Dynamic Simulation of Infectious Bacteria

Alex Goncalves

Contents

[Analysis 3](#_Toc98334775)

[Background 3](#_Toc98334776)

[Client Interview 4](#_Toc98334777)

[Existing systems 5](#_Toc98334778)

[Similar Existing product 5](#_Toc98334779)

[Objectives 6](#_Toc98334780)

[Design 8](#_Toc98334781)

[Background 8](#_Toc98334782)

[Main Program Hierarchy Chart 10](#_Toc98334783)

[Flowchart for spawning mammalian cells 11](#_Toc98334784)

[Pseudocode for spawning mammalian cells 12](#_Toc98334785)

[Flowchart for spawning bacterial cells 13](#_Toc98334786)

[Pseudocode for spawning bacterial cells 14](#_Toc98334787)

[Hierarchy diagram for Cell Interactions 14](#_Toc98334788)

[Pseudocode for Interactions between Mammalian Cells 16](#_Toc98334789)

[Pseudocode for elastic collisions 16](#_Toc98334790)

[Critical Path 17](#_Toc98334791)

[Implementation 17](#_Toc98334792)

[Overview 18](#_Toc98334793)

[Systems Flow Diagram 19](#_Toc98334794)

[Algorithms 20](#_Toc98334795)

[Documenting design 20](#_Toc98334796)

[Part 1 – Mammalian Cells 21](#_Toc98334797)

[Part 2 – Bacterial Cells 22](#_Toc98334798)

[Part 3 – Bacterial-mammalian interactions / finalizing simulations 22](#_Toc98334799)

[Part 4 – UI Management 24](#_Toc98334800)

[Testing 30](#_Toc98334801)

[Code 39](#_Toc98334802)

[M cell spawning script 51](#_Toc98334803)

[Mammalian Collision handling script 53](#_Toc98334804)

[Bacteria cell spawner 55](#_Toc98334805)

[Collision Handling script for bacteria 57](#_Toc98334806)

[Infection script 59](#_Toc98334807)

[Bacteria Fade out script 63](#_Toc98334808)

[Duplication script 64](#_Toc98334809)

[Duplication bug solution 66](#_Toc98334810)

[End scene Transition 68](#_Toc98334811)

[Static variable script 70](#_Toc98334812)

[Wall spawning algorithm 71](#_Toc98334813)

[UI handling scripts 73](#_Toc98334814)

[main menu UI script 73](#_Toc98334815)

[Pre simulation UI script 74](#_Toc98334816)

[Simulation UI script 75](#_Toc98334817)

[End Scene UI script 79](#_Toc98334818)

[Evaluation 80](#_Toc98334819)

[Review of Objectives 80](#_Toc98334820)

[Client Feedback on Client testing 80](#_Toc98334821)

[Evaluation 80](#_Toc98334822)

# Analysis

## Background

My project is to create a dynamic simulation which models interactions between mammalian cells and bacterial cells. This simulation would be able to display the speed at which bacteria can exponentially replicate. I am making this project to help aid understanding of the certain types of interactions between human cells and bacteria and how altering specific features of the infecting bacteria may alter the rate of spread at a simplified level. My client is Tracy Nevitt, of Neem biotech, who specializes in research into how different compounds and molecules affect a given bacteria, my solution would allow for the ability to change the parameters of the bacteria in accordance with how the molecule in question is known to affect bacterial properties and would be able to simulate the actual effects on the virulence of the bacteria.

This project will be a multi-agent system. A multi-agent system consists of elements/objects which interact and react to each other autonomously. This type of system is powerful in simulating many different subject matters, any type of phenomena that involves many different variables to interact can be created using agents. These systems often have simple rules but due to the dynamic nature of the interactions between agents, very complex emergent behaviors can be modeled.

An agent can be any active entity in the model, e.g. people, vehicles, animals, or whatever is relevant to the system. In this simulation the agents will be mammalian and bacterial cells.

I will be creating this project using Unity with C# to create a desktop application and several libraries and using object-oriented programming as C# is very well suited to object oriented and, in this simulation, OOP is very applicable with each cell being able to be an object. There are multiple areas of interest in this project, but the main hurdle is how interactions between cells will be handled as I want movement to be fluid and organic,

## Client Interview

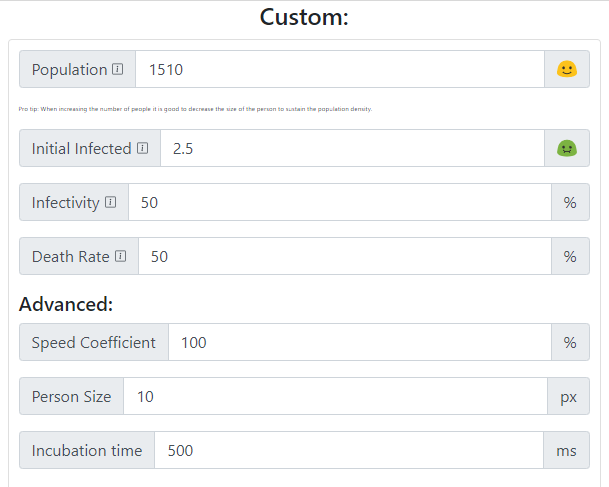
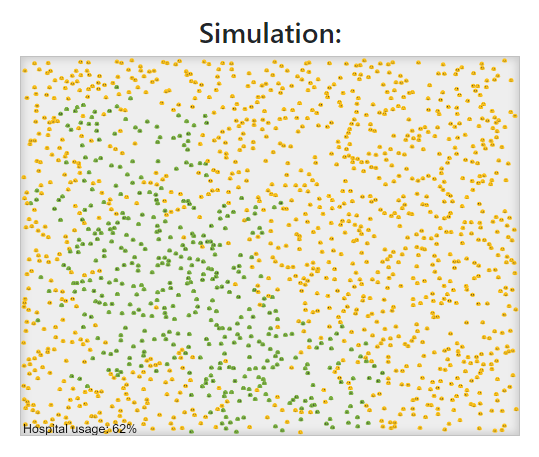
* What does the project need to be able to simulate –
  + bacterial cells invading multiplying cell – antibodies stop bacteria from entering mammalian cell
* Who is going to be using it –
  + client and colleagues of client
* What platform does it need to be –
  + desktop app
* Technical requirements for program –
  + simple enough that non tech savvy users
* What will it be used for –
  + major pipeline of Neem is to identify molecules that have antibacterial effects, and this could be used to help see the effectiveness of those molecules given certain parameters on the virulence of the bacteria
* Scalability, how many cells should it be able to simulate –
  + unknown too many variables to know as storage and performance allowance is different for each device
* Level of detail in cells e.g. how should it look visually –
  + Mammalian cells are circular and bacterial cells are capsule shaped, bacteria are to be much smaller than the mammalian cells
* How should the mammalian cells react if infected
  + If they die, they turn red and fade away, if they survive they turn green and stay in the simulation
* Should the cells be limited by the boundary of the screen (ie should cells roam and move around off the screen, or bounce off of the edge of the screen) –
  + Yes, they should bounce off of screen boundaries
* How do you currently view cell and bacterial interactions (what do you currently use)
  + They must be prepared on slides and viewed under a microscope
* What should be available to the user –
  + simply just a way to start the simulation and modify the parameters of the simulation
* Do bacterial cells randomly float around or do they actively move towards cells at a certain distance-
  + they float randomly until contacting mammalian cell
* How many bacterial cells are created after they infect a mammalian cell –
  + Between 2 and 15
* Do the mammalian cells need to be able to duplicate –
  + no, they do not need to, cells used like this are made to have a slow rate of cell division

## Existing systems

After speaking with the client, It was discovered that they currently have no method of simulating cells in this way.

