Experimental Analysis of Probabilistic Smart Terrain in Unity

by

Emile Boulos

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Computing and Information Systems

YOUNGSTOWN STATE UNIVERSITY

May 2018

Experimental Analysis of Probabilistic Smart Terrain in Unity

Emile Boulos

I hereby release this thesis to the public. I understand that 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:

Emile Boulos, Student Date

Approvals:

John Sullins, Thesis Advisor Date

Yong Zhang, Committee Member Date

Abdu Arslanyilmaz, Committee Member Date

Sal Sanders, Dean of Graduate Studies Date

Abstract

Smart Terrain is a unique algorithm where objects that are part of the surrounding environment play a tremendous role in giving a non-player character (NPC) important values it needs to distinguish a proper action to take to perform intelligently. A more specific example where this algorithm is used is in the game, The Sims. Objects such as the terrain and refrigerators calculate values and communicate them to the NPC to set a route it should follow to reach its goal or satisfy its desire. This implementation is better known as Probabilistic Smart Terrain because it incorporates probabilities that provide some uncertainty as to whether the target object can satisfy the NPC's need or not. This paper expresses more features of the implementation of this algorithm. This was done by adding upon Joseph Korchnak's implementation of this algorithm in a Unity 2.5D game and includes many bugs fixes, extra levels, and more established objectives to better express the versatility of the Artificial Intelligence (AI). It also contains an experimental analysis of participants that is based on opinion through survey. The game itself involves a guard, the NPC, that patrols multiple game objects (coins and the exit), all of which have different values, on a maze-like board setting. The Probabilistic Smart Terrain algorithm calculates the probability of where to go based on variables of the distance the guard is from the coin or exit, the time since they were last checked, and the value of that coin or the exit. This algorithm can then provide the artificially intelligent expression of the guard patrolling the areas of higher priority (where the player may be more likely to go) and thereby allowing the assessment of algorithm.

Acknowledgements

Thank you Dr. Sullins for the opportunity to work on this project. Thank you, Joe for your guidance in the beginnings of my participation in this research. Thank you, Google for your resources.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Motivation

The AI of a game must perform as the story-assimilated character would to effectively grasp the attention of player and to give them the feeling they are actually playing against an "intelligent" opponent. The process in the design of the AI is to add upon this persuasion of immersion into the game by adding upon what is already present in the games graphical design, story, and rules. This means that for the AI to be considered very well done, it needs to elicit the proper behaviors to the player. These behaviors can be understood through the actions of the AI (e.g. a beast fighting back when you strike it). Naturally, the process of coordinating all the pieces from the hardware level to the software level can provide some challenges, as implementing more sophisticated algorithms can be very costly. Nonetheless, AI is a key factor for the player's immersion into a game and will continue having great importance for any designer attempting to create a seemingly intelligent NPC or obstacle. This paper builds on Probabilistic Smart Terrain (Sullins 2010) by furthering the implementation of the algorithm that was started by Joe Korchnak (which itself was built on a tutorial for Unity called 2D Roguelike) into a more full-fledged game to better assess the AI's versatility, with multiple levels and even multiple guards in one of the levels.

1.2 Contributions

1. The main contributions of this thesis are the extra features added to transform the attempted game into a more full-fledged game, which can be added upon and tested further to better understand how well this algorithm fairs with players. The original attempt ran into some bugs with the guards and had no obtainable objective for the player to participate in. To assess the capacity of the guards' seeming intelligences, I first had to finish these aspects of the game as well as add other features, so players and researchers could better assess the potential of the algorithm. This included giving players the ability to "pick up" coins and remove them from the game.
2. Make this knowledge part of the guard's behavior, causing them to follow routes that no longer involved removed coins.
3. Adding "exits" to the next level and making them a "goal" to be patrolled by the guards similar to the coins.
4. Giving guards the ability to end the game by catching the player, giving players the motivation to understand the guards' behavior well enough to avoid them.
5. Creating multiple levels of increasing difficulty.

As mentioned before, the game was originally a tutorial for Unity and was built upon using the same assets, but with our own algorithmic features and objectives ("Unity - 2D Roguelike Tutorial" 2016). Should any further testing or modification be desired by any

researcher or player, all the information on the features and the main algorithm are shown below.

Another contribution of this thesis was to explore ways for multiple guards to appear to work together without having them communicate directly. This was done by embedding the desired behavior in the design of a level with multiple guards. Adding such intelligence to the world the characters exist in (rather than to the characters themselves) is the basis of smart terrain.

One final contribution is the design of a procedure to have actual players evaluate the apparent intelligence of the AI, the main way to test the playability of that AI in a game. This includes a script for familiarizing players with the game, and a set of questions that players would be asked after playing the game.

1.3 Research Questions

The main research questions were related to the player evaluation of the probabilistic smart terrain algorithm, including;

1. Did guards prioritize guarding the most valuable treasures (as well as the level exit)?
2. Did guards change their behavior as the player removed treasures, moving to cover treasures not yet stolen by the player?
3. Did multiple guards on the same level appear to "cooperate", each covering their own area within a level?

1.4 Organization

We begin with Section 1: Introduction, the overview of the thesis. Section 2: Background and Related Work, which explains a little about the algorithm, its predecessor, and its beginning implementation in a Unity 2D game. Section 3: Further Implementation in an Unity 2D game explains all of the extra features added to make the game playable. Section 4: Experimental Analysis demonstrates how the game meets the goals of the algorithm, as well as suggestions for surveying actual players in the future. Section 5: Future Work gives suggestions to better the game and its capacity at implementing this algorithm.

Chapter 2

BACKGROUND AND RELATED WORK

2.1 Smart Terrain

A predecessor to the Probabilistic Smart Terrain idea is the "Smart Terrain" idea (Doyle, 1999). Used in the game, *The Sims* (Doornbos 2001), the characters of the game follow routes toward specific objects that could satisfy their "needs". One of these needs is to satisfy their own hunger when they have not eaten in a while and this can be done by following a route to an object that can satisfy this need, a refrigerator. This only captures one of the many needs that these characters can possess. The process of having multiple separate objects that these characters can route towards to satisfy these separate needs is what makes up "Smart Terrain". The key feature that uniquely defines the exclusiveness

of “Smart Terrain” is having the artificially intelligent portion placed not in the character, but in the terrain itself, giving guiding signals to character about where to go.

2.2 Probabilistic Smart Terrain

Probabilistic Smart Terrain builds upon “Smart Terrain” by including multiple objects within the route a character could take and by altering the certainty that an object can satisfy a need. Characters would then be encouraged to alter their routes to explore objects that might meet their needs. This can allow an NPC to move more intelligently and once again, add to the immersion of a player into the game.

The equation behind the values given to the tiles is a modified form of the original Probabilistic Smart Terrain, called the follow equation (Sullins 2010). The original equation is expressed below and following it is the modified form expressed in C#.

$$\mathbf{B}(p_i, a_i, K_i, t) = a_i - (a_i - p_i) / (1 + e^{-t + K_i})$$

- p_i is the original probability that object i meets the need, as defined by the level designer.
- a_i is the modified probability (either 0 or 1) created once the object was explored.
- t is the number of time units since the character last explored the object.
- k_i is the *expected time to knowledge expiration* for object i.

```
private double CalculateTileWeights(int tile)
{
    double answer =
        (0 - (0 - ValueOfGold) / (1 + Mathf.Exp ((float)(-TimeGold + RemeberGold)))) / GoldArray[tile] +
        (0 - (0 - ValueOfSilver) / (1 + Mathf.Exp ((float)(-TimeSilver + RemeberSilver)))) / SilverArray [tile]  +
        (0 - (0 - ValueOfCopper) / (1 + Mathf.Exp ((float)(-TimeCopper + RemeberCopper)))) / CopperArray [tile] +
        (0 - (0 - ValueOfExit) / (1 + Mathf.Exp ((float)(-TimeExit + RemeberExit)))) / ExitArray [tile];

    return answer;
}
```

In this game, we have a_i set to 0 as the original probability and p_i as the value of the objects being patrolled. The t variable counts the number of time units since the NPC last checked the object and is calculated with k_i to impact how long the NPC should wait before checking that object again. To finalize this equation, the distance of tiles between each object are represented at the divisor of each equation. This and the other two components assess the worthiness of each game object on the board.

2.3 Unity

This game is a continuation of the one started by Joseph Korchnak. He used the assets from the Unity tutorial, Roguelike 2D, and implemented the Probabilistic Smart Terrain algorithm into those assets. With this game, the guard would patrol an efficient route based on the value of the object, the distance the guard is from the object, and a timing variable since the object was last checked. He created one level, comprising of a maze-like board of his own layout of the wall tiles and coins (Figure 1). The player could also move tile to tile. The algorithm was implemented, but the game was unfinished and left in this state.

Figure 1: Original game

(Unity versions used to alter this game were 5.4.1f1, 5.5.0f3 and 2017.1.0f3.)

Chapter 3

Further Implementation in a Unity 2D game

The previous work (Korchnak, 2018) did a great deal towards implementing a game based on probabilistic smart terrain, but was lacking in many key features to make the game testable or more importantly, playable. The original game did not allow players to pick up the coins or have any value attached to increment the score should the player be able to obtain them, a key requirement in motivating the player to move towards the coins. There

was no function for the guard to penalize the player and if the player got within a range from the guard, the guard would simply stop, giving the player no incentive to avoid guards. There were no extra levels following the first, important to evaluate the performance of the AI in different game configurations. To begin the assessment process of this algorithm from a gaming perspective, these characteristics and more needed to be fulfilled.

3.1 Coins

To begin, I coded the ability to pick up coins and have the coins give a value to the player's score, thus giving the player their objective. Coins in the original level were moved around and extra coins were added to allow the guard to take a more intricate route when patrolling (Figure 2). I also added the exit tile to the calculations of the algorithm, so that the guard would also patrol the exit, making the game a little more challenging. This suggests more attention from the player of the guards' movements will be necessary to participate in the game well. Players could now be attentive in avoiding the guards, steal treasures, and make their way to the exit.

Figure 2: Coins Obtainable, Extra Coins, Exit Tile, and Score Display!

3.2 Guards

The zombie-looking creature on the board acts as the guard of the game, which patrols the coins and the exit. The first of my alterations to the guard is the ability to continue pursuing their route to patrol the coins even when approaching the player. Originally, the guard would keep still when within a certain range of the player. The issue was in the chase algorithm which was competing for priority over the Probabilistic Smart Terrain algorithm. The issue was fixed upon the removal of the function.

The second alteration to the guard is the ability to prevent the player from reaching their goal. Originally, upon approaching the guard, nothing would harm the player, so there was nothing to stop the player from the objective. After my modifications, should the

player come within a 2-tile radius of the guard, the player is caught and the game was over (Figure 3). A unique quality of this feature is that the guard can “hear” the player through wall tiles if they are close enough (Figure 3). With the guard now able to be a threat to the player, the game has become “playable”.

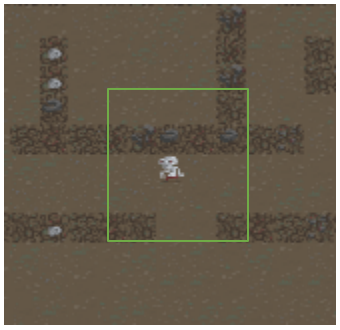

Figure 3: 2-tile Radius of Death

I have also altered the “personalities” of the guards across the levels by changing their time delays. This not only allows the guard to move faster or slower depending on my preference, but also has a great impact on their apparent intelligence. For the Probabilistic Smart Terrain algorithm to work properly in a dynamic sense, three variables must be incorporated in its calculation. The value of the object, the distance the guard is from the object, and the time since the guard last visited the object are these three variables. Should the player steal one or more of the coins, the guard should then dynamically alter their route to guard the other coins instead of patrolling empty areas. When shortening the time delay of the guard, the guard would move from tile to tile more quickly because the updates for the guard would be happening more rapidly. This would not allow the timing variable to increment high enough to impact the algorithm to move the guard more intelligently, but more predictably. When moving quickly, the guard will repeat the same route to areas

where coins were programmed to be, even after being retrieved by the player. This gives the guard a more “frantic” or “foolish” personality trait (Figure 4). Upon reaching the final level, the time delay is much greater, so the timing variable has a much greater impact on the algorithm. The guards in this level move slowly but alter their path when a coin is stolen. The guards will rarely revisit that area, making them seem more “cautious” and “intelligent”. To summarize, the time delay becomes greater as you pass through each of the three levels, which makes the guards move slower, but more intelligently.

Figure 4: “Foolish Guard”

3.3 Boards Levels

To further test the versatility of this algorithm, a setting that would require greater attention of the guards’ movements by the player was provided (Figure 6, 7, 8). Each level

was designed manually and each with more complexity than the last. The arrangement of the coins and exit, the spawning locations of the guards, and the arrangement and amount of wall tiles per level were all experimented with to test how the algorithm would work and prioritize the different coins, as well as to prevent the guards from hovering over or around any specific object because of its algorithmic value being too great. Note that tweaking in-game parameters to improve playability is a standard part of game design.

Figure 5: Level One

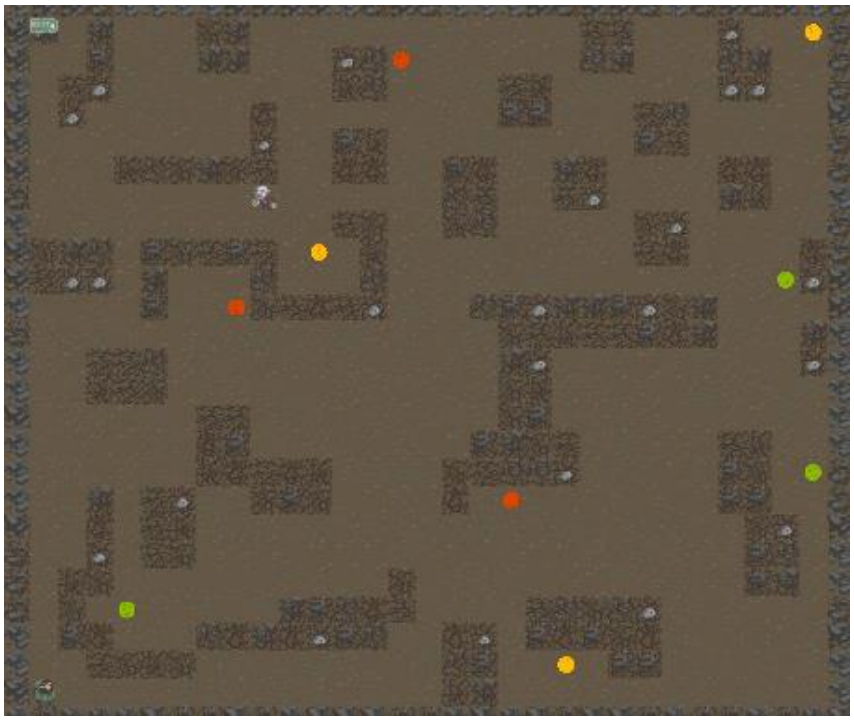

Figure 6: Level Two

Figure 7: Level Three

3.4 Multiple Guards and Emergent Behavior

As described above, the final level included multiple guards, a common feature in many games. In such games, players expect those guards to "cooperate" in certain ways, possibly by splitting up to cover different areas, or at least patrolling separate sections of the level. This is very important, as guards that all tried to patrol the same area of a level would not appear to be "intelligent".

This concept is very difficult to implement in probabilistic smart terrain, however, as each non-player character bases its movement on the locations and value of the coins, and not on the locations and movements of other NPCs. Some work has been done in modifying probabilistic smart terrain in this direction (Sullins 2011), but comes at a cost to the efficiency of the algorithm. The NPCs would either have to communicate with a complex central system that coordinated the actions of all of them, or communicate separately with every other NPC every move.

I designed an alternative method, based on the main concept of smart terrain, in which "intelligence" is embedded in the world around the characters rather than the characters themselves (Doyle, 1999). Specifically:

- The level is designed in a more "maze-like" structure, that increases the actual distances between the two guards, in terms of tiles they would have to move to reach each other's area.
- Coins of high value were also placed in such a way that each guard would be forced to return to them on a regular basis, as their timeouts expired.

