PROBABILISTIC SMART TERRAIN

ALGORITHM

By

Mohd Faseeh

Submitted in the partial fulfillment of the Requirement

for the degree of

Master of Computing and Information Systems

YOUNGSTOWN STATE UNIVERSITY

May 2016

Probabilistic Smart Terrain Algorithm

Mohd Faseeh

I hereby release this thesis to the public. This thesis will be made available from the OhioLINK ETD Center and the Maag Library Circulation Desk for public access. I also authorize the University or other individuals to make copies of this thesis as needed for scholarly research.

Signature: ______________________ 04/27/2016

Mohd Faseeh, Student   Date

Approvals:

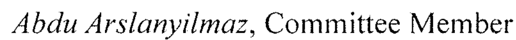
4/27/2016

John Sullins, Thesis Advisor   Date

______________________ 4/27/2016

Alina Lazar, Committee Member   Date

______________________ 4/27/2016



Abdu Arslanyilmaz, Committee Member   Date

______________________ 5-3-16

Sal Sanders, Associate Dean of Graduate Studies   Date

// Abstract

Smart Terrain is an algorithm that is used to find the object that meets the needs transmit signal to the non-player character with those needs influencing the character to move towards those objects. We describe how the probabilistic reasoning can be implemented on it deploying the object that it may meet a need with a given probability. The expected distance can be measured in terms of probability and distance that meet the needs, allowing the non-player character to follow the route in the game.

This algorithm can be used to manage a character's need to direct them which objective has a priority or which objectives are profitable to them. With a smart terrain, this algorithm defines how to find the goal in terms of probability and distance. This algorithm defines how the character is behaving as Human for making decisions.

We implement the algorithm as a Unity 3D Game using Waypoint and Navigation Mesh where the objective is to find and collect some valuable objects and stay away from the guards guarding the objects, while navigating in a maze like game world. The algorithm finds a path based on the concept of adjacent routes in the game such that it makes it difficult for the player to stay away from the guards. The player on the other hand is controlled by the user. The algorithm searches for possible paths and then makes a decision based on the calculations on probabilities and distances as discussed in detail in the paper.

Several features other than path finding such as ray casting and navigation mesh are also implemented to make the game feel life like. The guards behave intelligently and the

algorithm changes the probabilities of player being in a particular area of the game world with time. This makes the game even tough to win.

Acknowledgement

I am deeply grateful to my advisor *Dr. John Sullins* for giving me the confidence to explore my research interest and the guidance to avoid getting lost in my exploration. Dr. Sullins is a fabulous advisor with sharp mind, perspective and very helpful. I have been amazingly fortunate to have an advisor who gave me the freedom to explore on my own and at the same time the guidance to recover when my steps faltered. His patience and support helped me overcome my crisis situations and finish this research.

My committee member, *Dr. Alina Lazar*, has been always there to listen and give advice. I am deeply grateful to her for the long discussion that helped me sort out the technical details of my work. I am thankful to her for holding me to a high research standard and strict validation for each research results and thus teaching me how to do research,

My other committee member, *Dr. Abdu Arslanyilmaz*, he is one of the best teacher that I have had in my life. He sets high standards for his students and he encourages and guides them to meet those standards. He introduced me to the Unity programming which inspired me to work on my research. I am indebted to him for his continuous encouragement and guidance.

Most importantly, none of this would have been possible without the love and patience of my family. My immediate family, to whom this research is dedicated to, has been a constant source of love, concern, support and strength all these years. My best friend who has given the strength whenever I lost the confidence. I am grateful to have such a friend in my life.

Finally, I appreciate the financial support from the Department of Computer Science and the STEM College at YSU. I am grateful to them for supporting me during my graduate studies.

Table of Content

List of Tables…………………………………………………………ix

List of Figures……………………………………………………….…..x

SECTION 1:

Introduction…………………………………………………….……1

1.1 Motivation………………………………………………………………1

1.2 Contribution…………………………………………………………….1

1.3 Organization…………………………………………………………….3

SECTION 2:

Background…………………………………………….………….……4

2.1 Smart Terrain……………………………………………………………4

2.2 Probabilistic Smart Terrain…………………………………………………..4

2.2.1 Assumptions………………………………………………………..5

2.2.2 Plausible decision making with multiple objects……………………………..6

2.3 Previous Algorithm…………………………………………………………7

2.3.1 Modified Algorithm…………………………………………………..8

2.4 Waypoints…………………………………………………………………...8

2.5 Unity 3D……………………………………………………………………9

SECTION 3: Implementation of the Algorithm……………………….11

3.1 Defining the problem……………………………………………………………11

3.2 Representing the game with probabilities and waypoints…………………….....11

3.3 Implementing the world in Unity……………………………………………..13

3.4 Nav Mesh………………………………………………………………………16

3.5 Ray Cast and Player Pursuit…………………………………………………20

3.6 Collision detection and Treasure collection…………………………………….20

SECTION 4: Probabilistic Smart Terrain…………………………….…….22

4.1 Route Creation…………………………………………….........................23

4.2 Route Evaluation……………………………………………………………25

4.3 Sample Run…………………………………………………………………26

4.4 Modified probabilities over time……………………………………………27

SECTION 5: Future Work……………………………………………...29

REFERENCE…………………………………………………….…....31

APPENDIX…………………………………………………….…32

List of Table:

Score Board……………………………………………………………………12

Defining Probabilities…………………………………………………………13

List of Figures

Figure 1: Character moving towards the room having high probability……………..…...6

Figure 2: Character plausible decision for moving towards high probability………..…...7

Figure 3: Waypoints inside the rooms……………………………………………….…...9

Figure 4: View of a 3D game……………….…………………………………………...14

Figure 5: Smart Terrain Game……………………………………………………………..15

Figure 6: Implementation of Nav mesh agent………………………………………...16

Figure 7: View of a game in top level in XZ plane……………………………….......17

Figure 8: Animation State Machine………………………………………………………..18

Figure 9: Adjacent rooms…………………………………………………………………..22

Figure 10: Chosen route…………………………………………………………………27

SECTION 1: INTRODUCTION

1.1 Motivation

Many computer games involve a player exploring a level while avoiding guards of some sort, where the guards are controlled by the gaming artificial intelligence. This can be a difficult problem, as it is easy for such guards to either appear too “smart” by using an algorithm such as A* to move directly towards the player (whose location they should not know), or too “stupid” by simply following a predictable predefined patrol. A third approach is to introduce p*robabilities*, influencing the guard to move to rooms where the player is considered most likely to be.

Previous research[1] has introduced probabilities into non-player character navigation. However, it has been limited to moving between individual tiles on a level, rather than considering paths between rooms as a whole. Evaluation of the algorithm has also been limited, as no actual interactive game has been created to allow actual players to evaluate the apparent intelligence of non-player characters controlled by the probabilistic algorithms. This thesis is meant to address these issues.

1.2 Contributions

The main contributions of this thesis are:

1. *Providing an interactive testbed for probabilistic game algorithms*.

   The only way to reliably test any algorithm related to gaming is have actual typical players play the game. These players would evaluate the apparent intelligence of the non-player characters such as the guards, and make suggestions for improving their performance.

We have implemented an interactive game (with 8 rooms and two guards), where the goal of the player is to collect "treasures" while avoiding guards. The game is implemented in Unity, taking advantage of its abilities to automatically navigate between waypoints when moving between rooms, and to detect and move towards the player if they are visible to the guard.

Researchers may plug in their own gaming AI to control the behavior of the guards in terms of rooms to move to next. This would allow researchers to focus on the gaming AI and have it evaluated by players without having to worry about these implementation details.

2. *Exploring a probabilistic algorithm based on routes between rooms as a whole.*

Previous work has focused on deciding which individual tile on a level to move to next, using map flooding to avoid walls, and must be rerun each time the non-player character moves to a new tile.

This work takes advantage of the ability of Unity to automatically navigate from one waypoint to another while avoiding walls, without having to consider low-level constructs such as tile. Rather than thinking one small tile at a time, we propose an algorithm that plans an entire route through the level in advance. This route operates in terms of paths between waypoints set in adjacent rooms, letting Unity handle the details of creating and following that path. The route is also dynamic, in the sense that

it can be interrupted if the player becomes visible to the guard, and can also be affected by learned information about rooms already explored.

1.3 Organization

This thesis is organized as follows. The next section gives background on previous work in probabilistic smart terrain, as well as gaming concepts such as waypoints, navigational meshes, and ray casting. Section 3 describes the testbed game level used in this work, and the components of Unity used to implement it as an interactive game that would be interesting to players. Section 4 describes the modified probabilistic smart terrain algorithm based on waypoints and routes rather than tiles. Section 5 describes future directions for this work.

SECTION 2: BACKGROUND

2.1 *Smart Terrain:*

Smart Terrain is a useful algorithm for moving an object in a tiled world. It has been used in a number of games as the Sims due to its efficiency and ease of implementation. The idea of Smart Terrain can be summarized as below:[1]

- A character may have one or more need such as hunger, energy, and fun.
- Objects in the character meet those needs such as Refrigerator meets the hunger needs and Bed meets the energy need.

