively large mass, suddenly stopped without decelerating.

Once the maze has rotated to the next desired orientation, another Newton hinge is created with a zero torque. This hinge simply acts as an additional constraint on the maze's motion, effectively serving as a ‘brake’. In addition to stopping the rotation of the maze, this additional constraint serves to keep the maze from falling when the rotation hinge is removed in preparation for the next move. Even if no gravitational force is applied to the maze itself, the marble inside of it has a gravitational force, and would therefore cause the maze to fall if it is not constantly held up.

Coordination of Knowledge Engine and Physics Engine

In addition to performing the physical moves to be simulated, the Simulation Controller provides the overall control of the system, integrating the physical simulation with the artificial intelligence. It uses feedback from the Physics Engine to maintain the history of previous moves and their results (in the form of the 'physical state graph' discussed previously). This historic data is used by the Knowledge Engine in its decision-making process. The Simulation Controller then carries out the actions chosen by the Knowledge Engine in the manner described above.

The model used to handle this coordination is a finite state machine. The possible states for the system are:

- `WAITING`
- `STABLE`
- `ACCELERATING`
- `DECELERATING`
- `SOLVED`
- `FINISHED`
- `FROZEN`

Each time an update is requested of the Newton physics engine, it invokes a callback function to set the desired forces on the objects of the system. As well as being able to effect the simulation, this callback function has access to the current status of the objects in the simulation. It has therefore been implemented as the main function of the Simulation Controller. The states, the activities associated with them, and the transitions between them are as follows:

- WAITING:

  This is the initial state of the system when it is first started, as well as the state after a move has been completed. When in this state, the Simulation Controller monitors the kinetic energy of the marble to determine when it has come to rest. The marble must remain at rest for a specified period of time, otherwise brief periods when the marble is changing direction or temporarily balanced on the edge of a 'cliff' could be falsely identified as a stable state. Once stability is detected, the condition of the marble must be tested to determine if it is actually resting in a 'valid' location, or if it is simply balanced between states, in the 'Frozen' condition covered previously. If this problem is detected, the state is changed to FROZEN to serve as an indicator that the maze is not solvable (the

methods presented in Section 4.7 '*Improved Control Scheme*' are designed to avoid this condition as much as possible). Once true stability is determined, the state is transitioned to STABLE.

- STABLE:

  This is the state in which the previous move is finally complete. The actual results of the previous move are determined from the Physics Engine and stored in the physical state graph. The location of the marble is checked to determine if it is in the target cell. If it is, the state is transitioned to SOLVED. Otherwise the queue of moves to be performed next is checked. If there are no additional moves queued up, the Knowledge Engine is invoked. It uses data from the physical state graph as well as knowledge of the geometry of the maze itself and the physics involved to determine the next move or moves. These moves are added to the queue of moves to be processed. If no new moves can be made (the maze is determined to be unsolvable), the state is transitioned to FINISHED.

  If further moves are possible, the next move to be processed is popped off the queue and the next desired orientation is determined. Using this and the current orientation, the axis of rotation is determined and created in the Physics Engine. The constraint corresponding to the axis of rotation for the previous move is removed. The constraint which acted as the brakes during the WAITING state is also removed to allow rotation about the new axis. The state is transitioned to ACCELERATING to begin the next move.

- ACCELERATING:

  In this state, a positive torque about the axis of rotation is applied to perform the desired move. The physics engine is queried to determine the current orientation of the maze. When its rotation has exceeded half the desired sweep angle, the state is transitioned to DECELERATING.

- DECELERATING

  During deceleration, a negative torque is applied about the axis of rotation to begin slowing the maze down. When the maze has attained (within an acceptable tolerance) its desired orientation, the 'brake' constraint is created to stop the movement of the maze. Although the movement of the maze has been halted, the marble inside is still moving to its final resting location. The state is transitioned to WAITING until that final location can be determined.