## Similar Existing product

Many similar solutions simulate rate of spread of infection in a population e.g. a pandemic. While this is not the same can give similar insight into exponential spread. An example is this website used to simulate the spread of covid-19 among a population.



|  |  |
| --- | --- |
| What I would like to recreate in my project | What I would change in my interpretation |
| * Simulation involves actual agents moving on the screen and interacting | * Have both mammalian cells and bacterial cells instead of just healthy and sick |
| * Properties of infection can be altered which change the rate of spread | * This example is web-based, and my solution Is going to be an application |
| * The number of agents that the simulation starts with can be determined by the user | * Upon infecting bacteria multiply and multiple are released when the host cell dies |
|  | * There is no cap on the number of healthy agents to begin with so the user may start the simulation with more agents than there is memory for |

## Objectives

1. Menus
   1. On startup window
      1. Option to begin a new simulation
         1. Clicking on run simulation takes user to pre run settings
      2. Option to end program
   2. Design
      1. Large buttons used for menu navigation
      2. Text must be readable, choose suitable font and colors
   3. In Simulation
      1. Option to pause simulation
         1. Option to end simulation is presented
         2. Option to reset simulation with current settings
         3. Option to create a new simulation
      2. Ability to change speed of simulation
      3. Ability to see the amount of time lapsed since the simulation began
   4. Pre simulation Menu
      1. Input for number of mammalian cells
      2. Input for number of bacterial cells
      3. Input for percentage for infection success rate
      4. Input for the amount of bacterial cells per successful infection
2. Simulation
   1. Cells cannot go beyond the bounds of the window and are reflected on contact with no change in velocity
   2. When simulation is paused cells of all types stop moving and interacting
   3. Mammalian cell properties
      1. behavior
         1. Cells should not move around the screen too violently
         2. If a cell dies it disappears and its effect on neighboring cells is removed
      2. Design
         1. Each cell is a rigid body (i.e. no deformation due to forces)
         2. Each cell is lightly colored
   4. Bacterial cell properties
      1. Behavior
         1. Bacterial cells spawn with a random velocity at a random position on the screen
         2. Bacterial cells bounce off other bacterial cells
         3. Bacterial cells should have the ability to adhere to mammalian cells before infection
      2. Infecting Mammalian Cells
         1. Infection is determined by not only contact with mammalian cell but also the chance of infection property of the bacteria
         2. A bacterial cell can infect a mammalian cell when it meets it
         3. The bacteria then fades away
         4. If infection is not successful, the mammalian cell turns green to indicate it has survived
         5. If infection is successful, the bacteria enter mammalian cell and duplicates up to 10 copies of itself
         6. After infected mammalian cell dies the cells duplicated inside are given the same velocity as the mammalian cell they infected
      3. Design
         1. Each cell is a rigid body, cannot be deformed by forces
         2. Bacterial cells oblong and ovular in shape

# Design

## Background

My project will be a simulation that models interactions between human and bacterial cells, the simulation must be able to allow the user to alter the parameters of the bacteria before the simulation using sliders and settings. The human cells must move in random directions and velocities, to model Brownian motion with particles moving randomly.

The simulation itself should be comprehendible and not too cluttered, possibly with a key to understand what is on the screen and an option to slow down or increase the speed of the simulation.

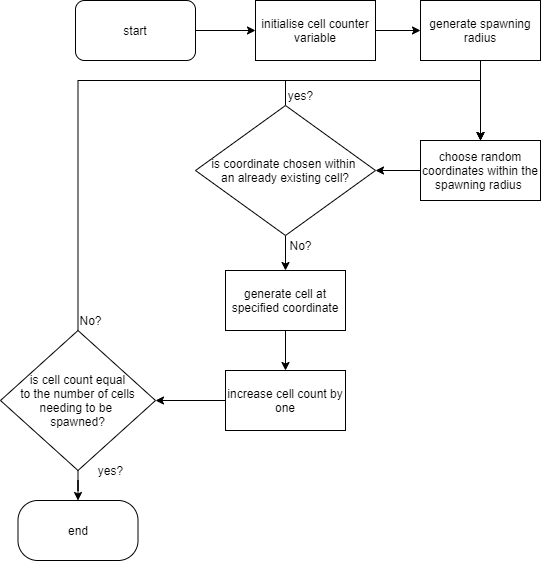
The UI for the pre-simulation scene should be easy to understand and contain the following parameters, number of bacterial cells, number of mammalian cells, chance of infection, and number of bacterial cells per infection.

## Main Program Hierarchy Chart

Timeline

Description automatically generated with medium confidence

## Flowchart for spawning mammalian cells



## Pseudocode for spawning mammalian cells

spawningMCells()

NewCell = null

MammalianCellCount = input(“number of mammalian cells”)

SpawnRadius = input(“spawn radius”)

SpawnPoint = new vector(0, 0)

For (int I = 0; I <= mammalianCellCount)

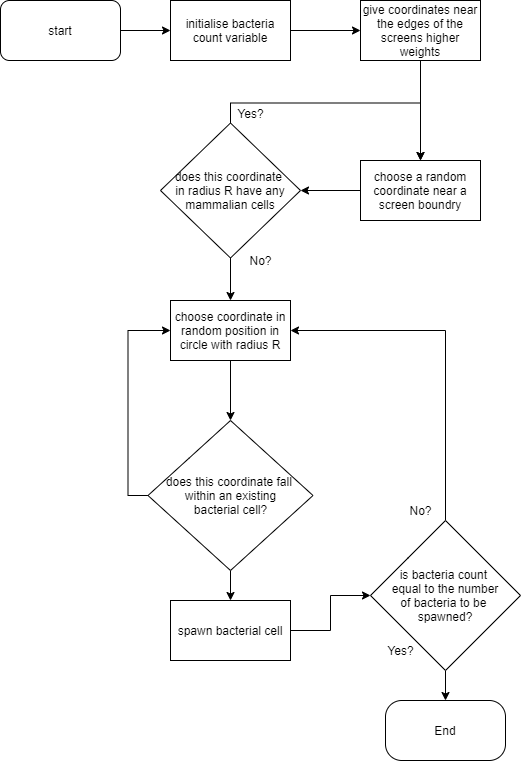
RandomX = random int <= spawnRadius

RandomY = random int <= spawnRadius

spawnPoint = new vector(randomX, randomY)

newCell = new MammalianCell(spawnoint)

## Flowchart for spawning bacterial cells



## Pseudocode for spawning bacterial cells

spawningBCells()

NewCell = null

BacterialCellCount = input(“number of bacterial cells”)

SpawnRadius = input(“spawn radius”)

SpawnPoint = new vector(0, 0)

For (int I = 0; I <= BacterialCellCount)

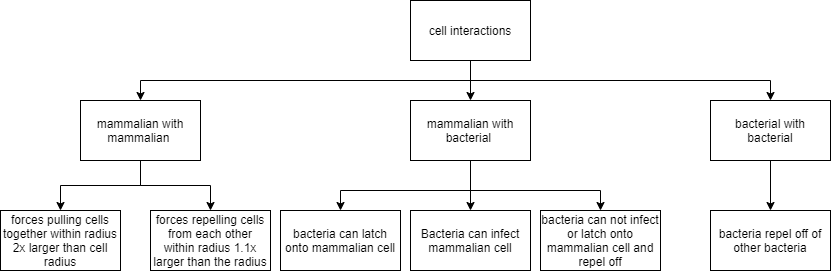
RandomX = random int <= spawnRadius

RandomY = random int <= spawnRadius

spawnPoint = new vector(randomX, randomY)

newCell = new BacterialCell(spawnoint)

## Hierarchy diagram for Cell Interactions



## Pseudocode for Interactions between Mammalian Cells

interCellularForces()

cellApproach = collider.listen()

while simulation = RUNNING

if cellApproach == True

If approachingCell.type == mammalian

approachingCell.addForce(xDirection, yDirection, strength = TBT)

#comment# force applied towards center of cell

distanceBetweenCells = sqrt(differenceInX^2 + differenceInY^2)

if distanceBetweenCells <= (1.1 x cell.radius)

approachingCell.addForce(-xDirection, -yDirection, strength=TBT)

else

pass

else

Pass

## Pseudocode for elastic collisions

ElasticCollisions(body1, body2) #only called if a collision is detected

vectorBetweenBodies = body1.position - body2.position

body1.directionVector = body1.directionVector + vectorBetweenBodies

body2.directionVector = body2.directionVector - vectorBetweenBodies

This resolves the directions bodies should travel after a collision depending on the vector between the bodies