The character is influenced to move towards those objects by “signals” sent by those objects which have the following properties:[2]

- It has a finite (and possibly fairly small) range. That is, a character should not be influenced by objects implausibly far away.
- The “strength” of the broadcast signal decreases with distance, allowing closer objects a greater degree of influence.
- The signal moves around rather than through obstacles. That is, the strength of the signal is proportional to the time it would take the character to reach the object (presuming the character must move around obstacles as well).

2.2 *Probabilistic Smart Terrain:*

Probabilistic reasoning can be added to the smart terrain algorithm, enabling an object to broadcast that *it may meet a need* with a given probability instead of *it meets the need with probability* of 1. This causes a character to monitor those rooms which have high probability of a character’s need.

These probabilities are used to find the expected distance of an object that meets a need allowing nonplayer characters to make plausible decisions about which direction to move towards in an uncertain environment.[1]

2.2.1 Assumptions:

There are some assumptions about the probabilities in this paper:

- There will not be any specific algorithm to create these probabilities-; that is, they have been created by the designer or are based on some logical calculation and concept. They are programmer defined numbers.
- There is only one single need that the character is trying to meet such as hunger and only objects related to that such as refrigerator are broadcasting
- There is assumption of conditional independence, which means that whether or not an object meets the need does not actually depend on any other object in that area.
- Once the character has reached the object and discovered that whether it does or does not meet the need, those probabilities will be changed to either 1 or 0.

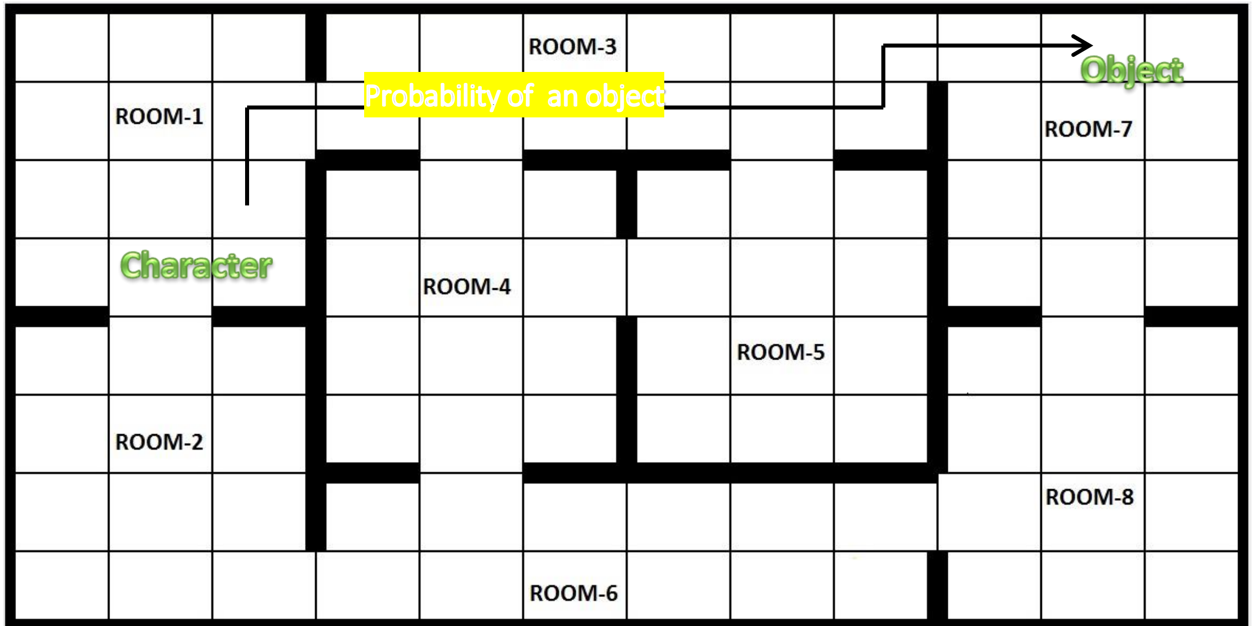


Figure 1: Character moving towards the room having high probability

2.2.2 Plausible decision making with multiple objects:

This becomes quite interesting when there are multiple objects and the character needs to determine which move is the best for meeting the need[3]. In this case, the character will move towards the closest object with the highest probability of meeting the need. However, there will be many cases in which there is not a clear best solution.

For example, consider a scenario of a hunger needs in a house with two refrigerators R1 & R2. The character with a hunger need is 8 tiles away from R1 with the chance of 70% probability of containing food and 6 tiles from R2 which has 60% chance of food. It's not obvious that whether the character should move towards R1 or R2. The character should behave in a plausible manner from the point of view of the player.[1] If they do not make

plausible move then the game will be a failure in terms of the view of a player. So which path should be taken by character?

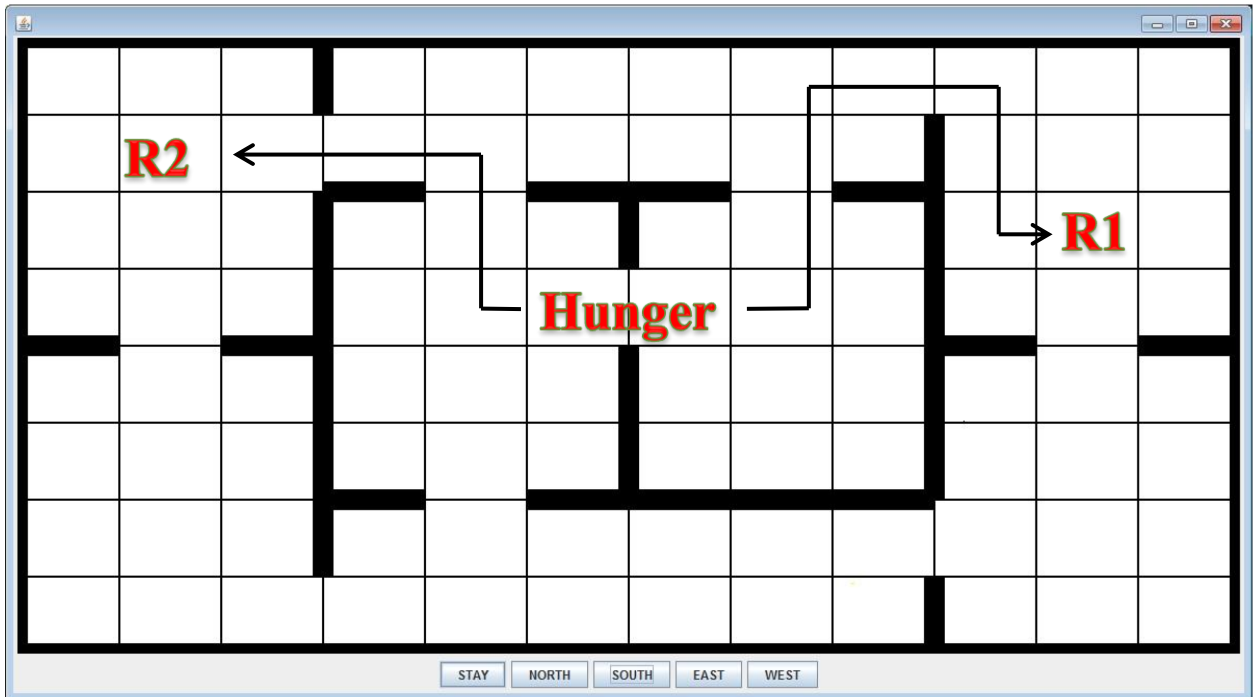


Figure 2: Character plausible decision for moving towards high probability

2.3 Previous Algorithm:

This algorithm was accomplished with a simple version of "Map Flooding" often used for influence maps in which the signal strength decreases by some value (such as 1) for every non-wall tile between the object and the character. Given this influence map, a character may reach the object by simply moving in the direction of the highest signal value.

In order to determine which of the four surrounding tiles to move to, a measure was based on an estimate the expected distance from each tile to some other object that meets

the need that allows comparison between them used to ultimately determine the “best” to move to in terms of distances and probabilities.[1]

2.3.1 Modified Algorithm:

The algorithm proposed in this thesis is based upon marker/waypoints within rooms as a whole, rather than individual tiles that a room is composed of. Each marker (visible in our world) gives us the exact position of the room’s geometrical center in the XZ plane.

In addition, we create an entire world “route” through all of the rooms in advance (avoiding cycles), rather than rerunning the algorithm at each new tile. This greatly increases the efficiency of the algorithm.