6.6 Rendering Engine

Computer graphics is an essential component of this thesis. The three dimensional view of the maze, and the animation of its solution, is critical to the development, testing, and final display of the system. To handle the computer graphics, the libraries OpenGL and GLUT [Angel, 2003] were utilized. Since this is a simulation, rather than an interactive application, the GLUT (the OpenGL Utility Toolkit) API was able to provide all the user interface functionality needed, and greatly simplified that aspect of the development. By acting as an abstraction layer between the domain-specific code and the Windows operating system, it enabled the development to be concentrated on the physics model and on the AI needed to solve the problem, rather than on the low-level interface functionality that is specific to Windows.

The geometry used to create the marble and maze is actually quite simple, consisting of only a sphere and square polygons. There are however some interesting challenges created by the numerous polygons involved, and the fact that the geometry must appear transparent, since everything of interest takes place inside of the maze. As explained in Section 6.2 '*Computer Model*', long continuous walls which span multiple cells of the maze are actually constructed of numerous individual smaller walls, one for each side of each cell. This presents a problem when attempting to display them as if they were a

continuous wall. If the walls are outlined as in Figure 43 below, the numerous lines outlining the polygons make the representation very confusing. If however the polygons are not outlined as in Figure 44, the delineation between the cells is not as apparent, especially if it were displayed on a monitor with limited capability.

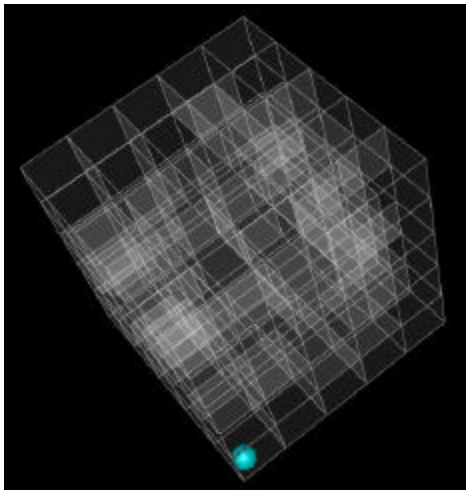

Figure 43: All Polygons Outlined

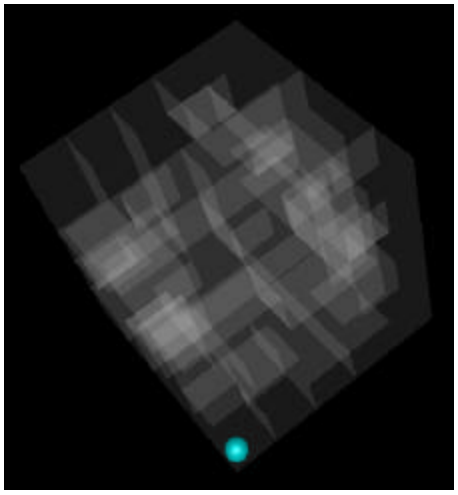

Figure 44: No Polygons Outlined

The solution to this was to determine and display only the edges of polygons where they intersect at a right angle. The effect is to visually 'stitch together' polygons that meet edge-to-edge to form a single wall section. As can be seen in Figure 45, this significantly enhances the viewer's ability to understand the geometry. This detail – determining where to draw the line segments representing the edges – actually takes more computational time than generating the maze itself, but fortunately it only needs to be done once for each maze.

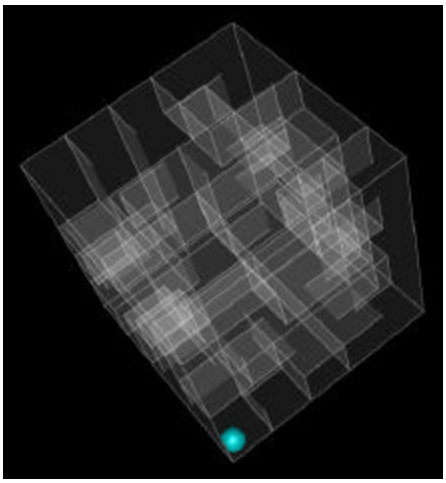

Figure 45: Selective Edges Drawn

Another challenge for the Rendering Engine comes from the fact that the game requires both opaque objects (the marble) and transparent objects (the walls). This required careful consideration of the order in which things were rendered, when to enable or disable lighting, and when to enable or disable depth testing. The solution that yields visually acceptable results is to draw the translucent polygons of the walls first with lighting and depth-testing disabled (so they do not block the view of the marble), drawing the solid sphere of the marble with lighting enabled, and then drawing the edges of the walls (as determined above) with depth-testing enabled to make it clear that the marbles are in fact behind the walls.