The combined result of this was to effectively “restrict” each guard’s movements to a specific area of the level, as shown in Figure 8.

Figure 8: Localized Guards

This also has aspects of “emergence”, an AI concept that refers to groups of nonintelligent entities combining in such a way that the group as a whole appears to be “intelligent”. Even though there is no actual cooperation between the guards, they should appear to be working together to guard the level.

Chapter 4

Experimental Analysis

4.1 Implementation Results

Once the changes in the previous chapter were implemented, we then ran the game to make sure each of the main objectives of the AI were accomplished.

4.1.1 Prioritization

As expected, the guards visited gold coins more often than silver coins, and silver coins more often than copper coins, due to the gold coins being given the highest probability, the silver coins being given a lower probability, and copper coins being given the lowest probability. In addition, the guards also regularly visited the exit tile that was added, as that was given the highest probability of all. This matches how players should expect an “intelligent” guard NPC to act, trying to guard the more valuable items while keeping the player from escaping.

4.1.2 Path Modification

The modification that allowed players to pick up the coins successfully resulted in the guards changing their paths to no longer include coins that were no longer there, and focus more on coins that still remained.

One very interesting effect of this was that the guards began to check the exit more frequently as the coins were taken. While this was simply an effect of fewer other objects attracting the guard away, it gave the impression of a guard seemingly changing its priorities from guarding coins to preventing the player from escaping. In particular, if the player managed to steal all of the coins the guard seemed to "race" towards the exit and hover there to cut off the player, due to the exit being the only object that attracted it (and of course, being the only object the player was unable to steal).

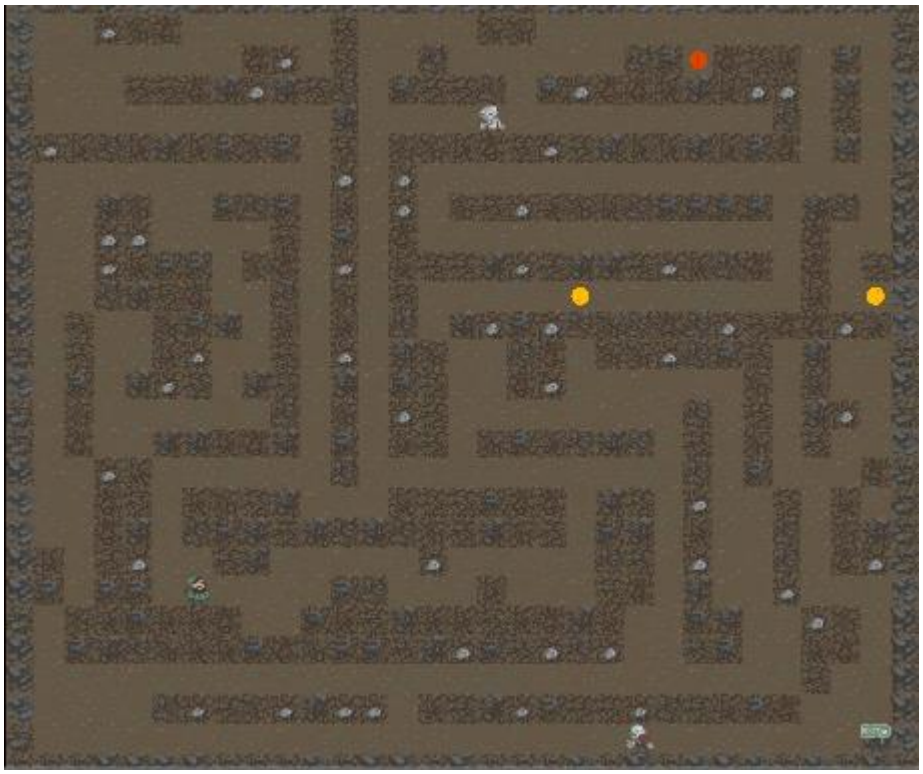

Figure 9: Less Coins, More Caution

4.1.3 Multiple Guards

In the case of multiple guards, we did observe that each kept to its own "area" of the level, repeatedly moving between the coins in its vicinity (Figure 8). It did take some work in experimenting with different level configurations of the level and different coin placements, but this would be something we would expect a level designer of an actual game to do in order to maximize the playability of a level.

The only time this behavior changed was when the player stole most or all of the coins from the vicinity of one of the guards. At that point, the only objects left would be the coins in the area of the other guard or the exit, and the guard would begin to move in

one of those directions (Figure 9). However, this was also plausible behavior on the part of an “intelligent” guard, as the least intelligent action would be to continue to patrol an area with no coins left.

Finally, if the player manages to steal most of the coins on this entire level then both guards will hover around the exit. This was the one case where it was impossible to keep the guards from colliding, as there was no other object left to draw them away from the exit. This might still make sense from the point of view of a player, however, as it could look like the guards are “teaming up” to prevent the player from escaping.

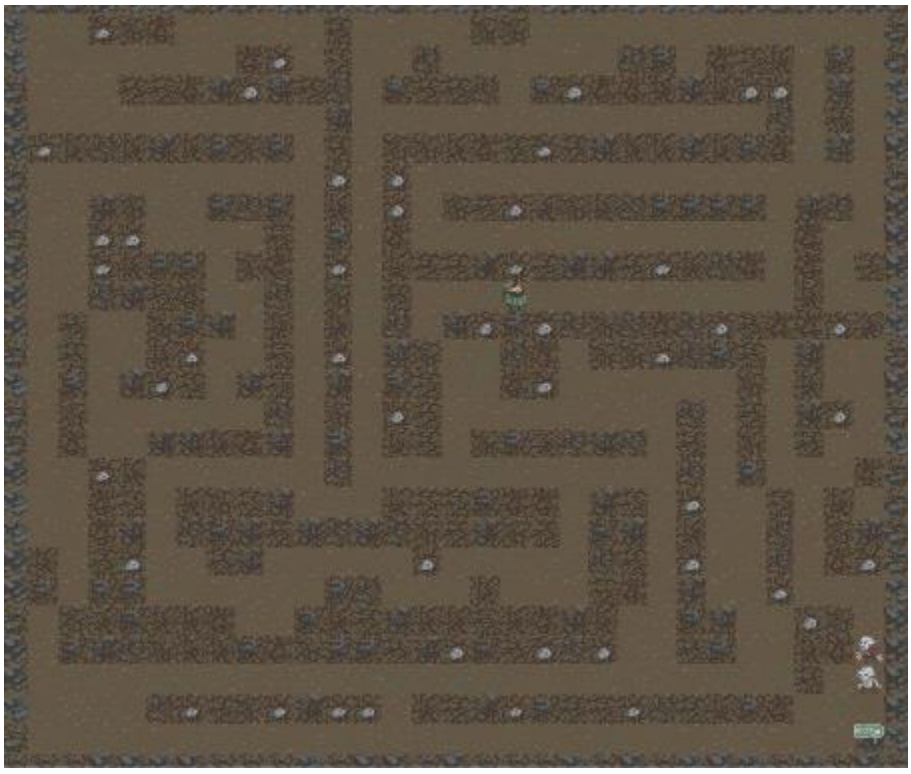

Figure 10: Is it too late to give back the coins?

4.2 Outline for Future Player Evaluation of Game

The main goal of artificial intelligence in a game is to provide the player with an immersive experience by providing opponent non-player characters that make choices the player considers "intelligent". The most effective way to evaluate whether or not the game AI accomplishes this is simply to let players play the game, and then give their opinions of the apparent intelligence of the NPCs. To test the effectiveness of this algorithm from a gaming sense, we will need participants to play the game and give us their opinions on the guards.

Any such participants will need instruction on the game before the test can begin. We have developed a small script and pictures (Figure 9, 10) to explain the game (not the algorithm) to the participant:

"Here we have a tile-based game on a maze-like board in 2.5D. This means that the game is 2-dimensional, but is played from an aerial view of the game, so the character can move up, down, left, and right. Even though this game is tiled-based, the tiles that make up the board are not visible as separate tiles. The characters make their movements in the manner of one tile to the next, okay?"

"You only move with the arrow keys shown on the keyboard (point them out) and your objective is to steal all of the coins in the level before going to the exit. There are three

levels and each level has an exit and several gold, silver, and copper coins. The silver coins are portrayed as green-colored coins and the copper coins as red-colored coins."

"Something important to mention are the guards that attempt to stop you from achieving your goal by guarding the coins. If you fall within the range of 'hearing', it's game over. Yep, I said hearing. The guards are blind zombies that listen for their prey and can hear two tile spaces away from themselves (show picture). Also, if you think you are safe hiding behind a wall from your opponent, you are unless the wall is only a single layer (show picture). The guards can hear you through this and will eat you so please try your best at staying out of its range."

"Your player is a sneaky thief who takes pride in his work, but is obviously very afraid of the guards so he moves cautiously and will scratch at the walls if you run into them out of desperation to get out."

"Please forgive the lag in-between the loading of the levels. There is a lot the gaming engine is doing in the background. You have three attempts you can take to get through all the levels. Again, please stay out of range of the guard and be attentive to its movements. I will explain the game further upon your completion of the survey. Thank you. Any questions?"

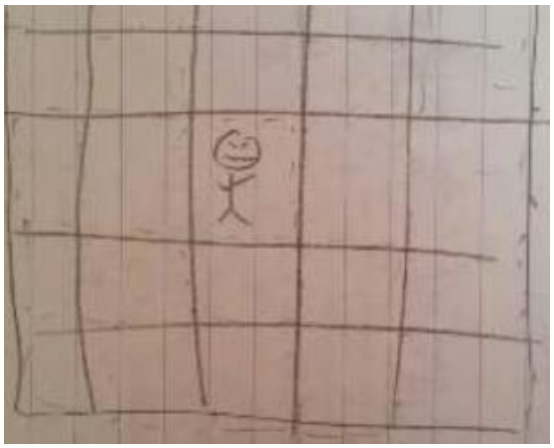

Figure 11: Zombie's Range of "Hearing"

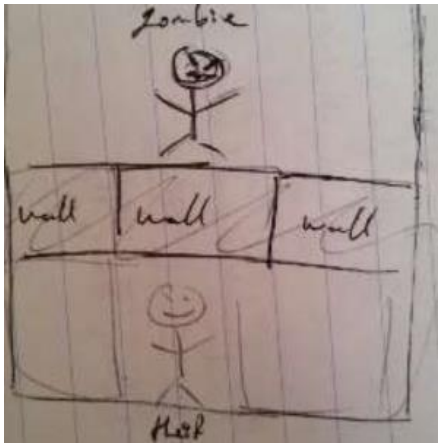

Figure 12: Range of "Hearing" through Wall

We also propose a set of questions to be asked after the game is played, meant for players to give their evaluation of the apparent intelligence of the guards. These would include general questions about the difficulty of the game, and specific questions about 1) how predictable of random the guard paths appear, 2) how they appear to prioritize the gold, silver, and copper coins as well as the exit, and 3) how well multiple guards appear

to cooperate. This could be a Likert-scaled survey, with 1 being “strongly disagree” and 5 being “strongly agree”:

Guards are a challenge to avoid	1 2 3 4 5
It was difficult to steal all of the coins	1 2 3 4 5
It was difficult to escape to the next level	1 2 3 4 5
Guards appear to be moving intelligently	1 2 3 4 5
Guards appear to be moving randomly	1 2 3 4 5
Guards appear to be moving in predictable pattern	1 2 3 4 5
Guards appear to prioritize gold	1 2 3 4 5
Guards appear to prioritize silver	1 2 3 4 5
Guards appear to prioritize copper	1 2 3 4 5
Guards appear to prioritize the exit	1 2 3 4 5
Multiple guards made the game harder	1 2 3 4 5
Multiple guards did well at covering entire level	1 2 3 4 5
Guards on same level appear to cooperate	1 2 3 4 5

Chapter 5

Future Work

The main idea to consider is to better create features around this algorithm to perform at a higher and more convincing capacity.

1. Have participants test the game for statistical analysis. This will help guide the process of what features to add and modify to promote the believability of the AI. It will also allow for a better comparison between these two groups and their separate believability in the AI.
2. There was an attempt to create even shorter paths or to block the exit with wall tiles so that the wall breaking function would be of use to the player, but the guard would

not dynamically view the wall broken as a new floor tile that could be walked upon. I thought it would be interesting to give the exit tile an extremely high value that the terrain would only send the guard if the player should break wall tiles surrounding the entirety of the exit, which would create a new path of value to the guard. Unfortunately, the tiles are seemingly preloaded before the level begins and would not register as floor tiles when the wall tiles are broken. It could be of interest to find a way to add this feature, so that players can notice the guard taking new routes according to the shorter paths they may have created.

3. Level 3 was the closest attempt to success I had in implementing the use of two guards that would coordinate patrolling the treasures and exit of the level. There is still the issue of their collision, which keeps them both at a standstill for long periods of time or for the rest of the duration of the level. It would help if there was some instruction given to guards when running into other guards of what to do instead of standing still. The probabilistic smart terrain algorithm could also be modified to base actions on the other NPCs as well as the goals (Sullins 2011), but at a cost of greater algorithm complexity and possibly slower run times.
4. Modify the manual setup of the original levels or add another level to have for participants for greater accuracy in our analyses of the performance and believability of the Probabilistic Smart Terrain algorithm.

