UNIVERSITY OF EDINBURGH

MASTER'S THESIS

Discrete SIR Game for Understanding Epidemics

Author:
Mateo VARGAS

Supervisor:
Dr. Stephen GILMORE

A thesis submitted in fulfillment of the requirements for the degree of Master's of Science in Computer Science

in the

University of Edinburgh
School of Informatics

August 15, 2017

Declaration of Authorship

I, Mateo VARGAS, declare that this thesis titled, "Discrete SIR Game for Understanding Epidemics" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"...we believe that games are both a reflection of and a participator in human culture. Playing is as old as people are, and games offer us ways to laugh, think, collaborate, escape..."

Austin Walker, "A Note on Trump, Waypoint, and Why We Play"

University of Edinburgh

Abstract

College of Science and Engineering
School of Informatics

Master's of Science in Computer Science

Discrete SIR Game for Understanding Epidemics

by Mateo VARGAS

The work contained within sought to develop a digital game that conveyed the core principles of the SIR model for epidemics and improve the learning outcomes for students when learning this model. Initial work resulted in a simple C# that simulated the SIR model. This was then extended to become the central behavior in a short game. Both the simple script and the model in the game were evaluated to ensure their accurate portrayal of the model. The prototype for this game was then deployed to several participants in two short evaluation experiments. Though neither work demonstrated a conclusive trend, the work within provides some insight into creating a game that conveys information about the SIR model of epidemics. ...

Acknowledgements

I would like to thank my father, mother, and sister for all their support from across the ocean, as well as Dr. Gilmore for all his advice and support. I would also like to extend my gratitude to my friend, Daniel, for allowing the use of his music...

Contents

Dedicated to my father, Rodrigo...

Chapter 1

Introduction

Games have been a feature of human civilization since some of the earliest societies in recorded history. Citizens of the nation-state of Ur played board games as part of their leisure time [1], and evidence has been found of early dice use on the Indian subcontinent since around 2300 B.C.E. [11]. Now, with the explosion of digital technology in recent decades, games have expanded into the digital realm and proliferated greatly. Because of the relative youth of the medium, research into the effects of playing digital games has really only just begun. Recent research has increasingly examined the use of games (and digital games in particular) in improving learning outcomes for developing pupils. Studies such as Rowe et al.'s [13] and Pesare et al.'s [12] demonstrated that games-based learning through specially designed games in conjunction with traditional instruction can have a positive effect on learning in a student's education. A large reason for this bloom in research is that games as a medium seem to uniquely engage players through their interactivity in ways that older forms of entertainment do not. Developers have increasingly been creating commercial release games that attempt to impart some knowledge on the subject they are simulation. Many of these games have seen success and have been praised by experts. These games have been developed and convey some educational benefits in a myriad of areas, such as *Mini Metro* and its simple simulation of urban planning [43] or *Kerbal Space Program* and its simulation of space exploration and rocket science [44].

The Susceptible-Infected-Recovered (SIR) model for disease is a widely-used mathematical model which describes the way a particular disease travels through a population. It is an important part of epidemiology and is taught to many students in health-related fields. The mathematical equations dictate the way individuals change from susceptible, infected, and recovered, with specific constants to represent contacts with infected individuals and the rate of recovery. This disease model has been used to describe epidemics such as the "Hong Kong Flu" epidemic of 1968 and is commonly taught in epidemiology courses [41]. Learning the equations and understanding how they work concretely, however, is a difficult proposition when simply reading the numbers and variables used in the model. Additionally, many of the individuals learning this model may not necessarily be as proficient in mathematical modeling as they are in other fields. Thus, learning an abstract, math heavy model such as this may be more difficult for these individuals. Using a game that simulates the SIR model within a manipulable digital experience may then allow students to cognitively anchor these abstract equations onto a concrete example, thus improving their learning outcomes. By playing a game that they can interact with and view changes based on their actions, the model may become more than just a set of numbers and variables to them.

The objective of this project is to address the question: can a digital game improve the learning outcomes of students that are learning the SIR model of epidemics? It would seem that the ways games engage people and deliver immediate feedback make them a valuable tool in learning many things. Thus, a digital game may prove to be an ideal way with which to give students a better understanding of the SIR model for epidemics. Within the constraints of this project, this prototype of a digital game provides a proof of concept that teaching the SIR model may benefit from a companion digital game with which players interact.

The remainder of this work will detail the separate steps taken to complete this project. Chapter Two will examine the mathematics underpinning the SIR model and delve into current avenues of research in games-based learning. It will also provide some background on the environment in which the game was developed, the Unity Integrated Development Environment. Chapter Three will detail the process of development for this application, from planning to the programming. This will include the a discussion of designing the game, as well as go into detail on the development of the scripts necessary to complete the game. Chapter Four will examine the results of an initial evaluation of the model and then examine the results obtained via a cohort of testers. Chapter Five will provide a review of the work completed, as well as any conclusions that can be drawn from this project.

Chapter 2

Background

The SIR model is a potentially useful tool epidemiologists and public health workers use to understand the spread of epidemics within a population. These same individuals, however, are required to handle epidemics in a more direct manner; the abstract manner mathematical model does not lend itself to a concrete understanding of the spread of disease in real-world scenarios. This challenge may be overcome through the thoughtful use of games-based learning and serious games, which both have been shown to have increased student's knowledge acquisition and content understanding in a variety of scenarios [2]. This section will comprise an overview of the SIR model itself, followed by an overview of past and current research into games-based learning and serious games, and end with a description of the Unity Integrated Development Environment and the Unity Engine.

2.1 Games For Learning

Games have been an integral part of human culture for centuries, with some of the earliest societies engaging in play as a way to spend their limited leisure time. Recently, researchers have been focusing in on both the positive and negative outcomes of playing games, in the physical realm and in the virtual realm. Garris et al. indicate that the outcomes for playing games can be divided between skills based learning outcomes, cognitive outcomes, and affective outcomes [4]. At a later point, Wouters et al. proposed that games have four kinds of learning outcomes, which can be broken down into cognitive skills, motor skills, affective learning, and communicative learning [45]. In recent years, the proliferation of video game consoles, mobile phones/tablets, and digital marketplaces has made digital games easier to acquire for more people, thus increasing the time modern people engage with digital games. With that in mind, recent research has been increasingly focused on harnessing digital games specifically as a way to increase engagement with certain materials and topics.

Initially, the focus of investigations into the outcomes of playing digital games was on the negative outcomes; however, this focus has shifted as more research has been published. Early work into the positive benefits of playing games found that there is a link between playing violent video games and improved visual spatial abilities [2]. From this seed of research grew the tree that encompasses games-based learning, or the creation of games with educational, rather than entertainment, purposes in mind. Learning is most effective when students are actively engaged with problem based methods that provide immediate feedback, as Boyle et al. describe [6]. While intended for leisurely engagement, games provide the features Boyle describes as an integral part of their medium, emphasizing the potential benefits when they are used in an educational context and providing a link between playing games

and learning. The National Research Council of the United States put forth that games are promising because they motivate players with challenges and provide rapid feedback, while having their instructions tailored to their learning needs [7]. This link has been observed in real-world scenarios, from grade school classes to medical training, reported by Rowe et al. [13] and Pesare et al. [12].

2.1.1 Learning Games in School

To investigate this link between playing games and learning, Rowe et al. set out to examine the ways narrative can assist in engaging students and increasing their learning. They developed the game *Crystal Island*, a narrative-centered game with the objective of aiding in students understanding of material they learn in a traditional academic environment. This game was intended to work in conjunction with the traditionally-taught curriculum from the North Carolina School system's eighth grade biology course [13]. They developed the game with Valve Software's Source engine, a widely available platform with which to design digital games. The game itself tells the tale of a visitor traveling to an island that is in the midst of a mysterious epidemic. The player character is then tasked with the investigation of the disease, with the ultimate objective of finding the cure and stopping the spread of the disease. Each student's overall performance in understanding the material is evaluated using a combination of factors, such as sixteen multiple choice questions, the number of game objectives completed, and the final game score that was calculated for the student [13]. Evaluating along these measures, Rowe et al. observe that an increased level of engagement with the game correlated with improvement on the exit quiz. They measured engagement through the use of a presence questionnaire, originally created by [10]. This suggests that the students that were more active in playing the game experience a larger learning gain, which they noted occurred regardless of the prior knowledge level of the individual [13]. Additionally, these same trends were observed across groups of individuals with little prior gaming experience and groups with plenty of gaming experience [13]. It is important, however, to note that players who did demonstrate a greater prior content knowledge showed a higher engagement level, despite their findings that engagement measures were more associated with the post-test score. Rowe et al. point out that the key element to this improvement was a carefully crafted story and gameplay elements that reflect the lessons they seek to teach [13]. *Crystal Island* and the ensuing experiments demonstrate that narrative games can indeed have educational benefits when designed and played in conjunction with a traditional course curriculum.

2.1.2 Learning Games in Other Contexts

In a later study, Pesare et al. sought to develop games to assist in the training of medical professionals and patients. They aimed to develop one game which focused on training medical professionals of all levels and another with the goal of training patients how to manage their own illnesses [12]. Their objective was to improve the training of these populations through the use of games in combination with traditional training methods.

The first game simulated clinical cases, and used real patient data to create different scenarios for the different tiers of medical professionals, from nurses to specialists [12]. The game presents a player with a particular case, pulled from the database of anonymous patient information, and presents the player with several options. The options the player is presented with depends on their particular role in

the hospital; nurses get different options than physician's assistants, who themselves have different options than would a doctor. The game then rewards players for taking the correct course of action, according to the real patient data [12]. According to the player's performance, the game would then record the score and level for the player and update the case study to a different scenario. The game itself recorded a player's score and level, ranks (used to measure a participant's progress), and rewards (bonuses). The player performance was evaluated using a combination of the final game score and the performance on a pre- and post-test.

In the second game, the goal was to train patients to better manage their diseases. This game was designed to be played with the aid of a medical professional, who acts as sort of a game manager. The patient plays as an avatar in a role-playing game where progress is measured by the number of correct answers a player gives to problems posed to the game avatar [12]. Answering a question correctly allows the player to increase their score, while incorrect answers will cause the player to lose a life, with the ubiquitous standard of three lives being given at the beginning of the game. Each question was designed to have the player learn something about managing their disease. Once a player loses all three of their lives by giving incorrect answers, the medical professional assigned to their case would intervene and restart the game. The medical professional will then examine the learning gap and assign other training material to aid in supporting the patient [12]. Each patient's performance was then evaluated using a combination of their game score and pre- and post-tests.

While both of these games had small sample sizes in terms of their evaluations, their results are promising. These games demonstrate that games-based learning can be an interesting avenue to improve education and training in the medical field. Furthermore, they demonstrate that using games in both academic contexts and work training contexts can be beneficial for the participants. Games for learning now address a myriad of different disciplines, such as business, computing, geography, health, and science [2]. Because of these previous studies, it would seem that, when teaching students the SIR model, they would benefit from the use of an interactive game to simulate the abstract model. To develop such a game, the Unity development environment would be beneficial for its ease of access and use.

2.2 The SIR Model

Mathematical models have been used to describe many of the natural phenomena that humans have observed in the world, and the spread of disease is no different. The SIR model, specifically, is a mathematical model that simulates the spread of an infectious disease through a fixed population. The SIR model was developed in the wake of the Spanish Influenza Epidemic of 1918-1919 that killed between 20 and 50 million individuals worldwide [42]. First proposed by Kermack and McKendrick of the Royal School of Physicians, Edinburgh, in 1927, they observed that the termination of an infection is subject to the population density, infection, recovery, and death rates [9]. This led them to propose the SIR model as a way of explaining the rapid rise and then decline of infected patients seen in epidemics throughout history [9]. The model gained widespread use and has been used to model epidemics throughout the Twentieth century, such as the "Hong Kong Flu" epidemic of 1968. It utilizes a system of equations to describe the transition of individuals between the susceptible population (individuals who have yet to achieve immunity), the infected population (individuals who are carrying the disease), and the recovered population

(individuals who have either achieved immunization naturally or otherwise, as well as the dead). For this particular model, the independent variable is time, primarily measured in days. The dependent variable is the number of people in each population. The equations for each population are seen here as a function of t (time in days), with N representing the total population [41]:

$$S(t) = S$$

$$I(t) = I$$

$$R(t) = R$$

$$S(t) + I(t) + R(t) = N$$

These same equations can then be converted to model the fractions of the total population that each contain, with N symbolizing the entire population count. Each of these equations must then be able to be summed up to equal one at each time point [41]:

$$s(t) = \frac{S(t)}{N}$$

$$i(t) = \frac{I(t)}{N}$$

$$r(t) = \frac{R(t)}{N}$$

$$s(t) + i(t) + r(t) = 1$$

The first order derivatives of each of these equations, then, represent the rates of change for each subpopulation. These rates are determined by two different constants, which vary depending on the disease. The first constant represents the fixed number of contacts that occur each day, symbolized by b. The second constant, symbolized by k, is the fraction of individuals that recover during a given day [41]. With this in mind, the rate of change for each subpopulation is then modeled with the following formulas, with the first set representing the rates of change when working with the first system of equations presented and the second representing when working with the second system of equations:

$$\frac{dS}{dt} = -bS(t)I(t)$$

$$\frac{dR}{dt} = kI(t)$$

$$\frac{dI}{dt} = bS(t)I(t) - kI(t)$$

$$\frac{ds}{dt} = -bs(t)i(t)$$

$$\frac{dr}{dt} = ki(t)$$

$$\frac{di}{dt} = bs(t)i(t) - ki(t)$$

From these equations, it is evident that, at each time point t, the susceptible population is reduced as individuals are infected with the disease. For the recovered individuals, the subpopulation grows as a fraction of the infected population recovers

each day. The infected population, then, increases with the amount of individuals that change from susceptible to infected and decreases with the amount of infected individuals that recover each day. To those with a background in mathematics, understanding this may be trivial; however, not all health officials who would benefit from a deep understanding of this model have such a background. Utilizing games in conjunction with traditional teaching methods, then, allows the students to cognitively anchor these abstract concepts through the interaction with a virtual scenario.

2.3 Unity Integrated Development Environment

The Unity Integrated Development Environment (Unity IDE) has gained traction within the games industry recently because of its availability and the powerful tools which it provides to developers. Specifically amongst small, third-party developers, the Unity IDE has become the most popular tool, with over 34% of the top 1000 mobile games having been developed using Unity [21]. Unity is even being increasingly used by larger game development studios such as Ubisoft, amongst others. There are games of all stripes being created in this engine, with releases such as *Lara Croft: Go*, *Pokémon: Go*, *Cities: Skylines*, and *Firewatch*. Because of its free availability, as well as the amount of platforms Unity supports (everything from mobile devices to home video game consoles), it affords a unique set of powers with which to create video games [21]. The most important aspect of this is that Unity comes ready with a library that dictates the behavior of the Unity Engine. This Unity Engine provides many of the object behaviors that are required to function in the game, such as the definition of game objects, physics components, and so on. These can all be seen on the Unity Documentation page [40]. In addition, Unity provides a Unity Asset Store, which allows developers to scour the storefront for any necessary asssets, such as sound, sprites, scripts, library extensions and so on. For this project, only art assets and a single library extension was obtained using the asset store (these will be discussed in further detail in Chapter Three). The Editor allows for the manipulation of the game objects, whose code is dictated through the Unity Engine's library.

2.3.1 Unity Editor

The Unity Editor is where most of the design work takes place. Within this frame is contained all of the options to visually inspect the developing game, as well as any view of the different game objects that make up a single scene and the files that are imported for the current project. The toolbar at the top provides the developers ways to adjust objects in the scene, either by moving the object or changing the object's scale. It also provides ways for the developer to run the game and pause whenever necessary.

The Project Window allows for a structured view of all the files that are imported into the current project. This also constitutes the main file structure of the overall application. From here, a developer can view any files that are imported and add them onto game objects as needed, through a series of menus or by dragging and dropping. These assets can be imported from local files or obtained via the Unity Asset store, a digital marketplace for prefabricated assets. From here, one can create scripts which are then opened in MonoDevelop with all the requisite libraries and classes to function within the Unity environment.