## Critical Path

# Implementation

Checklist

* Rendering mammalian cells {}
* Implementing collision detectors {}
* Implement interaction forces for mammalian cells {}
* Create spawning algorithms {}
* Implement screen boundary restrictions {}
* Rendering bacterial cells {}
* Implement collision between bacterial cells {}
* Implement ability for bacteria to latch onto mammalian cells {}
* Implement bacterial cell duplication {}
* Add user ability to alter bacterial cell attributes {}
* Implement random color for bacteria and mammalian cells
  + Additional possible capabilities
* Display statistics of simulation with graphs and tables
* Capability for mammalian cells to repel/fight back against bacterium
* Development of new strain of bacteria
* Implement more variety in the shapes of the cells to appear more interesting

Troubleshooting

* First attempted to attach spawning script to cell prefab, this caused an infinite chain of cells spawning more cells on instantiation, discovered that script can be attached to the camera
* When running spawning script all cells were spawned directly on top of each other, causing repulsion forces to cancel out, random spawning points were allocated to each cell
* Noticed a large spike in performance drop when many interactions were happening, discovered that I had left a debugging print statement in the collision detection algorithm which was severely decreasing performance
* When implementing bacterial cells, cells were not being rendered to the screen, after troubleshooting it was discovered that the bacterial cell objects were being instantiated with a coordinate with a Z-axis component, making it spawn outside of the camera plane
* When implementing colliders to bacterial cells I noticed that the mammalian cells were attracted to them, the solution to this problem was to add a conditional statement to the mammalian cell attraction algorithm which checks the tag of the object in the collider trigger (with the prefabs of each object I am creating a new tag to distinguish different cell types)
* Increased mass of mammalian cells to reduce bouncing against bacterial cells which is more accurate

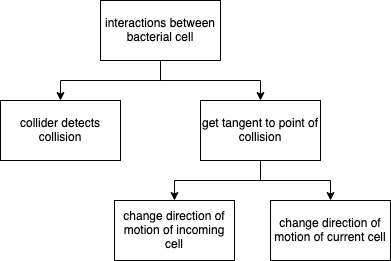
## Overview

After speaking to my client, having implemented the algorithms above, it was brought to my attention that the way in which I had modeled the interactions between mammalian cells was inaccurate to what my client wanted. Instead of forces pulling cells together causing grouping, they should instead bounce off each other and move in random directions, simulating Brownian motion. This type of motion is more accurate to the movement of white blood cells which is what the client wanted them to emulate. The new more accurate algorithms can be found in the algorithms section.

## Systems Flow Diagram

Graphical user interface

Description automatically generated



The algorithms for collisions between mammalian-mammalian and bacterial-bacterial collisions are similar as the algorithm just accesses the rigidbody and collider components of each cell and changes them accordingly

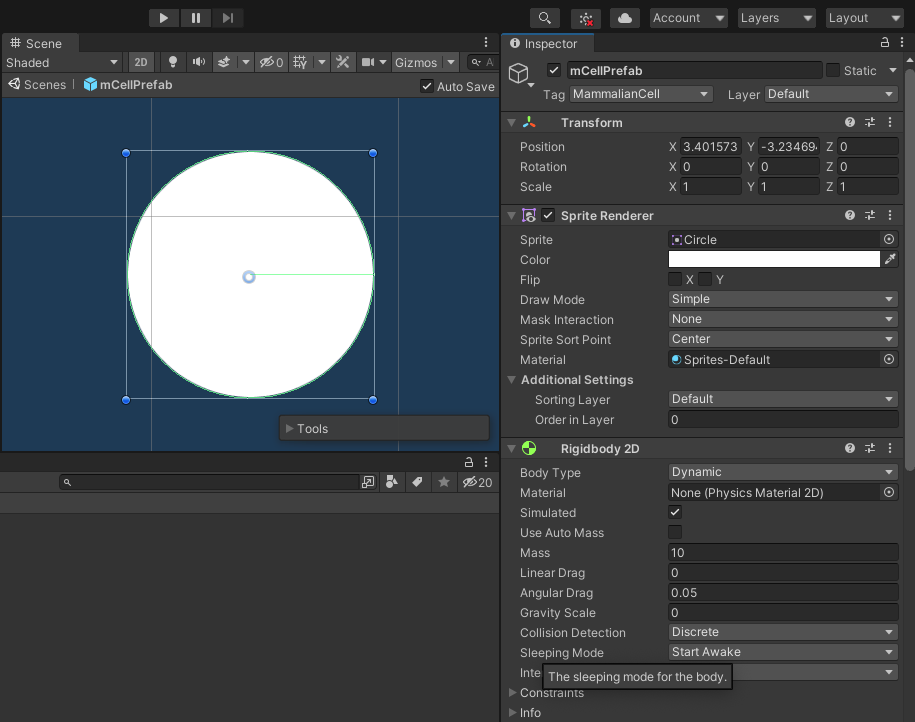
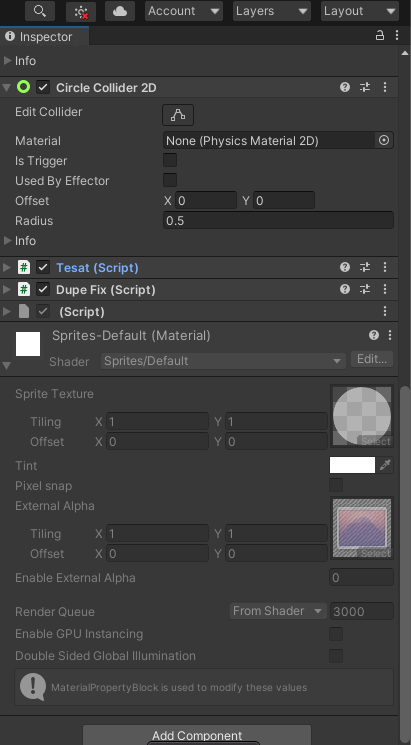
## Algorithms

### Documenting design

To allow for efficient use of time while programming I split the project into smaller chunks to be completed and tested separately. The project is comprised of four smaller chunks.

#### Part 1 – Mammalian Cells

The first part of the program I began working on was creating the mammalian cells and their interactions with each other. First there is the cell itself, I opted to go with a simple circular shape and because white blood cells are being modeled, I kept the white material that they are instantiated with. Then a RigidBody component is attached so that so that the unity engine knows to include it in each physics update, the rigidbody component also gives the object aspects such as mass, velocity, and can make it affected by gravity and drag. To give the cell the ability to sense other cells it encounters, I give each mammalian cell a collider component, this means that if another cell hits the mammalian cell, it can collect data on this cell to respond accordingly. The collider is just slightly larger than the objects themselves as objects would need to overlap otherwise, this way, they get close enough to look as though they are touching without needing to overlap.