Another function that must be handled is the timing of the graphical updates. Updating the display every time the physics simulation is updated is not a good idea. Generally, the Newton physics engine can compute the new state of the system in far less time than it takes to render the objects and swap the frame buffers. If the graphics were updated every time the simulation was updated, the result would be an incredibly slow motion. One possible solution would be to advance the simulation by a larger time increment each time Newton is invoked, but this leads to unacceptable inaccuracies in the simulation.

A better solution, which was implemented for this thesis, is to advance the simulation as frequently as possible, and maintain a running timer to keep track of when the graphical

display is due to be updated. When a certain elapsed time (corresponding to the desired frame rate) has passed, the Rendering Engine updates the display with the current state of the geometry, which is obtained from Newton in the form of transformation matrices for each object. Although this results in the physics simulation being advanced several times between graphical frames, the changes from one frame to the next are very minute. The resulting effect is a smooth animation of a real-time accurate simulation.

7. Summary & Conclusions

For this thesis, algorithms were developed for finding the physical solution to a random three dimensional maze. The 'Physical solution' consists of being able to roll a marble from the starting cell to a target cell of the maze by rotating the maze itself and relying on the physics simulation to move the marble. To implement the solution, a software framework was developed which integrated the artificial intelligence, physics simulation, and computer graphics required to determine and animate the solution.

The intelligent agent controlling the maze depends on the physical simulation to determine the results of its actions. The agent's incomplete knowledge of the outcome of each move, before making it, creates much of the challenge for this thesis. The physical moves are not reversible, and therefore traditional search algorithms, which are reliant upon backtracking, can not be applied directly.

Several solution algorithms were developed to solve the physical puzzle. These algorithms all use elements of graph theory to expand the agent's knowledge of the maze being solved and the results of specific actions. The algorithms differ in the level of additional knowledge utilized pertaining to the maze's geometry and the physics involved. In general, it was shown that increasing the 'intelligence' of the agent significantly improves its performance. A modified control scheme was also developed which improves the physical actions that can be made by the agent. This was shown to reduce problems which occurred as part of the physical simulation.

The dependence upon the physical simulation was also shown to affect the analysis of the algorithms developed. All of the solution algorithms are complete in the sense that they are able to solve any random maze, but only under the conditions that a physical solution is possible, the control mechanism is capable of performing the necessary actions, and the marble does not enter a section of the maze from which it cannot escape. Interestingly, none of the algorithms developed can be guaranteed to find the optimal physical solution, since this would require perfect knowledge of the results of a given move before it is

made, which is not possible. However, despite the limitations imposed by the physical simulation, the best algorithm developed was able to improve the performance (as measured by the number of mazes that could be solved) over a simple blind search by over 400%, while requiring 69% fewer moves.

In this thesis, artificial intelligence, physics simulation, and computer graphics are successfully integrated to develop and display the solution to an interesting problem that could not be solved without each of the three technologies.

8. References

E. Angel, *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*, 2003, Pearson Education.

D. Astle, K. Hawkins, *Beginning OpenGL Game Programming*, 2004, Premier Press.

M. Atkin, P. Cohen, *Using Simulation and Critical Points to Define States in Continuous Search Spaces* in *Proceedings of the 32nd Conference on Winter Simulation*, 2000, Society for Computer Simulation International, Pages: 464 – 470.

S. Blackman, *Serious games...and less!* in *ACM SIGGRAPH Computer Graphics Archive*, Volume 39, Issue 1, Feb. 2005, ACM Press, New York, NY, USA, Pages: 12 – 16.