The Hierarchy Window is where all of the current game objects that are required for the current scene can be viewed. These game objects can be anything from

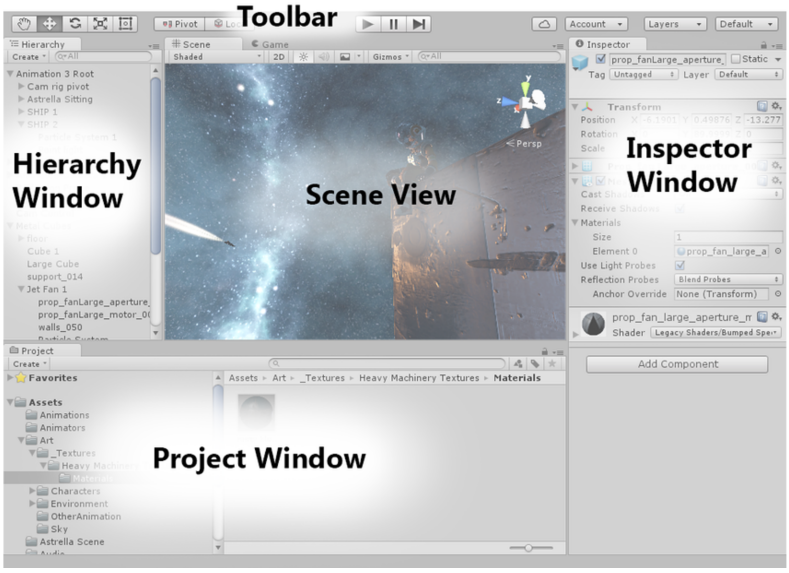


FIGURE 2.1: "Unity Editor Window [27]"

art, programming scripts, cameras, UI canvases, and so on. Objects that are being worked on can be selected in this window for closer inspection. This will also display which objects are not destroyed by the Unity engine between loaded scenes.

The Inspector Window allows the developer to more closely inspect individual game objects. Because each game object can have components attached to them, this window allows a developer to view and adjust the attached components. These components can include physics components to detect collision, rendering components to render artwork, scripts to determine behavior, text boxes, buttons, and so on. Scripts in particular have a unique feature; when a particular variable is public in the corresponding script, the developer can simply drag and drop the required object into the designated slot in the Inspector Window. Thus, variables can be assigned within the Unity Editor and not explicitly done via scripting (although that is also a viable option). Where necessary, this work will explain components required for this particular endeavor.

2.3.2 Unity Engine

The Unity Engine provides many libraries of methods and classes available for developers to dive straight into creating scripts for their games. These libraries dictate the behavior of many game functions, such as `UnityEngine.SceneManagement`, which provides methods to load and quit scenes that are in the build [40]. It also provides classes that describe different components that may be needed for game objects, such as the `SpriteRenderer` class, which describes an object that allows the engine to render sprites [35]. Engineering the libraries and classes to demonstrate the behavior required for a particular project is then up to the programmer.

For this particular project, the libraries required from the Unity Engine were the `UnityEngine.SceneManagement` and the `UnityEngine.UI`. The `SceneManagement` library was used in this particular project to load between the four scenes created for the prototype, as well as quit the application when desired. The classes utilized for scripting were:

- Sprite (Sprite object for use in 2D gameplay [34]),

- SpriteRenderer (renders a Sprite for 2D graphics [35]),
- Rigidbody2D (Rigidbody physics component for 2D sprites [32]),
- Box Collider2D (Collider for 2D physics representing an axis-aligned rectangle [17]),
- Canvas (element that can be used for screen rendering [20]),
- Button (a button that can be clicked on to trigger events [18]),
- Text (the default graphic to draw font data to screen [36]),
- and AudioClip (a container for audio data [15])

The documentation for all of these libraries and classes can be found at [40]. These classes will be discussed more in detail where appropriate when discussing the methodology.

Chapter 3

Methodology

To create this game, there were two phases of work done, broken up by evaluation phases. The first phase involved writing a script that simulated the spread of an infection through a population solely via programmatic conventions and instructions. This model would accurately portray the SIR model. The second phase involved creating the four scenes necessary for the overall prototype. This included editing and extending the original script from phase one to utilize the Unity engine's extensive library of classes and methods. Every script was written in C# for this particular project. What follows is a detailed explanation of the necessary scripting components for each phase of work, as well as an explanation of the unique techniques in Unity that were used.

3.1 Simple SIR Model in C#

The initial phase done was a preliminary foray into designing and implementing a model that accurately simulates the SIR model entirely programmatically. This created a solid foundation to use for later scripting work when implementing the same behavior in a game environment. With an accurate model with which to use as a base, there was less time spent planning and implementing how to approach this model within a game environment. To do this, time was spent in deciding which data structure would be best to use as the backbone to the model, as well as the variables which the model would track. The constructor is seen in Figure 3.1. Please refer to the figure(s) as needed.

3.1.1 SIR Model Constructor

The constructor, sirModel, takes in four values that dictate the behavior of the overall model: the initial numbers of susceptible and infected individuals (int `susceptible` and int `infected` respectively), the contact rate (double `b`) and the recovery rate (double `k`). It utilizes a Dictionary, called population, that keeps track of every individual in the population and their status. The individual's key is a unique number, while the value for each key is a string that indicates their status: "susceptible," "infected," or "recovered." A Dictionary was chosen as the most-apt data structure because of its O(1) retrieval time [3]. Knowing that the model would most likely need a quick way to look up a given individual and change their status, the O(1) look-up time would be crucial in implementing an efficient model. The individuals are then added to the Dictionary, one by one, the "susceptible" individuals being added at the front and the "infected" individuals being added immediately after. This would always add up to the total population, as the "recovered" population is initially set to zero. It also ensures that the contact rate and the recovery rate are set to the passed in parameters of `b` and `k`, so the model is aware of these values. A series of getter

and setter methods were then written to be able to get and set the susceptible count, the infected count, the recovered count, the contact rate, and the recovery rate.

```
/**
 *Class that defines the SIR Model of infection. This version was used to produce the initial phase data and tests
 *that are included in the evaluation. This is the basis of the class that will be modified and extended to work within
 *Unity.
 **/
public class sirModel{

    private Dictionary<int, string> population; //dictionary for entire pop. with <individual id, infection status>
    private int susceptible_count;
    private int infected_count;
    private int recovered_count;
    private double contacts;
    private int total_pop;
    private double recovery_rate;


    /**
     *Public constructor to create an instance of the SIR model. It takes the intial number of susceptible individuals,
     *infected individuals, the contact rate, and the recovery rate as arguments. This will be adjusted for the model
     *that will be used in the game (as the first two parameters will have to be linked to the game objects that
     *represent individuals.
     **/
    public sirModel(int susceptible, int infected, double b, double k){

        population = new Dictionary<int, string> ();
        susceptible_count = susceptible;
        infected_count = infected;
        recovered_count = 0;
        total_pop = susceptible_count + infected_count + recovered_count;
        contacts = b;
        recovery_rate = k;

        int i = 0;
        while (i < susceptible) {

            population.Add (i, "susceptible");
            i++;

        }

        for (int j = 0; j < infected; j++) {

            population.Add ((i) + j, "infected");
        }

    }
```

3.1.2 Infection and Recovery

The method that is instrumental in eliciting behavior that simulates the SIR model is the `infect_and_recover()` method. This method is the primary way the epidemic behavior is translated into programmatic code in this script. Originally, the plan was to implement the infection and recovery behavior as separate methods; however, when considering the mathematical equations (see Chapter2), the rate of infection and the rate of recovery must be calculated at once for the same time point. This is because the equations both have dependencies on the count of infected individuals. When shifting the focus to the in-game model, however, this method of calculation changes.

This method begins by calculating the rate of infection and the rate of recovery for the current time point. In this particular case, these equations are calculated as rates in terms of the whole population; this is opposed to viewing the total population as one and using fractions of one. The SIR model's equations have been detailed for both scenarios and can be seen in Chapter 2. This simplified the programmatic instructions that were needed to complete the rest of the method. The two variables, `rate_of_infection` and `rate_of_recovery`, determine the amount of individuals that change status at the given time point. The loops are then what actually ensure that the changes in status are made.

```
//Method to infect and recover a certain amount of people on the given day.
public void infect_and_recover(){


    //THIS WILL CHANGE WHEN SWITCHING TO THE GAME. WILL BE BASED ON ACTUAL GAME COLLISIONS
    double rate_of_infection = -(contacts) * ((double)susceptible_count / (double)total_pop) * (infected_count);
    double rate_of_recovery = recovery_rate * (infected_count);

    //loop over to infect
    int i = 0;
    for (int j = 0; j < population.Count; j++) {

            if (population [j] == "susceptible" && i > rate_of_infection) {
            population [j] = "infected";
            infected_count++;
            susceptible_count--;
            i = i - 1;
        }
    }

    //loop over to recover
    int k = 0;
    for (int j = population.Count-1; j >= 0; j--){

        if (population [j] == "infected" && k < rate_of_recovery) {

            population [j] = "recovered";
            infected_count--;
            recovered_count++;
            k = k + 1;

        }
    }
}
```

The first loop is where the infection spread occurs. There is a variable `i` which is set to zero. This variable `i` keeps track of the amount of infectious contacts that have occurred up to this point. While it is greater than the `rate_of_infection` variable, there are still infectious contacts to occur for this time point. It then loops over from zero to the total population count. At each iteration, it checks the status of the current individual. If the individual is "susceptible" and `i` is still greater than the `rate_of_infection`, then the current individual will change from "susceptible" to "infected," incrementing the `infected_count` and decrementing the `susceptible_count` variables. It also then will decrement `i` by one (because there an infectious contact). Once this loop is completed, the infectious contacts for the current time point are completed.

The second loop is where the recovery occurs. Because the Dictionary is set up so that the infected individuals are added after the susceptible individuals, this loop goes through the dictionary in reverse order. It keeps track of a variable, `k`, which keeps track of the amount of individuals that have recovered for this time point. Every iteration then checks if the current individual is "infected" and if `k` is still less than the rate of recovery. If both of these hold true, the current individual will be set to "recovered." The `infected_count` will then decrement and recovered_count and `k` will increment.

Following this pattern, the script demonstrated the requisite desired behavior for an infection that follows the SIR model. The results from evaluating this will be examined in Chapter Four. The creation of this script allowed for a better understanding in how best to approach implementing this model inside the game itself, which was the ultimate goal of this exploratory phase. While the infection and recovery portions of the model change from this script to the in-game model, the data structures and values for the in-game model remain quite similar. This will be covered in Section 3.5.

3.2 Planning and Design of the Game

With a simple SIR model implemented, the next step shifted focus to the planning and designing of the game. To do this, a game design document and a state diagram

of the different game states were created. A game design document is a document that aims to express some vision for the game and describes possible content [14]. This, in combination with a state diagram outlining the expected sequence of events for a level, created a clearer path to follow while completing this project. Due to the smaller size of this project compared to the larger-scale projects that occur in the industry, many parts of the traditional game design documents were taken out.

3.2.1 Game Design Document

Game Design Document: Xeno-plague

Contents:

- Description of the game
- The story
  - State Diagram of game
  - Day-to-Day scene
  - Instruction scene
- Mockups of scenes

Description of the Game:

This game is a simple resource manager. The player is tasked will be given a choice to immediately be able to quarantine two individuals or wait a certain amount of time to be able to create a vaccine and begin vaccinating individuals. As time passes, the player must try and contain the infection while also managing to simultaneously saving as much energy as possible and keeping the colony productive. Quarantining individuals doesn't take energy but takes away from the colony's productivity. Vaccination costs energy and takes time to develop initially but does not take away from productivity.

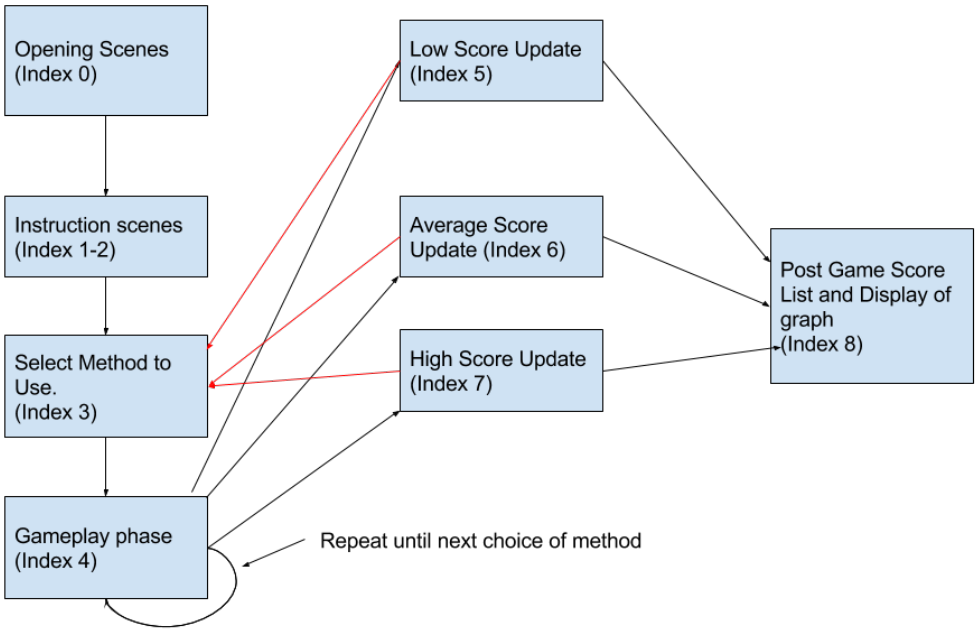
Every few days or so, you will be asked whether you want to invest in creating a vaccine, quarantine, or do nothing. During the gameplay phase, you will see lots of individuals on the page (the amount depends on how well the game can run it). They will be moving randomly about the ship. If you have the option of quarantining, you can drag and drop the requisite amount of individuals into the quarantine room, at the cost of productivity. If you have the option of vaccination, you can click on the individuals you want to vaccinate. This pattern will repeat until the infection is completed (i.e. all individuals are in the recovered state). Player performance will be measured in a final high score; this is then dependent on a few factors, such as the amount of energy used, the amount of production completed, the maximum number of infected individuals, and the amount of time the infection lasts. A high score will minimize the energy, infected individuals, and time, while maximizing production completed.

The story:

The story will be presented in between gameplay phases, when deciding which actions to take. Certain thresholds will have to be met for a particular scenario to be presented. Each state is unique due to text that will appear, as well as options presented to the player.

This document records the initial plan for the game. The plan was to create a resource manager, where you have to manage some resource (in this case "power") to

be able to create vaccines to stem some of the infection. Please refer to both Figures 3.3 and 3.4 as necessary. There originally was going to be a second resource to manage, but that was not completed in favor of simplifying the mechanics. Instead, it was left that the alternative to vaccinating individuals was to quarantine individuals for no cost. The document demonstrates the idea that the game would progress, with some form of in-game time representing a "day," and ask the player whether they would like to create vaccine, at the cost of some power. This would occur multiple times through the game, although the player becomes unable to create more vaccines once power becomes zero. The level would continue until the end of the infection. Then, the game would display a graph that represents the three subpopulations, with the aim of helping the players cognitively anchor the level they just experienced onto a more traditional graphical representation of this model. A state diagram was also created to layout how the game progresses from event to event. While the state diagram originally thought of each box as its own scene, many of the boxes involved in the game-play phases simply became events during the main game scene.



3.3 Scene One: The Main Menu

The first scene in the game is the main menu. From here, the player is able to initiate the game, quit the game, and/or view the credits. To create the menu, art assets, specifically the background image, were acquired through the Unity Asset Store. The music required was obtained via consent through a musician based in Los Angeles, Kyng Cold. The sound effects were acquired through the Unity Asset Store. This scene consists of six main game objects: the `Main Camera`, `EventSystem`, `StartMenu`, `QuitMenu`, `soundManager`, and `Credits`.

The `Main Camera` is an object of class Camera that is responsible for rendering everything that the player will see on screen [19]. This object will repeat in every scene. Because it is an object within Unity's scripting API, the `Main Camera` is not adjusted in any way and thus does not appear in any script. Thus, the `Main Camera` remains in place and does not move; the only task it has is to render the single screen with which the players interact. Because of this, there is no script attached to the object. The only adjustments to the default settings were to set the camera's Z-axis position to -10 via the Inspector Window in Unity. When the camera had the Z-axis position of zero, the necessary game objects were not visible. This is because the camera, when at Z-axis position zero, will look behind the game objects and never render the objects. To counteract this, lowering the depth to -10 ensured that the camera could actually see the game objects that need rendering.