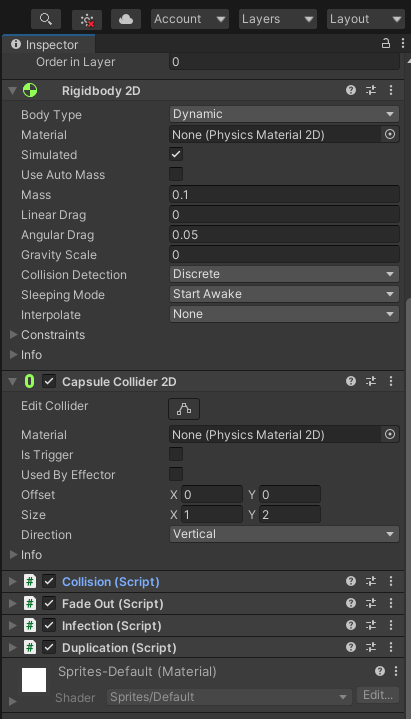
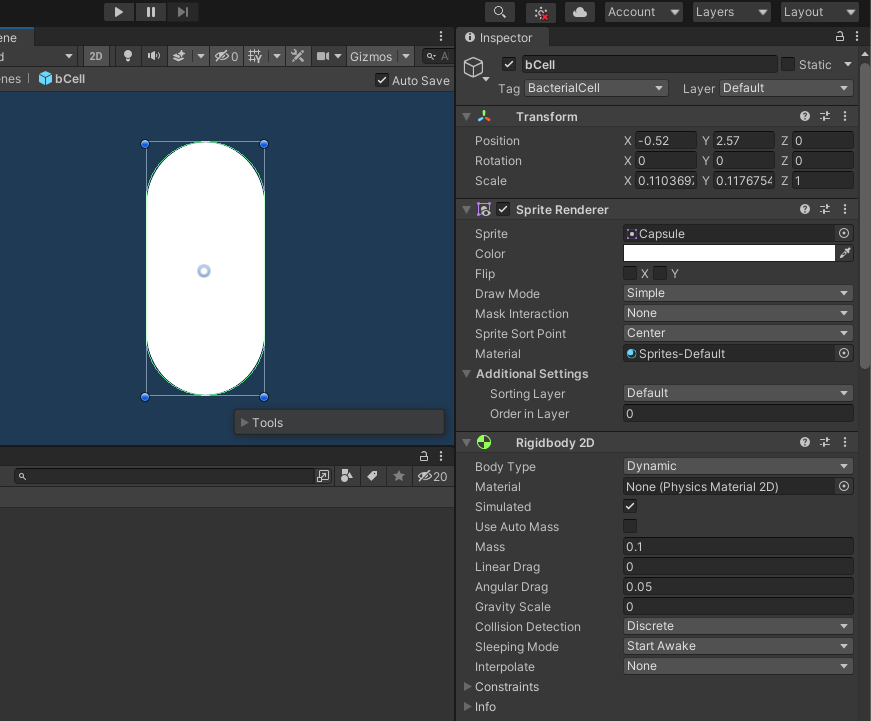
After configuring the cell, I began to work on the correlating scripts. For mammalian cells there are only two scripts, A spawning algorithm, and a collision handling script. While testing the spawning algorithm I tried attaching the script to the mammalian cells themselves, but this led to the problem of infinite loops of instantiating where new mammalian cells would then instantiate new cells and so on and so forth, causing major lag spikes. To avoid this problem, I found that I could attach the script to the camera object and the script would only run once. The rest of the scripts handle mammalian-bacterial interactions and will be discussed in section three.

To ensure that cells knew the type of cell they are interacting with, I created multiple tags, including, MammalianCell, BacterialCell, UnaffectedMammalianCell, and InfectedMammalianCell. These tags are just simple strings that are held by the cell objects, and when a collision is detected a cell can see the tag.

#### Part 2 – Bacterial Cells

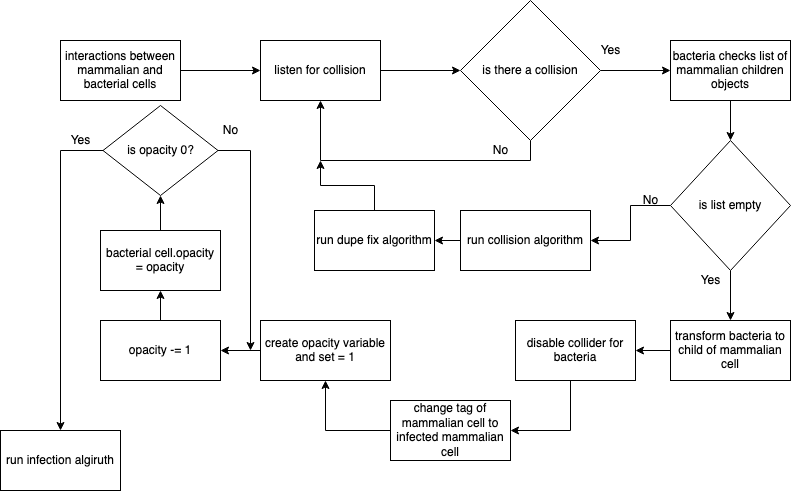
The second part I began focusing on was creating bacterial cells. The cells have similar simple designs using the default capsule sprite to simulate a bacterial cell more accurately. These cells similarly have the same rigidbody and collider components, the only difference is in the scripts that they use.

The scripts solely in charge of bacterial to bacterial interactions are similar to those of the mammalian cell, with a spawning algorithm and a collision handling script. But the bacterial cells also have more functions, one of which is the duplication script. This script is called upon when the cell detects it has infected, and does exactly what the name suggests, handles the duplication of the bacteria upon a successful infection. The rest of the scripts handle mammalian-bacterial interactions and will be discussed in section three.

Part 3 – Bacterial-mammalian interactions / finalizing simulations

This section I began work on the scripts that handled interactions with mammalian cells and bacterial cells. The first of which was alterations with the collision algorithms so that the bacteria would not just bounce off the mammalian cells. After this I began work on the infection script which is attached to the bacterial cell and runs when contact with a mammalian cell is detected. After this script runs there is a chance that the mammalian cell survives the infection and continues into the simulation, but there is also a chance that the mammalian cell dies and disappears from the simulation, upon disappearing, the cell object is destroyed, this chance is determined by the user before the simulation runs. If the infection is successful, the duplication script is run.

As the mammalian cell is infected, to indicate its current state it begins changing colors, if the cell survives, it begins to phase to green, and if the cell dies, it slowly changes to red and then begins to fade away.



This is the variant of the algorithm which goes over the state of events if the infected mammalian cell dies.

During testing of these interactions I found that with specific circumstances there would be a large burst of bacterial cells that spawned, after much troubleshooting I discovered that this happened when a mammalian cell happened to be on top of the spawning point of duplicating bacterial cells. To solve this problem, I created a separate script that detects if this happens. This script is attached to all mammalian cells as these cells have access to all of the data of the attached bacterial cells, more details of this solution can be found in the code section.

It was also found that mammalian cells were rebounding off of bacterial cells in a way that is unrealistic to how they would normally interact, where the mammalian cell would completely rebound after colliding with a smaller bacterial cell. As the mammalian cell is in comparison, much larger than the bacteria, its movement should be much less affected by interactions with bacterial cells. To make this the case I increased the mass of the mammalian cells which made interactions much more accurate.

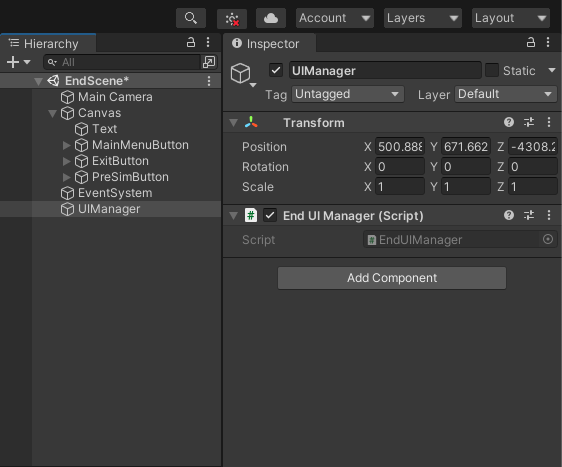
While testing the simulation, it was discovered that sometimes under certain circumstances the collision script would rebound cells at odd and unrealistic angles, after some troubleshooting the solution could not be found and I assumed that it was just a fundamental problem with how I had implemented collision handling, and I hope to fix this problem as an extension task in my free time.