APPENDIX - Code

```
using UnityEngine;
using System.Collections;


using System.Collections.Generic;       //Allows us to use Lists
.
using UnityEngine.UI;                   //Allows us to use UI.

public class GameManager : MonoBehaviour
{
    public float levelStartDelay = 2f;                          //T
ime to wait before starting level, in seconds.
    public float turnDelay = 0.1f;                              //D
elay between each Player turn.
    public int playerScore = 0;                                  //
Starting value for Player score points.
    public static GameManager instance = null;                  //S
tatic instance of GameManager which allows it to be accessed by a
ny other script.
    [HideInInspector] public bool playersTurn = true;        //Bo
olean to check if it's players turn, hidden in inspector but publ
ic.


    public int enemyDelay = 25;                                  //
delay before the enemy can move
    public int playerDelay = 20;


    private Text levelText;                                      //
Text to display current level number.
    private GameObject levelImage;                              //I
mage to block out level as levels are being set up, background fo
r levelText.
    private BoardManager boardScript;                          //St
ore a reference to our BoardManager which will set up the level.
    private int level = 3;                                      //C
urrent level number, expressed in game as "Day 1".
    private List<Enemy> enemies;                              //Lis
t of all Enemy units, used to issue them move commands.
    private bool doingSetup = true;                              //
Boolean to check if we're setting up board, prevent Player from m
oving during setup.
```

```
    //level 1 dumb guard who runs around checking coins not payin
g attention to anything (bc moving too fast to register timeout)
    //level 2 moves slower but smarter
    //level 3 moves even slower but even smarter, will change rou
te (bc can register the timing)


    //Awake is always called before any Start functions
    void Awake()
    {
        //Check if instance already exists
        if (instance == null)

            //if not, set instance to this
            instance = this;

        //If instance already exists and it's not this:
        else if (instance != this)

            //Then destroy this. This enforces our singleton patt
ern, meaning there can only ever be one instance of a GameManager
.
            Destroy(gameObject);

        //Sets this to not be destroyed when reloading scene
        DontDestroyOnLoad(gameObject);

        //Assign enemies to a new List of Enemy objects.
        enemies = new List<Enemy>();

        //Get a component reference to the attached BoardManager
script
        boardScript = GetComponent<BoardManager>();

        //Call the InitGame function to initialize the first leve
l
        InitGame();
    }

    //This is called each time a scene is loaded.
    void OnLevelWasLoaded(int index)
```

```
    {
        //Add one to our level number.
        level++;
        //Call InitGame to initialize our level.
        InitGame();
    }

    //Initializes the game for each level.
    void InitGame()
    {

        //While doingSetup is true the player can't move, prevent
 player from moving while title card is up.
        doingSetup = true;

        //Get a reference to our image LevelImage by finding it b
y name.
        levelImage = GameObject.Find("LevelImage");

        //Get a reference to our text LevelText's text component
by finding it by name and calling GetComponent.
        levelText = GameObject.Find("LevelText").GetComponent<Tex
t>();

        //Set the text of levelText to the string "Day" and appen
d the current level number.
        levelText.text = "Day " + level;

        //Set levelImage to active blocking player's view of the
game board during setup.
        levelImage.SetActive(true);

        //Call the HideLevelImage function with a delay in second
s of levelStartDelay.
        Invoke("HideLevelImage", levelStartDelay);

        //Clear any Enemy objects in our List to prepare for next
 level.
        enemies.Clear();

        //Call the SetupScene function of the BoardManager script
, pass it current level number.
        boardScript.SetupScene(level);
```

```
    }


    //Hides black image used between levels
    void HideLevelImage()
    {
        //Disable the levelImage gameObject.
        levelImage.SetActive(false);

        //Set doingSetup to false allowing player to move again.
        doingSetup = false;
    }

    //Update is called every frame.
    void Update()
    {
        //Check that playersTurn or enemiesMoving or doingSetup a
re not currently true.
        if(doingSetup)

        //If any of these are true, return and do not start MoveE
nemies.
        return;

        //Start moving enemies.
        if (enemyDelay == 0) {

            StartCoroutine (MoveEnemies ());

            if (level == 3) {

                enemyDelay = 45;
            }
            else if (level == 2) {

                enemyDelay = 25;
            }

            else{

            enemyDelay = 19;
```

```
            }
        }

        enemyDelay--;

        if (playerDelay == 0)
        {
            playersTurn = true;

            playerDelay = 20;

        }

        playerDelay--;

    }

    //Call this to add the passed in Enemy to the List of Enemy o
bjects.
    public void AddEnemyToList(Enemy script)
    {
        //Add Enemy to List enemies.
        enemies.Add(script);
    }


    //GameOver is called when the player reaches 0 food points
    public void GameOver()
    {
        //Set levelText to display number of levels passed and ga
me over message
        levelText.text = "After " + level + " day(s), you died.";

        //Enable black background image gameObject.
        levelImage.SetActive(true);

        //Disable this GameManager.
        enabled = false;
    }

    public void YouWin()
    {
        if(playerScore<1375)
```

```
        {
            //Set levelText to display number of levels passed an
d game over message
            levelText.text = " Not enough treasure. You Lose!!";

            //Enable black background image gameObject.
            levelImage.SetActive(true);

            //Disable this GameManager.
            enabled = false;
        }
        else
        {
        //Set levelText to display number of levels passed and ga
me over message
        levelText.text = "You Win!";

        //Enable black background image gameObject.
        levelImage.SetActive(true);

        //Disable this GameManager.
        enabled = false;
        }
    }

    //Coroutine to move enemies in sequence.
    IEnumerator MoveEnemies()
    {

        //Wait for turnDelay seconds, defaults to .1 (100 ms).
        yield return new WaitForSeconds(turnDelay);

        //If there are no enemies spawned (IE in first level):
        if (enemies.Count == 0)
        {
            //Wait for turnDelay seconds between moves, replaces
delay caused by enemies moving when there are none.
            yield return new WaitForSeconds(turnDelay);
        }

        //Loop through List of Enemy objects.
        for (int i = 0; i < enemies.Count; i++)
        {
```

```
            //Call the MoveEnemy function of Enemy at index i in
the enemies List.
            enemies[i].MoveEnemy ();

            //Wait for Enemy's moveTime before moving next Enemy,

            yield return new WaitForSeconds(enemies[i].moveTime);
        }
    }
}




using UnityEngine;
using System;
using System.Collections.Generic;         //Allows us to use List
s.
using Random = UnityEngine.Random;         //Tells Random to use
the Unity Engine random number generator.



public class BoardManager : MonoBehaviour
{
    // Using Serializable allows us to embed a class with sub pro
perties in the inspector.
    [Serializable]
    public class Count
    {
        public int minimum;             //Minimum value for our C
ount class.
        public int maximum;             //Maximum value for our C
ount class.


        //Assignment constructor.
        public Count (int min, int max)
        {
            minimum = min;
            maximum = max;
        }
    }
```

```
    public int columns = 100;
     //Number of columns in our game board.
    public int rows = 100;
     //Number of rows in our game board.
    public Count wallCount = new Count (5, 9);
     //Lower and upper limit for our random number of walls per l
evel.
    public Count foodCount = new Count (1, 5);
     //Lower and upper limit for our random number of food items
per level.**************
    public GameObject exit;
   //Prefab to spawn for exit.
    public GameObject gold;
    public GameObject silver;
    public GameObject copper;
    public GameObject[] floorTiles;
      //Array of floor prefabs.
    public GameObject[] wallTiles;
     //Array of wall prefabs.
    public GameObject[] enemyTiles;
      //Array of enemy prefabs.
    public GameObject[] outerWallTiles;
      //Array of outer tile prefabs.

    public FloodTiles floodScript;

    private Transform boardHolder;
     //A variable to store a reference to the transform of our Bo
ard object.
    private Transform wallHolder;
    private List <Vector3> gridPositions = new List <Vector3> ();
    //A list of possible locations to place tiles.
    private List <Vector3> wallPositions = new List <Vector3> ();


    //Clears our list gridPositions and prepares it to generate a
 new board.
    void InitialiseList ()
    {

        //Clear our list gridPositions.
```

```
        gridPositions.Clear ();
        makeWallList ();


        //Loop through x axis (columns).
        for(int x = 0; x < columns-1; x++)
        {
            //Within each column, loop through y axis (rows).
            for(int y = 0; y < rows-1; y++)
            {
                //At each index add a new Vector3 to our list wit
h the x and y coordinates of that position.
                gridPositions.Add (new Vector3(x, y, 0f));
            }
        }
    }

    void InitialiseList_2 ()
    {

        //Clear our list gridPositions.
        gridPositions.Clear ();
        wallPositions.Clear ();
        makeWallList_2 ();

        //Loop through x axis (columns).
        for(int x = 0; x < columns-1; x++)
        {
            //Within each column, loop through y axis (rows).
            for(int y = 0; y < rows-1; y++)
            {
                //At each index add a new Vector3 to our list wit
h the x and y coordinates of that position.
                gridPositions.Add (new Vector3(x, y, 0f));
            }
        }
    }

    void InitialiseList_3 ()
    {

        //Clear our list gridPositions.
        gridPositions.Clear ();
```

```
        wallPositions.Clear ();
        makeWallList_3 ();

        //Loop through x axis (columns).
        for(int x = 0; x < columns-1; x++)
        {
            //Within each column, loop through y axis (rows).
            for(int y = 0; y < rows-1; y++)
            {
                //At each index add a new Vector3 to our list wit
h the x and y coordinates of that position.
                gridPositions.Add (new Vector3(x, y, 0f));
            }
        }
    }


    void makeWallList_3()
    {
        wallPositions.Add (new Vector3(4, 1, 0));
        wallPositions.Add (new Vector3(5, 1, 0));
        wallPositions.Add (new Vector3(6, 1, 0));
        wallPositions.Add (new Vector3(7, 1, 0));
        wallPositions.Add (new Vector3(8, 1, 0));
        wallPositions.Add (new Vector3(9, 1, 0));
        wallPositions.Add (new Vector3(10, 1, 0));
        wallPositions.Add (new Vector3(11, 1, 0));
        wallPositions.Add (new Vector3(13, 1, 0));
        wallPositions.Add (new Vector3(14, 1, 0));
        wallPositions.Add (new Vector3(15, 1, 0));
        wallPositions.Add (new Vector3(16, 1, 0));
        wallPositions.Add (new Vector3(17, 1, 0));
        wallPositions.Add (new Vector3(18, 1, 0));
        wallPositions.Add (new Vector3(19, 1, 0));
        wallPositions.Add (new Vector3(20, 1, 0));
        wallPositions.Add (new Vector3(21, 1, 0));
        wallPositions.Add (new Vector3(22, 1, 0));
        wallPositions.Add (new Vector3(23, 1, 0));
        wallPositions.Add (new Vector3(24, 1, 0));
        wallPositions.Add (new Vector3(25, 1, 0));

        wallPositions.Add (new Vector3(26, 2, 0));
```

```
wallPositions.Add (new Vector3(1, 3, 0));
wallPositions.Add (new Vector3(2, 3, 0));
wallPositions.Add (new Vector3(3, 3, 0));
wallPositions.Add (new Vector3(4, 3, 0));
wallPositions.Add (new Vector3(5, 3, 0));
wallPositions.Add (new Vector3(6, 3, 0));
wallPositions.Add (new Vector3(7, 3, 0));
wallPositions.Add (new Vector3(8, 3, 0));
wallPositions.Add (new Vector3(9, 3, 0));
wallPositions.Add (new Vector3(10, 3, 0));
wallPositions.Add (new Vector3(11, 3, 0));
wallPositions.Add (new Vector3(12, 3, 0));
wallPositions.Add (new Vector3(13, 3, 0));
wallPositions.Add (new Vector3(14, 3, 0));
wallPositions.Add (new Vector3(15, 3, 0));
wallPositions.Add (new Vector3(16, 3, 0));
wallPositions.Add (new Vector3(17, 3, 0));
wallPositions.Add (new Vector3(18, 3, 0));
wallPositions.Add (new Vector3(19, 3, 0));
wallPositions.Add (new Vector3(22, 3, 0));
wallPositions.Add (new Vector3(23, 3, 0));
wallPositions.Add (new Vector3(26, 3, 0));
wallPositions.Add (new Vector3(27, 3, 0));

//wallPositions.Add (new Vector3(0, 4, 0));
wallPositions.Add (new Vector3(1, 4, 0));
wallPositions.Add (new Vector3(2, 4, 0));
wallPositions.Add (new Vector3(3, 4, 0));
wallPositions.Add (new Vector3(4, 4, 0));
wallPositions.Add (new Vector3(5, 4, 0));
wallPositions.Add (new Vector3(6, 4, 0));
wallPositions.Add (new Vector3(9, 4, 0));
wallPositions.Add (new Vector3(10, 4, 0));
wallPositions.Add (new Vector3(11, 4, 0));
wallPositions.Add (new Vector3(12, 4, 0));
wallPositions.Add (new Vector3(13, 4, 0));
wallPositions.Add (new Vector3(14, 4, 0));
wallPositions.Add (new Vector3(15, 4, 0));
wallPositions.Add (new Vector3(16, 4, 0));
wallPositions.Add (new Vector3(17, 4, 0));
wallPositions.Add (new Vector3(18, 4, 0));
wallPositions.Add (new Vector3(19, 4, 0));
wallPositions.Add (new Vector3(22, 4, 0));
```

```
wallPositions.Add (new Vector3(23, 4, 0));
wallPositions.Add (new Vector3(26, 4, 0));
wallPositions.Add (new Vector3(27, 4, 0));

wallPositions.Add (new Vector3(0, 5, 0));
wallPositions.Add (new Vector3(2, 5, 0));
wallPositions.Add (new Vector3(3, 5, 0));
wallPositions.Add (new Vector3(10, 5, 0));
wallPositions.Add (new Vector3(11, 5, 0));
wallPositions.Add (new Vector3(15, 5, 0));
wallPositions.Add (new Vector3(19, 5, 0));
wallPositions.Add (new Vector3(20, 5, 0));
wallPositions.Add (new Vector3(22, 5, 0));
wallPositions.Add (new Vector3(25, 5, 0));

wallPositions.Add (new Vector3(0, 6, 0));
wallPositions.Add (new Vector3(2, 6, 0));
wallPositions.Add (new Vector3(3, 6, 0));
wallPositions.Add (new Vector3(6, 6, 0));
wallPositions.Add (new Vector3(7, 6, 0));
wallPositions.Add (new Vector3(19, 6, 0));
wallPositions.Add (new Vector3(20, 6, 0));
wallPositions.Add (new Vector3(13, 6, 0));
wallPositions.Add (new Vector3(22, 6, 0));
wallPositions.Add (new Vector3(25, 6, 0));
wallPositions.Add (new Vector3(27, 6, 0));
wallPositions.Add (new Vector3(28, 6, 0));

wallPositions.Add (new Vector3(2, 7, 0));
wallPositions.Add (new Vector3(3, 7, 0));
wallPositions.Add (new Vector3(5, 7, 0));
wallPositions.Add (new Vector3(6, 7, 0));
wallPositions.Add (new Vector3(7, 7, 0));
wallPositions.Add (new Vector3(8, 7, 0));
wallPositions.Add (new Vector3(9, 7, 0));
wallPositions.Add (new Vector3(10, 7, 0));
wallPositions.Add (new Vector3(11, 7, 0));
wallPositions.Add (new Vector3(12, 7, 0));
wallPositions.Add (new Vector3(13, 7, 0));
wallPositions.Add (new Vector3(14, 7, 0));
wallPositions.Add (new Vector3(15, 7, 0));
wallPositions.Add (new Vector3(16, 7, 0));
wallPositions.Add (new Vector3(17, 7, 0));
```

```
wallPositions.Add (new Vector3(19, 7, 0));
wallPositions.Add (new Vector3(20, 7, 0));