The `EventSystem` is a game object that is used by default in every Unity scene, much like the `Main Camera`. This class dictates the sending of events to objects based on some form of input, such as the keyboard, mouse, touch, or other custom input [23]. For this project, keyboard and mouse are the only forms of input used. This game object, then, is responsible for monitoring for any mouse presses on the Main Menu screen and sending that notification to the scripts that utilize this input on the Main Menu.

The `StartMenu`, `QuitMenu`, and `Credits` are all game objects of class `Canvas`. The Canvas class is a class used for rendering the screen [20]. Each of these Canvases have game objects attached to them that are the actual buttons, text, and scripting needed to create the UI behavior necessary for the menu. Each Canvas represents the three different UI menus available to the player through the main menu. These menus are the Start Menu, where the player can start or quit, the Quit Menu, where the player can close the game, and the Credits, where the player can find the proper attributions.

The `StartMenu` has a Unity Image class object attached to it that is used as the background, and has the background Sprite set as the source image. This Canvas

has a few other Unity game objects as sub-objects. These are a Unity Image object, `Image`, which displays the background, a Unity Text object, `Title`, which displays the title of the game, and three Unity Button objects, `StartButton`, `QuitButton`, and `CreditsButton`. Each of these buttons are interactive and take you to either the main game, the quit menu, or the credits. Additionally, the `StartMenu` Canvas has an attached script, named `menuScript`. This script will be discussed in further detail shortly.

The `QuitMenu` and the `Credits` have a similar structure to the `StartMenu`. They both are Canvas class objects that also have an Image object, a Text object, and two Button objects. However, these Canvases do not have a `menuScript` attached; each Canvas is controlled via the script attached to the `StartMenu` canvas.

The `soundManager` is simply an object with a single attached component, which is the `soundController` script. This game object is explicitly responsible for the sound production throughout the game and it remains active in every scene. The scripts, `menuScript` and `soundController`, are the vital part of the functionality of this scene.

3.3.1 menuScript

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

/**
 *Class that controls the Start menu of the game.
 **/
public class menuScript : MonoBehaviour {


    public Canvas quitMenu;
    public Button startText;
    public Button exitText;
    public Canvas creditsMenu;
    public Button creditsText;

    /**
     *Method that starts the menu upon entering the scene. Gets the components for the text and menu, and disables the
     *quit menu.
     **/
    void Start () {

        quitMenu = quitMenu.GetComponent<Canvas> ();
        creditsMenu = creditsMenu.GetComponent<Canvas> ();

        startText = startText.GetComponent<Button> ();
        exitText = exitText.GetComponent<Button> ();
        creditsText = creditsText.GetComponent<Button> ();

        quitMenu.enabled = false;
        creditsMenu.enabled = false;
    }

    /**
     *Method that enables the Quit menu popup.
     * */
    public void ExitPress(){

        quitMenu.enabled = true; //shows quit menu
        startText.gameObject.SetActive(false);//hides start text
        exitText.gameObject.SetActive(false); //hides exit text
        creditsText.gameObject.SetActive(false);

    }
```

The `menuScript` class dictates the interactive behavior and options that the player has while on the main menu. It is attached as a component to the `StartMenu`, because it primarily dictates the behavior of this menu, which is the parent of the other two menus. `QuitMenu` and `Credits` are simply sub-menus of the `StartMenu`. Because of this, the `menuScript` class contains two public[1] variables, `quitMenu` and `creditsMenu`, that represent the two canvases used for the sub-menus. The `menuScript` also defines the three buttons, `startText`, `exitText`, and `creditsText`

[1]Besides defining access to variables as seen in other frameworks, the public keyword in Unity provides an additional functionality. A variable set to public in a script is then accessible to manipulation in the Unity Editor via the Inspector window. From there, a developer can set what they would like this specific variable to be, without explicitly stating it programmatically in the script. Assigning game objects in this way results in these becoming attached to components of the game object to which the script is attached.

that are the buttons players can manipulate. Please refer to Figures 3.5 and 3.6 as needed.

The `Start()` method is called upon loading a scene in which its parent class exists automatically through the Unity game engine [2]. The first task is to ensure that the class variables are all pointing to the correct items. This involves using the Unity provided method `GameObject.GetComponent<T type>()`. This method specifically returns the component of type `type`, if one is attached to the current game object [25]. For this specific script, the Canvas and Button components are attached to the `StartMenu` Canvas through the Inspector window, and dragged into the available slots. Thus, the `StartMenu` ensures that `quitMenu` points to the `QuitMenu` Canvas, `creditsMenu` points to the `Credits` Canvas, `startText` points to the `StartButton`, `exitText` points to the `QuitButton`, and `creditsText` points to the `CreditsButton`. The next few lines ensure that the sub-menus are hidden at the beginning. This is done by setting the `enabled`[3] variable to false.

The next five methods determine what occurs when buttons are pressed on the main menu. `ExitPress()`, `CreditsPress()`, `NoPress()`, `StartLevel()`, and `ExitGame()`, are all assigned to the correct Buttons via the Unity Editor. For Button components, the Inspector window contains a section entitled `OnClick()`. This interface allows the developer to select which game object contains the script that dictates its behavior (in this case `StartMenu`) and then select the method to call when the `EventSystem` detects a press on the Button.

`ExitPress()` dictates the behavior of the application when the player presses the `QuitButton`. When a player presses the `QuitButton`, the `EventSystem` will call the method. This method will then set `quitMenu.enabled` to true, making the `QuitMenu` Canvas visible to the player and laying it over the main menu. Additionally, it will hide the three buttons that make up the main menu, `startText`, `exitText`, and `creditsText`. This is done by using the `gameObject.SetActive(bool boolean)`[4] method and setting it to false for all three. `CreditsPress()` determines the behavior of the application when the player presses the `CreditsButton`. It follows an almost identical pattern as `ExitPress()`. It sets `creditsMenu.enabled` to true so the menu is then overlaid onto the `StartMenu` Canvas. It then proceeds to deactivate the `startText`, `exitText`, and `creditsText` Buttons by using the same `gameObject.SetActive(bool boolean)` method.

[2] Along with `Update()` and `FixedUpdate()`, `Start()` is automatically created when creating a script through the Unity Editor. This is because all scripts that work within the Unity Engine inherit from MonoBehavior, which dictates many of the Unity specific behaviors and provides these methods to inherit.

[3] The `enabled` variable is an inherited variable from the class `MonoBehavior`. Its purpose is to determine whether an object will be updated or not. When set to true, the component will be visible and updated by calling its `Update()` method. When set to false, the component is inactive.

[4] `gameObject.SetActive(bool boolean)` is a method of the gameObject class (which all the components derive from as well) and determines whether the given component is active at the current moment [26].

```
    public void CreditsPress(){

        creditsMenu.enabled = true;
        startText.gameObject.SetActive (false);
        exitText.gameObject.SetActive (false);
        creditsText.gameObject.SetActive (false);


    }

    /**
     *Method that closes the Quit menu popup and returns to the Start menu after enabling the Quit menu.
     * */
    public void NoPress(){

        quitMenu.enabled = false; //hides quit menu
        creditsMenu.enabled = false;
        startText.gameObject.SetActive(true); //shows start text
        exitText.gameObject.SetActive(true); //shows exit text
        creditsText.gameObject.SetActive(true);

    }

    /**
     *Method that starts the next scene upon pressing the start button.
     **/
    public void StartLevel(){

        SceneManager.LoadScene (1);

    }

    /**
     *Method that exits the game after pressing the exit button on the quit menu.
     **/
    public void ExitGame(){

        Application.Quit ();

    }
}
```

The `NoPress()` method is called when pressing the No Button on the `QuitMenu`. This will only be available to be called after the `ExitPress()` or the `CreditsPress()` method is called (because otherwise the particular Button that calls this method is hidden). The purpose of this method is to return to the `StartMenu` from the `QuitMenu` or the `CreditsMenu`. It does this by setting both `quitMenu.enabled` to false and `creditsMenu.enabled` to false, thus hiding these menus. Additionally, it uses `gameObject.SetActive(bool boolean)` to ensure that the necessary buttons from the `StartMenu` Canvas are activated.

The `StartLevel()` method is called when pressing the `startText` Button on the `StartMenu` Canvas. This method has a single line of instruction, with its only goal to load the next Scene in the game. To do this, the `UnityEngine.SceneManagement` [5] library is used to enable access to the `SceneManager.LoadScene(int buildIndex)` method, which loads the Scene with the passed in index in the game's build.

`ExitGame()` is called when pressing the Yes Button on the `QuitMenu`. This method is explicitly tasked with closing the application from this Button. To do so, it calls the method `Application.Quit()`, which quits the application entirely. This method is available via the Application class that exists in the `UnityEngine` library.

3.3.2 soundController

The `soundController` class is attached to the `soundManager` game object and determines the sounds to play throughout the game. It contains five local variables, an AudioSource[6] `efxSource`, which determines the source for sound effects, an AudioSource `musicSource`, which determines the source for music, an instance of itself, `instance`, and two floats to determine a range of pitches. The `instance` variable exists because this class follows a singleton design pattern and ensures that there is only one `soundController` class in existence. This same design pattern will be seen in another script, the `gameManager`. The `musicSource` variable is set via the Inspector window, which also provides an interface to determine whether

[5] `UnityEngine.SceneManagement` allows the management of Scenes at run-time during a game [33].

[6] AudioSource is a class that represents the source of the audio and can be represented in 2D or 3D. It contains a variable, clip, that determines the audio data to play [16].

the music plays immediately. In this case, the music starts when the game starts and continues throughout.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/* Class that defines the sound controller to control all... */
public class soundController : MonoBehaviour {

    public AudioSource efxSource;
    public AudioSource musicSource;
    public static soundController instance = null;
    public float lowPitchRange = .95f;
    public float highPitchRange = 1.05f;


    /* Method to initialize the sound controller. Follows the... */
    void Awake () {

        if (instance == null) {

            instance = this;

        }
        else if (instance != this) {

            Destroy (gameObject);

        }

        DontDestroyOnLoad (gameObject);

    }

    /* Method to play a single sound. ... */
    public void PlaySingle (AudioClip clip){

        efxSource.clip = clip;

        efxSource.Play ();

    }

    /* Method to randomize sound. Remained unused in prototype. ... */
    public void RandomizeSfx(params AudioClip[] clips){

        int randomIndex = Random.Range (0, clips.Length);

        float randomPitch = Random.Range (lowPitchRange, highPitchRange);

        efxSource.pitch = randomPitch;

        efxSource.Play ();

    }

}
```

The method `Awake()` overrides the `Awake()` method inherited via `MonoBehavior` and ensures the singleton pattern remains[7]. Here, it checks to see if `instance` is currently set to `NULL` (which it will be when the game is first launched). If so, it sets `instance` to the current `soundController` object. If it is not `NULL`, it destroys the current `soundController`, because that would mean there is more than one instance of this class. The last method called when loading the script is

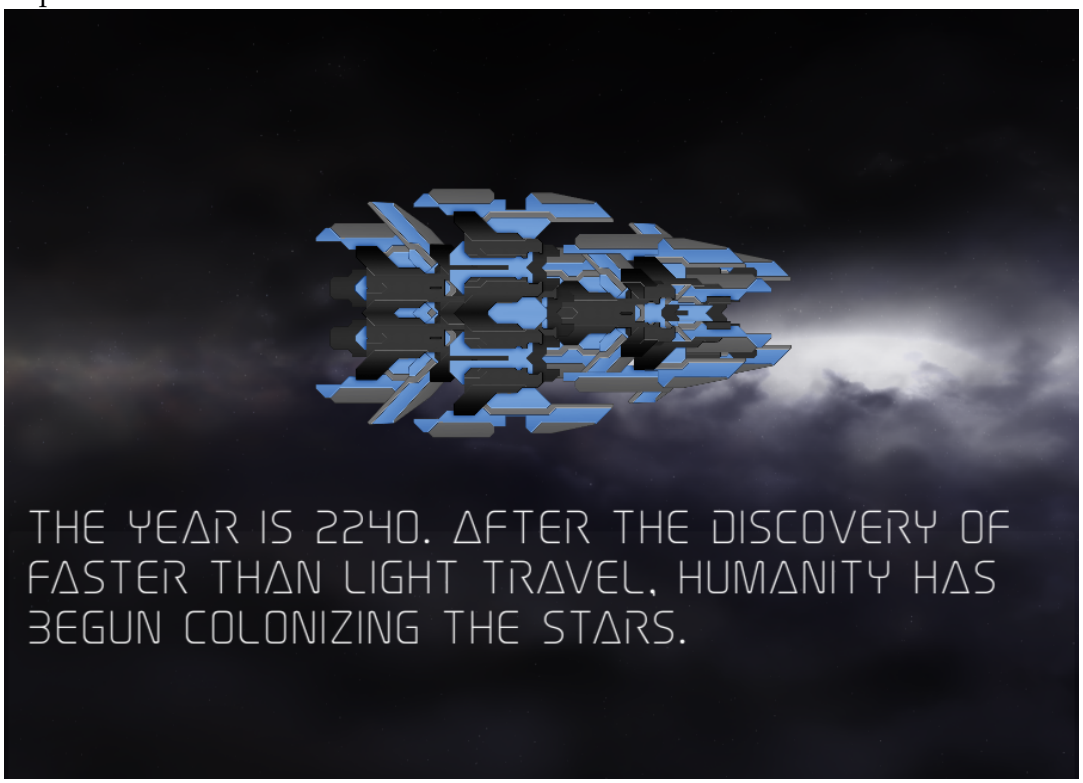
[7]The `Awake()` method is called when a script is initially being loaded by the Unity engine and is used instead of a traditional constructor in C# scripts written for Unity [28].

`DontDestroyOnLoad(gameObject object)`[8]. This ensures that the instance of `soundController` that is created remains active between scenes. This script will then be the only control of sound throughout the entire game.

The methods `PlaySingle(AudioClip`[9] `clip)` and `RandomizeSfx(params AudioClip[] clips)` are the methods that play the sound effects when necessary. These are called during game-play and are triggered by specific events in the game. `PlaySingle(AudioClip clip)` plays a single sound effect clip at its default pitch. It sets the variable `efxSource.clip` to the audio data to be played and then plays it. This is done by using the `AudioSource.Play()` method. The `RandomizeSfx(params AudioClip[] clips)` does much the same work, except that in the case it selects a random `AudioClip` to obtain data from, adjusts its pitch randomly, and then plays the given clip.

3.4 Scene Two: The Instruction Scene

The second scene is an introduction and instruction sequence. Here, the player reads a set of texts to understand the objectives of the game. It is here that the mechanics of the game are explained to the player, as well as a motivation as to why the player is playing. This scene consists of a text box that displays the instructions, as well as a sprite for aesthetics.



This particular scene consists of a `Main Camera` and an `EventSystem`, much like the previous scene, as well as a `Canvas`. The `Canvas` game object in this scene has an Image component attached which displays the background. In this scene, `Canvas` has three components as subcomponents: a `Panel`[10], and two Sprites. The

[8] The method `DontDestroyOnLoad(gameObject object)` is inherited from the Object class to the gameObject class. It ensures that the passed in target is not destroyed automatically when loading a new scene [29].

[9] A container for Audio data that provides methods to change and play said data [15].

[10] A Panel component in Unity is a specific UI component that is a combination of other UI classes.

two Sprites are shown for aesthetics and are activated depending on the text displayed.

The Panel is where most of the functionality for this scene occurs. The UI Panel has four sub-game objects: a Text component `GameText`, a script component `textboxManager`, and two Button components, `ContinueText` and `RepeatText`. These work in concert to allow the player to click the mouse to continue through the instructions that are displayed in the Panel. Once the player reaches a certain point in the dialogue, the Sprite changes. Upon reaching the end of the dialogue, the player then has the ability to continue to the game or repeat the instructions. All of this behavior is controlled through the `textboxManager` script.