2.4 Waypoints:

Waypoints are referred as an intermediate point on a route or line of travel or as a stopping place on a journey, and are a way to represent a character’s world as a simple graph. Such graphs are usually designed to allow a character to traverse their world by moving from one waypoint to another on the graph.[4]

Waypoints are an in-built feature in Unity game programming. Waypoints are used to define the location of each room, as well as for determining distance between rooms.[5]

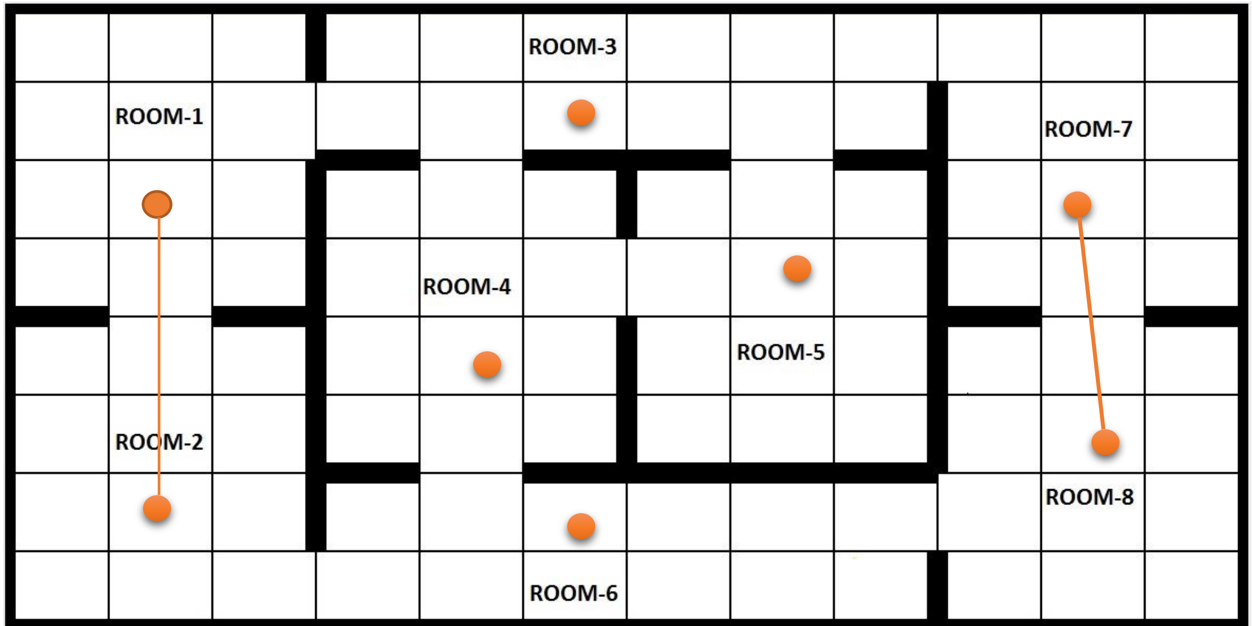


Figure 3: Waypoints inside the rooms

2.5 Unity 3D:

Unity is a cross platform game engine developed by Unity Technologies used to develop video games for PCs, consoles mobile devices and websites. First announced only for OS X at Apple's Worldwide Developers Conference in 2005, it has since been extended to target more than fifteen platforms. It is the default software development kit (SDK) for the Wii U. Five major versions of Unity have been released. At the 2006 WWDC trade show, Apple Inc named Unity as the runner up for its Best Use of Mac OS X Graphics category.

Unity is a cross platform game engine used to develop games for PCs, mobiles and websites. The game is designed in *Unity 5.3.4 version*. Unity has built-in features for defining and navigating between waypoints. In particular, there is a path finding script

called Navigation Component which is used to find the best available path, avoiding the walls and obstacles.[6]

SECTION 3: UNITY IMPLEMENTATION OF THE GAME

This section describes the implementation of the game in Unity 3D, in order to make it possible for actual player to play the game and evaluate the effectiveness of the gaming artificial intelligence. While this game system was used to implement the waypoint-based probabilistic smart terrain algorithm described in the next section, it could also be easily adapted to test any other waypoint-based gaming AI algorithms by modifying those sections of codes.

3.1 Defining the Game:

This research was inspired by a possible game scenario described by Dr. Sullins, a more complex version of a scenario used in his previous work. It was also used as a testbed for the algorithm described in the next section.[1]

The game world consists of eight rooms (as shown in the above figure). Several of the rooms contain "treasure", in the form of jewelry with different values. The goal of the Player is to steal as much jewelry as possible without being caught by the either of the two "Guards". There is an exit door on the left-top corner of the house. This door is used when the player steals all the jewelry and want to win the game. The goal of the Guards (implemented as a Gaming AI) is to present the player with as difficult challenge as possible, by searching for the player and attempting to catch the player if they are found.

3.2 Representing the Game with Probabilities & Waypoints:

We have assigned each of the eight rooms a particular probability, representing an estimate of the chance of the player moving into that room. We have also placed waypoints in all eight rooms to define a distance between each room (part of the

probabilistic smart terrain). We have also assigned four treasures in the form of jewelry in four different rooms. We define the value of each treasure in Table 1.

Jewelry	Points
Silver	**175**
Diamond	**100**
Statue	**200**
Gold	**150**

Table 1: Score Board

The chances of finding a player are defined to be the highest in the rooms nearest the exit door or rooms with treasure (the more expensive the treasure the higher the probability). In general, these probabilities are defined arbitrarily by the game designer and may be "tweaked" during game testing to create better game play (very commonly done with the parameter of gaming AI algorithms).

Rooms	Probabilities
Room 1	0.2
Room 2	0.4
Room 3	0.2
Room 4	0.2
Room 5	0.4
Room 6	0.2

Room 7	0.6
Room 8	0.5

Table 2: Defining Probabilities

Once the guard is in eye contact with the player, he will chase the player to their last position. If the guard catches the player, the game will be over and the Player loses, but the player is able to steal all the jewelry and exit through the door without being caught, the Player wins. Once the guard reaches the room, that room's probability will become zero and if the player is not there then the probability of that room will increase slowly as the guard moves away from the room (described in more details in the next section).

The player can take any path they want using the arrow keys to move, in order to collect all the valuable treasures and then exit the game without being captured by the guards. The guard uses algorithm described in the next section to make an informed decision about the order of rooms to explore. These decisions are meant to find the player in minimum time before he collects all the items and exits the game.

3.3 Implementing the world in Unity:

Unity is capable of handling 2D and 3D worlds very well. We are using the Unity 5.1 version, the latest release available during the time of the work. For this game, we have used the 3D world model. Although the game looks 2D to the player or any viewer, the game world is completely modeled in 3D. The projection is arranged top-down such that the world appears 2D to the viewer or player, as shown in Fig 7. This helps in evaluating the path finding algorithm and also better gameplay, as it gives the entire view of the game world to the player, including the location of each guard. The player then decides

where to move from their current location in the world in order to gain treasure while avoiding the guards.

The world consists of a floor which has a grid-like layout drawn on it. The floor lies in the XZ plane of the 3D coordinate system used by Unity. The walls in the game world are also in the XZ plane but since they have height, which is required to create the effect of rooms and separation of areas, the walls also extend to the third axis, the Y-axis. The walls are high enough (that is, extended into the Y-Axis far enough) to prevent the player and guards to cross them during the normal game course. The camera is fixed and placed at the top of the level while the level ground lies in the XZ plane, thereby making the users feel as if it is 2D game. The walls force the player and the guards to follow certain paths.

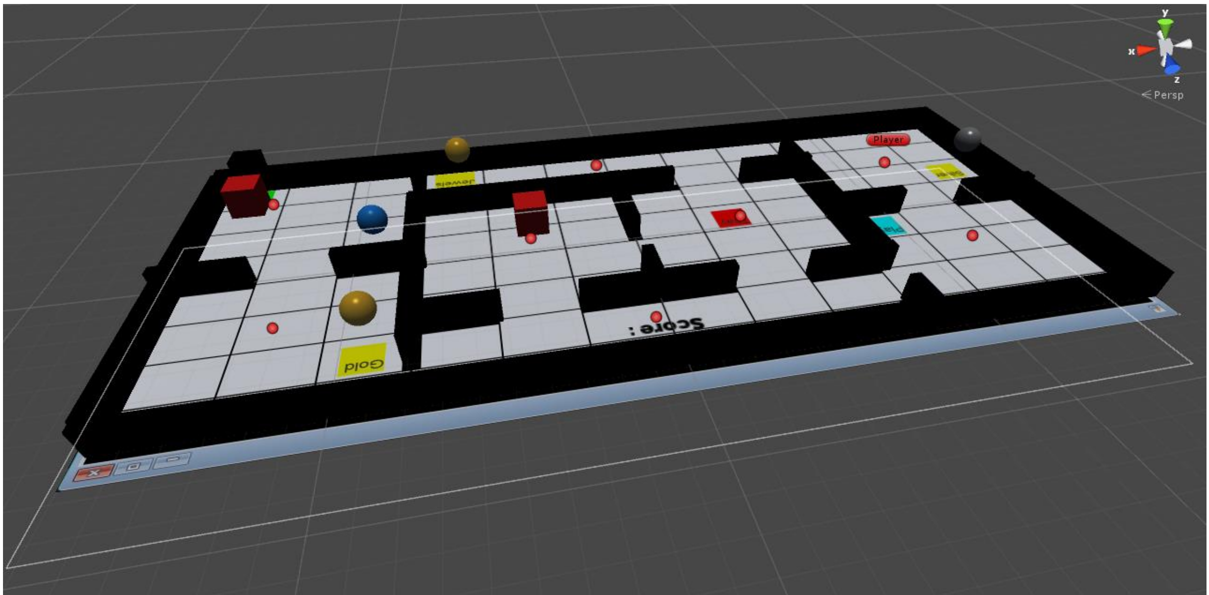


Figure 4: View of a 3D Game

This layout enables us to use the built-in unity features related to path-finding using Navigation agents and navigation meshes. The 3d mesh/cube that the floor comprises of is converted into a navigable mesh in which players and guards can move freely, except for the walls. Each object has a 3D physics model applied to it for collision detection, gravity etc.