wallPositions.Add (new Vector3(22, 7, 0));
wallPositions.Add (new Vector3(25, 7, 0));
wallPositions.Add (new Vector3(27, 7, 0));
wallPositions.Add (new Vector3(28, 7, 0));

wallPositions.Add (new Vector3(2, 8, 0));
wallPositions.Add (new Vector3(3, 8, 0));
wallPositions.Add (new Vector3(5, 8, 0));
wallPositions.Add (new Vector3(6, 8, 0));
wallPositions.Add (new Vector3(7, 8, 0));
wallPositions.Add (new Vector3(8, 8, 0));
wallPositions.Add (new Vector3(12, 8, 0));
wallPositions.Add (new Vector3(13, 8, 0));
wallPositions.Add (new Vector3(14, 8, 0));
wallPositions.Add (new Vector3(15, 8, 0));
wallPositions.Add (new Vector3(16, 8, 0));
wallPositions.Add (new Vector3(17, 8, 0));
wallPositions.Add (new Vector3(19, 8, 0));
wallPositions.Add (new Vector3(20, 8, 0));
wallPositions.Add (new Vector3(22, 8, 0));
wallPositions.Add (new Vector3(25, 8, 0));
wallPositions.Add (new Vector3(27, 8, 0));

wallPositions.Add (new Vector3(2, 9, 0));
wallPositions.Add (new Vector3(3, 9, 0));
wallPositions.Add (new Vector3(10, 9, 0));
wallPositions.Add (new Vector3(22, 9, 0));
wallPositions.Add (new Vector3(24, 9, 0));
wallPositions.Add (new Vector3(28, 9, 0));

wallPositions.Add (new Vector3(1, 10, 0));
wallPositions.Add (new Vector3(4, 10, 0));
wallPositions.Add (new Vector3(5, 10, 0));
wallPositions.Add (new Vector3(6, 10, 0));
wallPositions.Add (new Vector3(7, 10, 0));
wallPositions.Add (new Vector3(8, 10, 0));
wallPositions.Add (new Vector3(10, 10, 0));
wallPositions.Add (new Vector3(12, 10, 0));
wallPositions.Add (new Vector3(13, 10, 0));
wallPositions.Add (new Vector3(15, 10, 0));
wallPositions.Add (new Vector3(16, 10, 0));
```

```
wallPositions.Add (new Vector3(17, 10, 0));
wallPositions.Add (new Vector3(18, 10, 0));
wallPositions.Add (new Vector3(19, 10, 0));
wallPositions.Add (new Vector3(20, 10, 0));
wallPositions.Add (new Vector3(22, 10, 0));
wallPositions.Add (new Vector3(24, 10, 0));
wallPositions.Add (new Vector3(26, 10, 0));

wallPositions.Add (new Vector3(1, 11, 0));
wallPositions.Add (new Vector3(8, 11, 0));
wallPositions.Add (new Vector3(10, 11, 0));
wallPositions.Add (new Vector3(12, 11, 0));
wallPositions.Add (new Vector3(13, 11, 0));
wallPositions.Add (new Vector3(22, 11, 0));
wallPositions.Add (new Vector3(24, 11, 0));
wallPositions.Add (new Vector3(26, 11, 0));
wallPositions.Add (new Vector3(27, 11, 0));

wallPositions.Add (new Vector3(1, 12, 0));
wallPositions.Add (new Vector3(3, 12, 0));
wallPositions.Add (new Vector3(4, 12, 0));
wallPositions.Add (new Vector3(5, 12, 0));
wallPositions.Add (new Vector3(7, 12, 0));
wallPositions.Add (new Vector3(8, 12, 0));
wallPositions.Add (new Vector3(10, 12, 0));
wallPositions.Add (new Vector3(12, 12, 0));
wallPositions.Add (new Vector3(13, 12, 0));
wallPositions.Add (new Vector3(16, 12, 0));
wallPositions.Add (new Vector3(17, 12, 0));
wallPositions.Add (new Vector3(24, 12, 0));
wallPositions.Add (new Vector3(26, 12, 0));

wallPositions.Add (new Vector3(1, 13, 0));
wallPositions.Add (new Vector3(4, 13, 0));
wallPositions.Add (new Vector3(5, 13, 0));
wallPositions.Add (new Vector3(8, 13, 0));
wallPositions.Add (new Vector3(10, 13, 0));
wallPositions.Add (new Vector3(12, 13, 0));
wallPositions.Add (new Vector3(13, 13, 0));
wallPositions.Add (new Vector3(16, 13, 0));
wallPositions.Add (new Vector3(17, 13, 0));
wallPositions.Add (new Vector3(19, 13, 0));
wallPositions.Add (new Vector3(20, 13, 0));
```

```
wallPositions.Add (new Vector3(21, 13, 0));
wallPositions.Add (new Vector3(22, 13, 0));
wallPositions.Add (new Vector3(23, 13, 0));
wallPositions.Add (new Vector3(24, 13, 0));
wallPositions.Add (new Vector3(26, 13, 0));

wallPositions.Add (new Vector3(1, 14, 0));
wallPositions.Add (new Vector3(4, 14, 0));
wallPositions.Add (new Vector3(5, 14, 0));
wallPositions.Add (new Vector3(6, 14, 0));
wallPositions.Add (new Vector3(10, 14, 0));
wallPositions.Add (new Vector3(12, 14, 0));
wallPositions.Add (new Vector3(14, 14, 0));
wallPositions.Add (new Vector3(15, 14, 0));
wallPositions.Add (new Vector3(16, 14, 0));
wallPositions.Add (new Vector3(17, 14, 0));
wallPositions.Add (new Vector3(18, 14, 0));
wallPositions.Add (new Vector3(19, 14, 0));
wallPositions.Add (new Vector3(20, 14, 0));
wallPositions.Add (new Vector3(21, 14, 0));
wallPositions.Add (new Vector3(22, 14, 0));
wallPositions.Add (new Vector3(23, 14, 0));
wallPositions.Add (new Vector3(24, 14, 0));
wallPositions.Add (new Vector3(25, 14, 0));
wallPositions.Add (new Vector3(26, 14, 0));
wallPositions.Add (new Vector3(27, 14, 0));
wallPositions.Add (new Vector3(28, 14, 0));
wallPositions.Add (new Vector3(8, 14, 0));

wallPositions.Add (new Vector3(2, 15, 0));
wallPositions.Add (new Vector3(3, 15, 0));
wallPositions.Add (new Vector3(4, 15, 0));
wallPositions.Add (new Vector3(5, 15, 0));
wallPositions.Add (new Vector3(8, 15, 0));
wallPositions.Add (new Vector3(10, 15, 0));
wallPositions.Add (new Vector3(12, 15, 0));
wallPositions.Add (new Vector3(26, 15, 0));

wallPositions.Add (new Vector3(2, 16, 0));
wallPositions.Add (new Vector3(3, 16, 0));
wallPositions.Add (new Vector3(4, 16, 0));
wallPositions.Add (new Vector3(5, 16, 0));
wallPositions.Add (new Vector3(7, 16, 0));
```

```
wallPositions.Add (new Vector3(8, 16, 0));
wallPositions.Add (new Vector3(10, 16, 0));
wallPositions.Add (new Vector3(12, 16, 0));
wallPositions.Add (new Vector3(13, 16, 0));
wallPositions.Add (new Vector3(14, 16, 0));
wallPositions.Add (new Vector3(15, 16, 0));
wallPositions.Add (new Vector3(16, 16, 0));
wallPositions.Add (new Vector3(17, 16, 0));
wallPositions.Add (new Vector3(18, 16, 0));
wallPositions.Add (new Vector3(19, 16, 0));
wallPositions.Add (new Vector3(20, 16, 0));
wallPositions.Add (new Vector3(21, 16, 0));
wallPositions.Add (new Vector3(22, 16, 0));
wallPositions.Add (new Vector3(23, 16, 0));
wallPositions.Add (new Vector3(24, 16, 0));
wallPositions.Add (new Vector3(26, 16, 0));
wallPositions.Add (new Vector3(28, 16, 0));

wallPositions.Add (new Vector3(2, 17, 0));
wallPositions.Add (new Vector3(3, 17, 0));
wallPositions.Add (new Vector3(8, 17, 0));
wallPositions.Add (new Vector3(10, 17, 0));
wallPositions.Add (new Vector3(12, 17, 0));
wallPositions.Add (new Vector3(26, 17, 0));

wallPositions.Add (new Vector3(2, 18, 0));
wallPositions.Add (new Vector3(3, 18, 0));
wallPositions.Add (new Vector3(6, 18, 0));
wallPositions.Add (new Vector3(7, 18, 0));
wallPositions.Add (new Vector3(8, 18, 0));
wallPositions.Add (new Vector3(10, 18, 0));
wallPositions.Add (new Vector3(12, 18, 0));
wallPositions.Add (new Vector3(14, 18, 0));
wallPositions.Add (new Vector3(15, 18, 0));
wallPositions.Add (new Vector3(16, 18, 0));
wallPositions.Add (new Vector3(17, 18, 0));
wallPositions.Add (new Vector3(18, 18, 0));
wallPositions.Add (new Vector3(19, 18, 0));
wallPositions.Add (new Vector3(20, 18, 0));
wallPositions.Add (new Vector3(21, 18, 0));
wallPositions.Add (new Vector3(22, 18, 0));
wallPositions.Add (new Vector3(23, 18, 0));
wallPositions.Add (new Vector3(24, 18, 0));
```

```
//wallPositions.Add (new Vector3(25, 18, 0));
wallPositions.Add (new Vector3(26, 18, 0));
wallPositions.Add (new Vector3(27, 18, 0));

wallPositions.Add (new Vector3(10, 19, 0));
wallPositions.Add (new Vector3(12, 19, 0));

wallPositions.Add (new Vector3(0, 20, 0));
wallPositions.Add (new Vector3(1, 20, 0));
wallPositions.Add (new Vector3(2, 20, 0));
wallPositions.Add (new Vector3(3, 20, 0));
wallPositions.Add (new Vector3(4, 20, 0));
wallPositions.Add (new Vector3(5, 20, 0));
wallPositions.Add (new Vector3(6, 20, 0));
wallPositions.Add (new Vector3(7, 20, 0));
wallPositions.Add (new Vector3(8, 20, 0));
wallPositions.Add (new Vector3(10, 20, 0));
wallPositions.Add (new Vector3(12, 20, 0));
wallPositions.Add (new Vector3(13, 20, 0));
wallPositions.Add (new Vector3(14, 20, 0));
wallPositions.Add (new Vector3(15, 20, 0));
wallPositions.Add (new Vector3(16, 20, 0));
wallPositions.Add (new Vector3(17, 20, 0));
wallPositions.Add (new Vector3(18, 20, 0));
wallPositions.Add (new Vector3(19, 20, 0));
wallPositions.Add (new Vector3(20, 20, 0));
wallPositions.Add (new Vector3(21, 20, 0));
wallPositions.Add (new Vector3(22, 20, 0));
wallPositions.Add (new Vector3(23, 20, 0));
wallPositions.Add (new Vector3(24, 20, 0));
wallPositions.Add (new Vector3(25, 20, 0));
wallPositions.Add (new Vector3(27, 20, 0));

wallPositions.Add (new Vector3(10, 21, 0));
wallPositions.Add (new Vector3(25, 21, 0));
wallPositions.Add (new Vector3(27, 21, 0));

wallPositions.Add (new Vector3(3, 22, 0));
wallPositions.Add (new Vector3(4, 22, 0));
wallPositions.Add (new Vector3(5, 22, 0));
wallPositions.Add (new Vector3(6, 22, 0));
wallPositions.Add (new Vector3(7, 22, 0));
wallPositions.Add (new Vector3(8, 22, 0));
```

```
    wallPositions.Add (new Vector3(9, 22, 0));
    wallPositions.Add (new Vector3(10, 22, 0));
    wallPositions.Add (new Vector3(12, 22, 0));
    wallPositions.Add (new Vector3(13, 22, 0));
    wallPositions.Add (new Vector3(14, 22, 0));
    wallPositions.Add (new Vector3(15, 22, 0));
    wallPositions.Add (new Vector3(17, 22, 0));
    wallPositions.Add (new Vector3(18, 22, 0));
    wallPositions.Add (new Vector3(19, 22, 0));
    wallPositions.Add (new Vector3(20, 22, 0));
    wallPositions.Add (new Vector3(21, 22, 0));
    wallPositions.Add (new Vector3(22, 22, 0));
    wallPositions.Add (new Vector3(23, 22, 0));
    wallPositions.Add (new Vector3(24, 22, 0));
    wallPositions.Add (new Vector3(25, 22, 0));
    wallPositions.Add (new Vector3(27, 22, 0));

    wallPositions.Add (new Vector3(7, 23, 0));
    wallPositions.Add (new Vector3(8, 23, 0));
    wallPositions.Add (new Vector3(10, 23, 0));
    wallPositions.Add (new Vector3(13, 23, 0));
    wallPositions.Add (new Vector3(20, 23, 0));
    wallPositions.Add (new Vector3(21, 23, 0));
    wallPositions.Add (new Vector3(23, 23, 0));
    wallPositions.Add (new Vector3(24, 23, 0));
    wallPositions.Add (new Vector3(25, 23, 0));
    wallPositions.Add (new Vector3(27, 23, 0));

    wallPositions.Add (new Vector3(2, 24, 0));
    wallPositions.Add (new Vector3(3, 24, 0));
    wallPositions.Add (new Vector3(4, 24, 0));
    wallPositions.Add (new Vector3(10, 24, 0));
    wallPositions.Add (new Vector3(15, 24, 0));
    wallPositions.Add (new Vector3(16, 24, 0));


}



void makeWallList_2()
{
    wallPositions.Add (new Vector3(15, 0, 0));
```

```
wallPositions.Add (new Vector3(16, 0, 0));

wallPositions.Add (new Vector3(2, 1, 0));
wallPositions.Add (new Vector3(3, 1, 0));
wallPositions.Add (new Vector3(4, 1, 0));
wallPositions.Add (new Vector3(5, 1, 0));
wallPositions.Add (new Vector3(5, 1, 0));
wallPositions.Add (new Vector3(15, 1, 0));
wallPositions.Add (new Vector3(16, 1, 0));
wallPositions.Add (new Vector3(21, 1, 0));
wallPositions.Add (new Vector3(22, 1, 0));

wallPositions.Add (new Vector3(1, 2, 0));
wallPositions.Add (new Vector3(2, 2, 0));
wallPositions.Add (new Vector3(6, 2, 0));
wallPositions.Add (new Vector3(7, 2, 0));
wallPositions.Add (new Vector3(8, 2, 0));
wallPositions.Add (new Vector3(9, 2, 0));
wallPositions.Add (new Vector3(10, 2, 0));
wallPositions.Add (new Vector3(11, 2, 0));
wallPositions.Add (new Vector3(12, 2, 0));
wallPositions.Add (new Vector3(15, 2, 0));
wallPositions.Add (new Vector3(16, 2, 0));
wallPositions.Add (new Vector3(18, 2, 0));
wallPositions.Add (new Vector3(19, 2, 0));
wallPositions.Add (new Vector3(20, 2, 0));
wallPositions.Add (new Vector3(21, 2, 0));
wallPositions.Add (new Vector3(22, 2, 0));

wallPositions.Add (new Vector3(1, 3, 0));
wallPositions.Add (new Vector3(9, 3, 0));
wallPositions.Add (new Vector3(10, 3, 0));
wallPositions.Add (new Vector3(11, 3, 0));
wallPositions.Add (new Vector3(12, 3, 0));
wallPositions.Add (new Vector3(13, 3, 0));
wallPositions.Add (new Vector3(18, 3, 0));
wallPositions.Add (new Vector3(19, 3, 0));
wallPositions.Add (new Vector3(20, 3, 0));
wallPositions.Add (new Vector3(21, 3, 0));
wallPositions.Add (new Vector3(22, 3, 0));

wallPositions.Add (new Vector3(1, 4, 0));
wallPositions.Add (new Vector3(2, 4, 0));
```

```
wallPositions.Add (new Vector3(13, 4, 0));
wallPositions.Add (new Vector3(26, 4, 0));
wallPositions.Add (new Vector3(27, 4, 0));

wallPositions.Add (new Vector3(2, 5, 0));
wallPositions.Add (new Vector3(4, 5, 0));
wallPositions.Add (new Vector3(5, 5, 0));
wallPositions.Add (new Vector3(26, 5, 0));
wallPositions.Add (new Vector3(27, 5, 0));

wallPositions.Add (new Vector3(2, 6, 0));
wallPositions.Add (new Vector3(4, 6, 0));
wallPositions.Add (new Vector3(5, 6, 0));
wallPositions.Add (new Vector3(26, 6, 0));
wallPositions.Add (new Vector3(27, 6, 0));

wallPositions.Add (new Vector3(2, 7, 0));
wallPositions.Add (new Vector3(4, 7, 0));
wallPositions.Add (new Vector3(5, 7, 0));
wallPositions.Add (new Vector3(8, 7, 0));
wallPositions.Add (new Vector3(9, 7, 0));
wallPositions.Add (new Vector3(10, 7, 0));
wallPositions.Add (new Vector3(15, 7, 0));
wallPositions.Add (new Vector3(25, 7, 0));
wallPositions.Add (new Vector3(26, 7, 0));

wallPositions.Add (new Vector3(6, 8, 0));
wallPositions.Add (new Vector3(7, 8, 0));
wallPositions.Add (new Vector3(8, 8, 0));
wallPositions.Add (new Vector3(9, 8, 0));
wallPositions.Add (new Vector3(10, 8, 0));
wallPositions.Add (new Vector3(15, 8, 0));
wallPositions.Add (new Vector3(16, 8, 0));
wallPositions.Add (new Vector3(17, 8, 0));
wallPositions.Add (new Vector3(18, 8, 0));
wallPositions.Add (new Vector3(19, 8, 0));
wallPositions.Add (new Vector3(25, 8, 0));
wallPositions.Add (new Vector3(26, 8, 0));

wallPositions.Add (new Vector3(6, 9, 0));
wallPositions.Add (new Vector3(7, 9, 0));
wallPositions.Add (new Vector3(16, 9, 0));
wallPositions.Add (new Vector3(17, 9, 0));
```

```
wallPositions.Add (new Vector3(18, 9, 0));
wallPositions.Add (new Vector3(19, 9, 0));
wallPositions.Add (new Vector3(25, 9, 0));
wallPositions.Add (new Vector3(26, 9, 0));

wallPositions.Add (new Vector3(6, 10, 0));
wallPositions.Add (new Vector3(7, 10, 0));
wallPositions.Add (new Vector3(17, 10, 0));
wallPositions.Add (new Vector3(18, 10, 0));

wallPositions.Add (new Vector3(2, 11, 0));
wallPositions.Add (new Vector3(3, 11, 0));
wallPositions.Add (new Vector3(4, 11, 0));
wallPositions.Add (new Vector3(17, 11, 0));
wallPositions.Add (new Vector3(18, 11, 0));

wallPositions.Add (new Vector3(2, 12, 0));
wallPositions.Add (new Vector3(3, 12, 0));
wallPositions.Add (new Vector3(4, 12, 0));
wallPositions.Add (new Vector3(17, 12, 0));
wallPositions.Add (new Vector3(18, 12, 0));
wallPositions.Add (new Vector3(28, 12, 0));

wallPositions.Add (new Vector3(17, 13, 0));
wallPositions.Add (new Vector3(18, 13, 0));
wallPositions.Add (new Vector3(19, 13, 0));
wallPositions.Add (new Vector3(20, 13, 0));
wallPositions.Add (new Vector3(21, 13, 0));
wallPositions.Add (new Vector3(22, 13, 0));
wallPositions.Add (new Vector3(23, 13, 0));
wallPositions.Add (new Vector3(24, 13, 0));
wallPositions.Add (new Vector3(28, 13, 0));

wallPositions.Add (new Vector3(4, 14, 0));
wallPositions.Add (new Vector3(8, 14, 0));
wallPositions.Add (new Vector3(9, 14, 0));
wallPositions.Add (new Vector3(10, 14, 0));
wallPositions.Add (new Vector3(11, 14, 0));
wallPositions.Add (new Vector3(12, 14, 0));
wallPositions.Add (new Vector3(16, 14, 0));
wallPositions.Add (new Vector3(17, 14, 0));
wallPositions.Add (new Vector3(18, 14, 0));
wallPositions.Add (new Vector3(19, 14, 0));
```

```
wallPositions.Add (new Vector3(20, 14, 0));
wallPositions.Add (new Vector3(21, 14, 0));
wallPositions.Add (new Vector3(22, 14, 0));
wallPositions.Add (new Vector3(23, 14, 0));
wallPositions.Add (new Vector3(24, 14, 0));

wallPositions.Add (new Vector3(0, 15, 0));
wallPositions.Add (new Vector3(1, 15, 0));
wallPositions.Add (new Vector3(2, 15, 0));
wallPositions.Add (new Vector3(4, 15, 0));
wallPositions.Add (new Vector3(8, 15, 0));
wallPositions.Add (new Vector3(12, 15, 0));
wallPositions.Add (new Vector3(28, 15, 0));

wallPositions.Add (new Vector3(0, 16, 0));
wallPositions.Add (new Vector3(1, 16, 0));
wallPositions.Add (new Vector3(2, 16, 0));
wallPositions.Add (new Vector3(4, 16, 0));
wallPositions.Add (new Vector3(5, 16, 0));
wallPositions.Add (new Vector3(6, 16, 0));