3.4.1 textboxManager

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

/* Class that defines the controller for all the textboxes... */
public class textboxManager : MonoBehaviour {


    public Text text; //The text to display.
    public TextAsset text_file; //The text file to load.
    public Button continue_text; //Button to continue in certain scenarios.
    public Button repeat_text; //Button to repeat text in certain scenarios.
    public string[] text_lines; //An array of strings that contains each line in the text file.
    public int current_line; //Index of the current line.
    public int hide_ship; //Index where to hide a sprite in the scene
    public int repeat_instructions; //Index where to begin the repeat of instructions.
    public int end_at_line; //Index where the text file ends.
    public SpriteRenderer sprite_renderer_one; //Sprite to render.
    public SpriteRenderer sprite_renderer_two; //Sprite to render.


    /**Method to initialize the textManager and load the text files. Checks to make sure the text file is not null or
     * empty and, if not, loads them into the text_lines array.
     * */
    void Start () {

        continue_text.gameObject.SetActive (false);
        repeat_text.gameObject.SetActive (false);
        sprite_renderer_one.enabled = true;
        sprite_renderer_two.enabled = false;

        if (text_file != null) {

            text_lines = (text_file.text.Split('\n'));

        }

        if (end_at_line == 0) {

            end_at_line = text_lines.Length - 1;

        }

    }
```

The script `textboxManager` determines the dialogue being displayed in the `Panel`, as well as the `Sprite` to be displayed above it. Thus, this class contains multiple different public local variables. The variable `text` is the text to display, `text_file` is the file from which the text to display is obtained, `continue_text` and `repeat_text`, the two Buttons, and `text_lines`, an array of strings that represent a single line from `text_file`. It also contains a set of ints to keep track of where to trigger changes in the scene: `current_line`, which keeps track of the current line being displayed, `hide_ship`, the line to change sprites, and `end_at_line`, which determines when the file is completed. Additionally, it contains two `SpriteRenderer` objects, `sprite_renderer_one` and `sprite_renderer_two`, which render the

Sprites when active. `sprite_renderer_one` renders the ship Sprite and `sprite_renderer_two` renders the doctor Sprite.

Upon loading the Scene, the `Start()` method is called to set up the necessary parts of the dialogue box. The first two instructions ensure that the `continue_text` and `repeat_text` Buttons are set to inactive, as they are not needed until the end of the Scene. It then sets `sprite_renderer_one.enabled` to true so the ship is displayed, and sets `sprite_renderer_two.enabled` to false so the doctor is not displayed. It then checks to ensure that the `text_file` needed is not empty and then loads the file into `text_lines`, splitting at every newline character. The last task will be to ensure that `end_at_line` ends at the last line of dialogue available.

```
/**
 * Method to update the scene with current information. Uses a press of the left mouse button to progress text.
 * Calls specific methods and enables buttons depending on line index AND current scene.
 * */
void Update(){

    text.text = text_lines [current_line];

    if ((Input.GetKeyDown(KeyCode.Mouse0)) && current_line < end_at_line) {

        current_line = current_line + 1;

    }

    if (current_line == hide_ship && (SceneManager.GetActiveScene().buildIndex == 1 )) {

        sprite_renderer_one.enabled = false;
        sprite_renderer_two.enabled = true;

    }

    if ((current_line == repeat_instructions) && (SceneManager.GetActiveScene().buildIndex == 1 )) {

        continue_text.gameObject.SetActive (true);
        repeat_text.gameObject.SetActive (true);

    }

}
```

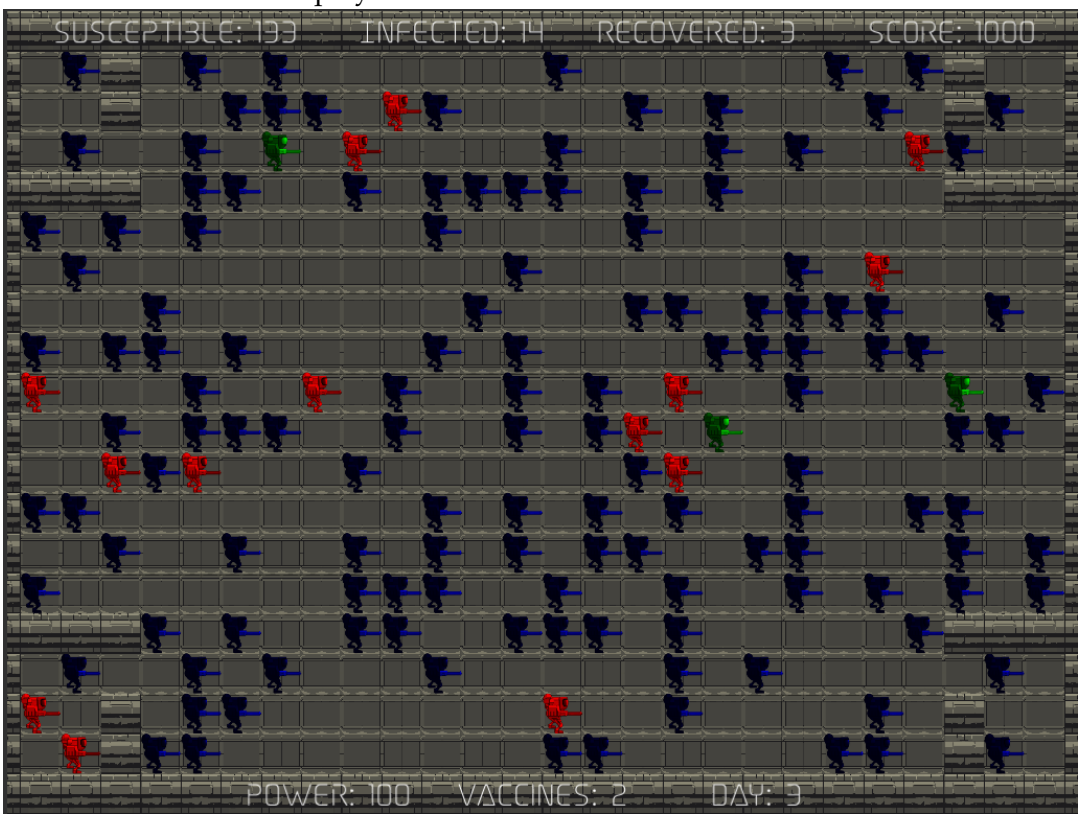
The `Update()` method is called every frame by the Unity Engine as the game is progressing. In this case, this is the engine responsible for monitoring and then displaying the correct items. It starts by setting `text` to the current line of text. It then checks if there is a mouse click; if so, the current line progresses to the next line. It checks then if `current_line` is equal to `hide_ship` and, if so, hides the ship Sprite by setting enabled to false and displays the doctor Sprite by setting enabled to true. It finally then checks if `current_line` is equal to `end_at_line` and, if so, displays the `continue_text` Button and `repeat_text` Button.

While the `continuePress()` method works similarly to the `StartLevel()` method seen in `menuScript`, the `repeatPress()` is different. As seen before, the Buttons are configured to trigger these methods when pressed. The `repeatPress()` method specifically ascertains that the correct Scene is being displayed and then deactivates the Buttons and reverts `current_line` to the value where the instructions begin. These methods all work together to create the dialogue that leads a player into the game-play phase.

3.5 Scene Three: Gameplay

The third scene comprises all of the actual game-play. This scene is also the most complex in the amount of inter-working parts that demonstrate the behavior seen on screen. After continuing past the instructions detailed in Scene Two, the game

board is then laid out and displayed, and allows the player to start working on limiting the infection in whatever way they see fit. Each A.I. controlled individual is colored according to their status in terms of the model; the blue is "susceptible," red is "infected," and green is "recovered." It allows for the player to use the numeric keys 1-4 to close the doors (with each key representing a door, left to right and top to bottom). Additionally, players can pause the game at any time using the space key. This allows the player to focus in and distribute the vaccines available by clicking on individuals. The pausing was implemented because it seemed to be a too difficult to click on an individual while moving, thus rendering the difficulty more in the click action rather than in the overall scenario. Every five seconds, a pop-up menu will appear, prompting the player to choose whether to create a new vaccine or not, at the expense of some power. This pattern of game-play continues as a loop until there are no more "infected" individuals left in the game-play scene. This is because, once there are no more "infected" individuals, the infection spread is stopped and the infection lifespan is completed. The player also consistently has feedback as the progress of the infection through the UI text. With each update of the frame, the game displays the new amount of "susceptible," "infected," and "recovered" individuals at the top of the game board. The number of vaccines and the amount of days into the infection are displayed on the bottom.



This scene consists of the `Main Camera`, the `EventSystem`, a script called `gameManager`, and a `Canvas` for UI. The `gameManager` script is key to this entire scene working. While the game object is itself the script `gameManager`, it has two more scripts attached to it as components of the `gameManager` script: the `boardManager`, and the `sirGameModel`. These are used along with `gameManager` to set the layout of the board, set the individuals, and set up the model, saving which individuals are being set to "susceptible" and which are being set to "infected." Each Sprite is generated by `boardManager` and it then becomes a game object in and of itself in the Hierarchy

panel on the Unity Editor. This is important, because the door tiles and the individual tiles each have their own scripts attached to dictate their behavior; these are `doorController` and `character` respectively. The `sirGameModel` script is an edited version of the `sirModel` script seen in Chapter 3, Section 3.1. This script is the controller of the model and sends updates to the other scripts so that they reflect the changes needed to keep the game running smoothly. It then becomes a quasi-form of the model-view-controller, with the `sirGameModel` script serving as the model and updating the view through the character script and the `gameManager`. This was chosen primarily to protect the SIR model from the rest of the scripts. The `sirGameModel` script is the only script that keeps track of the infection status of individuals and the counts of each the subpopulations. It is also the only script that can change the status of an individual. Every other script communicates with this model to affect the changes required due to their behavior.

The `Canvas`, in this case, consists of all the objects that make up the UI. This includes the Text objects that display the amount of individuals in each population, the number of vaccines, power, score, and days. These are updated regularly as the game progresses. All of the text on screen as UI is individually its own UI Text object that is a sub-object of the `Canvas`. There are additionally two Canvases that are sub-objects to the Canvas: `PauseMenu` and `VaccineMenu`. `PauseMenu` only appears when the escape key is pressed and provides options for a player in a traditional pause menu format. The `VaccineMenu` appears every so often and prompts the player to choose to make a new vaccine or not. These are controlled by the scripts `pauseController` and `vaccineMenuController` respectively.

3.5.1 gameManager

The `gameManager` script is set up similarly to the `soundController` script seen in Chapter 3, Section 3.3.2, although it contains many more local variables than `soundController`. This class contains an instance of `boardManager` and `sirGameModel`, as well as an instance of itself. It also contains lists to store the number of days, susceptible individuals, infected individuals, and recovered individuals accrued through the model's progression. Additionally, it contains three ints to keep track of score, the total population, the maximum number of infected, and the current power. It has a float that keeps track of the time passed in game (used for the vaccine choices) as well as whether the pause menu is active or not. Finally, it contains multiple Text objects that display the UI necessary, as well as two Image objects to display the pause menu and vaccine menu.

Much like with the `soundController` script, this class follows the singleton design pattern when being initialized. It checks to see if `instance` is set to null and, if so, sets it to the current instance. Otherwise, it destroys the object using `Destroy (gameObject object)`. Once it ensures that instance is pointing to this, it then calls `DontDestroyOnLoad(gameObject object)` to retain it between Scenes. This `Awake()` method differs from the one in `soundController` in that, after ensuring the singleton pattern, there is still more work for the `gameManager` to accomplish. It points `board_script` to the `boardManager` component and `sir_model` to the `sirGameModel` component. It then initializes all the lists necessary to keep track of the data accrued via the model. It sets the `time_to_choose` to 5 seconds and sets `level` to three to determine the amount of individuals laid out onto the board. It then calls `InitGame()`, which initializes the rest of the necessary components for the game.

```
ublic class gameManager : MonoBehaviour {

    public static gameManager instance = null;
    private boardManager board_script;
    public sirGameModel sir_model;
    public List<int> day;
    public List<int> susceptible_count;
    public List<int> infected_count;
    public List<int> recovered_count;
    public int total;
    public int level;
    public int score;
    private int max_infected;
    private int power;
    private bool paused;
    private float time_to_choose;

    private Text susceptible_text;
    private Text infected_text;
    private Text recovered_text;
    private Text vaccine_text;
    private Text score_text;
    private Text power_text;
    private Text day_text;
    private Image pause_menu;
    private Image vaccine_menu;

    /* Initializes the game manager overall. ... */
    void Awake () {

        if (instance == null) {

            instance = this;

        } else if (instance != this) {

            Destroy (gameObject);

        }

        //ENSURES THAT GAMEOBJECT AND ALL ATTACHED OBJECTS ARE RETAINED BETWEEN SCENES!

        DontDestroyOnLoad (gameObject);



        board_script = GetComponent<boardManager> ();
        sir_model = GetComponent <sirGameModel> ();

        day = new List<int> ();
        susceptible_count = new List<int> ();
        infected_count = new List<int> ();
        recovered_count = new List<int> ();

        time_to_choose = Time.fixedTime + 5.0f;
        level = 3;

        InitGame ();

    }
```

```
/* Method to initialize the game. Sets up the board and... */
void InitGame(){

    if (level == 1) {

        board_script.SetupScene (50);

    } else if (level == 2) {

        board_script.SetupScene (100);

    } else if (level == 3) {

        board_script.SetupScene (150);

    } else if (level == 4) {

        board_script.SetupScene (200);

    }

    //board_script.SetupScene ();
    sir_model.setupModel ();

    score = 1000;
    power = 100;
    paused = false;
    max_infected = 0;
    total = board_script.individual_count;

    add_data ();

    susceptible_text = GameObject.Find ("susceptible_text").GetComponent<Text> ();
    infected_text = GameObject.Find ("infected_text").GetComponent<Text> ();
    recovered_text = GameObject.Find ("recovered_text").GetComponent<Text> ();
    vaccine_text = GameObject.Find ("Vaccines").GetComponent<Text> ();
    score_text = GameObject.Find ("Score").GetComponent<Text> ();
    power_text = GameObject.Find ("Power").GetComponent<Text> ();
    day_text = GameObject.Find ("Days").GetComponent<Text> ();
    pause_menu = GameObject.Find ("PauseMenu").GetComponent<Image> ();
    vaccine_menu = GameObject.Find ("VaccineMenu").GetComponent<Image> ();

    // graph = GameObject.Find ("Graph").GetComponent<Image> ();

    updateText ();


}
```

`InitGame()` then proceeds to dictate laying out the board, initializing the model, and initializing the UI. It first checks to verify how many individuals will be used in the scene, from 50 individuals to 200 mapping to levels 1 to 4. Depending on the level, it will then call `board_script.SetupScene(int individuals)`, which will be discussed shortly. It sets up the initial values for `score`, `power`, `paused`, and `max_infected`. It then records the initial data to the local lists by calling a method `add_data()`, which simply adds the number of individuals in each subpopulation, the day, and the maximum number of infected to their corresponding containers. It then points the UI to their components on the canvas and calls `updateText()`, which simply sets the Text components to the current values obtained via the `sirGameModel`.

```
    void Update(){

        if (SceneManager.GetActiveScene () == SceneManager.GetSceneByBuildIndex (2)) {

            if (pause_menu.gameObject.activeInHierarchy == true) {

                Time.timeScale = 0;


            } else if (Input.GetKeyDown (KeyCode.Space)) {

                if (paused == true) {

                    Time.timeScale = 1;

                    paused = false;

                }
                else if (paused == false) {

                    Time.timeScale = 0;

                    paused = true;

                }


            }
            else if (Time.fixedTime >= time_to_choose) {

                calculateScore ();

                vaccine_menu.gameObject.SetActive (true);

                Time.timeScale = 0;

                time_to_choose = Time.fixedTime + 5.0f;

                add_data ();

                calculateScore ();


            }
            else {

                updateText ();

            }

        }

    }
}
```

The third vital method for this class is the `Update()` function. It begins by ensuring the Scene index is correct. It checks to see if the pause menu is activated using the `gameObject.activeInHierarchy`[11] variable and, if it is active, uses the `Time.timeScale`[12] variable to pause the game. This is done by setting `timeScale` to zero. It then also checks if the escape key is pressed and, if so, pauses or unpauses the game based on the value of the `paused` boolean variable. Lastly, it checks to make sure that the variable `Time.fixedTime` is greater than the `time_to_choose` variable and, if so, presents the vaccine menu, calculates the score through a simple arithmetic method that takes into account `power`, and adds the data to the lists in `gameManager`. Otherwise, it simply updates the UI text. This all ensures that, as each frame is updates, the view is displaying the correct information, as well as allowing the player to reach the pause menu.

[11] `activeInHieracy` is simply a boolean that serves as a flag for whether a particular `gameObject` is currently active in the scene or not [24].

[12] `Time.timeScale` comes from Unity's class Time and defines the scale at which time is passing in the game [37].