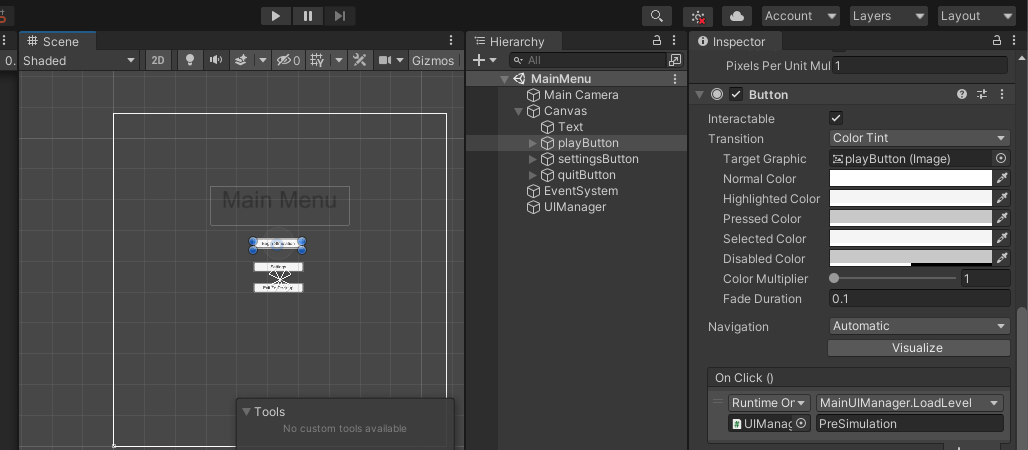
#### Part 4 – UI Management

The whole program is composed of scenes, these scenes are separate and are the spaces each menu takes place in, there is a scene for the main menu, a scene for the pre simulation settings, a scene for the simulation itself, and a scene for the end of the simulation.

Each scene has its own script which handles the functions of the UI elements in it. Scene management scripts themselves are rather simple but to be able to apply the functions of the script it must be attached to a game object in that scene.

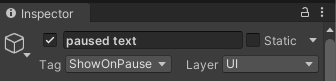
The UI script acts as a library of functions that can be assigned to different actions in the program.

As the script only contains functions, It must be attached to the scenes UI manager object so that the functions may be assigned to different UI elements

If we want to give a specific script to a button, the UI manager object is attached to the button in the onClick() list. Once attached, if the function requires input an input field displayed next to it. In the example the play button on the main menu is being given the UI manager and from this library of functions the load level function is picked, because the function requires a string input, there is an input field next to the field for the UI manager object, containing the text ”PreSimulation”.

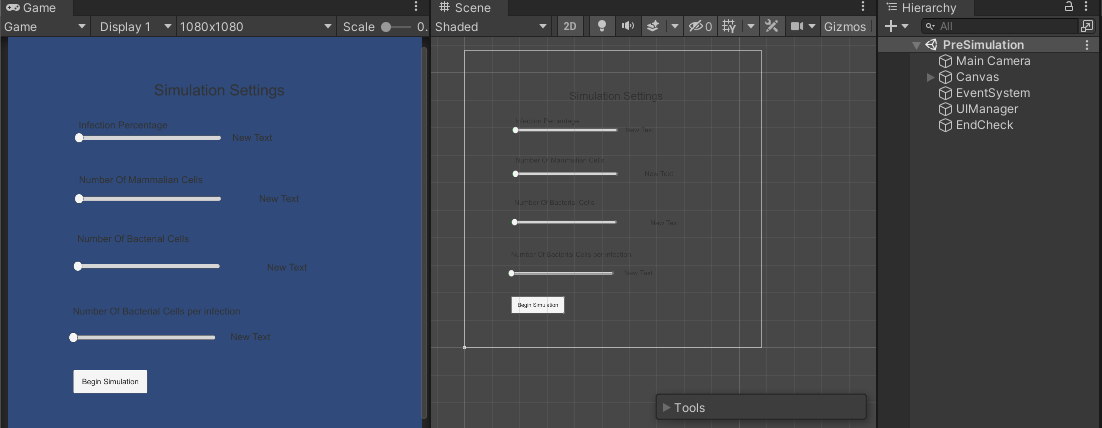
Pause menu

To get the elements of the pause menu to appear on screen only when the game is paused, I gave all of these elements the show on pause tag.

The UI script for the simulation scene then has a list of all of the objects with the tag show on pause. Then the function called showPaused detects if the game is paused and sets the objects in the list to active if the game is paused .

UI elements on the screen must be placed manually in the editor and come with default textures supplied by unity libraries.

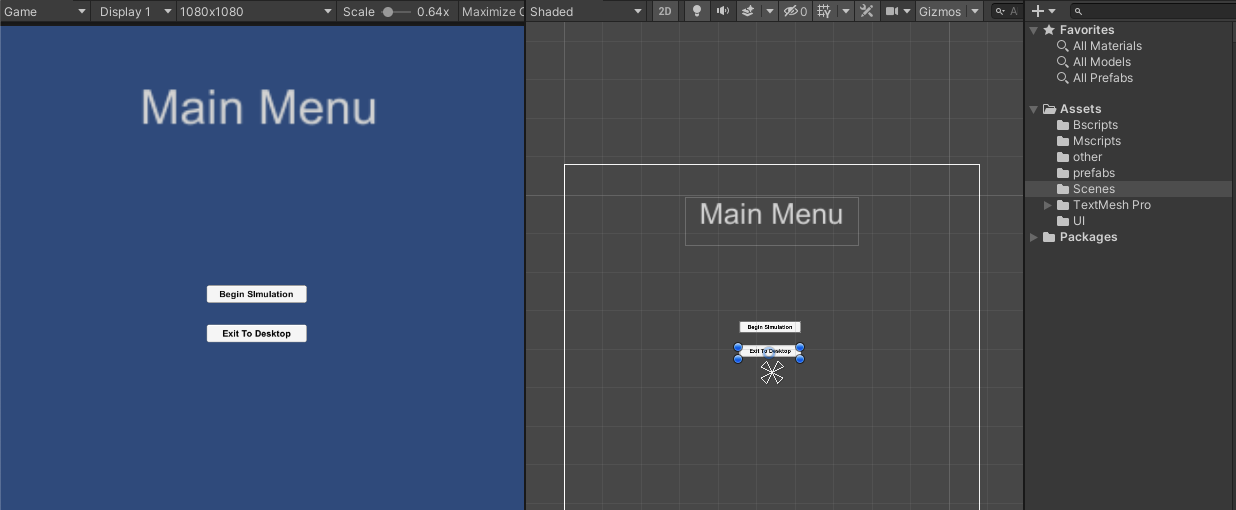
User input

For user input I opted to use simple sliders and buttons to allow for a simple and intuitive way to interact with the program. This also means that there is no chance for the user to input data which is not acceptable removing the need for data validation. The default values and colors for these buttons and sliders can be altered in the unity interface by viewing them in the inspector tab.

Scene 1 – Main menu

The main menu is very simple with just two buttons, being, begin new simulation, and exit to desktop.

To make the text more readable I have made the font size for the main menu text large and white to pop out against the blue background. The text within the buttons themselves has been made bold and a dark shade of black as to be more readable at a distance.

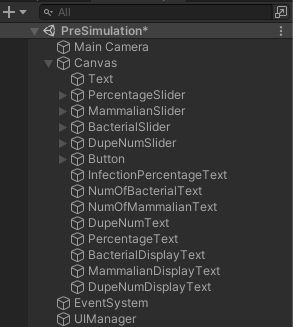


Scene 2 – Pre Simulation

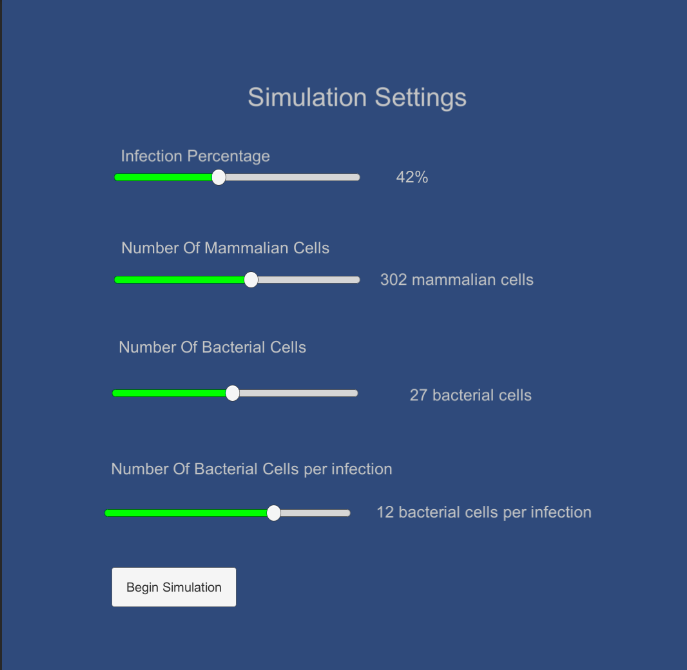
The pre simulation scene is where the user can alter the properties of the bacteria for the simulation. This scene is more complex with four sliders and a button to control the virulence of the bacteria and to begin the simulation.