wallPositions.Add (new Vector3(7, 16, 0));
wallPositions.Add (new Vector3(8, 16, 0));
wallPositions.Add (new Vector3(12, 16, 0));
wallPositions.Add (new Vector3(22, 16, 0));
wallPositions.Add (new Vector3(23, 16, 0));
wallPositions.Add (new Vector3(28, 16, 0));

wallPositions.Add (new Vector3(11, 17, 0));
wallPositions.Add (new Vector3(12, 17, 0));
wallPositions.Add (new Vector3(15, 17, 0));
wallPositions.Add (new Vector3(16, 17, 0));
wallPositions.Add (new Vector3(22, 17, 0));
wallPositions.Add (new Vector3(23, 17, 0));

wallPositions.Add (new Vector3(15, 18, 0));
wallPositions.Add (new Vector3(16, 18, 0));
wallPositions.Add (new Vector3(19, 18, 0));
wallPositions.Add (new Vector3(20, 18, 0));
wallPositions.Add (new Vector3(25, 18, 0));
wallPositions.Add (new Vector3(26, 18, 0));

wallPositions.Add (new Vector3(3, 19, 0));
wallPositions.Add (new Vector3(4, 19, 0));
```

```
wallPositions.Add (new Vector3(5, 19, 0));
wallPositions.Add (new Vector3(6, 19, 0));
wallPositions.Add (new Vector3(7, 19, 0));
wallPositions.Add (new Vector3(8, 19, 0));
wallPositions.Add (new Vector3(11, 19, 0));
wallPositions.Add (new Vector3(12, 19, 0));
wallPositions.Add (new Vector3(15, 19, 0));
wallPositions.Add (new Vector3(16, 19, 0));
wallPositions.Add (new Vector3(19, 19, 0));
wallPositions.Add (new Vector3(20, 19, 0));
wallPositions.Add (new Vector3(25, 19, 0));
wallPositions.Add (new Vector3(26, 19, 0));

wallPositions.Add (new Vector3(8, 20, 0));
wallPositions.Add (new Vector3(11, 20, 0));
wallPositions.Add (new Vector3(12, 20, 0));
wallPositions.Add (new Vector3(22, 20, 0));
wallPositions.Add (new Vector3(23, 20, 0));

wallPositions.Add (new Vector3(1, 21, 0));
wallPositions.Add (new Vector3(8, 21, 0));
wallPositions.Add (new Vector3(17, 21, 0));
wallPositions.Add (new Vector3(18, 21, 0));
wallPositions.Add (new Vector3(22, 21, 0));
wallPositions.Add (new Vector3(23, 21, 0));

wallPositions.Add (new Vector3(1, 22, 0));
wallPositions.Add (new Vector3(2, 22, 0));
wallPositions.Add (new Vector3(11, 22, 0));
wallPositions.Add (new Vector3(12, 22, 0));
wallPositions.Add (new Vector3(17, 22, 0));
wallPositions.Add (new Vector3(18, 22, 0));
wallPositions.Add (new Vector3(25, 22, 0));
wallPositions.Add (new Vector3(26, 22, 0));

wallPositions.Add (new Vector3(2, 23, 0));
wallPositions.Add (new Vector3(6, 23, 0));
wallPositions.Add (new Vector3(7, 23, 0));
wallPositions.Add (new Vector3(11, 23, 0));
wallPositions.Add (new Vector3(12, 23, 0));
wallPositions.Add (new Vector3(20, 23, 0));
wallPositions.Add (new Vector3(21, 23, 0));
wallPositions.Add (new Vector3(25, 23, 0));
```

```
    wallPositions.Add (new Vector3(26, 23, 0));

    wallPositions.Add (new Vector3(2, 24, 0));
    wallPositions.Add (new Vector3(6, 24, 0));
    wallPositions.Add (new Vector3(7, 24, 0));
    wallPositions.Add (new Vector3(20, 24, 0));
    wallPositions.Add (new Vector3(21, 24, 0));
    wallPositions.Add (new Vector3(25, 24, 0));
    wallPositions.Add (new Vector3(26, 18, 0));


}

void makeWallList()
{

    wallPositions.Add (new Vector3(0, 3, 0));
    wallPositions.Add (new Vector3(1, 3, 0));
    wallPositions.Add (new Vector3(4, 3, 0));

    for (int i = 0; i < 8; i++) {
        wallPositions.Add (new Vector3(5, i, 0));
    }

    for (int i = 0; i < 3; i++) {
        wallPositions.Add (new Vector3(i, 12, 0));
    }

    for (int i = 0; i < 6; i++)
    {
        wallPositions.Add (new Vector3(5 + i, 12, 0));
    }

    wallPositions.Add (new Vector3(6, 7, 0));
    wallPositions.Add (new Vector3(7, 7, 0));
    wallPositions.Add (new Vector3(10, 3, 0));
    wallPositions.Add (new Vector3(10, 5, 0));
    wallPositions.Add (new Vector3(10, 7, 0));
    wallPositions.Add (new Vector3(11, 3, 0));
    wallPositions.Add (new Vector3(11, 5, 0));
    wallPositions.Add (new Vector3(11, 7, 0));
```

```
for (int i = 0; i < 8; i++)
{
    if (i == 4)
        continue;

    wallPositions.Add (new Vector3(12, i, 0));
}

wallPositions.Add (new Vector3(21, 6, 0));
wallPositions.Add (new Vector3(21, 16, 0));
wallPositions.Add (new Vector3(21, 17, 0));
wallPositions.Add (new Vector3(16, 18, 0));

for (int i = 0; i < 3; i++) {

    wallPositions.Add(new Vector3(i, 12, 0));
}

for (int i = 0; i < 6; i++) {

    wallPositions.Add(new Vector3(i + 5, 12, 0));
}

for (int i = 0; i < 7; i++) {

    wallPositions.Add(new Vector3(i + 13, 12, 0));
}

for (int i = 0; i < 4; i++) {

    wallPositions.Add(new Vector3(i, 15, 0));
}

for (int i = 0; i < 10; i++) {

    wallPositions.Add(new Vector3(i + 6, 15, 0));
}

for (int i = 0; i < 5; i++) {

    wallPositions.Add(new Vector3(i + 18, 15, 0));
}
```

```
for (int i = 0; i < 4; i++) {

    wallPositions.Add(new Vector3(i + 24, 14, 0));
}

for (int i = 0; i < 3; i++) {

    wallPositions.Add(new Vector3(i, 20, 0));
}

for (int i = 0; i < 3; i++) {

    wallPositions.Add(new Vector3(i + 5, 20, 0));
}

for (int i = 0; i < 5; i++) {

    wallPositions.Add(new Vector3(i + 9, 20, 0));
}

for (int i = 0; i < 3; i++) {

    wallPositions.Add(new Vector3(i + 15, 20, 0));
}

for (int i = 0; i < 6; i++) {

    wallPositions.Add(new Vector3(i + 21, 19, 0));
}

wallPositions.Add (new Vector3 (27, 17, 0));

for (int i = 0; i < 3; i++) {

    wallPositions.Add(new Vector3(i + 16, 17, 0));
}

for (int i = 0; i < 3; i++) {

    wallPositions.Add(new Vector3(i + 22, 11, 0));
}
```

```
for (int i = 0; i < 2; i++) {

    wallPositions.Add(new Vector3(i+27, 11, 0));
}

for (int i = 0; i < 3; i++) {

    if (i == 1)
        continue;

    wallPositions.Add(new Vector3(i + 19, 8, 0));
}

for (int i = 0; i < 5; i++) {

    wallPositions.Add(new Vector3(i + 24, 4, 0));
}

for (int i = 0; i < 3; i++) {

    wallPositions.Add(new Vector3(7, i+16, 0));
}

for (int i = 0; i < 3; i++) {

    wallPositions.Add(new Vector3(7, i+22, 0));
}

for (int i = 0; i < 4; i++) {

    wallPositions.Add(new Vector3(13, i+16, 0));
}

for (int i = 0; i < 4; i++) {

    wallPositions.Add(new Vector3(13, i+21, 0));
}

for (int i = 0; i < 2; i++) {

    wallPositions.Add(new Vector3(16, i+21, 0));
}
```

```
for (int i = 0; i < 2; i++) {

    wallPositions.Add(new Vector3(18, i+18, 0));
}

for (int i = 0; i < 8; i++) {

    if (i == 4)
        continue;

    wallPositions.Add(new Vector3(19, i, 0));
}

for (int i = 0; i < 4; i++) {

    wallPositions.Add(new Vector3(21, i+21, 0));
}

for (int i = 0; i < 4; i++) {

    wallPositions.Add(new Vector3(22, i+6, 0));
}

for (int i = 0; i < 3; i++) {

    wallPositions.Add(new Vector3(22, i+12, 0));
}

for (int i = 0; i < 3; i++) {

    wallPositions.Add(new Vector3(24, i, 0));
}

for (int i = 0; i < 6; i++) {

    wallPositions.Add(new Vector3(24, i+5, 0));
}

for (int i = 0; i < 3; i++) {

    wallPositions.Add(new Vector3(25, i+15, 0));
}
```

```
        for (int i = 0; i < 3; i++) {

            wallPositions.Add(new Vector3(26, i+15, 0));
        }

    }


    //Sets up the outer walls and floor (background) of the game
board.
    void BoardSetup ()
    {
        //Instantiate Board and set boardHolder to its transform.
        boardHolder = new GameObject ("Board").transform;

        //Loop along x axis, starting from -
1 (to fill corner) with floor or outerwall edge tiles.
        for(int x = -1; x < columns + 1; x++)
        {
            //Loop along y axis, starting from -
1 to place floor or outerwall tiles.
            for(int y = -1; y < rows + 1; y++)
            {
                //Choose a random tile from our array of floor ti
le prefabs and prepare to instantiate it.
                GameObject toInstantiate = floorTiles[Random.Rang
e (0,floorTiles.Length)];

                //Check if we current position is at board edge,
if so choose a random outer wall prefab from our array of outer w
all tiles.
                if(x == -1 || x == columns || y == -
1 || y == rows)
                    toInstantiate = outerWallTiles [Random.Range
(0, outerWallTiles.Length)];

                //Instantiate the GameObject instance using the p
refab chosen for toInstantiate at the Vector3 corresponding to cu
rrent grid position in loop, cast it to GameObject.
                GameObject instance =
                    Instantiate (toInstantiate, new Vector3 (x, y
, 0f), Quaternion.identity) as GameObject;
```

```
                //Set the parent of our newly instantiated object
 instance to boardHolder, this is just organizational to avoid cl
uttering hierarchy.
                instance.transform.SetParent (boardHolder);
            }
        }
    }


    //RandomPosition returns a random position from our list grid
Positions.
    Vector3 RandomPosition ()
    {
        //Declare an integer randomIndex, set it's value to a ran
dom number between 0 and the count of items in our List gridPosit
ions.
        int randomIndex = Random.Range (0, gridPositions.Count);

        //Declare a variable of type Vector3 called randomPositio
n, set it's value to the entry at randomIndex from our List gridP
ositions.
        Vector3 randomPosition = gridPositions[randomIndex];

        //Remove the entry at randomIndex from the list so that i
t can't be re-used.
        gridPositions.RemoveAt (randomIndex);

        //Return the randomly selected Vector3 position.
        return randomPosition;
    }


    //LayoutObjectAtRandom accepts an array of game objects to ch
oose from along with a minimum and maximum range for the number o
f objects to create.
    void LayoutObjectAtRandom (GameObject[] tileArray, int minimu
m, int maximum)
    {
        //Choose a random number of objects to instantiate within
 the minimum and maximum limits
        int objectCount = Random.Range (minimum, maximum+1);
```

```
        //Instantiate objects until the randomly chosen limit objectCount is reached
        for(int i = 0; i < objectCount; i++)
        {
            //Choose a position for randomPosition by getting a random position from our list of available Vector3s stored in gridPosition
            Vector3 randomPosition = RandomPosition();

            //Choose a random tile from tileArray and assign it to tileChoice
            GameObject tileChoice = tileArray[Random.Range (0, tileArray.Length)];

            //Instantiate tileChoice at the position returned by RandomPosition with no change in rotation
            Instantiate(tileChoice, randomPosition, Quaternion.identity);
        }
    }

    void PlaceEnemy(int amount)
    {

        for (int i = 0; i < amount; i++)
        {
            //Choose a position for randomPosition by getting a random position from our list of available Vector3s stored in gridPosition
            Vector3 randomPosition = new Vector3(27f, 16f, 0f);

            //Choose a random tile from tileArray and assign it to tileChoice
            GameObject tileChoice = enemyTiles[Random.Range(0, enemyTiles.Length)];

            //Instantiate tileChoice at the position returned by RandomPosition with no change in rotation
            Instantiate(tileChoice, randomPosition, Quaternion.identity);
        }
```

```
    }

    void PlaceEnemy_2(int amount)
    {

        for (int i = 0; i < amount; i++)
        {
            //Choose a position for randomPosition by getting a r
andom position from our list of available Vector3s stored in grid
Position
            Vector3 randomPosition = new Vector3(13f, 9f, 0f);

            //Choose a random tile from tileArray and assign it t
o tileChoice
            GameObject tileChoice = enemyTiles[Random.Range(0, en
emyTiles.Length)];

            //Instantiate tileChoice at the position returned by
RandomPosition with no change in rotation
            Instantiate(tileChoice, randomPosition, Quaternion.id
entity);
        }

    }

    void PlaceEnemy_3(int amount)
    {

        for (int i = 0; i < amount; i++)
        {
            //Choose a position for randomPosition by getting a r
andom position from our list of available Vector3s stored in grid
Position
            Vector3 randomPosition = new Vector3(28f, 22f, 0f);

            //Choose a random tile from tileArray and assign it t
o tileChoice
            GameObject tileChoice = enemyTiles[Random.Range(0, en
emyTiles.Length)];

            //Instantiate tileChoice at the position returned by
RandomPosition with no change in rotation
            Instantiate(tileChoice, randomPosition, Quaternion.id
```

```
entity);
        }

    }

    void PlaceCoins()
    {
        Instantiate(gold, new Vector3(10f,1f,0f), Quaternion.iden
tity);

        Instantiate(silver, new Vector3(17f, 18f, 0f), Quaternion
.identity);

        Instantiate(copper, new Vector3(21f, 7f, 0f), Quaternion.
identity);

        Instantiate(silver, new Vector3(28f, 2f, 0f), Quaternion.
identity);

        Instantiate(gold, new Vector3(2f, 19f, 0f), Quaternion.id
entity);


    }

    void PlaceCoins_2()
    {
        Instantiate(gold, new Vector3(10,16,0), Quaternion.identi
ty);

        Instantiate(gold, new Vector3(19, 1, 0), Quaternion.ident
ity);

        Instantiate(gold, new Vector3(28, 24, 0), Quaternion.iden
tity);


        Instantiate(silver, new Vector3(3, 3, 0), Quaternion.iden
tity);

        Instantiate(silver, new Vector3(28, 8, 0), Quaternion.ide
ntity);
```

```
        Instantiate(silver, new Vector3(27, 15, 0), Quaternion.id
entity);


        Instantiate(copper, new Vector3(7, 14, 0), Quaternion.ide
ntity);

        Instantiate(copper, new Vector3(17, 7, 0), Quaternion.ide
ntity);

        Instantiate(copper, new Vector3(13, 23, 0), Quaternion.id
entity);

    }

    void PlaceCoins_3()
    {
        Instantiate(gold, new Vector3( 1, 5,0), Quaternion.identi
ty);

        Instantiate(gold, new Vector3(18, 15, 0), Quaternion.iden
tity);

        Instantiate(gold, new Vector3(28, 15, 0), Quaternion.iden
tity);


        Instantiate(silver, new Vector3(8, 4, 0), Quaternion.iden
tity);

        Instantiate(silver, new Vector3(27, 13, 0), Quaternion.id
entity);

        Instantiate(silver, new Vector3(24, 21, 0), Quaternion.id
entity);

        Instantiate(silver, new Vector3(9, 23, 0), Quaternion.ide
ntity);


        Instantiate(copper, new Vector3(3, 14, 0), Quaternion.ide
ntity);
```

```
        Instantiate(copper, new Vector3(14, 13, 0), Quaternion.id
entity);

        Instantiate(copper, new Vector3(22, 23, 0), Quaternion.id
entity);

    }


    void LayoutWalls()
    {
        int i = 0;
        foreach (Vector3 wall in wallPositions) {

            GameObject tileChoice = wallTiles [Random.Range (0, w
allTiles.Length)];
            GameObject instance = Instantiate (tileChoice, wall,
Quaternion.identity) as GameObject;
            i++;

            instance.transform.SetParent (wallHolder);
        }

    }



    //SetupScene initializes our level and calls the previous fun
ctions to lay out the game board
    public void SetupScene (int level)
    {

        if(level==4){
            GameManager.instance.YouWin ();
        }

        if (level == 3) {
            PlaceCoins_3 ();
            BoardSetup ();
            InitialiseList_3 ();
            LayoutWalls ();
            PlaceEnemy_2 (1);
            PlaceEnemy_3 (1);
```

```
            Instantiate (exit, new Vector3 (28, 0, 0f), Quaternio
n.identity);
        }

        if (level == 2) {
            PlaceCoins_2 ();
            BoardSetup ();
            InitialiseList_2 ();
            LayoutWalls ();
            PlaceEnemy_2 (1);
            Instantiate (exit, new Vector3 (0, 24, 0f), Quaternio
n.identity);
        }

        if(level==1){

        PlaceCoins();
        //PlaceCoins_2 ();

        //Creates the outer walls and floor.
        BoardSetup ();

        //Reset our list of gridpositions.
        InitialiseList ();

        LayoutWalls ();

        //Instantiate a random number of enemies based on minimum
 and maximum, at randomized positions.
        PlaceEnemy (1);

        //Instantiate the exit
        Instantiate (exit, new Vector3 (columns - 1, rows - 1, 0f
), Quaternion.identity);
        }
    }
}



using UnityEngine;
using System.Collections;
```

```
public class Wall : MonoBehaviour
{
    public AudioClip chopSound1;                //1 of 2 audio cl