```
/* Method that calculates the current score. TO BE WORKED ON/ ... */
void calculateScore(){

    if (power < 100 && power >= 80) {

        score = score - 10;

    }
    else if (power >= 60 && power < 80) {

        score = score - 20;

    }
    else if (power < 60 && power >= 40) {

        score = score - 30;

    }
    else if (power < 40 && power >= 20) {

        score = score - 40;

    }
    else if (power < 20 && power >= 0) {

        score = score - 50;

    }

    score = score + (sir_model.get_recovered_count() * 5) - (sir_model.get_days()/5);
    score = (score / (max_infected / 10));

}
```

The game score is calculated here and takes into account the "power" left, the maximum number of infected, the recovered individuals, and the "days" that passed over the course of the game. The idea behind the calculation was to emphasize retaining "power" while also trying to limit the peak and length (in "days") of the infection. The purpose of this is to reflect that, in real world infections, resources are limited and a successful response would involve minimizing the number of infected individuals and the number of days the infection spreads.

3.5.2 boardManager

`boardManager` is a component of the `gameManager` object and controls the layout of the Sprite tiles that make up the game board. It does this by containing three ints, which make up the columns, rows, and the count of individuals, as well as arrays of `GameObjects` that contain the tiles, a `Transform`[13] for the board, and two lists, one with vectors and the other with `GameObjects`. The several methods needed to actually set up the board are called from the method `SetupScene(int individual_count)`. This method is called from the `gameManager` and proceeds to call all the necessary methods in `boardManager` to set up the board.

[13] A `Transform` is an object that determine the position, rotation, and scale of an object [38].

```
public GameObject[] floor_tiles;
public GameObject[] wall_tiles;
public GameObject[] outer_wall_tiles;
public GameObject[] individual_tiles;

private Transform board_holder;
private List <Vector3> grid_positions = new List<Vector3>();
private List <GameObject> characters = new List<GameObject> ();

/* Method that initializes a list of all the possible grid... */
void initializeList(){

    grid_positions.Clear ();

    for (int x = -10; x < columns+6; x++) {

        for (int y = -6; y < rows+2; y++) {

            grid_positions.Add (new Vector3 (x, y, 0f));

        }

    }

}

/* Method to set the game board up. Instantiates all the... */
void boardSetup(){

    board_holder = new GameObject ("Board").transform;

    for (int x = -10; x < columns+6; x++) {

        for (int y = -6; y < rows+2; y++) {

            GameObject to_instantiate = floor_tiles[0];
            GameObject instance;

            if(x == -10 || x == columns+5 || y == -6 || y == rows+1){

                to_instantiate = wall_tiles[0];
                Vector3 vector = new Vector3 (x, y, 0f);
                instance = Instantiate(to_instantiate, new Vector3(x, y, 0f), Quaternion.identity) as GameObject;
                instance.transform.SetParent(board_holder);
                grid_positions.Remove (vector);

            }
            else{

                instance = Instantiate(to_instantiate, new Vector3(x, y, 0f), Quaternion.identity) as GameObject;
                instance.transform.SetParent(board_holder);

            }

        }

    }

}
```

The `SetupScene(int individual_count)` method begins by calling `initializeList()`, which initializes the list of all possible positions in which to place times. It loops over the determined amount of columns and rows needed to lay the game board out. While the values of `columns` and `rows` are set to 12 each, these had to be adjusted to account for the X and Y placement of the grid in the Unity Editor. For each iteration, it adds a location to the list `grid_positions`, utilizing the `Vector3` object[14] to add the locations.

It then calls `boardSetup()`, which first determines the position, rotation, and scale for the board. In this case, it is the default, 2D plane setting. It then loops over each row and column, instantiating the necessary tiles. If the location happens to be on the edge of the grid, it instantiates a wall tile, which has a `BoxCollider2D` component that makes it impenetrable to other objects. It will also remove these edge locations so that nothing else can be placed there. Otherwise, if the location is not on the edge of the board, it will place a floor tile, which has no extraneous components attached.

[14] A `Vector3` is a Unity library class that describes a 3D vector [39].

The method then lays out the inner walls to create the rooms by calling `LayoutInnerWalls()`. The instructions to do this are similar to how the wall tiles were instantiated previously in `boardSetup()`, but the locations are chosen with intent and are not necessarily edges. Much like before, it removes these locations once the walls are instantiated.

```
/* Method to select a random positions vector from the... */
Vector3 RandomPosition(){

    int random_index = Random.Range (0, grid_positions.Count);
    Vector3 random_position = grid_positions [random_index];
    grid_positions.RemoveAt (random_index);
    return random_position;

}

/* Method to layout the character tiles at random on the... */
void LayoutIndividualAtRandom(GameObject[] array){

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

        Vector3 random_position = RandomPosition ();
        GameObject tile = array [0];
        GameObject instance = Instantiate (tile, random_position, Quaternion.identity);
        characters.Add (instance);

    }

}
```

Lastly, it calls `LayOutIndividualAtRandom(GameObject[] array)` to layout the different character tiles. This method utilizes the method `RandomPosition()` to choose a `Vector3` at random in which to place an individual. It does this for each and every individual that the `gameManager` dictates the game will have. The `RandomPosition()` method works by obtaining a randomly chosen index, using the `Random.Range(min, max)` method. This `Random` class is one provided by the Unity library. It then obtains the `Vector3` that is at that specific index, removes it and returns the `Vector3` obtained. Once this is returned in `LayoutIndividualAtRandom(GameObject array)`, it is then used to instantiate the character and then adds said character to a list of characters. Once the board's setup is complete the `sirGameModel` script then begins to initialize.

3.5.3 sirGameModel

```
/* Method to set up the SIR model. This is called from... */
public void setupModel(){

    done = false;

    population = new Dictionary<GameObject, string> ();

    boardManager board = GetComponent<boardManager> ();

    susceptible_count = board.individual_count - 10;
    infected_count = 10;
    recovered_count = 0;
    vaccine_counter = 2;
    day = 1;

    total_pop = susceptible_count + infected_count + recovered_count;

    susceptible_data.Add (timer_in_seconds, susceptible_count);
    infected_data.Add (timer_in_seconds, infected_count);
    recovered_data.Add (timer_in_seconds, recovered_count);

    List <GameObject> character_tiles = board.getCharacters();

    int i = 0;
    while (i < susceptible_count) {

        population.Add (character_tiles[i], "susceptible");
        i++;

    }

    for (int j = 0; j < infected_count; j++) {

        population.Add (character_tiles[(i + j)], "infected");

    }


    level_timer = 0.0f;
    time_for_vaccine = Time.fixedTime + 5.0f;

}
```

Much like `boardManager`, the `sirGameModel` script is a component of the `gameManager`. Unlike the previous, simple SIR script seen in Chapter 3, Section 3.1, this `sirGameModel` class was developed to work within Unity and take advantage of the features it provides. As opposed to the constructor used in the simple script, the method `setupModel()` is used[15]. This model serves as the backbone of the game and dictates the spread of infection. It is primarily backed by a dictionary that keeps track of each individual and their current status in the model. Using this, the other scripts can then get these values directly from the model so that they may update the view, through the use of a myriad of get and set methods, as the local variables for the model are private to provide protection to them. The model setup first initializes a dictionary, called `population`, that contains each individual on the board and their current status. It then points the local variable `board` to the `boardManager`. It then sets the initial values for each of the ints that represent the number of individuals in each population, which will always be the total population minus ten for susceptible, ten for infected, and 0 for recovered. It also sets the number of vaccines to two and the day to one. It then sets the value for the local variable `total_pop` to the combination of the susceptible, infected, and recovered count. It then gets the list of characters from `board` and loops over this list, setting the prerequisite portion to susceptible and infected (determined by the counts for both) and adding them to the dictionary. Afterwards, the script sets the time to create a vaccine to five seconds, which represents a single in-game "day." Each frame update then in the `FixedUpdate()` method waits for five seconds to pass before updating the day

[15]This is because classes derived from `MonoBehavior` cannot have custom parameters in its constructor, as well as being unable to be invoked with the new keyword.

number.

```
/* Method to set up the SIR model. This is called from... */
public void setupModel(){

    done = false;

    population = new Dictionary<GameObject, string> ();

    boardManager board = GetComponent<boardManager> ();

    susceptible_count = board.individual_count - 10;
    infected_count = 10;
    recovered_count = 0;
    vaccine_counter = 2;
    day = 1;

    total_pop = susceptible_count + infected_count + recovered_count;

    susceptible_data.Add (timer_in_seconds, susceptible_count);
    infected_data.Add (timer_in_seconds, infected_count);
    recovered_data.Add (timer_in_seconds, recovered_count);

    List <GameObject> character_tiles = board.getCharacters();

    int i = 0;
    while (i < susceptible_count) {

        population.Add (character_tiles[i], "susceptible");
        i++;

    }

    for (int j = 0; j < infected_count; j++) {

        population.Add (character_tiles[(i + j)], "infected");

    }


    level_timer = 0.0f;
    time_for_vaccine = Time.fixedTime + 5.0f;

}
```

The methods `infect(GameObject character_one, GameObject character_two)` and `recover(GameObject character)` methods in the `sirGameModel` are where most of the implementation differences lie between the simple script and the in-game model. For one, in the game model, the actions of infection and recovery are separated out into two different methods, as compared to the singular method in the simple script. Additionally, the core functionality of when the infect method is called is different than in the simple script. While `recover(GameObject character)` is called every five seconds (or one in-game "day"), `infect(GameObject character_one, GameObject character_two)` is only called upon the collision of two character Sprites. The way this action is triggered will be discussed shortly, but for the moment, it is suffice to know that when two character Sprites collide on the game board, the collision triggers this method to be called. It then takes the two character Sprites that collided and proceeds through the method. It first checks to make sure if one of the character Sprites is marked as "infected" in the `sirGameModel`'s dictionary. If neither is marked as "infected," the method simply returns. Otherwise, it chooses a random value in the range between zero and one thousand. That random value determines the probability that a certain contact will result in an infectious contact. Because we know that `b` represents a fixed number of infectious contacts, we can presume that there is some chance that a contact will not be infectious. Using data collected from the "Hong Kong Flu" epidemic as a real-life example, the estimate is that there is an infectious contact every two days and, thus `b` is set to one-half [41]. To recreate this, then, the random number is used to dictate the probability of a contact being infectious. This way, not every single contact results in the spread of an infection, $even if there is an "infected" and "susceptible" contact$. It then proceeds to check which character is the "infected," and enter the if-statement according to

which character Sprite is the "infected" party. If `random_chance` is greater than one thousand, the "susceptible" Sprite is then set as "infected" and the corresponding counts for "infected" and "susceptible" increment and decrement.

```
/* Method to determine the spread of the infection amongst... */
public void infect(GameObject moving_character, GameObject hit_character){

    if (population [moving_character] != "infected" && population [hit_character] != "infected") {

        return;

    }

    int random_chance = Random.Range (0, 1000); //CHANGE DEPENDING ON THE INFECTION RATE (USING 1/3)


    if ((population [moving_character] == "infected") && (population [hit_character] == "susceptible")) {

        if (random_chance > 500) {

            population [hit_character] = "infected";
            infected_count++;
            susceptible_count--;


        }
    }
    else if((population[moving_character] == "susceptible") && (population[hit_character] == "infected")){

        if(random_chance > 500){

            population[moving_character] = "infected";
            infected_count++;
            susceptible_count--;

        }

    }
}

/**
 *Method to determine the recovery of individuals. This method is called with every update, and is only called
 *for individuals who are infected. It takes in the current character that it is called in as an argument.
 * */
public void recover(GameObject character){

    int random_chance = Random.Range (0, 1000);

    if (random_chance < 333) {

        population [character] = "recovered";
        infected_count--;
        recovered_count++;

    }

}
```

The `recover(GameObject character)` method is then called every five seconds or one in-game "day." Much like the previous method, the `recover(GameObject character)` function first chooses an int between zero and one thousand to represent the chance that an individual recovers that day. If the random integer chosen is less than 333, then the character's status in the model is changed from "infected" to "recovered" and the corresponding counts increment and decrement.

```
/**
 *Method to vaccinate a unit. Takes in the unit to vaccinate as an argument.
 **/
public void vaccinate(GameObject character){

    if (vaccine_counter > 0) {

        population [character] = "recovered";
        susceptible_count--;
        recovered_count++;
        vaccine_counter--;

    }

}
```

The method to vaccinate is simple in this model. This particular method is triggered when the player clicks on a character Sprite with the left mouse-button. Once the method is called, it simply checks to ensure that there is an available vaccine to be used and, if so, sets the characters status to "recovered" in the model and increments and decrements the appropriate counters.

These three methods make up the bulk of the functionality of the game. Without these working in concert to accurately reflect the SIR model, there would be no real representation of the model. While they are not the same mathematical equations that the simple script emulated, these are meant to adhere to these equations through behavior dependent on what actually occurs in the game world. Approaching the model in this manner allows for the infection to spread as it would in the real world (by individuals coming into contact with one another) without necessarily adhering to the same rigidity that is seen when simply taking into account the mathematical equations. This means that they will reflect more of the randomness that is involved with real-life infection scenarios, while still following the SIR model.

3.5.4 movementController and character

```
/* Class that describes the behavior of the characters that... */
public class character : movementController {

    private SpriteRenderer renderer;
    private string status;
    private sirGameModel sir;
    float time_to_move;
    float time_to_recover;
    public AudioClip infect_sound;
    public AudioClip recover_sound;
    public AudioClip game_over_sound;

    /* Method to star the character script... */
    protected override void Start () {

        sir = GameObject.Find ("gameManager").GetComponent<sirGameModel> ();

        renderer = GetComponent<SpriteRenderer> ();

        time_to_move = Time.fixedTime + 0.5f;

        time_to_recover = Time.fixedTime + 5.0f;

        base.Start();
    }

    /* Method for a character to attempt a move in a... */
    protected override void attemptMove <T> (int x_direction, int y_direction){

        base.attemptMove <T> (x_direction, y_direction);

    }
```

The `movementController` class defines the behavior of how the individual Sprite characters move on the game board. This class is inherited by the character class and extended to define some of the methods left abstract in the `movementController` class. The `character` script is the one actually attached to the individual Sprites that are moving around the board. Upon the scene loading, each character loads this script by calling the `Start()` method. The method here first points the `character` script's local `sirGameModel` variable to the script that is attached to the `gameManager` script. It then gets the renderer variable to point to the `SpriteRenderer` that is attached to the current character. It sets the time gap between each characters

movement at half a second and then sets the time to recover to five seconds. The last portion simply calls the `Start()` method in the `movementController`. The `Start()` method in `movementController` then points its local variable `box_collider` to the `BoxCollider2D` component that is attached to the current character, as well as pointing `rigidbody_2d` to the `Rigidbody2D` component. It also has a variable, `inverse_move_time`, which inverses the time to move and is used to smooth out movements.