The navigable mesh is the region in which player and guards can move freely. This requires that the areas which the walls need to be excluded from the navigable mesh. Once we define the navigable mesh, we can then use the path finding scripts using the navigation agent and move within the mesh automatically finding the path between two coordinates (or Transform Vectors) in the mesh.

Figure 5 shows the navigable mesh.

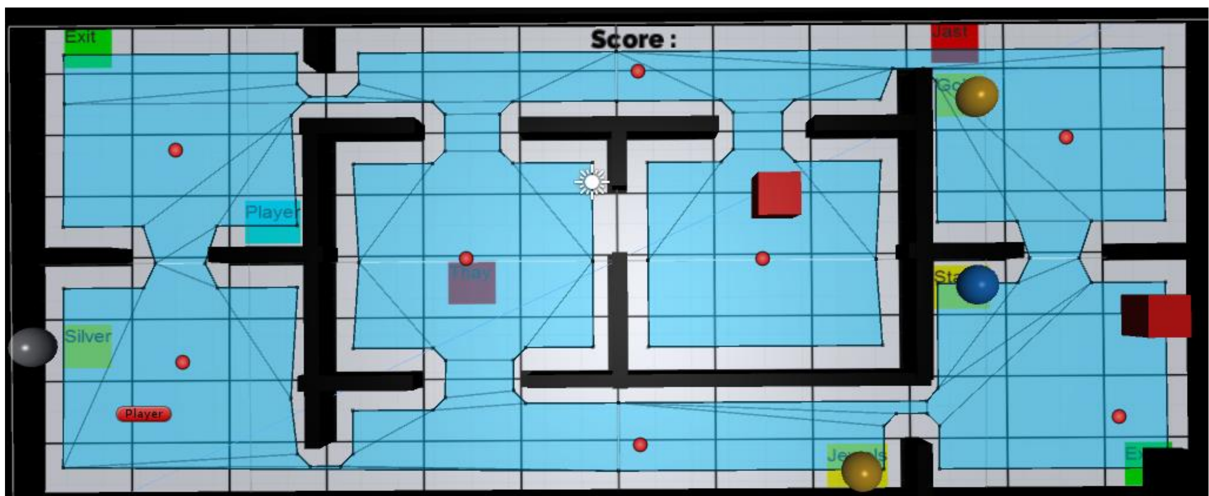


Figure 5: Smart terrain Game

The region shown in the blue is the navigable area i.e. the area in which players and guards are allows to navigate freely. The region marked in the white is the region is non-

navigable and the player as well as guards cannot navigate to these regions of the mesh. This gives us the effect of rooms and walls.

3.4 NavMesh Agent:

The NavMeshAgent components help to create characters which avoid each other while moving towards their goal. Agents reason about the game world using the NavMesh to reach specific waypoints while avoiding obstacle such as walls. Path finding and spatial reasoning are handled using the scripting API of the NavMesh Agent.[7]

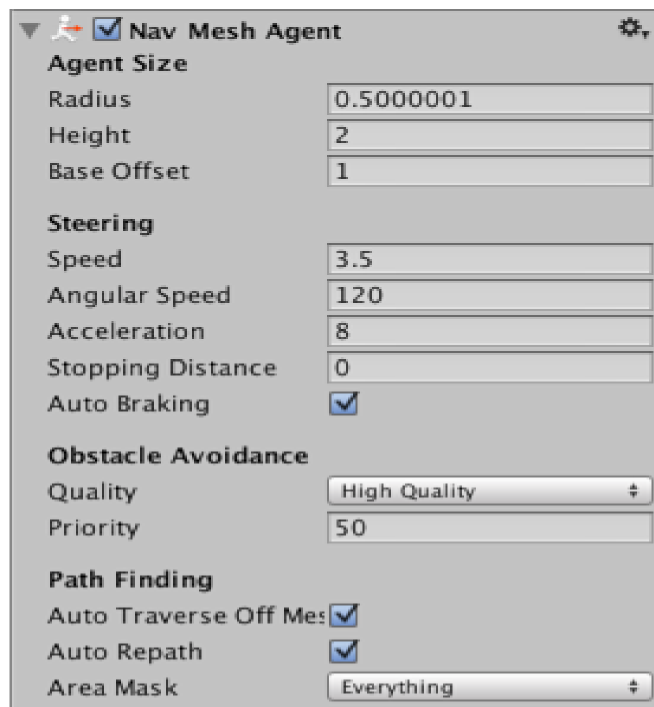


Figure 6: Implementation of Nav Mesh Agent

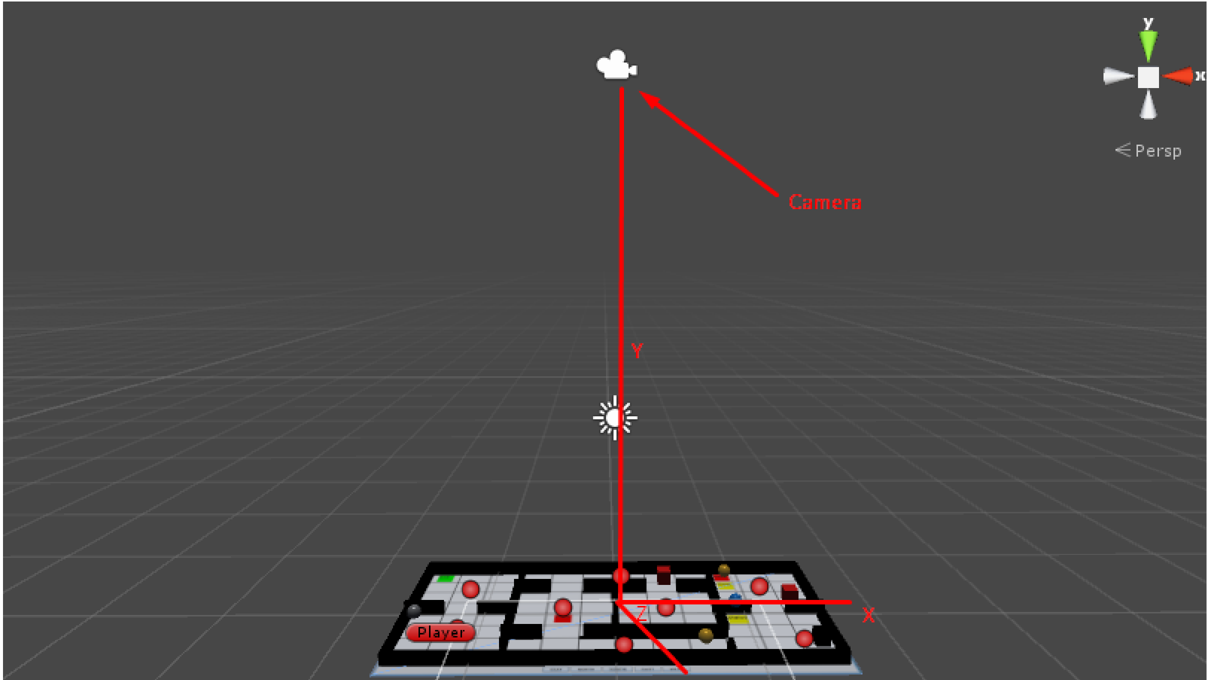


Figure 7: View of a game from the top level in XZ plane

The player object is a 2D sprite and is defined in the XZ plane. It is only visible from a distance in the Y-axis, perpendicular to the XZ plane in which our game world lives.

The player sprite is an animated sprite. It is a state machine where each state defined an animated state of the player sprite. The states are,

- Standing
- Walking Left
- Walking Right
- Walking Up
- Walking Down

The states change based on the user input. We keep the track of the states using a state parameter called **direction**. The state is changed based on the value of the **direction**

parameter. If the value of the direction parameter is 0, it means that the state of the player object sprite should be “standing”. If the **direction** parameter changes to 1, 2, 3 or 4, it means that the state will be changed to Walking Left, Walking Right, Walking Up and Walking Down. The states keep changing continuously to simulate an estimation of “walking”.

When the user is not pressing any key on the keyboard, the **direction** parameter is set to 0 which means the state of the sprite will be **Standing**, hence the standing animation. If the user presses the UP key on the keyboard, the **direction** parameter changes to 1, which changes the state from **Standing** to **Walking Up**. As soon as user leaves the key, the **direction** parameter changes back to 0 and the state hence changes to **Standing**. This keeps on happening for UP, LEFT, RIGHT and DOWN keys on the keyboard. Unity continuously monitors the presses and releases of these four keys and changes the **direction** parameter, which in turn changes the state. Below is the implemented state machine. The state in the orange is the **Standing** State.