ips that play when the wall is attacked by the player.
    public AudioClip chopSound2;                //2 of 2 audio cl
ips that play when the wall is attacked by the player.
    public Sprite dmgSprite;                    //Alternate sprit
e to display after Wall has been attacked by player.
    public int hp = 5;                            //hit points fo
r the wall. --
- not used, but functionality left for future use if necessary
    public LayerMask floorMask;

    private SpriteRenderer spriteRenderer;        //Store a compo
nent reference to the attached SpriteRenderer.

    public Vector2 currentPos;


    void Awake ()
    {
        //Get a component reference to the SpriteRenderer.
        spriteRenderer = GetComponent<SpriteRenderer> ();
        //currentPos = new Vector2(transform.position.x, transfor
m.position.y);
        setInitialTile (Physics2D.Linecast(new Vector2(transform.
position.x, transform.position.y),
            new Vector2(transform.position.x, transform.position.
y) + Vector2.up, floorMask).collider.gameObject);


    }

    void Start()
    {

    }

    public void setInitialTile(GameObject startTile)
    {

        startTile.GetComponent<Tile> ()._hasWall = true;
```

```
        startTile.GetComponent<BoxCollider2D> ().enabled = false;

    }


    //DamageWall is called when the player attacks a wall. --
- not used, but fuctionality left for future use if necessary
    public void DamageWall (int loss)
    {
        //Call the RandomizeSfx function of SoundManager to play
one of two chop sounds.
        SoundManager.instance.RandomizeSfx (chopSound1, chopSound
2);

        //Set spriteRenderer to the damaged wall sprite.
        spriteRenderer.sprite = dmgSprite;

        //Subtract loss from hit point total.
        hp -= loss;

        //If hit points are less than or equal to zero:
        if(hp <= 0)
            //Disable the gameObject.
            gameObject.SetActive (false);
    }



}


using UnityEngine;
using System.Collections;

public class Tile : MonoBehaviour{

    public int distanceGold = 100, distanceSilver = 100, distance
Copper = 100, distanceExit = 100;
    public bool _hasWall = false;

    public void setDistanceGold(int _distanceGold)
    {
        distanceGold = _distanceGold;
```

```
}

public void setDistanceSilver(int _distanceSilver)
{
    distanceSilver = _distanceSilver;
}

public void setDistanceCopper(int _distanceCopper)
{
    distanceCopper = _distanceCopper;
}

public void setDistanceExit(int _distanceExit)
{
    distanceExit = _distanceExit;
}

public int getDistanceGold()
{
    return distanceGold;
}

public int getDistanceSilver()
{
    return distanceSilver;
}

public int getDistanceCopper()
{
    return distanceCopper;
}

public int getDistanceExit()
{
    return distanceExit;
}

public bool hasWall()
{
    return _hasWall;
}

public void setWall()
```

```
    {
        _hasWall = true;
    }
}


using UnityEngine;
using System.Collections;

using System.Collections.Generic;        //Allows us to use Lists
.

public class FloodTiles : MonoBehaviour {

    public List<GameObject> tileList = new List<GameObject>();
    public LayerMask floorLayer;
    public List<GameObject> finshedTiles = new List<GameObject>()
;

    public void Start()
    {
        setInitialTile (Physics2D.Linecast(new Vector2(transform.
position.x, transform.position.y),
            new Vector2(transform.position.x, transform.position.
y) + Vector2.up, floorLayer).collider.gameObject);
    }

    public void setInitialTile(GameObject startTile)
    {
        tileList.Add (startTile);

        if (CompareTag ("Gold"))
        {
            startTile.GetComponent<Tile> ().distanceGold = 1;
        }
        if (CompareTag ("Silver"))
        {
            startTile.GetComponent<Tile> ().distanceSilver = 1;
        }
        if (CompareTag ("Copper"))
        {
            startTile.GetComponent<Tile> ().distanceCopper = 1;
        }
```

```
        if (CompareTag ("Exit"))
        {
            startTile.GetComponent<Tile> ().distanceExit = 1;
        }

        floodTiles ();

    }

    public void floodTiles()
    {
        int testExitSentinel = 0;
        //start the loop until no more tiles are left
        while (tileList.Count != 0 && testExitSentinel != 100)
        {

            testExitSentinel++;

            //Get the count and valid surrounding tiles
            int tileCount = 0;

            //Add the surrounding tiles to the list
            int currentTileAmount = tileList.Count;
            for(int i = 0; i < currentTileAmount; i ++)
            {
                RaycastHit2D[] surroundingTiles = new RaycastHit2
D[4];
                //Get the surronding tiles
                tileCount = getSurroundingTiles (surroundingTiles
, tileList[i]);

                //Add the tiles to the list
                for (int j = 0; j < 4; j++)
                {
                    if (surroundingTiles [j].collider != null) {
                    //Tile to be added
                        bool addTile = true;
                        //Check for duplicates
                        foreach(GameObject tileToCheck in tileLis
t)
                        {
                            if (surroundingTiles [j].collider.gam
eObject == tileToCheck)
```

```
                                addTile = false;
                            if (surroundingTiles [j].collider.Get
Component<Tile> ()._hasWall) {
                                addTile = false;
                            }
                        }



                        if(addTile)
                            tileList.Add(surroundingTiles[j].coll
ider.gameObject);

                    }

                    //Check for tiles in finished Tile List


                }

            }

            //Get the surrounding tiles of each tile and check di
stances
            currentTileAmount = tileList.Count;

            for(int i = 0; i < currentTileAmount; i++)
            {
                RaycastHit2D[] surroundingTiles = new RaycastHit2
D[4];

                Debug.Log("In Flood to check distances");

                tileCount = getSurroundingTiles (surroundingTiles
, tileList[i]);




                    if (gameObject.CompareTag ("Gold")) {
```

```
                        setDistance (tileList[i], surroundingTile
s, 'g');

                    }

                    if (gameObject.CompareTag ("Silver")) {
                        setDistance (tileList[i], surroundingTile
s, 's');

                    }

                    if (gameObject.CompareTag ("Copper")) {
                        setDistance (tileList[i], surroundingTile
s, 'c');

                    }
                    if (gameObject.CompareTag ("Exit")) {
                        setDistance (tileList[i], surroundingTile
s, 'e');

                    }



            }

            //Check to remove tiles
            currentTileAmount = tileList.Count;

            for(int i = 0; i < currentTileAmount; i++)
            {
                //Get the surrounding tiles for each tile in the
list
                RaycastHit2D[] surroundingTiles = new RaycastHit2
D[4];

                getSurroundingTiles(surroundingTiles, tileList[i]
);
                char type = ' ';
                int currentDistance = 0;
```

```
                if (gameObject.CompareTag ("Gold")) {
                    type = 'g';
                    currentDistance = tileList[i].GetComponent<Ti
le> ().distanceGold;
                }

                if (gameObject.CompareTag ("Silver")) {
                    currentDistance = tileList[i].GetComponent<Ti
le> ().distanceSilver;
                    type = 's';
                }

                if (gameObject.CompareTag ("Copper")) {
                    currentDistance = tileList[i].GetComponent<Ti
le> ().distanceCopper;
                    type = 'c';


                }

                if (gameObject.CompareTag ("Exit")) {
                    currentDistance = tileList[i].GetComponent<Ti
le> ().distanceExit;
                    type = 'e';
                    Debug.Log (currentDistance);
                }

                if (finshCheck (surroundingTiles, currentDistance
, type)) {

                    Debug.Log ("Current i value " + i);

                    finshedTiles.Add (tileList[i]);
                    tileList.RemoveAt(i);
                    currentTileAmount--;
                    i--;

                }


            }
```

```
        }//End while


    }//Flood tiles

    //Loop to remove a tile
    public void removeTile(GameObject tileToRemove)
    {
        int index = 0;
        int currentTileCount = tileList.Count;

        for (int i = 0; i < currentTileCount; i++) {
            Debug.Log ("Remove index" + index);

            if (tileList [i] == tileToRemove) {
                tileList.RemoveAt (index);
                index--;
                currentTileCount--;
                i--;
            }
                index++;


        }
    }

    //Get the count and set the surrounding tiles to the array
    public int getSurroundingTiles(RaycastHit2D[] surroundingTile
s, GameObject tile)
    {

        int tileCount = 0;

        tile.GetComponentInParent<BoxCollider2D> ().enabled = fal
se;

        RaycastHit2D upTile, rightTile, downTile, leftTile;
        bool hasUpTile = false, hasRightTile = false, hasDownTile
 = false, hasLeftTile = false;

        upTile = Physics2D.Linecast (Vector2.up + new Vector2 (ti
```

```
le.transform.position.x, tile.transform.position.y),
            new Vector2 (tile.transform.position.x, tile.transfor
m.position.y), floorLayer);

        rightTile = Physics2D.Linecast (Vector2.right + new Vecto
r2 (tile.transform.position.x, tile.transform.position.y),
            new Vector2 (tile.transform.position.x, tile.transfor
m.position.y), floorLayer);

        downTile = Physics2D.Linecast (Vector2.down + new Vector2
 (tile.transform.position.x, tile.transform.position.y),
            new Vector2 (tile.transform.position.x, tile.transfor
m.position.y), floorLayer);

        leftTile = Physics2D.Linecast (Vector2.left + new Vector2
 (tile.transform.position.x, tile.transform.position.y),
            new Vector2 (tile.transform.position.x, tile.transfor
m.position.y), floorLayer);

        //Check all tiles for null, hit current object or it has
a wall on it.

        if (upTile.collider != null && !(upTile.collider.GetCompo
nent<Tile> ().hasWall ()))
        {
            hasUpTile = true;

        }

        if (rightTile.collider != null && !(rightTile.collider.Ge
tComponent<Tile> ().hasWall ()))
        {
            hasRightTile = true;

        }

        if (downTile.collider != null && !(downTile.collider.GetC
omponent<Tile> ().hasWall ()))
        {
            hasDownTile = true;


        }
```

```
        if (leftTile.collider != null && !(leftTile.collider.GetC
omponent<Tile> ().hasWall ()))
        {
            hasLeftTile = true;
        }


        tile.GetComponentInParent<BoxCollider2D> ().enabled = tru
e;

        if (hasUpTile) {
            surroundingTiles [0] = upTile;
            tileCount++;
        }

        if (hasRightTile)
        {
            surroundingTiles [1] = rightTile;
            tileCount++;
        }



        if (hasDownTile)
        {
            surroundingTiles [2] = downTile;
            tileCount++;
        }



        if (hasLeftTile)
        {
            surroundingTiles [3] = leftTile;
            tileCount++;
        }



        Debug.Log (tileCount);

        return tileCount;
```

```
    }

    public void setDistance(GameObject tile, RaycastHit2D[] surro
undingTiles, char type)
    {
        Debug.Log ("In setDistance");


        foreach (RaycastHit2D distance in surroundingTiles)
        {


            Debug.Log ("In loop for distance");
            if (distance.collider != null) {
                switch (type) {
                case 'g':
                    Debug.Log ("Gold current tile" + tile.GetComp
onent<Tile> ().distanceGold);
                    Debug.Log ("Gold surrounding tile " + distanc
e.collider.GetComponent<Tile> ().distanceGold);
                    if (tile.GetComponent<Tile> ().distanceGold >
 distance.collider.GetComponent<Tile> ().distanceGold + 1) {
                        tile.GetComponent<Tile> ().distanceGold =
 distance.collider.GetComponent<Tile> ().distanceGold + 1;
                    }
                    break;
                case 's':
                    if (tile.GetComponent<Tile> ().distanceSilver
 > distance.collider.GetComponent<Tile> ().distanceSilver + 1) {
                        tile.GetComponent<Tile> ().distanceSilver
 = distance.collider.GetComponent<Tile> ().distanceSilver + 1;
                    }
                    break;
                case 'c':
                    if (tile.GetComponent<Tile> ().distanceCopper
 > distance.collider.GetComponent<Tile> ().distanceCopper + 1) {
                        tile.GetComponent<Tile> ().distanceCopper
 = distance.collider.GetComponent<Tile> ().distanceCopper + 1;
                    }
                    break;
                case 'e':
                    if (tile.GetComponent<Tile> ().distanceExit >
```

```
 distance.collider.GetComponent<Tile> ().distanceExit + 1) {
                        tile.GetComponent<Tile> ().distanceExit =
 distance.collider.GetComponent<Tile> ().distanceExit + 1;
                    }
                    break;
                default:
                    return;
                }
            }
        }

    }

    public bool finshCheck(RaycastHit2D[] _surroundingTiles, int
setDistance, char type)
    {
        bool toRemove = true;

        switch (type)
        {
        case 'g':
            foreach (RaycastHit2D tile in _surroundingTiles) {


                if (tile.collider != null) {

                    int distanceToGold = tile.collider.GetCompone
nt<Tile> ().distanceGold;

                    if ((distanceToGold == 100)) {
                        toRemove = false;
                    }

                    if (distanceToGold > setDistance + 1) {
                        toRemove = false;
                    }



                }

            }
            break;
```

```
        case 's':

            foreach (RaycastHit2D tile in _surroundingTiles) {

                if (tile.collider != null) {

                    int distanceToSilver = tile.collider.GetCompo
nent<Tile> ().distanceSilver;

                    if ((distanceToSilver == 100) || (distanceToS
ilver > setDistance + 1)) {
                        toRemove = false;
                    }
                }

            }


            break;
        case 'c':

            foreach (RaycastHit2D tile in _surroundingTiles) {
                if (tile.collider != null) {
                    int distanceToCopper = tile.collider.GetCompo
nent<Tile> ().distanceCopper;

                    if ((distanceToCopper == 100) || (distanceToC
opper > setDistance + 1)) {
                        toRemove = false;
                    }
                }

            }

            break;
        case 'e':

            foreach (RaycastHit2D tile in _surroundingTiles) {
                if (tile.collider != null) {
                    int distanceToExit = tile.collider.GetCompone
nt<Tile> ().distanceExit;

                    if ((distanceToExit == 100) || (distanceToExi
```

```
t > setDistance + 1)) {
                        toRemove = false;
                    }
                }

            }

            break;
        default:
            toRemove = false;
            break;
        }

        return toRemove;
    }



}


using UnityEngine;
using System.Collections;

public class SmartSelect : MonoBehaviour {

    public static double ValueOfGold, ValueOfSilver, ValueOfCoppe
r, ValueOfExit;
    public double TimeGold, TimeSilver, TimeCopper, TimeExit;
    public double RemeberGold, RemeberSilver, RemeberCopper, Reme
berExit;

    public int[] GoldArray;
    public int[] SilverArray;