```
/* Abstract class that dictates the underlying behavior of... */
public abstract class movementController : MonoBehaviour {

    public float move_time = 0.1f;
    public LayerMask blocking_layer;

    private BoxCollider2D box_collider;
    private Rigidbody2D rigidbody_2d;
    private float inverse_move_time;


    /* Method that initializes the components needed for... */
    protected virtual void Start () {

        box_collider = GetComponent<BoxCollider2D> ();
        rigidbody_2d = GetComponent<Rigidbody2D> ();
        inverse_move_time = 1f / move_time; //makes for more efficient computations

    }

    /**
     *Method to move a unit. Takes in an x and y direction to move in and returns a RaycastHit2D component that
     *dictates whether a move is possible or not.
     **/
    protected bool move(int x_direction, int y_direction, out RaycastHit2D hit){

        Vector2 start = transform.position; //casting this vector 3 into vector 2
        Vector2 end = start + new Vector2(x_direction, y_direction);

        box_collider.enabled = false;
        hit = Physics2D.Linecast (start, end, blocking_layer);
        box_collider.enabled = true;

        //Checking if space is open
        if (hit.transform == null) {

            StartCoroutine (smoothMovement (end));
            return true;

        }

        return false;

    }
```

To dictate where to move, the method `moveCharacter()` is called every half second in-game. This method determines the direction to move in and then proceeds with the movement by calling other methods. It first randomly selects an X and Y direction to go to, using a random number generator. Once that is chosen, these X and Y values are then passed into `character.attemptMove<T>(int x, int y)`. This method simply calls the method of the same name from the parent class, `movementController`. The `attemptMove<T> (int x, int y)` method in the parent class sets a variable, `hit`, to be a `RaycastHit2D` object [16] It then calls the `movementController`'s `move(int x, int y, out RaycastHit2D hit)` method.

The first line is to cast the current `Vector3` into a `Vector2` (for 2D physics) for the start point and then set a `Vector2` for an end point. It disables the `BoxCollider2D` component to allow for movement and assigns `hit` to a `Physics2D.Linecast`[17] object, taking in the parameters of the start vector, the end vector, and the layer that

[16] A `RaycastHit2D` object is a class that contains information about an object collision detected by a raycast in the 2D physics [31].

[17] This method returns a `RaycastHit2D` that is a line segment against two different colliders in the Scene [30].

the character Sprite is on, which is determined in the Unity Editor. It then re-enables the `BoxCollider2D` component, only to reset it. Then, it determines that if the space dictated by `hit` is empty (indicated by the transform variable being null), a coroutine[18] starts to ensure the smooth movement of the character to the end point and then returns true if so, false otherwise (which means the space is taken and there is a collision).

```
/* Method to infect a character after a game collision. It... */
public void infect (character hit_character){

    int current_infect_count = sir.get_infected_count ();

    sir.infect (this.gameObject, hit_character.gameObject);

    color_cue ();

    if (sir.get_infected_count() > current_infect_count) {

        soundController.instance.PlaySingle (infect_sound);

    }

}

/* Method to call the SIR model class's recover method on... */
public void recover(){

    sir.recover (this.gameObject);

    color_cue ();

    if (sir.get_individual_status (this.gameObject) == "recovered") {

        soundController.instance.PlaySingle (recover_sound);

    }

}

/* Method to change the color of the current character, ... */
public void color_cue(){

    if (sir.get_individual_status (this.gameObject) == "infected") {

        renderer.color = Color.blue;

    }
    else if (sir.get_individual_status (this.gameObject) == "recovered") {

        renderer.color = Color.green;

    }

}
```

Once the boolean is returned from `movementController`'s `move` method, the controller's `attemptMove` method can then verify whether the `RaycastHit2D` created from it, `hit`, is null (indicating empty space) and, if so, allows the move to occur. If not, it calls the `onCantMove(Component component)` method, which is implemented in the character class. This method then gets the character that has been hit and then calls the `infect(character hit_character)` method. This method simply calls the `sirGameModel`'s `infect` method (discussed in Chapter 3, Section 3.5.3). It then calls `color_cue()`, to change the color of the Sprite according to the status. THe script checks if the current number of infected is greater than the previous number of infected and, if so, plays the infection sound dictated by the `soundManager`. The `recover()` method works in much the same way, calling the `sirGameModel`'s `recover(GameObject object)` method, playing a recovery sound if there are more recovered individuals at the end of the method than there were at the beginning. Thus, both of these methods simply monitor when it is necessary to update the model, and then notify the model to do so. The model then

[18] A coroutine is a function that can be suspended until given specific instructions [22].

proceeds to work through the necessary instructions and the updated information is then sent back to the character class, which then edits the view that the player sees, thus fulfilling the quasi-model-view-controller pattern that was set out to be achieved.

```
/* Method that calls the vaccinate() method when a player... */
void OnMouseDown(){

    if (sir.get_vaccine_counters () > 0 && sir.get_individual_status(this.gameObject) == "susceptible") {

        vaccinate ();

    }

    color_cue ();

}

/* Method to vaccinate the individual. Calls the... */
void vaccinate(){

    sir.vaccinate (this.gameObject);

    if (sir.get_individual_status (this.gameObject) == "recovered") {

        soundController.instance.PlaySingle (recover_sound);

    }

}
```

To vaccinate, two methods are needed in the character class, `OnMouseDown()` and `vaccinate()`. The former method is triggered upon a notification from the EventSystem game object. When it detects a mouse click on a particular character, this method is called in response. When this method is called, it first checks whether there are enough vaccines available to use, as well as that the character you click on is a viable candidate for vaccination (i.e. the character is "susceptible" in the model). If so, it then calls the `vaccinate()` method in the character script and, once this method is completed, calls the `color_cue()` method to change the color of the character according to its status.

The `vaccinate()` method is then tasked with communicating with the model. It starts by calling the vaccinate method in `sirGameModel`. The logic and instructions for the change in status is then handled by the model. Once this is completed, `character.vaccinate()` then checks whether the current object has their status as "recovered," implying that this status changed due to this method. If so, it plays a simple chime to indicate that the status has changed.

```
    /* Method to update the current state of the Game Object... */
    void FixedUpdate(){

        color_cue ();

        if (game_manager.countDownDone == false) {

            return;

        }

        if ((sir.get_recovered_count() == sir.get_population().Count) ||
            (sir.get_susceptible_count() + sir.get_recovered_count() == sir.get_population().Count)) {

            soundController.instance.PlaySingle (game_over_sound);
            sir.set_done_flag(true);

            SceneManager.LoadScene (3);

        }

        if (Time.fixedTime >= time_to_move) {

            moveCharacter();

            if (sir.get_individual_status (this.gameObject) == "infected" && Time.fixedTime >= time_to_recover) {

                recover ();

                time_to_recover = Time.fixedTime + 5.0f;

            }


            time_to_move = Time.fixedTime + 0.5f;

        }

    }
}
```

Each character is updated each frame, with specific actions happening every two seconds and five seconds. When the `FixedUpdate()` method is called, it first calls `color_cue()` to make sure all the characters are of the correct color. It then checks to see if there are any "infected" individuals left. If there are not, it plays a game over sound and then loads the next scene. If not, then it continues to check if enough time has passed to move the character. Every half-second, a character calls `moveCharacter()`, initiating the movement sequence discussed above. It then checks if the current object is "infected" and if five seconds have passed. If so, it calls the recover() method in the character script. It does this until the end of the game-play scene, where the actual game displays the final results.

3.6 Scene Four: The Results

The final Scene in the game is a graph that displays the trajectory of the spread of infection amongst each of the subpopulations after playing through a full infection in the game. This Scene's purpose is to demonstrate back to the player a more traditional representation of what they just played, thereby linking it back to the mathematical equations the game is modeling. By viewing this Scene immediately after playing the game, the hope is that it is easier for a player to associate their experiences with what they see on the screen. That way, playing the game can quickly be linked to the more traditional ways the SIR model is taught. It simply displays the change in population over time for each of the three populations ("susceptible," "infected," and "recovered") as a line graph of number of individuals over time. This is presented, along with a final game score.

This scene consists of a `Main Camera`, `EventSystem`, and a `Canvas`. The `Canvas` simply details the `Text` seen on screen, as well as a Button to end the game. In this case, the `Main Camera` has a script attached to it, called `graphController`, which utilizes a library obtained via the Asset Store to create the graph. The Button is controlled by a script that is similar to the scripts used to control the menus in previous Scenes. While the Unity Engine provides a `LineRenderer` class that renders lines, it is severely limited in terms of abilities. Thus, `Vectrosity` was used to actually plot this graph. This library additionally also ensures that the lines do not need to update with every frame if static, but remain unchanged, which is ideal for this particular use.

```
public class graphController : MonoBehaviour {

    private Text title;
    private Text sus_text;
    private Text inf_text;
    private Text rec_text;
    private Text x_title;
    private Text y_title;
    private Text score;

    /* Method start up the graphy controller and lay out the graph... */
    void Start () {

        title = GameObject.Find ("title").GetComponent<Text> ();
        sus_text = GameObject.Find ("sus_text").GetComponent<Text> ();
        inf_text = GameObject.Find ("inf_text").GetComponent<Text> ();
        rec_text = GameObject.Find ("rec_text").GetComponent<Text> ();
        x_title = GameObject.Find ("x_title").GetComponent<Text> ();
        y_title = GameObject.Find ("y_title").GetComponent<Text> ();
        score = GameObject.Find ("score_text").GetComponent<Text> ();

        title.text = "SIR Model over Time in Days";
        sus_text.text = "Susceptible";
        inf_text.text = "Infected";
        rec_text.text = "Recovered";
        x_title.text = "Number of Days";
        y_title.text = "Number of Individuals";
        score.text = "Score: " + gameManager.instance.score.ToString ();

        int total = gameManager.instance.total;
        List<int> day = gameManager.instance.day;
        List<int> sus_count = gameManager.instance.susceptible_count;
        List<int> inf_count = gameManager.instance.infected_count;
        List<int> rec_count = gameManager.instance.recovered_count;

        for (int i = 1; i < day.Count; i++) {

            int x_one = ((day [i-1] * Screen.width) / day.Count);
            int x_two = ((day [i] * Screen.width) / day.Count);
            int y_sus_one = ((sus_count [i - 1] * Screen.height) / total);
            int y_sus_two = ((sus_count [i] * Screen.height) / total);
            int y_inf_one = ((inf_count [i - 1] * Screen.height) / total);
            int y_inf_two = ((inf_count [i] * Screen.height) / total);
            int y_rec_one = ((rec_count [i - 1] * Screen.height) / total);
            int y_rec_two = ((rec_count [i] * Screen.height) / total);

            VectorLine.SetLine (Color.black, new Vector2 (x_one, y_sus_one),
                new Vector2 (x_two, y_sus_two));
            VectorLine.SetLine (Color.red, new Vector2 (x_one, y_inf_one),
                new Vector2 (x_two, y_inf_two));
            VectorLine.SetLine (Color.green, new Vector2 (x_one, y_rec_one),
                new Vector2 (x_two, y_rec_two));