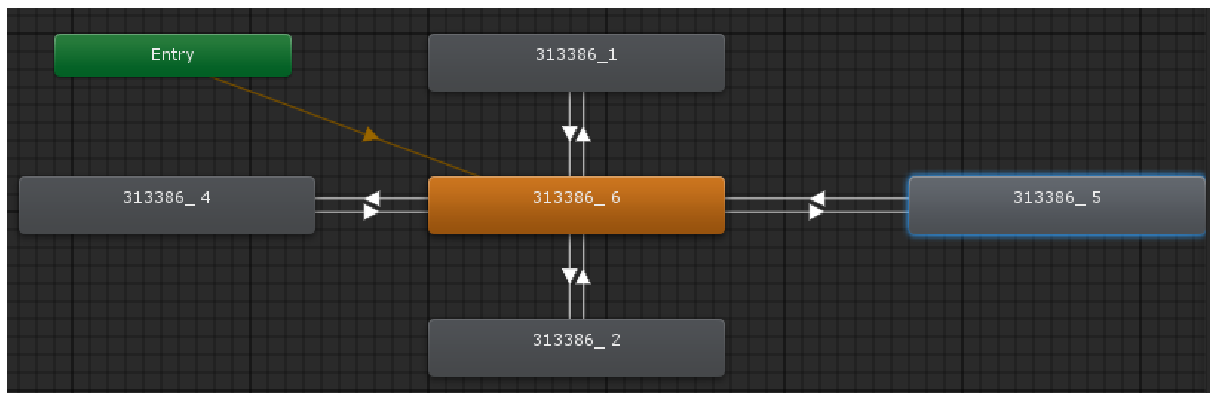


Figure 8: Animation State Machine

Each animation state consists of four frames.

1. Standing Frame

2. Walking Down Animation Frames

3. Walking Up Animation Frames

4. Walking Left Animation Frames

5. Walking Right Animation Frames

These animations execute based on the state of the sprite. The user presses different arrow keys, moving the player in XZ plane and also enabling different animation states on the player sprites. This gives a feel as the player is actually walking in the game world.

3.5 Ray casting and Player pursuit:

If while navigating on to a waypoint, the guard spots the player, it starts following the player with 2X speed. This gives a more dynamic and exciting gaming experience. It feels as if on spotting the player, the guards start running towards the player.

A Ray cast processing algorithm is used by Unity to make sure that the guards and player are in a straight line with no obstacles in between. A Ray cast is like a beam of light that is emitted from an object in a straight line in any required direction to detect position of other objects. This is a common way of creating effects of shooting objects in shooter games.

If there is no wall between them, the guards "see" the player and start running towards him for as long as they are in a straight line. If a wall comes in between, the guards lose the player and start following its original route again. The only way to trick the guards is not to come in a straight line with them, and if that happens, then the player needs to break the "line of sight" as soon as possible.

3.6 Collision Detection and Treasure Collection:

If the player moves to the exit, the game is complete and the player is declared as having WON. If the Guards catch the player before collecting all the items and exiting, the player is declared as having LOST. We have used collision detection to make sure that

the player has taken the treasure item. When the player collides with the item, we check for the item that the player has collided and once we find out the collider (we use the name), we can them remove it from the game world and increase the points of the player which are displayed on the top. Each item has its own defined points which are also totally defined in the algorithm. We can change the values which can increase of the decrease the importance of that item in the game world. However, this will not have any effect on the game algorithm and the AI as long as the probabilities of the rooms stay the same.

SECTION 4: PROBABILISTIC SMART TERRAIN ALGORITHM

The purpose of the algorithm is to find an efficient route through all of the rooms, in order to most quickly reach rooms that are most likely to contain the player based on the distances to the rooms and probabilities the player is in each room. These routes consist of paths between adjacent rooms (as the guard can only move between rooms that are connected), and do not contain the same path twice.

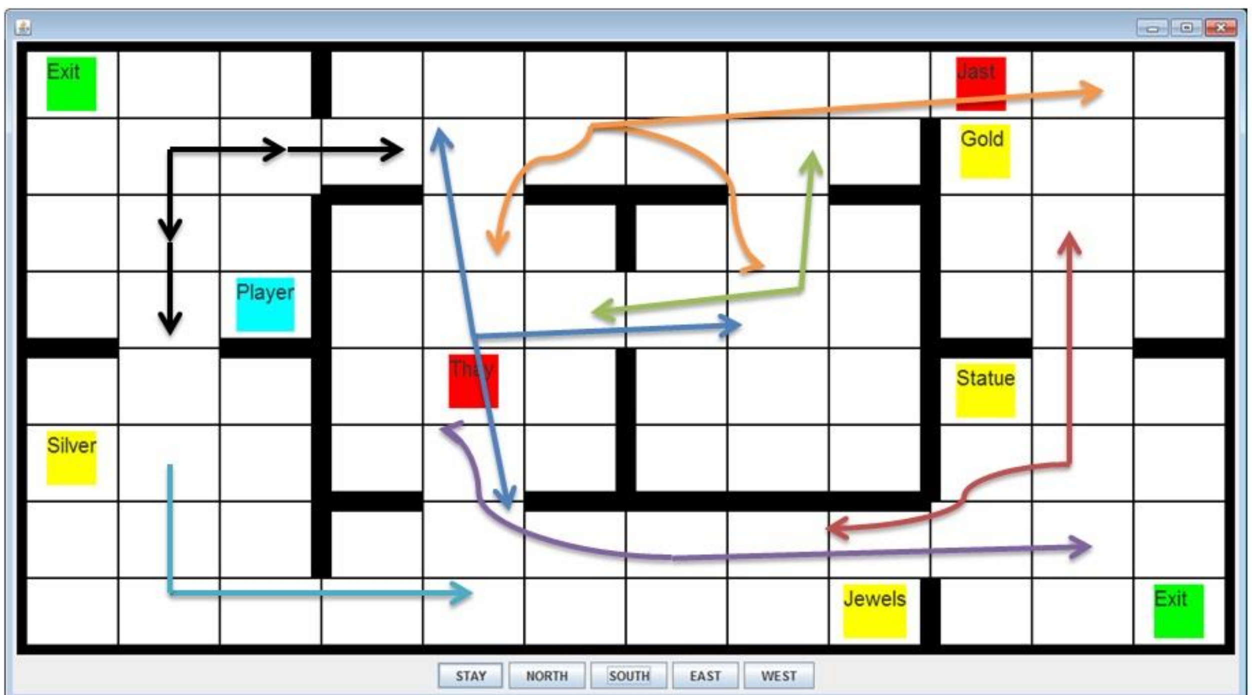


Figure 9: Adjacent routes

Once a room is reached, the algorithm is able to know whether the player is in the room. If the player is visible to the guard (that is there is no wall between the player and the guards based on the concept of Ray Cast), the Unity scripting moves the guard directly towards the player, interrupting its traversal of the route. Otherwise, the algorithm continues from that room to the next room in the route.

4.1 Route creation:

The algorithm starts by defining rooms and paths as data structures. We define a Path class and also a Room class (see the appendix for the code for these classes). The Path class has properties that define a unique path in space between two adjacent rooms. It has a source room and a destination room, the length of the path and a Boolean value to indicate whether the path has been taken previously by our NPC or not. Similarly, the Room class also has properties that identify a room, such as Room Number and other properties that indicate whether the room has already been navigated to or not.

We define the possible paths as the objects of class Path and all the rooms as objects of class Room. A route is a list of Path objects. So if a path exists from Room1 to Room8 via some other rooms, the Route can store all the Paths in order and define a unique route from Room1 to Room8. We add new routes to a list of routes also; this keeps on happening throughout the course of the algorithm. Redundant, duplicate and incomplete routes are removed from the list of routes and new routes are added to the list.

- Redundant routes are the routes that are already covered in other routes (that is the route that are subset of other route).
- Incomplete routes are the route that do not start from the source room which is room 1 or routes that do not cover all the rooms such routes are deleted from list of routes.
- Duplicate routes are the routes that make cycle or that concludes deadlock situation.

We start by defining one route for each room as a source. Each of these initial routes has only one Path. So in our case, for 8 rooms, we have 20 total initial routes containing paths between adjacent rooms, each containing a unique Path as defined during the start of the algorithm. We traverse through the list of routes over and over again, and pick up one route at a time we use loop to check all routes one by one starting from first route until the last route and for each route we execute the following.

For each route we select, we look at all other routes and try to joins chains. We do that by matching the source room number of each route's First Path to the destination room number of the selected route's last Path. If they match, we add a new Route to the list by joining the two Routes. This happens for every Route in the list until we visit all the rooms that is If a route does not contains all the rooms- by the end of the algorithm it is discarded. We also make sure that we do not create any cycles. We ensure this by checking the destinations of all the Paths in the Route. If any Path in the Route has a destination equal to any of the source of any other path in the route, then we drop that Route altogether.