T. Cazenave, *Optimizations of Data Structures, Heuristics, and Algorithms for Path-Finding on Maps*, 2006, IEEE Symposium on Computational Intelligence and Games Pages: 27 – 33.

D. Conger, *Physics Modeling for Game Programmers*, 2004, Thomson Course Technology.

D. Eberly, *Game Physics*, 2004, Elsevier.

J. Grefenstette, C. Ramsey, A. Schultz, *Learning Sequential Decision Rules Using Simulation Models and Competition* in *Machine Learning*, 1990, Pages 355-381.

J. Jerez, *Newton Game dynamics tutorial*, http://www.newtondynamics.com, 2000-2004.

R. Jones, David J. Thuente, *The Role of Simulation in Developing Game Playing Strategies* in *Proceedings of the 23rd Annual Symposium on Simulation,* 1990, IEEE Press, Piscataway, NJ, USA, Pages: 89 – 97.

I. Karpov, T. D'Silva, C. Varrichio, K. Stanley, R. Miikkulainen, *Integration and Evaluation of Exploration-Based Learning in Games,* 2006, IEEE Symposium on Computational Intelligence and Games, Pages: 39 – 44.

S. Koenig, *A comparison of Fast Search Methods for Real-Time Situated Agents* in *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems,* 2004, IEEE Computer Society, Pages: 864 – 871.

R. Korf., *Artificial Intelligence Search Algorithms* in *Handbook of Algorithms and Theory of Computation* , 1999, CRC Press.

E. Lengyel, *Mathematics for 3D Game Programming & Computer Graphics*, 2002, Charles River Media, Pages: 62 – 64.

M. Lewis, J. Jacobson, *Game Engines in Scientific Research* in *Communications of the ACM archive,* Volume 45, Issue 1, Jan. 2002, ACM Press, New York, NY, USA, Pages: 27 – 31.

T. McReynolds, D. Blythe, *Advanced Graphics Programming Using OpenGL*, 2005, Elsevier.

P. Norvig, *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp,* 1992, Morgan Kaufman Publishers.

S. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach*, 1995, Prentice-Hall.

D. Vrajitoru, W. Knight *Finite Graphs and Their Applications - C243 Lecture Notes*, 2004.

M. Weiss, *Data Structures and Problem Solving Using C++*, 2000, Addison Wesley Longman.

P. Winston, *Artificial Intelligence – Third Edition*, 1992, Patrick H. Winston.

Gaffer.org, *Fix Your Timestep!*, URL: http://www.gaffer.org/game-physics/fix-your-timestep.

Newton Game Dynamics, URL: http://www.physicsengine.com.

OpenGL.org, *Transparency, Translucency, and Blending*, URL: http://www.opengl.org/resources/faq/technical/transparency.htm.

Vita

Dana Cremer was born in South Bend, Indiana on January 14, 1965. He entered Indiana University South Bend in 1985. In 1988 he received the I.U.S.B. Excellence Award in Mathematics, and completed a Bachelor's Degree in Mathematics in 1989, graduating with *Highest Distinction*.

He began working on a Master of Science degree in Applied Mathematics and Computer Science at Indiana University South Bend in 2003, which he expects to complete in 2007.

Dana has worked for Honeywell since 1990, designing and developing software to support engineering and manufacturing. The technologies he has been able to study and implement include:

- Object-oriented programming languages
- Numerical Analysis
- Expert Systems
- Distributed software architectures
- Programming of computer-controlled coordinate measuring machines

Several of the systems he has developed have earned him recognition:

- Presented at IBM's Solutions 2000 conference in Las Vegas in 2000
- Wrote an article, "Using EJBs as Components to Interface with Legacy Systems" that was published by IBM in the December 2000 edition of IBM's WebSphere Developer Technical Journal.
- Presented at the WebSphere Technical Exchange in Orlando in 2001
- Designed and developed a system, the Matrix to MACPAC Interface, which was featured in Sun Microsystems' book, J2EE Technology in Practice.

Dana's current primary academic interests are Computer Graphics and Artificial Intelligence.