        }


    }
```

The script for this is relatively simple. Upon loading the scene, the `Start()` function is called and ensures that the `Text` local variables point to the `Text` components that make up the UI. It then sets up the Text to display the titles and legend of the graph. It then gets the data recorded via the `gameManager` script and points local list containers to the data. It then loops through all the data, calculating the coordinates for each time point, adjusting the values so that they are scaled for the screen size. This is done by getting two list entries at a time from each list, creating the start point and end point of each Vector. Once these values are calculated, `Vectrosity`'s `VectorLine.SetLine` method is utilized to plot every single vector that is calculated right above. This method simply plots the Vector between the two given points, thus creating the line graph.

Chapter 4

Evaluation

The evaluation of this project is broken down into two separate evaluations. The first section of the evaluation was done to evaluate the accuracy of the underlying model that was constructed programmatically. This involved verifying the simple C# script that was written as an exploratory venture, as well as verifying that the implemented model in the game-play level retains that accuracy. This was done prior to the completion of the prototype. Once the prototype was finished, the second evaluation period commenced. This involved recruiting several participants to complete a short pre-quiz to assess their knowledge and skill before there are any changes, playing the game, and then completing a post-quiz. This was followed up by a second experiment that was a variation on the first. The following will display the results of each of these evaluation measures and discuss the results.

4.1 Evaluating the Model

The first step in evaluating this project was to validate that the simple model written in C#, as well as the more complex in-game model, are accurately portraying the behavior of the SIR model. Accuracy involved comparing these models to the details in the original paper proposed by Kermack and McKendrick [9], as well as to previously implemented SIR models. This exploratory evaluation was vital to the project overall because a fundamental flaw in the understanding of the model would lead to a flawed representation of the model in the game. This would undermine any effort in investigating if playing a game is an effective way of helping individuals learn this model. Thus, this is an important early step in the evaluation of the game as a whole.

4.1.1 Evaluating the Simple C# Script

The simple C# model was the first model to be evaluated, as it was done in part to explore methods of implementing the SIR model using C#. This experiment was run within Terminal on Apple's OSX, with the script printing out a CSV file for the population data captured at each time-point for each subpopulation. Those CSV files were labeled, saved, and then used to produce three CSV files that contained an average over five trial runs. Each time a trial was run, the CSV files produced were reviewed to ensure that at each time point, the three subpopulations added up to equal the total population. This process was repeated for differing total populations:

- 100 individuals
- 1000 individuals
- 10.000 individuals

- 100.000 individuals

After running five trials for each total, the results were then averaged out to create an average for each subpopulation at each time-point. This, then, was the data used to create the graphs that are below.

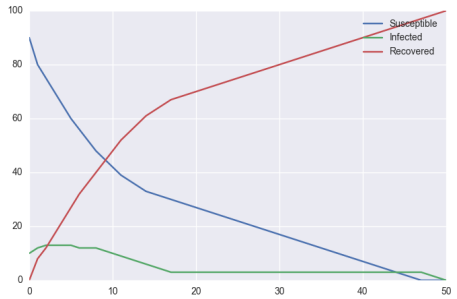


FIGURE 4.1: "Total population of 100 over time in days"

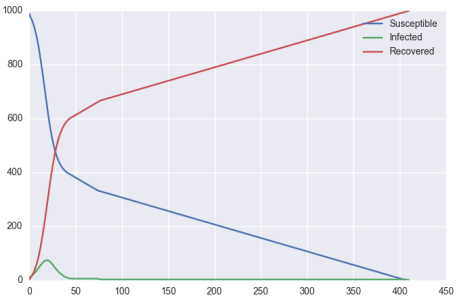


FIGURE 4.2: "Total population of 1000 over time in days"

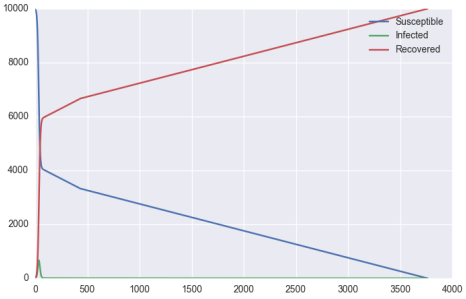


FIGURE 4.3: "Total population of 10.000 over time in days"

These all demonstrate similar behavior. The peak of the infection (i.e. the highest amount of infected individuals) occurs relatively early on in all four scenarios. This is to be expected, as in the initial phases of an infection there are many more individuals that are viable candidates to be infected. This spreads the disease more quickly and effectively than in the latter stages of an epidemic. Each of these graphs demonstrate that the epidemic increases until the density of the susceptible population reaches a certain threshold; once that threshold is reached by the susceptible population, the epidemic starts to slow down, as seen by the steady decrease of infected individuals from the peak of infection. Kermack and McKendrick observed

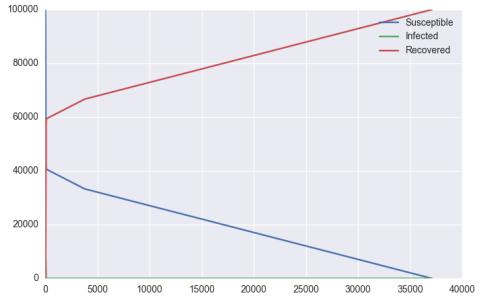


FIGURE 4.4: "Total population of 100.000 over time in days"

that an epidemic would continue spreading until a certain threshold is met, whereupon it begins to subside[9]. The behavior observed in these graphs align themselves with this observation.

Notice as well that, as the populations grew larger, the infection spread more effectively and quickly. This reflects part of the model, that changes in population density can vary the size of the epidemic; Kermack and McKendrick point out that the size of an epidemic increases rapidly as density increases [9]. Thus, it makes sense that, in the implemented model, the epidemic speed and severity increases as the amount of individuals increase. In the first graph, where the density is low, the infection is slow and does not spread as quickly, while the third and fourth graphs demonstrate a rapid rise in the infected population early in the infection, adhering to this principle.

4.1.2 Evaluating the In-Game Model

Using the simple script as a base, the in-game model was implemented to utilize more of the capabilities of Unity. The model was implemented as discussed above in Chapter 3. Much like for the simple C# script, this model was evaluated by running five different trials and then producing an averaged results table. This table was then used to create a graph representation of the course of the infection. This particular model was tested with 100 individuals, the medium setting for the game. Whereas the simple C# script could be tested with varying sizes of populations, the in-game model was limited in terms of size, so the medium population size supported was used to investigate the model's accuracy (as the model can hold up to 200 individuals on its game board while still allowing for character movement).

This model follows the principles set for by Kermack and McKendrick. The epidemic spreads and peaks at certain point. That certain point, then, is the high point for the infection, as well as the threshold for the susceptible population. Past this threshold, the epidemic starts to slowly stop, until the epidemic is completely over. While these results are similar to those above, the results differ greatly from the results obtained in the simple C# scripts 100 individuals scenario. However, this is due to the fundamental differences in the way these models are implemented. The C# script, taking place in an abstract manner, does not have the same physical space constraints the in-game model does for individuals, which may be why the infection does not spread rapidly in this scenario. The population density for the in-game model plays a much more important role in the spread of infection, simply because there is only a limited amount of space for the individual Sprites to move. Thus, the density dictates the spread of the epidemic for the in-game model more than it does in the simple C# script.

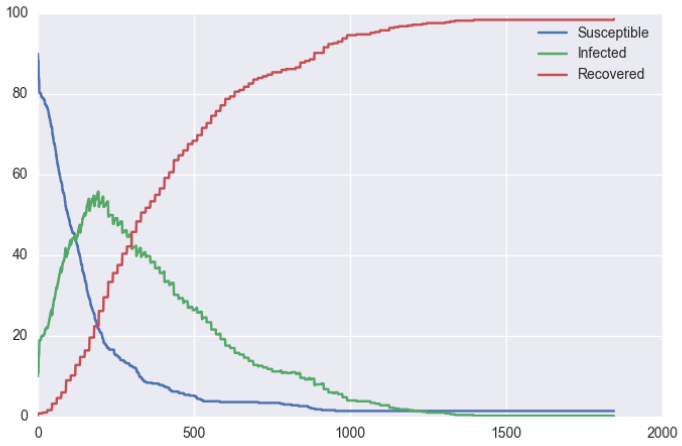


FIGURE 4.5: "In-Game Model with population of 100 over time in seconds"

4.1.3 Comparison to Previous Implementations

The final step in evaluating the accuracy of this model was to compare it to several different implementations of the model, completed by other groups. In doing this, an outside model, created in a different language and manner, could verify the accuracy of model completed for this particular work. These models were obtained from work previously done at the University of Warwick in the United Kingdom, as well as Duke University in the United States.

The model implemented by the University of Warwick followed the assumptions of a simple SIR model, presuming that the total population stays fixed. Additionally, their rates are determined in days and presume homogeneous mixing[8]. Their model was implemented in Python, as opposed to the C# language used for this project. The resulting data after running their model is seen below.

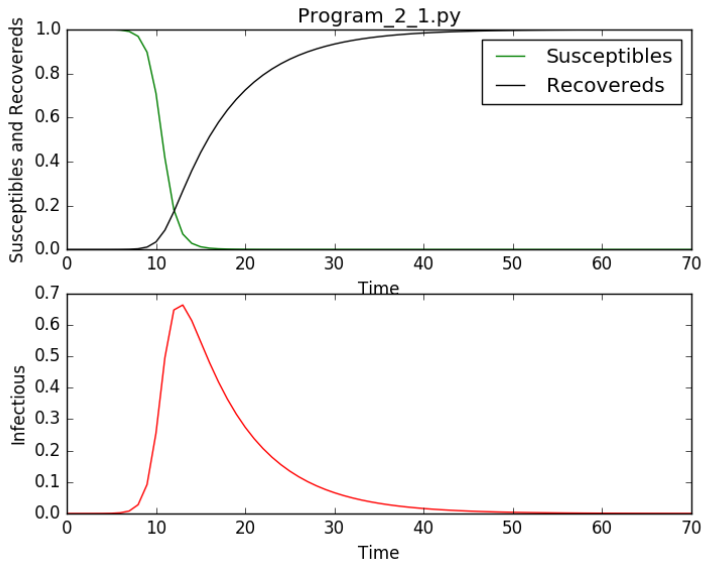


FIGURE 4.6: "Results from the University of Warwick Model[8]"

The resulting figure is similar to the simple C# script and very closely follows the in-game model. There is peak of infection that matches where the susceptible and the recovered graphs intersect; this is the threshold mentioned by Kermack and McKendrick[9]. It also demonstrates that, much as the models implemented for this project show, the peak of infection tends to happen early in the infection. This Python model demonstrates that, for this project, the models implemented are accurate in their functioning. The behavior seen in the independently developed Python model is similar to the behavior in both models, thus verifying that the models used in this project are accurate depictions of the SIR model.

A second model used to verify the SIR model was independently developed by researchers at Duke University. This particular model was developed in MATLAB. They used the data from the "Hong Kong Flu" Epidemic as the basis of their model. Their results are displayed below.

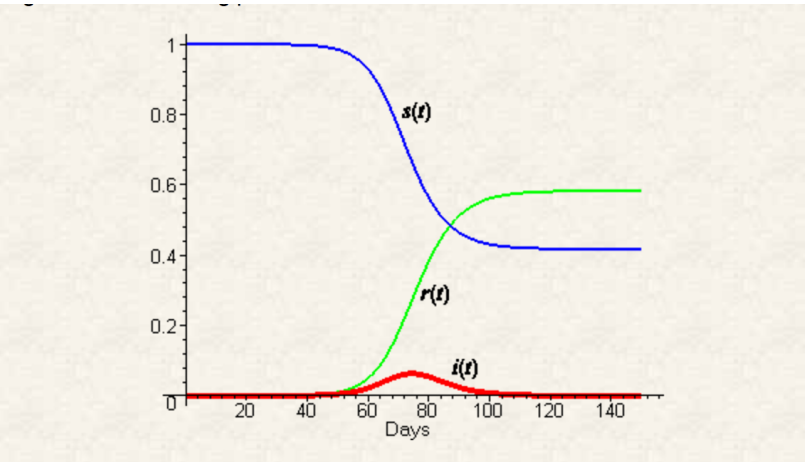


FIGURE 4.7: "Results from the Duke University Model[41]"

In this implemented model, the peak of the infection is much later than the previously seen models. Because this is based on real world data, however, the behavior may be slightly different than models created in a more sterile and controlled environment. Despite this, however, the same pattern of behavior in terms of the growth of the epidemic remains. The epidemic peaks close to when the susceptible and recovered populations intersect. Thus, when the epidemic reaches a certain threshold, the spread of disease starts to slow down and will eventually stop.

In comparing the two models created for this project with two different models created independently, the purpose was to ensure that the understanding of this model was sufficient for the needs of this project. This was a vital part of this project. The model's accuracy is vital in using this as an educational tool; verifying this accuracy was an important step in understanding whether the presented work simulated the SIR model. Following the comparison of two models, it is clear that both implemented models accurately depict the SIR model.

4.2 User Evaluation and Assessments

The next phase of evaluation involved evaluating the game's effects on players. This was broken into two different experiments. In the first, users were tasked with completing a short quiz that contained several questions regarding the SIR model, as well as questions asking their previous experience with video games, the SIR model,

and differential equations. They would then play the prototype to its completion, and immediately complete a post-quiz afterwards, which was similar to the pre-quiz but contains five new questions. They would then also input any comments or bugs they encountered while playing the prototype. The second experiment was similar but added one single step to the process. This step involved reading a short description of the SIR model, followed by spending a two-minute word search. The participant would then proceed with the rest of the experiment identically to the first group. This way, there could be some insight into whether the learning gains are improved when the game is presented in conjunction with more traditional literature on the SIR model.

4.2.1 Experiment One Results

These participants were tasked with taking a short, ten question quiz with questions pertaining to the SIR model. They would also be asked to rate the experience with video games, the SIR model, and differential equations on a a scale of 1 to 5, with 1 being none at all and 5 being extensive. Afterwards, they played the prototype to its completion. Upon finishing the prototype, they were then tasked with completing another quiz. This post-quiz contained five questions that were in the previous quiz, with their answers shifted in position, and five new questions. They were then asked to present any comments about the game that they played. Presented here are the results of the first experiment.

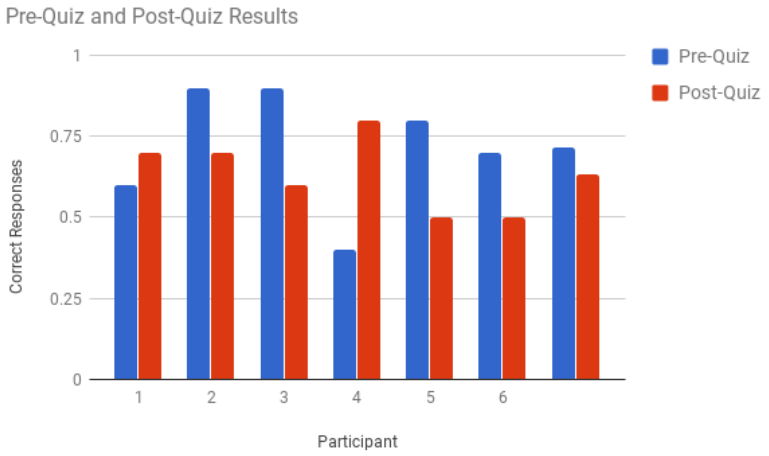


FIGURE 4.8: "Quiz Results from Experiment One"

Evidently, the participants did not all see an increase in performance. Participants 01 and 04 both saw an increase in performance from the pre-quiz to the post-quiz. However, participants 02, 03, 05, and 06 all saw a reduction in performance between the pre-quiz and the post-quiz. So, while it may seem that some participants benefited from playing the game, some did not necessarily see their performance increase after playing the game. This finding was unexpected, as one would expect that playing the game would help players better understand the SIR model. However, this could be caused by the previous experience the participants had going into the experiment. Because every participant rated their experience with the SIR model as little to none, it may be that their performances on both quizzes are affected by this. Those with high pre-quiz score and low post-quiz score (such as Participant 03) may have been more of a product of guessing rather than foreknowledge, leading

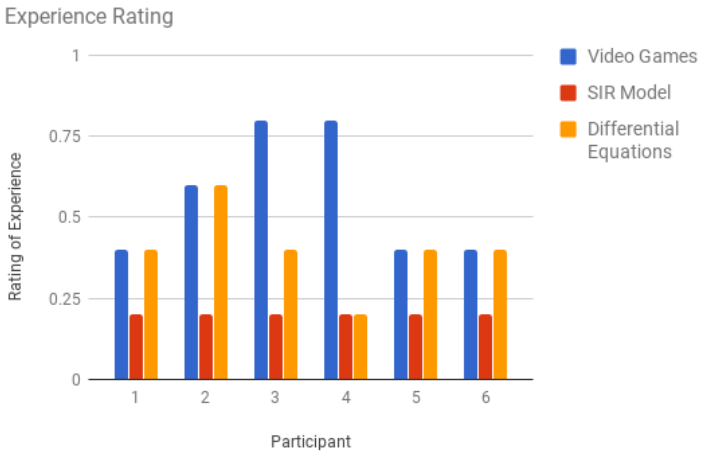


FIGURE 4.9: "Experience Ratings according to Users"

one to conclude that the high score on their pre-quiz is rather misleading. This is also a victim of sample size; with a larger group of testers, it would have been possible to recruit more participants with experience in the SIR model.

It would seem that playing the game on its own, without any short discussion of the SIR model in a traditional manner, does necessarily impart any learning benefits. However, as mentioned before, the cohort recruited did not have experience with the SIR model, so one cannot say that this is a representative sampling. It may be beneficial to examine how participants do in this same scenario, but with more knowledge of the SIR model going into the experiment. Thus, the next experiment was undertaken to examine this very thing.

4.2.2 Experiment Two Results

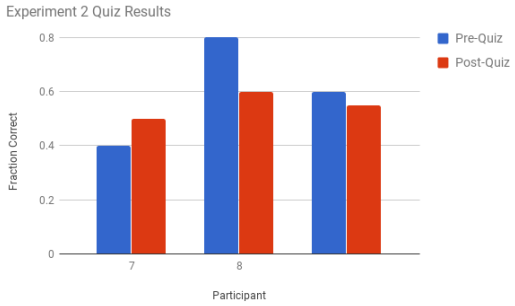


FIGURE 4.10: "Quiz Results from Experiment Two"

This experiment was an extension of the previous experiment. Much like before, participants completed a short pre-quiz, played the prototype, and then completed a post-quiz. However, as opposed to the previous experiment, participants in this experiment began by reading a short, traditional explanation of the SIR, provided by lecturers at Duke University [41]. This was done to provide some context to the participants, as many of the participants had little to no previous experience with the model. The participants would then play a word search for two minutes. That way, the participant would not necessarily have what they read about the SIR model fresh in their memory; it acts as a sort of mental palate cleanser. They would then

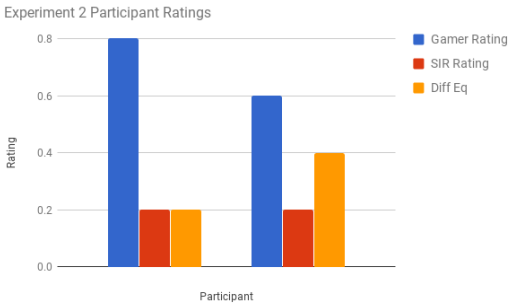


FIGURE 4.11: "Experiment Two Experience Ratings"

complete the experiment as set up in the first experiment. They would complete the pre-quiz and rate their experience with video games, the SIR model, and differential equations. They would then play the game, and complete a post-quiz. This way, the game would be played in conjunction with a more traditional instruction method of this model.

While one might have expected the added portion of instruction beneficial to the learning of the participants, the results of this experiment are not conclusive in any manner. Though Participant 07 managed to improve, Participant 08 diminished in performance. However, as compared to the previous experiment, the average of both participants pre- and post-quiz scores demonstrate a smaller reduction in performance. Once again, though, the size of the cohort limits the ability to make any conclusive declarations about the data. This second experiment, however, would be worth further investigation in a wider reacher experiment.