As the data from the sliders is being altered, I made a script in the UI manager that took the data from the script, rounded it to whatever may be necessary and then output it to a text object next to the slider so that the user can see the value they are choosing.

I chose to make the input fields sliders as this is a simple way of visualizing and seeing what is being input. Each slider has a different lowest value and a different highest value which suits it best, eg infection percentage has a range of values from 0 to 100, whereas number of cells per infection only has a range of 5 to 15. As the sliders for number of mCells, number of bCells, and bCells per infection must be whole numbers, The values are locked to integers and cannot output anything but an integer.

When looking at the Hierarchy tab in the unity editor, it can be seen that each slider has two corresponding text objects, one to display what the slider alters, and one to display the current value stored in the slider.

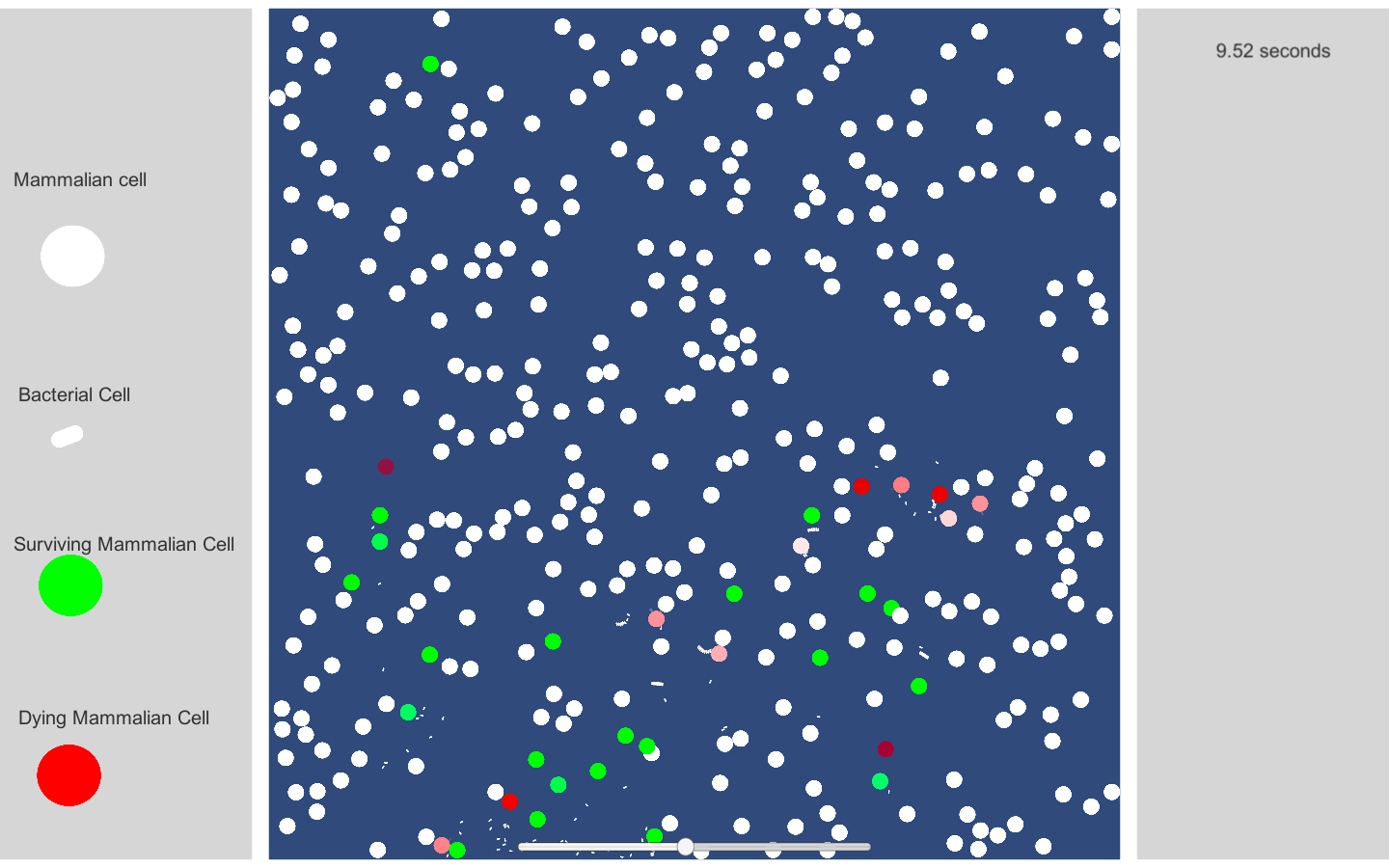
To make the sliders easier to see I added a green backdrop to the sections already covered.

When attempting to create this scene and make it so that the user input data actually affected the simulation, I found that variables could not be transferred across scenes. To fix this problem I came up with a solution which involves making a static variable script which stored all of the variables that needed to be fed into the next scene. By making the variables public static variables. They can be accessed through that script across to other scripts with a simple reference eg: in the staticVar file it may be “public static variableNameHere”, and to access it in a different script “StaticVar.variableNameHere”.