    public int[] CopperArray;
    public int[] ExitArray;

    public double UpTile, RightTile, LeftTile, DownTile;

    public int tileToMoveTo; // 0 = Up, 1 = right, 2 = down, 3 =
left
    public int TimeDelay = 0;
```

```
public int ResetTime = 60;

// Use this for initialization
void Start () {

    GoldArray = new int[4];
    SilverArray = new int[4];
    CopperArray = new int[4];
    ExitArray = new int[4];

    for (int i = 0; i < 4; i++) {

        GoldArray [i] = 0;
        SilverArray [i] = 0;
        CopperArray [i] = 0;
        ExitArray [i] = 0;

    }

    TimeGold = 10;
    TimeSilver = 10;
    TimeCopper = 10;
    TimeExit = 10;

    ValueOfGold = 400;
    ValueOfSilver = 170;
    ValueOfCopper = 80;
    ValueOfExit = 40;

    RemeberGold = 30;
    RemeberSilver = 30;
    RemeberCopper = 45;
    RemeberExit = 25;
    }



// Update is called once per frame
void Update () {

    if (TimeDelay == 0) {

        TimeDelay = ResetTime;
```

```
            TimeCopper++;
            TimeGold++;
            TimeSilver++;
            TimeExit++;

        }

        TimeDelay--;

    }

    public int SelectTile()
    {



        UpTile = CalculateTileWeights (0);

        RightTile = CalculateTileWeights (1);

        DownTile = CalculateTileWeights (2);

        LeftTile = CalculateTileWeights (3);

        if (UpTile > RightTile && UpTile > DownTile && UpTile > L
eftTile) {
            return 1;
        } else if (RightTile > DownTile && RightTile > LeftTile)
{
            return 2;
        }
        else if (DownTile > LeftTile) {
            return 3;
        }
        else {
            return 4;
        }

    }

    //0 = up
    private double CalculateTileWeights(int tile)
```

```
    {
        double answer =
            (0 - (0 - ValueOfGold) / (1 + Mathf.Exp ((float)(-
TimeGold + RemeberGold)))) / GoldArray[tile] +
            (0 - (0 - ValueOfSilver) / (1 + Mathf.Exp ((float)(-
TimeSilver + RemeberSilver)))) / SilverArray [tile]  +
            (0 - (0 - ValueOfCopper) / (1 + Mathf.Exp ((float)(-
TimeCopper + RemeberCopper)))) / CopperArray [tile] +
            (0 - (0 - ValueOfExit) / (1 + Mathf.Exp ((float)(-
TimeExit + RemeberExit)))) / ExitArray [tile];
        //Debug.Log (TimeExit + " " + RemeberExit + " " + ExitArr
ay [tile] + " " + ValueOfExit);
        //Debug.Log (TimeGold + " " + RemeberGold + " " + GoldArr
ay [tile] + " " + ValueOfGold);

        return answer;
    }
}


using UnityEngine;
using System.Collections;


//The abstract keyword enables you to create classes and class me
mbers that are incomplete and must be implemented in a derived cl
ass.
public abstract class MovingObject : MonoBehaviour
{
    public float moveTime = 0.1f;           //Time it will take
object to move, in seconds.
    public LayerMask blockingLayer;           //Layer on which c
ollision will be checked.


    public BoxCollider2D boxCollider;       //The BoxCollider2D
 component attached to this object.
    private Rigidbody2D rb2D;               //The Rigidbody2D co
mponent attached to this object.
    private float inverseMoveTime;          //Used to make move
ment more efficient.
```

```
    //Protected, virtual functions can be overridden by inheritin
g classes.
    protected virtual void Start ()
    {
        //Get a component reference to this object's BoxCollider2
D
        boxCollider = GetComponent <BoxCollider2D> ();

        //Get a component reference to this object's Rigidbody2D
        rb2D = GetComponent <Rigidbody2D> ();

        //By storing the reciprocal of the move time we can use i
t by multiplying instead of dividing, this is more efficient.
        inverseMoveTime = 1f / moveTime;
    }


    //Move returns true if it is able to move and false if not.
    //Move takes parameters for x direction, y direction and a Ra
ycastHit2D to check collision.
    protected bool Move (int xDir, int yDir, out RaycastHit2D hit
)
    {
        //Store start position to move from, based on objects cur
rent transform position.
        Vector2 start = transform.position;

        // Calculate end position based on the direction paramete
rs passed in when calling Move.
        Vector2 end = start + new Vector2 (xDir, yDir);

        //Disable the boxCollider so that linecast doesn't hit th
is object's own collider.
        boxCollider.enabled = false;

        //Cast a line from start point to end point checking coll
ision on blockingLayer.
        hit = Physics2D.Linecast (start, end, blockingLayer);

        //Re-enable boxCollider after linecast
        boxCollider.enabled = true;

        //Check if anything was hit
```

```
        if(hit.transform == null)
        {
            //If nothing was hit, start SmoothMovement co-
routine passing in the Vector2 end as destination
            //StartCoroutine (SmoothMovement (end));


            rb2D.MovePosition (end);



            //Return true to say that Move was successful
            return true;
        }

        //If something was hit, return false, Move was unsuccesfu
l.
        return false;
    }


    //Co-
routine for moving units from one space to next, takes a paramete
r end to specify where to move to.
    protected IEnumerator SmoothMovement (Vector3 end)
    {
        //Calculate the remaining distance to move based on the s
quare magnitude of the difference between current position and en
d parameter.
        //Square magnitude is used instead of magnitude because i
t's computationally cheaper.
        float sqrRemainingDistance = (transform.position - end).s
qrMagnitude;




        //While that distance is greater than a very small amount
 (Epsilon, almost zero):
        while(sqrRemainingDistance > float.Epsilon)
        {
            //Find a new position proportionally closer to the en
d, based on the moveTime
```

```
            Vector3 newPostion = Vector3.MoveTowards(rb2D.positio
n, end, inverseMoveTime * Time.deltaTime);

            //newPostion = new Vector3((float)((int)newPostion.x)
, (float)((int)newPostion.y), 0f);

            //Call MovePosition on attached Rigidbody2D and move
it to the calculated position.
            rb2D.MovePosition (newPostion);

            //Recalculate the remaining distance after moving.
            sqrRemainingDistance = (transform.position - end).sqr
Magnitude;



            //Return and loop until sqrRemainingDistance is close
 enough to zero to end the function
            yield return null;
        }
    }


    //The virtual keyword means AttemptMove can be overridden by
inheriting classes using the override keyword.
    //AttemptMove takes a generic parameter T to specify the type
 of component we expect our unit to interact with if blocked (Pla
yer for Enemies, Wall for Player).*****
    protected virtual void AttemptMove <T> (int xDir, int yDir)
        where T : Component
    {
        //Hit will store whatever our linecast hits when Move is
called.
        RaycastHit2D hit;

        //Set canMove to true if Move was successful, false if fa
iled.
        bool canMove = Move (xDir, yDir, out hit);

        //Check if nothing was hit by linecast
        if(hit.transform == null)
            //If nothing was hit, return and don't execute furthe
r code.
```

```
            return;

        //Get a component reference to the component of type T at
tached to the object that was hit
        T hitComponent = hit.transform.GetComponent <T> ();

        //If canMove is false and hitComponent is not equal to nu
ll, meaning MovingObject is blocked and has hit something it can
interact with.
        if(!canMove && hitComponent != null)

            //Call the OnCantMove function and pass it hitCompone
nt as a parameter.
            OnCantMove (hitComponent);
    }


    //The abstract modifier indicates that the thing being modifi
ed has a missing or incomplete implementation.
    //OnCantMove will be overriden by functions in the inheriting
 classes.
    protected abstract void OnCantMove <T> (T component)
        where T : Component;
}

using UnityEngine;
using System.Collections;

using System.Collections.Generic;       //Allows us to use Lists
.

//Enemy inherits from MovingObject, our base class for objects th
at can move, Player also inherits from this.
public class Enemy : MovingObject
{
    public int playerDamage;                                //The am
ount of food points to subtract from the player when attacking.
    public AudioClip attackSound1;                          //First
 of two audio clips to play when attacking the player.
    public AudioClip attackSound2;                          //Secon
d of two audio clips to play when attacking the player.

    public LayerMask floorLayer;
```

```
    public Vector2 closeGold, closeSilver, closeCopper, closeExit
;
    public int distanceToGold, distanceToSilver, distanceToCopper
, distanceToExit;
    public Tile currentTile, TileUp, TileDown, TileRight, TileLef
t;

    public float smartDistance, prevSDist;

    public Vector2 target, player;                              //
Transform to attempt to move toward each turn.


    private Animator animator;                                  //Varia
ble of type Animator to store a reference to the enemy's Animator
 component.


    //Start overrides the virtual Start function of the base clas
s.
    protected override void Start ()
    {
        //Register this enemy with our instance of GameManager by
 adding it to a list of Enemy objects.
        //This allows the GameManager to issue movement commands.
        GameManager.instance.AddEnemyToList (this);

        //Get and store a reference to the attached Animator comp
onent.
        animator = GetComponent<Animator> ();

        //Find the Player GameObject using it's tag and store a r
eference to its transform component.
        //target = GameObject.FindGameObjectWithTag("Player").tra
nsform;

        distanceToGold = 100;
        distanceToSilver = 100;
        distanceToCopper = 100;
        distanceToExit = 100;

        target = new Vector2 (1, 0);
```

```
        //Call the start function of our base class MovingObject.
        base.Start ();
    }


    //Override the AttemptMove function of MovingObject to includ
e functionality needed for Enemy to skip turns.
    //See comments in MovingObject for more on how base AttemptMo
ve function works.
    protected override void AttemptMove <T> (int xDir, int yDir)
    {

        //Call the AttemptMove function from MovingObject.
        base.AttemptMove <T> (xDir, yDir);

    }

    public void getTiles()
    {
        Vector2 start = new Vector2 (transform.position.x, transf
orm.position.y);
        RaycastHit2D CastUp, CastDown, CastRight, CastLeft;
        Vector2 endUp, endDown, endRight, endLeft;
        RaycastHit2D[] rayCastArray = new RaycastHit2D[4];
        int lastGold, lastSilver, lastCopper, lastExit;
        int random = Random.Range (1, 3);

        lastGold = 100;
        lastSilver = 100;
        lastCopper = 100;
        lastExit = 100;

        endUp = start + Vector2.up;
        endDown = start + Vector2.down;
        endRight = start + Vector2.right;
        endLeft = start + Vector2.left;

        closeGold = endUp;
        closeSilver = endUp;
        closeCopper = endUp;
        closeExit = endUp;
```

```
        boxCollider.enabled = false;


        CastUp = Physics2D.Linecast(endUp, start, floorLayer);
        CastDown = Physics2D.Linecast(endDown, start, floorLayer)
;
        CastRight = Physics2D.Linecast (endRight, start, floorLay
er);
        CastLeft = Physics2D.Linecast (endLeft, start, floorLayer
);

        rayCastArray [0] = CastUp;
        rayCastArray [2] = CastDown;
        rayCastArray [1] = CastRight;
        rayCastArray [3] = CastLeft;

        for (int i = 0; i < 4; i++) {
            GetComponent<SmartSelect> ().GoldArray [i] = 100;
            GetComponent<SmartSelect> ().SilverArray [i] = 100;
            GetComponent<SmartSelect> ().CopperArray [i] = 100;
            GetComponent<SmartSelect> ().ExitArray [i] = 100;
        }


        for (int i = 0; i < 4; i++)
        {
            if (rayCastArray [i].collider != null) {

                GetComponent<SmartSelect> ().GoldArray [i] = rayC
astArray [i].collider.GetComponent<Tile> ().distanceGold;
                GetComponent<SmartSelect> ().SilverArray [i] = ra
yCastArray [i].collider.GetComponent<Tile> ().distanceSilver;
                GetComponent<SmartSelect> ().CopperArray [i] = ra
yCastArray [i].collider.GetComponent<Tile> ().distanceCopper;
                GetComponent<SmartSelect> ().ExitArray [i] = rayC
astArray [i].collider.GetComponent<Tile> ().distanceExit;

                if (rayCastArray [i].collider.GetComponent<Tile>
().distanceGold < lastGold) {
                    lastGold = rayCastArray [i].collider.GetCompo
nent<Tile> ().distanceGold;
                    closeGold = new Vector2 (rayCastArray [i].col
```

```
lider.GetComponent<Tile> ().transform.position.x,
                        rayCastArray [i].collider.GetComponent<Ti
le> ().transform.position.y);
                } else if (rayCastArray [i].collider.GetComponent
<Tile> ().distanceGold == lastGold) {
                    if (random == 1) {
                        closeGold = new Vector2 (rayCastArray [i]
.collider.GetComponent<Tile> ().transform.position.x,
                            rayCastArray [i].collider.GetComponen
t<Tile> ().transform.position.y);
                    }
                }

                if (rayCastArray [i].collider.GetComponent<Tile>
().distanceSilver < lastSilver) {
                    lastSilver = rayCastArray [i].collider.GetCom
ponent<Tile> ().distanceSilver;
                    closeSilver = new Vector2 (rayCastArray [i].c
ollider.GetComponent<Tile> ().transform.position.x,
                        rayCastArray [i].collider.GetComponent<Ti
le> ().transform.position.y);
                } else if (rayCastArray [i].collider.GetComponent
<Tile> ().distanceSilver == lastSilver) {

                    if (random == 1) {
                        closeSilver = new Vector2 (rayCastArray [
i].collider.GetComponent<Tile> ().transform.position.x,
                            rayCastArray [i].collider.GetComponen
t<Tile> ().transform.position.y);

                    }

                }

                if (rayCastArray [i].collider.GetComponent<Tile>
().distanceCopper < lastCopper) {
                    lastCopper = rayCastArray [i].collider.GetCom
ponent<Tile> ().distanceCopper;
                    closeCopper = new Vector2 (rayCastArray [i].c
ollider.GetComponent<Tile> ().transform.position.x,
                        rayCastArray [i].collider.GetComponent<Ti
le> ().transform.position.y);
                } else if (rayCastArray [i].collider.GetComponent
```

```
<Tile> ().distanceCopper == lastCopper) {

                    if (random == 1) {
                        closeCopper = new Vector2 (rayCastArray [
i].collider.GetComponent<Tile> ().transform.position.x,
                            rayCastArray [i].collider.GetComponen
t<Tile> ().transform.position.y);
                    }

                }

                if (rayCastArray [i].collider.GetComponent<Tile>
().distanceExit < lastExit) {
                    lastExit = rayCastArray [i].collider.GetCompo
nent<Tile> ().distanceExit;
                    closeExit = new Vector2 (rayCastArray [i].col
lider.GetComponent<Tile> ().transform.position.x,
                        rayCastArray [i].collider.GetComponent<Ti
le> ().transform.position.y);
                } else if (rayCastArray [i].collider.GetComponent
<Tile> ().distanceExit == lastExit) {

                    if (random == 1) {
                        closeExit = new Vector2 (rayCastArray [i]
.collider.GetComponent<Tile> ().transform.position.x,
                            rayCastArray [i].collider.GetComponen
t<Tile> ().transform.position.y);
                    }

                }
            } else {

                GetComponent<SmartSelect> ().GoldArray [i] = 100;
                GetComponent<SmartSelect> ().SilverArray [i] = 10