Chapter 5

Conclusion

Presented in the preceding chapters is a game with the intention of teaching players the SIR model of infection. Through this work, an application was created in which players can play their way through an infection, trying to do their best to contain the outbreak while retaining resources. Utilizing the libraries available within the Unity development environment and engine, the game simulates the spread of an epidemic while adhering to the SIR model of epidemics. This allows players to interact with a simulated epidemic and learn how simple steps like quarantine and vaccination can affect it.

Several scripts were written to create the simulated epidemic behavior. Some control the behavior of the application overall, while some control what was occurring on the screen. The vital scripts of gameManager, boardManager, and sirGameModel control the behavior of the game overall. gameManager keeps track of relevant data such as "power," "score," and "day," as well as calling both the boardManager and sirGameModel to initialize the game overall. boardManager defines the class that dictates the way the board is laid out. sirGameModel defines the behavior of SIR model in the game. The scripts character and movementController, as well as menuScript, dictate much of the view, by controlling the character Sprites and the UI.

Through the evaluation of the created model, it is evident that the application correctly simulated the SIR model of epidemics. As seen in Chapter 4, an exploratory script directly implemented the model in only C# and resembled the expected behavior of the SIR model. The ideas utilized for this exploratory work then created the basis for which the model was extended for use in the application. Once the sirGameModel script was implemented, this was then evaluated for accuracy as well. This model also demonstrated the expected behavior of an epidemic according to the SIR model. Knowing the game correctly simulated the SIR model, the prototype of the game was then completed and then the user evaluation phase commenced. Two experiments were set up for participants. In first experiment, they simply completed a pre-quiz, rate their previous experience with the model and video games, play the prototype, and then take a post-quiz. In the second experiment, participants read a short description of the SIR model, played a word game, and then proceeded through the same steps that the participants in the first experiment did. These two modest evaluations cannot conclusively demonstrate that the game improved the learning outcomes of the evaluators. While some participants in both experiments did see an increase in performance between the pre-quiz and the post-quiz, some saw their performance diminish. The limitation of the small sample size hindered the creation of consensus as to whether this game successfully improves the learning outcomes for students learning the SIR model.

Despite this, however, enough individuals that do show some improvement. Expanding the evaluation cohort would prove beneficial and allow for a greater understanding as to the learning effects of this work. Because the game's lessons on the SIR model are not explicitly mentioned but rather gathered as a result of playing the game, the game may work most effectively in conjunction with formal lessons on the model. Expanding the second study would provide the most beneficial insight into the ways this game effects learning outcomes.

Appendix A

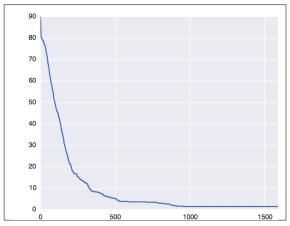
Quizzes

Pre-Survey

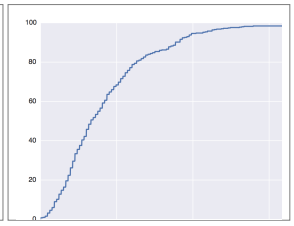
Please answer these questions prior to playing the single level prototype

* Required

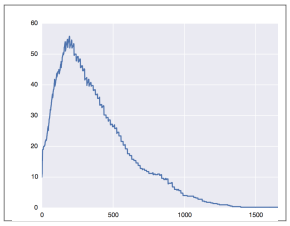
1. **What curve best represents the infected population over time in an epidemic?** *
Mark only one oval.



Option 1

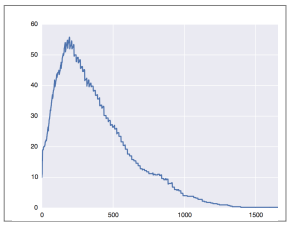


Option 2

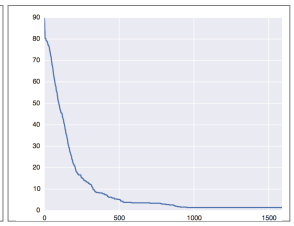


Option 3

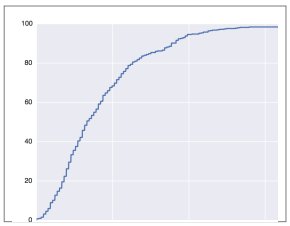
2. **What curve best represents the susceptible population over time in an epidemic?** *
Mark only one oval.



Option 1

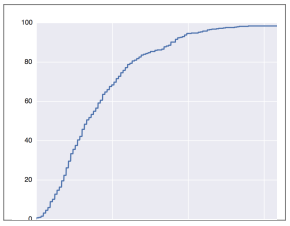


Option 2

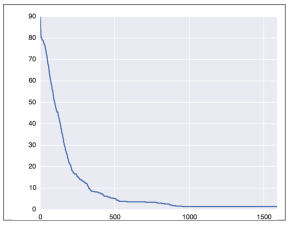


Option 3

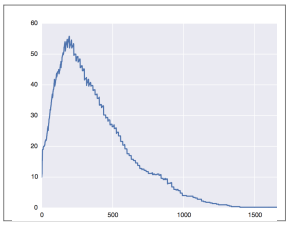
3. **What curve best represents the recovered population over time in an epidemic?** *
Mark only one oval.



Option 1



Option 2



Option 3

4. **How does the SIR model assume that susceptible and infected individuals eventually shift to the recovered population?**
Mark only one oval.

- Death
- Natural Recovery
- Vaccination
- Quarantine
- All of the Above

5. **True or False: On average, some fraction of the infected population will shift over to the recovered population each day.** *
Mark only one oval.

- True
- False

6. **True or False: An epidemic (in the SIR model) generally ends before the population of susceptible individuals is exhausted.** *
Mark only one oval.

- True
- False

7. **True or False: At each stage of an epidemic, adding the number of susceptible, infected, and recovered individuals will always equal to the total population.** *
Mark only one oval.

- True
- False

8. **How would you expect the number of susceptible individuals to change over time during an epidemic using the SIR model?**
Mark only one oval.

- ◯ Increase
- ◯ Decrease
- ◯ Stay Fixed
- ◯ Increase then decrease
- ◯ Decrease then increase

9. **What assumption does the SIR model make in regards to population size?** *
Mark only one oval.

- ◯ It increases over time.
- ◯ It decreases over time.
- ◯ It stays fixed.

10. **What is the amount of recovered individuals dependent on?** *
Mark only one oval.

- ◯ Amount of susceptible individuals
- ◯ Amount of infected individuals

11. **How much experience would you say you've had playing video games?** *
Mark only one oval.

	1	2	3	4	5	
None at all	◯	◯	◯	◯	◯	Extensive

12. **How much experience would you say you've had with the SIR epidemic model?** *
Mark only one oval.

	1	2	3	4	5	
None at all	◯	◯	◯	◯	◯	Extensive

13. **How much experience would you've had with differential equations?** *
Mark only one oval.

	1	2	3	4	5	
None at all	◯	◯	◯	◯	◯	Extensive

A.1 Post-Quiz

Post-Survey

Please complete this after you have played the single level prototype.

* Required

1. **How would you expect the number of susceptible individuals to change over time during an epidemic using the SIR model?** *
   Mark only one oval.
   - ◯ Decrease
   - ◯ Stay Fixed
   - ◯ Increase then decrease
   - ◯ Increase
   - ◯ Decrease then increase

2. **For the SIR model, what type of mixing is assumed?** *
   Mark only one oval.
   - ◯ Homogenous
   - ◯ Heterogenous

3. **True or False: At each stage of an epidemic, adding the number of susceptible, infected, and recovered individuals will always equal to the total population.** *
   Mark only one oval.
   - ◯ True
   - ◯ False

4. **How is the relative contagiousness of the disease measured in the SIR model?** *
   Mark only one oval.
   - ◯ Rate of change between susceptible to infected
   - ◯ Rate of change between infected and recovered
   - ◯ The number of close contacts per infected individual

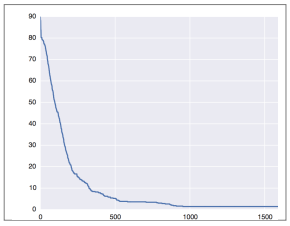
5. **What is herd immunity?** *
   Mark only one oval.
   - ◯ When there are no infected individuals left to spread the disease
   - ◯ When enough individuals are recovered to reduce the spread of disease
   - ◯ When there are no longer enough susceptibles in the population to spread the disease

6. **True or False: On average, some fraction of the infected population will shift over to the recovered population each day.** *
   Mark only one oval.
   - ◯ True
   - ◯ False

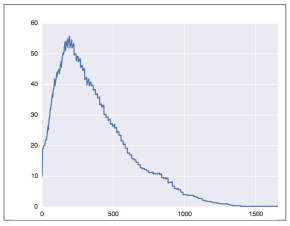
7. **What is the size of the infected population like at the peak of an epidemic?** *
   Mark only one oval.
   - ◯ Relatively high compared to total population
   - ◯ Relatively low compared to total population
   - ◯ Relatively average compared to total population

8. **How does the SIR model assume that susceptible and infected individuals eventually shift to the recovered population?** *
   Mark only one oval.
   - ◯ Vaccination
   - ◯ Death
   - ◯ Natural Recovery
   - ◯ Quarantine
   - ◯ All of the Above

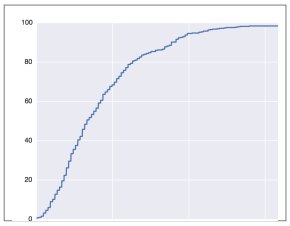
9. **What curve best represents the susceptible population over time in an epidemic?** *
Mark only one oval.



Option 1



Option 2



Option 3

10. **What is the amount of recovered individuals dependent on?** *
Mark only one oval.

Amount of susceptible individuals
Amount of infected individuals

11. **How would you rate your understanding of epidemics after playing this game, in terms of what you understood before?** *
Mark only one oval.

	1	2	3	4	5	
Very little						A lot

12. **Did you feel like the game conveyed some information in regards to the spread of diseases?** *

13. **Were there any game mechanics that you didn't understand or were poorly explained?** *

14. **Do you feel that there are improvements that could be made to the game?**

15. **Were there any bugs you encountered? If so, do you recall what happened?**

Powered by
Google Forms

Bibliography

[1] Google Arts and The British Museum Culture. *The Royal Game of Ur*. 2016. URL: https://www.google.com/culturalinstitute/beta/asset/the-royal-game-of-ur/MwE2MMZNSKiTwQ (visited on 07/17/2017).

[2] Thomas M. Connolly et al. "A systematic literature review of empirical evidence on computer games and serious games". In: *Computers and Education* 59 (2012), pp. 661–686.

[3] Microsoft Corporation. *Dictionary <TKey, TValue> Class*. URL: https://msdn.microsoft.com/en-us/library/xfhwa508(v=vs.110).aspx (visited on 07/19/2017).

[4] Rosemary Garris, Robert Ahlers, and James E. Driskell. "Games, motivation, and learning: A research and practice model". In: 2002.

[5] Mike Geig. In: *SamsTeachYourself Unity Game Development*. 2013.

[6] Thomas Hainey, Thomas Connolly, and Elizabeth Boyle. "A Survey of Students' Motivations for Playing Computer Games: a Comparative Analysis of Three Studies in Higher Education". In: *Proceedings of the 3rd European Conference on Game Based Learning*. Ed. by Maja Pivec. Academic Conferences and Publishing Limited (ACPIL), 2009, pp. 154–163.

[7] Margaret A. Honey. "Learning Science Through Computer Games and Simulations". In:

[8] Matt J. Keeling and Pejman Rohani. "Simple SIR Model". In: *Modeling Infectious Disease in Humans and Animals*. 2007, pp. 19–27.

[9] W. O. Kermack and A. G. McKendrick. "A Contribution to the Mathematical Theory of Epidemics". In: *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* 115.772 (1927), pp. 700–721. ISSN: 09501207. URL: http://www.jstor.org/stable/94815.

[10] Scott W. McQuiggan et al. "Story-Based Learning: The Impact of Narrative on Learning Experiences and Outcomes". In: *Intelligent Tutoring Systems*. 2008.

[11] Penn Museum. *The Indian Games of Pachisi, Chaupar, and Chausar*. 2016. URL: https://www.penn.museum/sites/expedition/the-indian-games-of-pachisi-chaupar-and-chausar/ (visited on 07/17/2017).

[12] Enrica Pesare et al. "Game-based learning and Gamification to promote engagement and motivation in medical learning contexts". In: *Smart Learning Environments*. 2016.

[13] Jonathan P. Rowe et al. "Integrating Learning, Problem Solving, and Engagement in Narrative-Centered Learning Environments". In: *I. J. Artificial Intelligence in Education* 21 (2011), pp. 115–133.

[14] Tim Ryan. *The Anatomy of a Design Document, Part 1: Documentation Guidelines for the Game Concept and Proposal*. 1999. URL: http://www.gamasutra.com/view/feature/131791/the_anatomy_of_a_design_document_.php?page=1 (visited on 07/19/2017).

[15] Unity Technologies. *AudioClip*. 2017. URL: https://docs.unity3d.com/ScriptReference/AudioClip.html (visited on 07/17/2017).

[16] Unity Technologies. *AudioSource*. 2017. URL: https://docs.unity3d.com/ScriptReference/AudioSource.html (visited on 07/21/2017).

[17] Unity Technologies. *BoxCollider2D*. 2017. URL: https://docs.unity3d.com/ScriptReference/BoxCollider2D.html (visited on 07/17/2017).

[18] Unity Technologies. *Button*. 2017. URL: https://docs.unity3d.com/ScriptReference/UI.Button.html (visited on 07/17/2017).

[19] Unity Technologies. *Camera*. 2017. URL: https://docs.unity3d.com/ScriptReference/Camera.html (visited on 07/21/2017).

[20] Unity Technologies. *Canvas*. 2017. URL: https://docs.unity3d.com/ScriptReference/Canvas.html (visited on 07/17/2017).

[21] Unity Technologies. *Company Facts*. 2017. URL: https://unity3d.com/public-relations (visited on 07/16/2017).

[22] Unity Technologies. *Coroutine*. 2017. URL: https://docs.unity3d.com/ScriptReference/Coroutine.html (visited on 07/21/2017).

[23] Unity Technologies. *EventSystem*. 2017. URL: https://docs.unity3d.com/Manual/EventSystem.html (visited on 07/21/2017).

[24] Unity Technologies. *gameObject.activeInHierarchy*. 2017. URL: https://docs.unity3d.com/ScriptReference/GameObject-activeInHierarchy.html (visited on 07/21/2017).

[25] Unity Technologies. *GameObject.GetComponent*. 2017. URL: https://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html (visited on 07/21/2017).

[26] Unity Technologies. *GameObject.SetActive*. 2017. URL: https://docs.unity3d.com/ScriptReference/GameObject.SetActive.html (visited on 07/21/2017).

[27] Unity Technologies. *Learning the Interface*. 2017. URL: https://docs.unity3d.com/Manual/LearningtheInterface.html (visited on 07/16/2017).

[28] Unity Technologies. *MonoBehavior.Awake*. 2017. URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html (visited on 07/21/2017).

[29] Unity Technologies. *Object.DontDestroyOnLoad*. 2017. URL: https://docs.unity3d.com/ScriptReference/Object.DontDestroyOnLoad.html (visited on 07/21/2017).

[30] Unity Technologies. *Physics2D.Lincast*. 2017. URL: https://docs.unity3d.com/ScriptReference/Physics2D.Linecast.html (visited on 07/21/2017).

[31] Unity Technologies. *RaycastHit2d*. 2017. URL: https://docs.unity3d.com/ScriptReference/RaycastHit2D.html (visited on 07/21/2017).

[32] Unity Technologies. *Rigidbody2D*. 2017. URL: https://docs.unity3d.com/ScriptReference/Rigidbody2D.html (visited on 07/17/2017).

[33] Unity Technologies. *SceneManager*. 2017. URL: https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html (visited on 07/21/2017).

[34] Unity Technologies. *Sprite*. 2017. URL: https://docs.unity3d.com/ScriptReference/Sprite.html (visited on 07/17/2017).

[35] Unity Technologies. *Sprite Renderer*. 2017. URL: https://docs.unity3d.com/ScriptReference/SpriteRenderer.html (visited on 07/17/2017).

[36] Unity Technologies. *Text*. 2017. URL: https://docs.unity3d.com/ScriptReference/UI.Text.html (visited on 07/17/2017).

[37] Unity Technologies. *Time.timeScale*. 2017. URL: https://docs.unity3d.com/ScriptReference/Time-timeScale.html (visited on 07/21/2017).

[38] Unity Technologies. *Transform*. 2017. URL: https://docs.unity3d.com/ScriptReference/Transform.html (visited on 07/21/2017).

[39] Unity Technologies. *Vector3*. 2017. URL: https://docs.unity3d.com/ScriptReference/Vector3.html (visited on 07/21/2017).

[40] Unity Technologies. *Welcome to the Unity Scripting Interface*. 2017. URL: https://docs.unity3d.com/ScriptReference/index.html (visited on 07/17/2017).

[41] *The SIR Model for the Spread of Disease*. https://services.math.duke.edu/education/ccp/materials/diffcalc/sir/sir1.html. Accessed: 2017-04-06.

[42] Terrence M. Tumpey et al. "Characterization of the Reconstructed 1918 Spanish Influenza Pandemic Virus". In: *Science* 310.5745 (2005), pp. 77–80. ISSN: 0036-8075. DOI: 10.1126/science.1119392. eprint: http://science.sciencemag.org/content/310/5745/77.full.pdf. URL: http://science.sciencemag.org/content/310/5745/77.

[43] Jarrett Walker. *Learning from Mini Metro*. 2016. URL: http://humantransit.org/2014/12/learning-how-transit-works-from-mini-metro.html (visited on 07/17/2017).

[44] Sam White. *Minecraft in Space: why NASA is embracing Kerbel Space Program*. 2014. URL: https://www.theguardian.com/technology/2014/may/22/kerbal-space-program-why-nasa-minecraft (visited on 07/17/2017).

[45] Pieter Wouters, Erik D. van der Spek, and Herre van Oostendorp. "Towards the Development of a Games-Based Learning Evaluation Framework". In: *Current Practices in Serious Game Research: A Review from a Learning Outcomes Perspective*. Ed. by Thomas Connolly, Mark Stansfield, and Thomas Hainey. 2009, pp. 232–250.