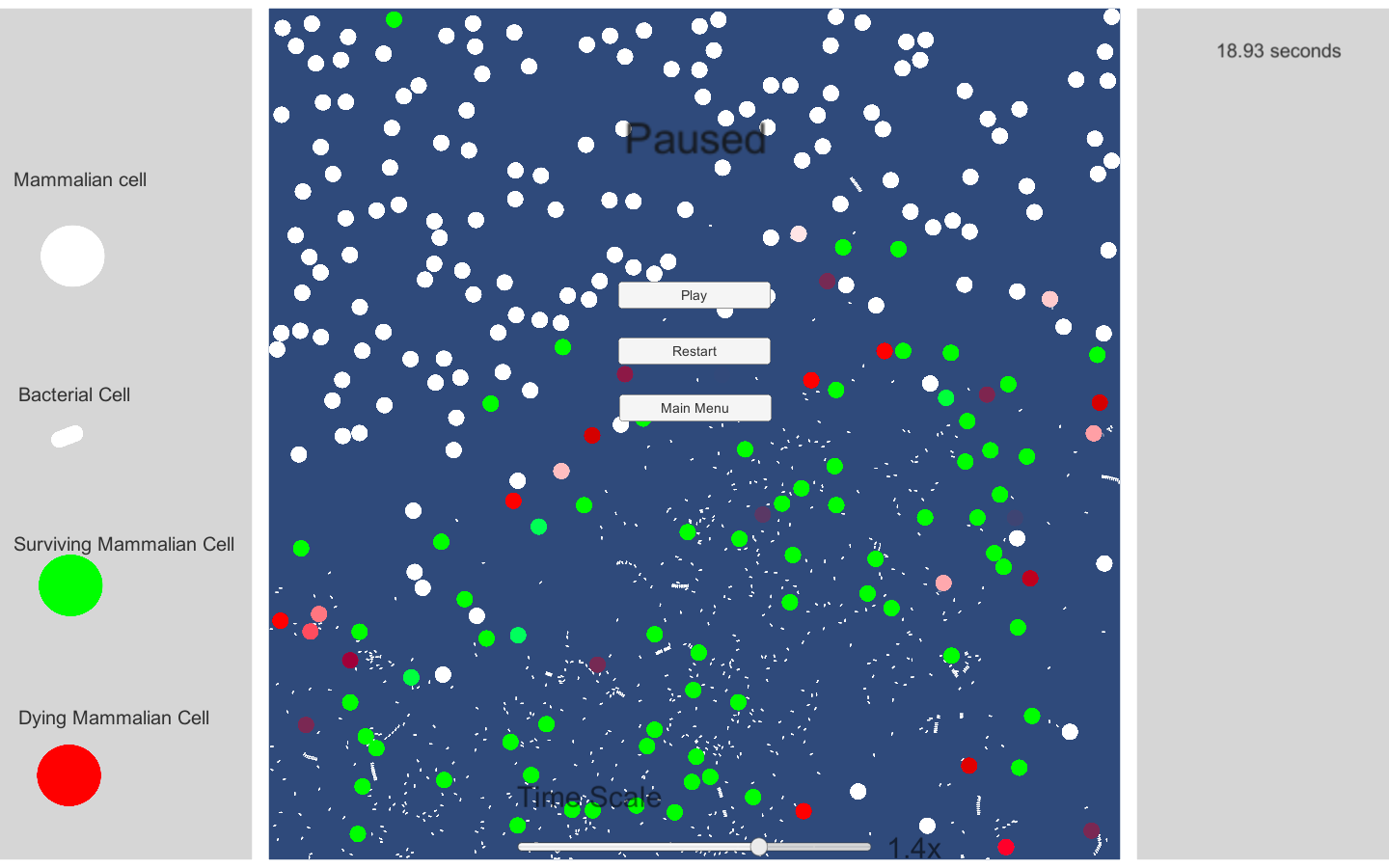
Scene 3 – Simulation

This scene is where the simulation takes place.

Some of the UI elements here only appear when the game is paused. These include the button to resume, which changes the time scale back to one, the new simulation button which clears static variables and brings the user back to the pre simulation scene, and the Exit to desktop button, which closes the program.



The other on-screen elements include text objects next to act as names in the key provided on the left of the screen, detailing each type of cell and their states with a text element to describe it. The other item on the screen is a timer which keeps track of how many seconds have passed since the beginning of the simulation, I achieved this by adding a small statement to the UI managing script for this scene which rounds the deltaTime() and adds it to a time variable giving the time to 2dp.

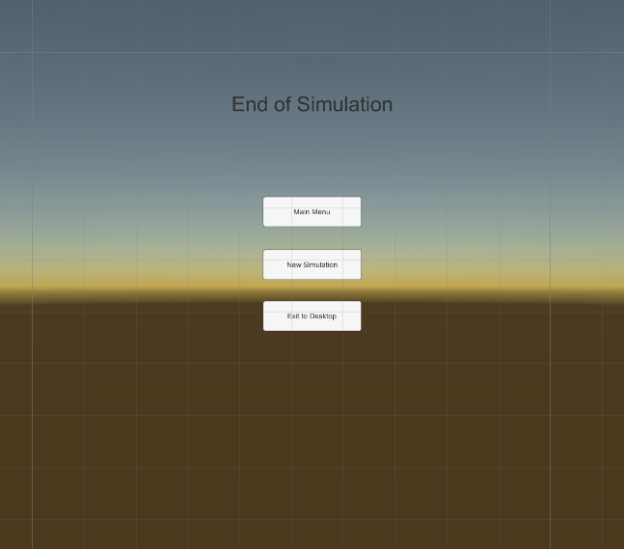


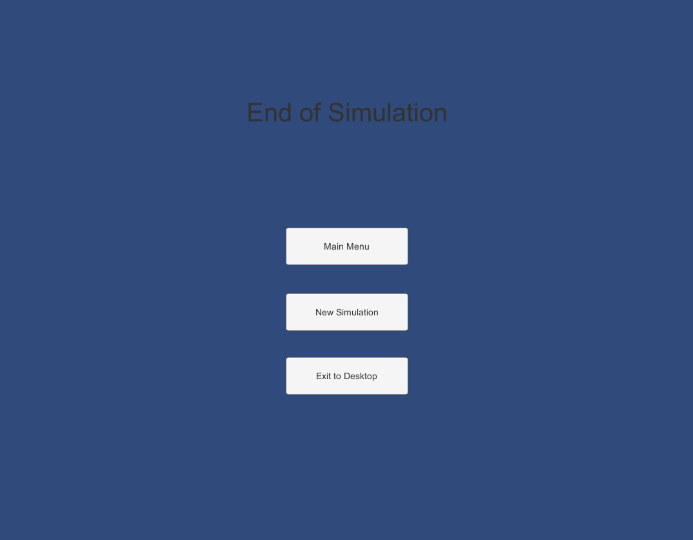
When the game is paused the simulation stops, i.e. the time scale is set to zero. At the bottom of the screen a slider can be seen, this slider is used to adjust the time scale of the simulation, with a range of 0.1x to 2x speed. This is useful, as if the performance begins to dip, the timescale can be dropped to retain the performance.

Scene 4 – End screen

The final scene is very simple and similar to the main menu allowing for traversal to the main menu, to the pre simulation scene, and to quit to the desktop.

This scene only appears after the simulation has reached its end, i.e. either all of the currently alive mammalian cells are infected, by iterating over the list of mammalian cells and checking their tags.