Finally, we filter the Routes in the list. We drop all the Routes that do not traverse through all the rooms. Then we filter by checking the Routes' source. If the source is not the current room, we drop it from consideration.

It is possible that this may result in an exponential number of routes in the worst case. However, we argue that this is not a major consideration since most levels in actual games:

- Contain a limited number of rooms in each.

- Have relatively few routes available in order to constrain the choices of the player and present more of a challenge.

We are willing to accept this possible cost in order to increase our chances of finding a good route. In practice, this algorithm performs quickly in the testbed in 0.07 seconds.

4.2 Route Evaluation:

Once a list of legal routes has been created, a second algorithm is then used to estimate which remaining route is best, based on a metric that uses length of the paths in each room Route and the Probabilities that are stored in the Room class. All the Routes are subjected to this calculation and the one with the lowest result is selected. This gives a sequence of rooms to visit with the high the likelihood that we will find the player quickly.

The metric is calculated as a summation of products of distance and probabilities.

$$\text{Decision Metric} = \sum_{i=0}^{n} \big(1 - \mathrm{P(i)}\big) * \frac{d\{i,i-1\}}{i+1}$$

{1-P (i)} = Probability of not having a Player in i^{th} room.

d {i, (i-1)} = Distance between Room i and (i-1)

We select the route with the minimum value of this metric. The metric is designed to favor routes that have the following characteristic:

- Lower total route distance, in terms of distance d {i, i-1} between adjacent rooms in the route.
- Routes that explore higher probability rooms early. This is the effect of dividing the probability of not finding the player at step i by i-1.

We take this list of Rooms and the navigation component to navigate the guards from one Room to the first room in the route we have chosen. Once, the next room to move to has been chosen, we use the built in navigation features in Unity to move to a waypoint set in that room. The process begins again from this point.

4.3 Sample Run:

As an example, we show the result of running the algorithm in our testbed level. Specifically, we assume that the guard is in room 1 (at the upper left) and wishes to find a good route based on the probabilities and distance between rooms given previously.

The best route chosen by the metrics explores rooms in the order

1 → **2 → 4 → 5 → 6 → 3 → 7 → 8**

as shown in figure 10 below.

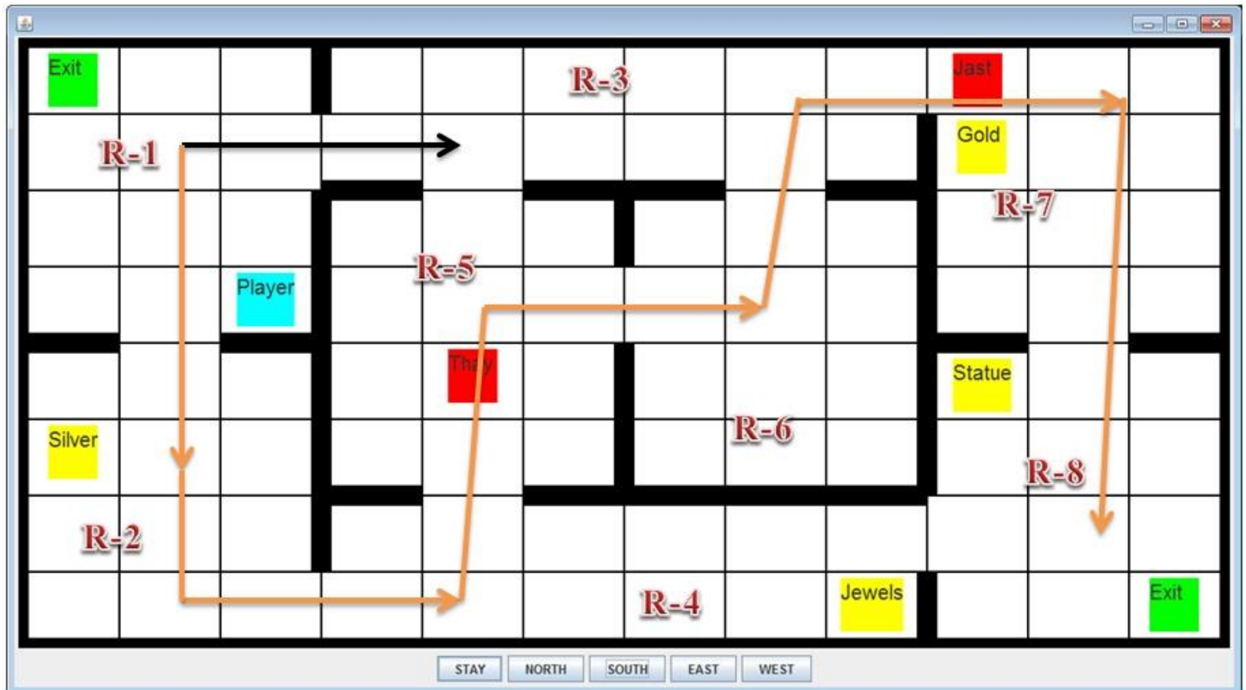


Figure 10: Chosen route

We argue that this is a plausibly good combination of probabilities and distances. The route explores two rooms with treasure early, but uses the “short cut” through rooms 5 and 6 to create a route with a relatively short overall distance.

4.4 Modified probabilities over time:

The guards navigate from one Room to another and reach the last room in the path and hence, and complete the route taken and visiting all the rooms. The algorithm then recalculates the paths again and finds the good path again based on the probabilities which have now changed depending on time. When a guard visits a room, if a player is not in the room its current probability is set to 0.

However, there is always a possibility that the player may still visit the room in the future, so its probability should increase again as the guard starts visiting other rooms.

Specifically, the probabilities increase by 1% of its original value every 5 seconds until it eventually returns to its original value. If a guard traverses its entire route without catching the player, another route is then generated by using those probabilities. Since most of the probabilities will be different from the calculation of the initial route (as not all the probabilities will have returned to their original values yet), it is very likely a different route will be chosen. This is important, as following the same patrol every time would make the guard appear less intelligent. These parameters are totally configurable and changing these values (the time span of 5 seconds and factor of 1%) may result in different outcomes, so finding the best values would also be part of the game testing process.

SECTION 5: FUTURE WORK

There are number of different direction that this research could take, given more time:

1. Let actual players evaluates the game algorithms:

   Unfortunately time did not permit this game to be used by actual players, in order for them to evaluate the apparent intelligence of the guards. However, since this algorithm has been implemented in a playable Unity game, it would be easy to do this in the future. It would also be easy for future research to implement their own gaming AI as part of this system, and then allow players to evaluate it as well.

2. Test the algorithm on different level configurations:

   As of now, the algorithm considers only the 8 rooms and the predefined paths between these rooms to find out a good route. It can be improved to work efficiently with more rooms and more paths. This would be helped by having the program read this information from files rather than having it be "hardwired", as it presently is, or by having the game automatically generate levels.

3. Test the algorithm in other scenarios:

   There are currently two guards and exactly four treasures with the probabilities given above. We can further test the system with different numbers off guards and treasures, or with different probabilities associated with those treasures.

4. Creating more cooperation between guards when determining the paths.

   Currently, each guard determines its route independently. Major improvement would be to implement cooperation between guards, so as to avoid checking for

rooms that are have been recently checked by other guards. That would increase apparent intelligence of the guard.

5. Increase the difficulty of the game with time.

   We can improve the algorithm to increase the difficulty of the game and making guards better at guarding as the player progresses. More intelligent guards will become highly protective of collectibles as the player progresses through the level. This can be done by increasing the probabilities associated with the treasure rooms over time. We can also slow down the speed of the player in the game to so that it would be hard for the player to collect the item or we can increase the speed of the guard while monitoring the rooms and when the guard and player are in the same straight line, we can increase the speed of the guard unto 4 times its original speed to catch the player.

REFERENCES

[1] J. R. Sullins, "Probabilistic smart terrain," *Int. J. Artif. Intell. Tools*, vol. 19, no. 04, pp. 531–550, Aug. 2010.
[2] P. Doyle, "Virtual intelligence from artificial reality: Building stupid agents in smart environments," in *Proceedings of the AAAI'99 Spring Symposium on Artificial Intelligence and Computer Games*, 1999.
[3] "probability - Average Distance Between Random Points on a Line - Mathematics Stack Exchange." [Online]. Available: http://math.stackexchange.com/questions/195245/average-distance-between-random-points-on-a-line. [Accessed: 19-Apr-2016].
[4] "Artificial Intelligence for Games (The Morgan Kaufmann Series in Interactive 3D Technology)." [Online]. Available: http://dl.acm.org/citation.cfm?id=1199865. [Accessed: 20-Apr-2016].
[5] "Waypoint," *Wikipedia, the free encyclopedia*. 13-Mar-2016.
[6] "Unity (game engine)," *Wikipedia, the free encyclopedia*. 18-Apr-2016.
[7] P. Tozour and I. S. Austin, "Building a near-optimal navigation mesh," *AI Game Program. Wisdom*, vol. 1, pp. 298–304, 2002.