0;
                GetComponent<SmartSelect> ().CopperArray [i] = 10
0;
                GetComponent<SmartSelect> ().ExitArray [i] = 100;

            }

        }
```

```
        distanceToGold = lastGold;
        distanceToSilver = lastSilver;
        distanceToCopper = lastCopper;
        distanceToExit = lastExit;



        boxCollider.enabled = true;


    }

    public void selectTarget()
    {

        getTiles ();



        if (distanceToCopper == 2 )
        {
            GetComponent<SmartSelect> ().TimeCopper = 0;
        }
        if (distanceToGold == 2) {
            GetComponent<SmartSelect> ().TimeGold = 0;
        }

        if (distanceToSilver == 2) {
            GetComponent<SmartSelect> ().TimeSilver = 0;
        }
        if (distanceToExit == 2) {
            GetComponent<SmartSelect> ().TimeExit = 0;
        }

        player = new Vector2 (GameObject.FindGameObjectWithTag ("
Player").transform.position.x,
            GameObject.FindGameObjectWithTag ("Player").transform
.position.y);

        switch (GetComponent<SmartSelect> ().SelectTile ())
        {
        case 1:
            target = new Vector2 (0, 1);
```

```
            break;
        case 2:
            target = new Vector2 (1,0);
            break;
        case 3:
            target = new Vector2 (0, -1);
            break;
        case 4:
            target = new Vector2 (-1, 0);
            break;
        default:
            target = new Vector2 (0, 0);
            break;
        }



    }

    //if player within range, then caught
    public void withinRange (int xDir, int yDir)
    {
        //used to find the tag of the player and put it in a vari
able , so can apply it to this the if statement next, this method
 was also used to maybe find the coin in reference to the enemy
        GameObject player = GameObject.FindGameObjectWithTag ("Pl
ayer");

        //range of tiles around enemy - if player steps within th
at range of 2 tiles around, then you are caught and it's game ove
r
        if (player.transform.position.x <= (transform.position.x
+ 2) && player.transform.position.x >= (transform.position.x - 2)
 &&
            player.transform.position.y <= (transform.position.y
+ 2) && player.transform.position.y >= (transform.position.y - 2)
)
        {
            GameManager.instance.GameOver ();
        }
    }
```

```
    //MoveEnemy is called by the GameManger each turn to tell eac
h Enemy to try to move towards the player.
    public void MoveEnemy ()
    {
        //Called to select the target tile to move to
        //sets the target Vector2 to that tile
        selectTarget ();

        //Declare variables for X and Y axis move directions, the
se range from -1 to 1.
        //These values allow us to choose between the cardinal di
rections: up, down, left and right.
        int xDir = (int)target.x;
        int yDir = (int)target.y;

        withinRange(xDir,yDir);
        AttemptMove <Player> (xDir, yDir);
    }



    //OnCantMove is called if Enemy attempts to move into a space
 occupied by a Player, it overrides the OnCantMove function of Mo
vingObject
    //and takes a generic parameter T which we use to pass in the
 component we expect to encounter, in this case Player
    //--
- not used, but functionality left for future use if necessary
    protected override void OnCantMove <T> (T component)
    {
        //Declare hitPlayer and set it to equal the encountered c
omponent.
        //Player hitPlayer = component as Player;

        //Call the LoseFood function of hitPlayer passing it play
erDamage, the amount of foodpoints to be subtracted.
        //hitPlayer.LoseFood (playerDamage);

        //Set the attack trigger of animator to trigger Enemy att
ack animation.
        //animator.SetTrigger ("enemyAttack");
```

```
        //Call the RandomizeSfx function of SoundManager passing
in the two audio clips to choose randomly between.
        //SoundManager.instance.RandomizeSfx (attackSound1, attac
kSound2);
    }
}


using UnityEngine;
using System.Collections;
using UnityEngine.UI;   //Allows us to use UI.
using UnityEngine.SceneManagement;

//Player inherits from MovingObject, our base class for objects t
hat can move, Enemy also inherits from this.
public class Player : MovingObject
{
    public float restartLevelDelay = 1f;        //Delay time in s
econds to restart level.
    public int pointsGold = 100;
    public int pointsSilver = 50;
    public int pointsCopper = 25;
    public int wallDamage = 1;                  //How much dama
ge a player does to a wall when chopping it. --
- not used, but functionality left for future use if necessary
    public Text scoreText;                      //UI Text to di
splay current player total score.
    public Text healthText;                      //display curr
ent health --
- not used, but functionality left for future use if necessary
    public AudioClip moveSound1;                //1 of 2 Audio cl
ips to play when player moves.
    public AudioClip moveSound2;                //2 of 2 Audio cl
ips to play when player moves.
    public AudioClip eatSound1;                  //1 of 2 Audio
 clips to play when player collects a coin object.
    public AudioClip eatSound2;                  //2 of 2 Audio
 clips to play when player collects a coin object.
    public AudioClip drinkSound1;               //1 of 2 Audio c
lips to play when player collects a coin object.
    public AudioClip drinkSound2;               //2 of 2 Audio c
lips to play when player collects a coin object.
    public AudioClip gameOverSound;              //Audio clip t
```

```
o play when player dies.

    private Animator animator;                    //Used to store
 a reference to the Player's animator component.
    private int score;
    private int health=3;                         //player health
--- not used, but functionality left for future use if necessary
    private Vector2 touchOrigin = -
Vector2.one;    //Used to store location of screen touch origin f
or mobile controls.


    //Start overrides the Start function of MovingObject
    protected override void Start ()
    {
        //Get a component reference to the Player's animator comp
onent
        animator = GetComponent<Animator>();

        //Get the current food point total stored in GameManager.
instance between levels.
        score = GameManager.instance.playerScore;


        //Set the scoreText to reflect the current player total s
core.
        scoreText.text = "Score: " + score;

        //Call the Start function of the MovingObject base class.
        base.Start ();
    }


    //This function is called when the behaviour becomes disabled
 or inactive.
    private void OnDisable ()
    {
        GameManager.instance.playerScore = score;
    }


    private void Update ()
    {
```

```
        //If it's not the player's turn, exit the function.
        if(!GameManager.instance.playersTurn) return;

        int horizontal = 0;      //Used to store the horizontal m
ove direction.
        int vertical = 0;        //Used to store the vertical mov
e direction.

        //Check if we are running either in the Unity editor or i
n a standalone build.
        #if UNITY_STANDALONE || UNITY_WEBPLAYER

        //Get input from the input manager, round it to an intege
r and store in horizontal to set x axis move direction
        horizontal = (int) (Input.GetAxisRaw ("Horizontal"));

        //Get input from the input manager, round it to an intege
r and store in vertical to set y axis move direction
        vertical = (int) (Input.GetAxisRaw ("Vertical"));

        //Check if moving horizontally, if so set vertical to zer
o.
        if(horizontal != 0)
        {
            vertical = 0;
        }
        //Check if we are running on iOS, Android, Windows Phone
8 or Unity iPhone
        #elif UNITY_IOS || UNITY_ANDROID || UNITY_WP8 || UNITY_IP
HONE

        #endif //End of mobile platform dependendent compilation
section started above with #elif
        //Check if we have a non-
zero value for horizontal or vertical
        if(horizontal != 0 || vertical != 0)
        {
            //Call AttemptMove passing in the generic parameter W
all, since that is what Player may interact with if they encounte
r one (by attacking it)
            //Pass in horizontal and vertical as parameters to sp
ecify the direction to move Player in.
            AttemptMove<Wall> (horizontal, vertical);
```

```
        }
    }

    //AttemptMove overrides the AttemptMove function in the base
class MovingObject
    //AttemptMove takes a generic parameter T which for Player wi
ll be of the type Wall, it also takes integers for x and y direct
ion to move in.
    protected override void AttemptMove <T> (int xDir, int yDir)
    {
        //Call the AttemptMove method of the base class, passing
in the component T (in this case Wall) and x and y direction to m
ove.
        base.AttemptMove <T> (xDir, yDir);

        //Hit allows us to reference the result of the Linecast d
one in Move.
        RaycastHit2D hit;

        //If Move returns true, meaning Player was able to move i
nto an empty space.
        if (Move (xDir, yDir, out hit))
        {
            //Call RandomizeSfx of SoundManager to play the move
sound, passing in two audio clips to choose from.
            SoundManager.instance.RandomizeSfx (moveSound1, moveS
ound2);
        }

        //CheckIfGameOver ();

        //Set the playersTurn boolean of GameManager to false now
 that players turn is over.
        GameManager.instance.playersTurn = false;
    }


    //OnCantMove overrides the abstract function OnCantMove in Mo
vingObject.
    //It takes a generic parameter T which in the case of Player
is a Wall which the player can attack and destroy.
    //--
- not used, but functionality left for future use if necessary
```

```
    protected override void OnCantMove <T> (T component)
    {
        //Set hitWall to equal the component passed in as a param
eter.
        Wall hitWall = component as Wall;

        //Call the DamageWall function of the Wall we are hitting
.
        //hitWall.DamageWall (wallDamage);

        //Set the attack trigger of the player's animation contro
ller in order to play the player's attack animation.
        animator.SetTrigger ("playerChop");
    }


    //OnTriggerEnter2D is sent when another object enters a trigg
er collider attached to this object (2D physics only).
    private void OnTriggerEnter2D (Collider2D other)
    {
        //Check if the tag of the trigger collided with is Exit.
        if(other.tag == "Exit")
        {
            //Invoke the Restart function to start the next level
 with a delay of restartLevelDelay (default 1 second).
            Invoke ("Restart", restartLevelDelay);

            //Disable the player object since level is over.
            enabled = false;
        }

        //Check if the tag of the trigger collided with Gold
        else if(other.tag == "Gold")
        {
            //Add gold to score
            score += pointsGold;

            //GameObject guard = GameObject.FindGameObjectWithTag
("Enemy"); ---tried to reference ValueofGold from enemy at first

            //used static on ValueofGold, etc. to to keep values
consistent throughout files (scripts), so if we change it here ,
it'll change back at the original
```

```
            //need to toggle with values to make route more "inte
lligent looking"
            SmartSelect.ValueOfGold = 1;

            //Update scoreText to represent current total and not
ify player that they gained points
            scoreText.text = "+" + pointsGold + "Score: " + score
;

            //Call the RandomizeSfx function of SoundManager and
pass in two eating sounds to choose between to play the eating so
und effect.
            SoundManager.instance.RandomizeSfx (eatSound1, eatSou
nd2);

            //Disable the gold coin object the player collided wi
th.
            other.gameObject.SetActive (false);
        }

        //Check if the tag of the trigger collided with Silver
        else if(other.tag == "Silver")
        {
            //Add silver to score
            score += pointsSilver;

            SmartSelect.ValueOfSilver = 1;

            //Update scoreText to represent current total and not
ify player that they gained points
            scoreText.text = "+" + pointsSilver + "Score: " + sco
re;

            //Call the RandomizeSfx function of SoundManager and
pass in two drinking sounds to choose between to play the drinkin
g sound effect.
            SoundManager.instance.RandomizeSfx (drinkSound1, drin
kSound2);

            //Disable the silver coin object the player collided
with.
            other.gameObject.SetActive (false);
        }
```

```
        else if(other.tag == "Copper")
        {
            //Add points to players score points total
            score += pointsCopper;
            //GameObject guard = GameObject.FindGameObjectWithTag
("Enemy");
            SmartSelect.ValueOfCopper = 1;

            //Update scoreText to represent current total and not
ify player that they gained points
            scoreText.text = "+" + pointsCopper + "Score: " + sco
re;

            //Call the RandomizeSfx function of SoundManager and
pass in two drinking sounds to choose between to play the drinkin
g sound effect.
            SoundManager.instance.RandomizeSfx (drinkSound1, drin
kSound2);

            //Disable the copper coin object the player collided
with.
            other.gameObject.SetActive (false);
        }
    }


    //Restart reloads the scene when called.
    private void Restart ()
    {
    //Load the last scene loaded, in this case Main, the only sce
ne in the game.
    SceneManager.LoadScene("Main");
    }


    //LoseFood is called when an enemy attacks the player.
    //It takes a parameter loss which specifies how many points t
o lose.
    //--
- not used, but functionality left for future use if necessary
    public void LoseFood (int loss)
    {
        //Set the trigger for the player animator to transition t
```

```
o the playerHit animation.
        animator.SetTrigger ("playerHit");

        //Subtract lost food points from the players total.
        health -= loss;

        //Check to see if game has ended.
        CheckIfGameOver ();
    }


    //CheckIfGameOver checks if the player is out of food points
and if so, ends the game.
    //--
- not used, but functionality left for future use if necessary
    private void CheckIfGameOver ()
    {
        //Check if food point total is less than or equal to zero
.
        if(health <= 0)
        {
            //Call the PlaySingle function of SoundManager and pa
ss it the gameOverSound as the audio clip to play.
            SoundManager.instance.PlaySingle (gameOverSound);

            //Stop the background music.
            SoundManager.instance.musicSource.Stop();

            //Call the GameOver function of GameManager.
            GameManager.instance.GameOver ();
        }
    }
}
```

References

Doornbos, J. (2001). Those Darn Sims: What Makes Them Tick?, in lecture, *Game Developers*
Conference.

Doyle, P, (1999). Virtual Intelligence Form Artificial Reality: Building Stupid Agents in Smart Environments, in *Proc. AAAI '99 Spring Symposium on Artificial Intelligence and Computer Games*.

Korchnak, J. (2018). Implementation of Probabilistic Smart Terrain in Unity (Master's thesis, Youngstown State University, Youngstown, OH). Retrieved from https://etd.ohiolink.edu/.

Millington, I., & Funge, J. D. (2009). *Artificial intelligence for games*. Burlington, MA: Elsevier Morgan Kaufmann.

Sullins, John R. 2010. "PROBABILISTIC SMART TERRAIN." *International Journal on Artificial Intelligence Tools* 19 (4): 531–50. doi:10.1142/S0218213010000315.

Sullins, John R. 2011. "Multi-Agent Probabilistic Smart Terrain". CGAMES '11 Proceedings of the 2011 16th International Conference on Computer Games: 33-37

"Unity - 2D Roguelike Tutorial." 2016. *Unity*. Accessed November 26. https://unity3d.com/learn/tutorials/projects/2d-roguelike-tutorial.