Appendix: Code

Implementing the Player:

```
Public class PlayerControl: MonoBehaviour {

void Update () {
		if (Input.GetKey (KeyCode.LeftArrow)) {
			float x = gameObject.transform.position.x;
			float y = gameObject.transform.position.y;
			float z = gameObject.transform.position.z;
			x = x - stepSizeH;
			gameObject.transform.position = new
Vector3(x,y,z);

	GetComponent<Animator>().SetInteger("direction", 3);
		}
		else if (Input.GetKey (KeyCode.RightArrow)) {
			float x = gameObject.transform.position.x;
			float y = gameObject.transform.position.y;
			float z = gameObject.transform.position.z;
			x = x + stepSizeH;
			gameObject.transform.position = new
Vector3(x,y,z);

	GetComponent<Animator>().SetInteger("direction", 1);
		}
		else if (Input.GetKey (KeyCode.UpArrow)) {
			float x = gameObject.transform.position.x;
			float y = gameObject.transform.position.y;
			float z = gameObject.transform.position.z;
			z = z + stepSizeV;
			gameObject.transform.position = new
Vector3(x,y,z);

	GetComponent<Animator>().SetInteger("direction", 0);
		}
		else if (Input.GetKey (KeyCode.DownArrow)) {
			float x = gameObject.transform.position.x;
			float y = gameObject.transform.position.y;
			float z = gameObject.transform.position.z;
			z = z - stepSizeV;
			gameObject.transform.position = new
Vector3(x,y,z);

	GetComponent<Animator>().SetInteger("direction", 2);
```

```
		}
		else

	GetComponent<Animator>().SetInteger("direction", 4);
	}
```

For Collision:

```
void OnCollisionEnter (Collision col)
	{
		if (col.gameObject.name == "Jewels") {
			col.gameObject.SetActive(false);
			Debug.Log ("COLLIDED with " +
col.gameObject.name);
			points += 100;
			scoreText.text = "Score : " +
points.ToString() +" Points";
			jewel = true;
		}

		if (col.gameObject.name == "Gold") {
			col.gameObject.SetActive(false);
			Debug.Log ("COLLIDED with " +
col.gameObject.name);
			points += 150;
			scoreText.text = "Score : " +
points.ToString() +" Points";
			gold = true;
		}

		if (col.gameObject.name == "Silver") {
			col.gameObject.SetActive(false);
			Debug.Log ("COLLIDED with " +
col.gameObject.name);
			points += 175;
			scoreText.text = "Score : " +
points.ToString() +" Points";
			silver = true;
		}

		if (col.gameObject.name == "Statue") {
			col.gameObject.SetActive(false);
			Debug.Log ("COLLIDED with " +
col.gameObject.name);
			points += 200;
```

```
                scoreText.text = "Score : " +
points.ToString() +" Points";
                statue = true;
          }

          if (col.gameObject.name == "Exit" && (gold &&
silver && statue && jewel)) {
                //col.gameObject.SetActive(false);
                Debug.Log ("COLLIDED with " +
col.gameObject.name);
                scoreText.text = "YOU WON!!!";
                scoreText.fontSize = 24;
                scoreText.color = new Color(255f, 0f,0f);
                scoreText.transform.position = new
Vector2(scoreText.transform.position.x, 20f);
                Time.timeScale = 0;
          }

          if (col.gameObject.name == "Enemy" ||
col.gameObject.name == "Enemy2") {
                Debug.Log ("COLLIDED with " +
col.gameObject.name);
                scoreText.text = "YOU LOST! GAME OVER!";
                scoreText.fontSize = 24;
                scoreText.color = new Color(255f, 0f,0f);
                scoreText.transform.position = new
Vector2(scoreText.transform.position.x, 20f);
                Time.timeScale = 0;
          }
}}
```

For Path finding (Smart Terrain Algorithm)

```
public class zombie : MonoBehaviour {

     class Room {
          public int RoomNumber;
          public float RoomProbability;
          public bool Visited;
          public Room(){
          }
          public Room(int RN, float RP)
          {
                RoomNumber = RN;
```

```
                RoomProbability = RP;
                Visited = false;

            }
        }

        public struct Path
        {
            public int source_room_number;
            public int destin_room_number;
            public bool path_taken;
            public int length;
        }

        public Transform[] wayPoints;
        public Transform victim;
        private Vector3 victimsLastPosition;
        public NavMeshAgent navComponent;
        int index;
        int indicator = 1;
        float lastRecordedTime;
        private NavMeshPath meshPath;

        //Creating a Dictionary to store Room Numbers (int)
    with their respective probabilities (float)
        private List<Room> rooms = new List<Room>();
        private List<Room> SortedRooms = new List<Room> ();

        void Start () {
            navComponent = GetComponent<NavMeshAgent> ();
            index = 0;
            lastRecordedTime = 0;
            meshPath = new NavMeshPath ();

            List<Path> paths = new List<Path>();
            List<List<Path>> routes = new List<List<Path>>();

            Path one = new Path(); one.source_room_number =
    1; one.destin_room_number = 2; one.path_taken = false;
    one.length = 3;
            Path two = new Path(); two.source_room_number =
    2; two.destin_room_number = 1; two.path_taken = false;
    two.length = 3;
            Path three = new Path(); three.source_room_number
    = 2; three.destin_room_number = 4; three.path_taken =
    false; three.length = 4;
```

```
            Path four = new Path(); four.source_room_number =
4; four.destin_room_number = 2; four.path_taken = false;
four.length = 4;
            Path five = new Path(); five.source_room_number =
4; five.destin_room_number = 8; five.path_taken = false;
five.length = 5;
            Path six = new Path(); six.source_room_number =
8; six.destin_room_number = 4; six.path_taken = false;
six.length = 5;
            Path seven = new Path(); seven.source_room_number
= 8; seven.destin_room_number = 7; seven.path_taken =
false; seven.length = 3;
            Path eight = new Path(); eight.source_room_number
= 7; eight.destin_room_number = 8; eight.path_taken =
false; eight.length = 3;
            Path nine = new Path(); nine.source_room_number =
7; nine.destin_room_number = 3; nine.path_taken = false;
nine.length = 5;
            Path ten = new Path(); ten.source_room_number =
3; ten.destin_room_number = 7; ten.path_taken = false;
ten.length = 5;
            Path eleven = new Path();
eleven.source_room_number = 3; eleven.destin_room_number =
1; eleven.path_taken = false; eleven.length = 4;
            Path twelve = new Path();
twelve.source_room_number = 1; twelve.destin_room_number =
3; twelve.path_taken = false; twelve.length = 4;
            Path thirteen = new Path();
thirteen.source_room_number = 3;
thirteen.destin_room_number = 5; thirteen.path_taken =
false; thirteen.length = 3;
            Path fourteen = new Path();
fourteen.source_room_number = 5;
fourteen.destin_room_number = 3; fourteen.path_taken =
false; fourteen.length = 3;
            Path fifteen = new Path();
fifteen.source_room_number = 3; fifteen.destin_room_number
= 6; fifteen.path_taken = false; fifteen.length = 3;
            Path sixteen = new Path();
sixteen.source_room_number = 6; sixteen.destin_room_number
= 3; sixteen.path_taken = false; sixteen.length = 3;
            Path seventeen = new Path();
seventeen.source_room_number = 5;
seventeen.destin_room_number = 6; seventeen.path_taken =
false; seventeen.length = 3;
            Path eighteen = new Path();
eighteen.source_room_number = 6;
```

```
eighteen.destin_room_number = 5; eighteen.path_taken =
false; eighteen.length = 3;
          Path nineteen = new Path();
nineteen.source_room_number = 5;
nineteen.destin_room_number = 4; nineteen.path_taken =
false; nineteen.length = 3;
          Path twenty = new Path();
twenty.source_room_number = 4; twenty.destin_room_number =
5; twenty.path_taken = false; twenty.length = 3;

          paths.Add(one);
          paths.Add(two);
          paths.Add(three);
          paths.Add(four);
          paths.Add(five);
          paths.Add(six);
          paths.Add(seven);
          paths.Add(eight);
          paths.Add(nine);
          paths.Add(ten);
          paths.Add(eleven);
          paths.Add(twelve);
          paths.Add(thirteen);
          paths.Add(fourteen);
          paths.Add(fifteen);
          paths.Add(sixteen);
          paths.Add(seventeen);
          paths.Add(eighteen);
          paths.Add(nineteen);
          paths.Add(twenty);

          Room R1 = new Room(); R1.RoomNumber = 1;
R1.Visited = false; R1.RoomProbability = 0.2f;
          Room R2 = new Room(); R2.RoomNumber = 2;
R2.Visited = false; R2.RoomProbability = 0.4f;
          Room R3 = new Room(); R3.RoomNumber = 3;
R3.Visited = false; R3.RoomProbability = 0.2f;
          Room R4 = new Room(); R4.RoomNumber = 4;
R4.Visited = false; R4.RoomProbability = 0.2f;
          Room R5 = new Room(); R5.RoomNumber = 5;
R5.Visited = false; R5.RoomProbability = 0.4f;
          Room R6 = new Room(); R6.RoomNumber = 6;
R6.Visited = false; R6.RoomProbability = 0.2f;
          Room R7 = new Room(); R7.RoomNumber = 7;
R7.Visited = false; R7.RoomProbability = 0.6f;
          Room R8 = new Room(); R8.RoomNumber = 8;
R8.Visited = false; R8.RoomProbability = 0.5f;
```

```
            rooms.Add(R1);
            rooms.Add(R2);
            rooms.Add(R3);
            rooms.Add(R4);
            rooms.Add(R5);
            rooms.Add(R6);
            rooms.Add(R7);
            rooms.Add(R8);
```

Path that Leads to Cycle:

```
//identify the paths that can lead to cycles
                         List<int> indicesToRemove = new
List<int>();

                         if (nextPossibleRoutes == null)
                              continue;


                         foreach (Path nextRoute in
nextPossibleRoutes)
                         {
                              foreach (Path p in
currentRoute)
                              {
                                   if (p.source_room_number
== nextRoute.destin_room_number)
                                   {

     indicesToRemove.Add(nextPossibleRoutes.IndexOf(nextRou
te));
                                   }
                              }
                         }
                    }
```

Remove the Path that leads to cycle:

```
//remove the paths that are leading to cycles
                         for (int q = indicesToRemove.Count
- 1; q >= 0; q--)
                         {

     nextPossibleRoutes.RemoveAt(indicesToRemove[q]);
```

Calculating Path that traverses all the rooms:

```
//calculate only for a path that traverses all rooms
     List<Path> backupRoute = new List<Path>(routes[i]);

//creating new routes with next possible routes and adding
them to routes
foreach (Path p in nextPossibleRoutes)
{
List<Path> newRoute = new List<Path>(backupRoute);
newRoute.Add(p);
routes.Add(newRoute);
}
//remove original route
//routes.Remove(routes[i]);
}
iter--;
}
//Console.WriteLine("=======================================
=====");
routes = routes.Distinct().ToList();
int minMatricIndex = -1;

//creaing a new list of just the perfect paths
List<List<Path>> ChosenPaths = new List<List<Path>> ();
for (int i = 0; i < routes.Count; i++) {
if (routes [i] [0].source_room_number != 1)
continue;
if (rooms.Count - 1 == routes [i].Count) {
ChosenPaths.Add(routes[i]);
}
}

//calculating a metric *ignore the typo in the variable
name
double minMetric = 100;
for (int i = 0; i < ChosenPaths.Count; i++)
{
//calculate only for a path that traverses all rooms
double metric = 0;
for (int j = 0; j < ChosenPaths[i].Count; j++)
{
```

```
metric += ChosenPaths[i][j].length * (1 -
rooms[ChosenPaths[i][j].source_room_number -
1].RoomProbability);
Debug.Log (rooms[ChosenPaths[i][j].source_room_number -
1].RoomNumber + "=>");
}
if(minMetric > metric)
{
minMetric = metric; minMatricIndex = i;
}
Debug.Log (metric);
}
Debug.Log ("Choosing path” + minMatricIndex + " out of
total " + ChosenPaths.Count+" with metric " + minMetric);
Debug.Log ("Length of the chosen path is” + ChosenPaths
[minMatricIndex].Count);
for (int iterator = 0; iterator <
ChosenPaths[minMatricIndex].Count; iterator++) {
Room R = new
Room(ChosenPaths[minMatricIndex][iterator].source_room_numb
er,
rooms[ChosenPaths[minMatricIndex][iterator].source_room_num
ber - 1].RoomProbability);
SortedRooms.Add (R);
Debug.Log ("Added Room " + R.RoomNumber.ToString());

if(iterator == ChosenPaths[minMatricIndex].Count - 1)
{
Room R_LAST = new
Room(ChosenPaths[minMatricIndex][iterator].destin_room_numb
er,
rooms[ChosenPaths[minMatricIndex][iterator].destin_room_num
ber - 1].RoomProbability);
SortedRooms.Add (R_LAST);
Debug.Log ("Added Room " + R_LAST.RoomNumber.ToString());
}
}

}
```

For chasing the player:

```
public class EnemyController : MonoBehaviour {

    public GameObject target;
```

```
	Vector3 enemyPosition;
	Vector3 newPositionEnemy;
	float savedtime;
	// Use this for initialization
	void Start () {
		target = GameObject.Find ("Player");


		enemyPosition = transform.position;
		savedtime = Time.time;
	}

	// Update is called once per frame
	void Update () {
		Vector3 targetPosition =
target.transform.position;//Vector2.Lerp(transform.position
, new Vector2(target.transform.position.x,
target.transform.position.y), .01f);
		enemyPosition = transform.position;

		Vector3 right = new Vector3 (enemyPosition.x +
1.5f, enemyPosition.y, enemyPosition.z);
		Vector3 left = new Vector3 (enemyPosition.x -
1.5f, enemyPosition.y, enemyPosition.z);
		Vector3 up = new Vector3 (enemyPosition.x,
enemyPosition.y, enemyPosition.z + 0.75f);
		Vector3 down = new Vector3 (enemyPosition.x,
enemyPosition.y, enemyPosition.z - 0.75f);


		if (Time.time - savedtime > 1f) {


			savedtime = Time.time;



			if (Mathf.Abs (targetPosition.x -
enemyPosition.x) > Mathf.Abs (targetPosition.z -
enemyPosition.z)) {
				if (targetPosition.x >
enemyPosition.x)//move right
					transform.position = right;
				else if (targetPosition.x <
enemyPosition.x)//move left
					transform.position = left;
			} else {
```

```
                        if (targetPosition.z >
enemyPosition.z)//move up
                            transform.position = up;
                        else if (targetPosition.z <
enemyPosition.z)//move down
                            transform.position = down;
                    }



                }

            }
}
```

Updating Probability every 5 seconds:

```
Updating Probabilities every 5 seconds
            if (Time.time - lastRecordedTime > 5.0f) {
                foreach (Room room in SortedRooms) {
                    if(room.RoomProbability < 1.0f)
                        switch(room.RoomNumber){
                        case 1: room.RoomProbability +=
0.02f; break;
                        case 2: room.RoomProbability +=
0.04f; break;
                        case 3: room.RoomProbability +=
0.02f; break;
                        case 4: room.RoomProbability +=
0.02f; break;
                        case 5: room.RoomProbability +=
0.04f; break;
                        case 6: room.RoomProbability +=
0.02f; break;
                        case 7: room.RoomProbability +=
0.06f; break;
                        case 8: room.RoomProbability +=
0.05f; break;
                    }
                }
                lastRecordedTime = Time.time;
            }

            if (/*Time.time - lastRecordedTime > 20f ||*/
Vector3.Distance( navComponent.transform.position,
```

```
wayPoints[SortedRooms[index].RoomNumber - 1].position) <
1f) {
                SortedRooms[index].Visited = true;
                index = (index + 1) % wayPoints.Length;
                SortedRooms[index].RoomProbability = 0.00f;

                Debug.Log("Current Target is Room Number "+
(SortedRooms[index].RoomNumber - 1));

            }
                    RaycastHit hit;


     if(Physics.Raycast(navComponent.transform.position,
(victim.transform.position -
navComponent.transform.position), out hit, 50f))
            {

                if(hit.collider.name == "Player" ||
hit.collider.name == "Quad")
                {
                    indicator = 0;
                    navComponent.speed = 2f;

     navComponent.SetDestination(victim.transform.position)
;
                    victimsLastPosition =
victim.transform.position;


     NavMesh.CalculatePath(navComponent.transform.position,
victimsLastPosition, NavMesh.AllAreas, meshPath);
                    for (int i = 0; i <
meshPath.corners.Length-1; i++)

     Debug.DrawLine(meshPath.corners[i],
meshPath.corners[i+1], Color.red);
                }
                else
                {

                    if(indicator == 0){

     navComponent.SetDestination(victimsLastPosition);
                        navComponent.speed = 2f;
```

```
	if(Vector3.Distance(victimsLastPosition,
navComponent.transform.position) < 1.0f )
					indicator = 1;
				}

				else
				{

	navComponent.SetDestination(wayPoints[SortedRooms[inde
x].RoomNumber - 1].position);
					navComponent.speed = 1f;
				}



	NavMesh.CalculatePath(navComponent.transform.position,
navComponent.destination, NavMesh.AllAreas, meshPath);
				for (int i = 0; i <
meshPath.corners.Length-1; i++)

	Debug.DrawLine(meshPath.corners[i],
meshPath.corners[i+1], Color.red);
			}
		}
	}

	public static List<Path> getNextRouteCandidates(Path
path, List<Path> paths)
	{
		List<Path> nextRouteCandidates = new
List<Path>();
		nextRouteCandidates = paths.FindAll(r =>
r.source_room_number == path.destin_room_number);
		if (nextRouteCandidates.Count == 0)
			return null;
		return nextRouteCandidates;
	}


}
```