# Testing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test ID | Description | Data or Action | Expected Result | Actual Result | Comments |
| 1 | Main menu |  |  |  |  |
| 1.1 | Begin simulation button. | Transitions to the pre simulation scene. | Changes current scene to the scene with string name “preSimulation”. | As expected. |  |
| 1.2 | Exit to desktop button. | Runs the unity function Application.Quit() | Closes the application. | As expected. |  |
| 2 | Pre simulation Menu . |  |  |  |  |
| 2.1 | Infection percentage slider. |  |  |  |  |
| 2.1.1 | Is lowest slider value valid. | The lowest value should be an integer value of 0. | The value returned should be 0. | As expected. | See screenshot 2.1.1(1)  2.1.1(2) |
| 2.1.2 | Is greatest slider value valid. | The greatest value should be integer value of 100. | The value returned should be 100. | As expected. | See screenshot 2.1.2(1)  2.1.2(2) |
| 2.1.3 | Is the slider value passed to the corresponding text field. | The value from the slider is converted to a string and is passed to the text box present . | The value in the text box changes corresponding to changes in the slider. | As expected. | See screenshot 2.1.3 |
| 2.1.4 | Is the slider value passed to the simulation scene. | The value for the infection percentage is passed to the next scene and dictates the chance of infection. | The bacteria have a chance of infection equal to the value of the slider. | As expected. |  |
| 2.2 | Mammalian cell count slider. |  |  |  |  |
| 2.2.1 | Is lowest slider value valid. | The lowest value should be an integer value of 50. | The value returned should be 50. | As expected. | See screenshot 2.2.1 |
| 2.2.2 | Is greatest slider value valid. | The greatest value should be an integer value of 500. | The value returned should be 500. | As expected. | See screenshot 2.2.2 |
| 2.2.3 | Is slider value passed to the corresponding field. | The value from the slider is converted to a string and is passed to the text box present. | The value in the text box changes corresponding to changes in the slider. | As expected. | See screenshot 2.2.3 |
| 2.2.4 | Is slider value passed to the simulation scene. | The number of mammalian cells must be exactly equal to the number of mammalian cells present. | The value of the slider is passed to the next scene and corresponds to the number of mammalian cells. | As expected. | See screenshots 2.2.4(1)  2.2.4(2) |
| 2.3 | Bacterial cell count slider |  |  |  |  |
| 2.3.1 | Is lowest slider value valid. | The lowest value should be an integer value of 5. | The value returned should be 5. | As expected. | See screenshot 2.3.1 |
| 2.3.2 | Is greatest slider value valid. | The greatest value should be an integer value of 50. | The value returned should be 50. | As expected. | See screenshot 2.3.2 |
| 2.3.3 | Is slider value passed to the corresponding field. | The value from the slider is converted to a string and is passed to the text box present . | The value in the text box changes corresponding to changes in the slider. | As expected. | See screenshot 2.3.3 |
| 2.3.4 | Is slider value passed to the simulation scene. | The number of bacterial cells in the simulation scene must be equal to the number chosen on the slider | The value of the slider is passed to the next scene and corresponds to the number of bacterial cells. | As expected. | See screenshots 2.2.4(1)  2.2.4(2) |
| 2.4 | Number of bacteria per infection slider |  |  |  |  |
| 2.4.1 | Is lowest slider value valid. | The lowest value should be an integer value of 5. | The value returned should be 5. | As expected. | See screenshot 2.4.1 |
| 2.4.2 | Is greatest slider value valid. | The greatest value should be an integer value of 15. | The value returned should be 15. | As expected. | See screenshot 2.4.2 |
| 2.4.3 | Is slider value passed to the corresponding field. | The value from the slider is converted to a string and is passed to the text box present . | The value in the text box changes corresponding to changes in the slider. | As expected. | See screenshot 2.4.3 |
| 2.4.4 | Is the slider value passed to the simulation scene. | The number of bacterial cells per infection must be equal to the slider value. | The number of bacteria released after infection must be the same as the value in the slider. | As expected. | See screenshot 2.4.4(1)  2.4.4(2) |
| 2.5 | Begin simulation button. |  |  |  |  |
| 2.5.1 | Is text in button valid. | The text in the button should be the string “Begin Simulation”. | The button should say begin simulation. | As expected. | See screenshot 2.5.1 |
| 2.5.2 | Button press action. | The button must transition to the scene of name “SimulationScene”. | The button, when pressed moves to the next scene and begins the simulation. | As expected. |  |
| 3 | Simulation Scene. |  |  |  |  |
| 3.1 | Time since simulation began. |  |  |  |  |
| 3.1.1 | Does value change as simulation ensues. | As the simulation plays, every physics step the time between frames should be added to the time value and passed to the text box. | The value in the text box should increase as time progresses. | As expected. |  |
| 3.1.2 | Does the timer stop counting when the simulation is paused. | As the simulation is paused the timer should not be counting up as no physics steps are taking place. | The timer must stop increasing when the simulation is paused. | As expected. | The value stops increasing |
| 3.1.2 | Does the timer counting change when the speed of the simulation is changed. | If the speed at which the simulation is changed the time between frames changes, and therefore the timer counts at a different rate. | If the timescale is increased the timer should count faster, and if the timescale is decreased the timer should count slower. | As expected. | The slider changes at a slower rate with a lower time scale, and at a faster rate with a higher time scale. |
| 3.1.3 | is the timer value rounded to 2dp. | The physics step timer is very accurate to many dp, but only 2dp are required. | Is the timer value rounded to 2dp. | As expected, except for brief frames the rounding goes past 2dp, code needs to be updated. | See screenshot 3.1.3 |
| 3.2 | Pause functionality. |  |  |  |  |
| 3.2.1 | Does the simulation pause when the escape key is pressed | The signal for the game to pause is the key escape, if pressed the timescale is set to 0 | If escape is pressed the game is paused | As expected | When escape is pressed the simulation stops |
| 3.2.2 | Does the pause menu open when the game is paused | If the game is paused the menu items must be displayed | When the game is paused the menu options appear | As expected | See screenshot 3.2.2 |
| 3.2.3 | Pause Menu |  |  |  |  |
| 3.2.3.1 | does the un-pause button un-pause the simulation | When the un-pause button is pressed the simulation resumes and the menu disappears | On being pressed the timescale is returned to 1 and the pause menu is hidden | As expected | The timescale is set back to 1 and the menu is hiddem |
| 3.2.3.2 | Does pressing escape in the pause menu un-pause the simulation | If the simulation is paused and escape is pressed the timescale is set back to 1 and the menu disappears | If the escape key is pressed while the game is paused the game should un-pause and hide the menu | As expected | The timescale is set back to 1 and the menu is hidden |
| 3.2.3.3 | Does pressing the restart button reset the simulation | Upon pressing the reset button the simulation scene is reloaded and should restart with the same start variables | The simulation is reset with the same preSimulation settings | As expected | The simulation is cleared and is re-run with different random spawning coordinates |
| 3.2.3.4 | Does pressing the main menu button return to menu | When the main menu button is pressed the simulation should end and the scene is changed to the “mainMenuScene” | Upon pressing, the user should be take to the main menu | As expected | The user is moved to the main menu |
| 3.2.3.5 | Does an altered timescale affect the pause menu | If the timescale of the simulation is different upon pausing, the pause menu overrides that and sets the timescale to 0 | if the timescale is different when pausing, it is set to 0 | As expected |  |
| 3.3 | Time scale slider |  |  |  |  |
| 3.3.1 | Slider itself works in the same capacity as the sliders in the pre simulation |  |  |  |  |
| 3.3.2 | Is the slider value applied to the timescale | The time scale must be set to the value of the slider | If the slider is at 2, the timescale is doubled | As expected | The speed of the simulation changes in accord with the slider |
| 3.3.3 | Is the lowest value valid | The lowest value of the slider is the float 0.1 as if it reached 0 it would pause | The minimum value of the slider is 0.1 | As expected | See screenshot 3.3.3 |
| 3.3.4 | Is the greatest value valid | The greatest value of the slider should be the float 2.0 | The maximum timescale is 2x the normal simulation speed | As expected | See screenshot 3.3.4 |
| 3.3.5 | Is the slider value rounded to 2dp | The slider value should be rounded to a 2 decimal place float for simplicity | The slider value is converted to a 2dp float before being applied to the timescale | As expected |  |
| 3.3.6 | Is the timescale slider functioning in the pause menu | The slider should be changeable in the pause menu and the text box should change according to its value | The timescale should not change but the slider and the corresponding text should be changeable in the pause menu | The slider can be changed but the text does not |  |
| 3.3.7 | Does the change in slider change the corresponding text | The rounded value of the slider is passed to the text box | The text box displays a rounded value corresponding to the timescale | As expected |  |
| 3.4 | Cell interactions |  |  |  |  |
| 3.4.1 | Unaffected mammalian cells with mammalian cells | When two mammalian cells interact, the cells should rebound off of each other | They should bounce off of each other retaining the same velocity | As expected |  |
| 3.4.2 | Bacterial with bacterial cells | When two bacterial cells interact, the cells should rebound off of each other | They should bounce off of each other retaining the same velocity | As expected |  |
| 3.4.3 | Bacterial cell with unaffected mammalian cell | The mammalian cells velocity remains unchanged | The bacterial cell latches onto the mammalian cell | As expected |  |
| 3.4.3.1 | Bacterial cell begins to disappear | The opacity of the bacterial cell begins to decrease | The bacterial cell begins to disappear | As expected | See screenshot 3.4.3.1 |
| 3.4.3.2 | Does the mammalian cell begin to change color when infected | If the infection is successful the mammalian cell begins transitioning to red | When the cell is infected the cell turns red | As expected | Also shown in screenshot 3.4.3.1 |
| 3.4.3.2.1 | Does the mammalian cells opacity change when infection is successful | The opacity of the mammalian cell should slowly decrease until it cannot be seen | The mammalian cell slowly disappears | As expected | See screenshot 3.4.3.2.1 |
| 3.4.3.2.2 | Do the bacteria duplicate | the bacteria must duplicate when the cell is fully invisible | N number of bacteria spawn from the mammalian cell | As expected | See screenshot 3.4.3.2.2 (1)  3.4.3.2.2 (2) |
| 3.4.3.3 | Does the mammalian cell change color when infection is unsuccessful | If the infection is unsuccessful then the mammalian cell transitions to green | The cell turns green if it survives the infection | As expected | See screenshot 3.4.3.3 |
| 3.4.4 | Bacterial cell with infected mammalian cell/ unaffected mammalian cell | The bacterial cell should bounce off of the cell with the same velocity it approached with | The bacterial cell bounces off the mammalian cell | As expected |  |
| 3.5 | Simulation end |  |  |  |  |
| 3.5.1 | Does the simulation end when all mammalian cells are dead | If there are no mammalian cells alive for 5 seconds the scene is changed to the endScene scene | If there are no mammalian cells then the simulation ends | As expected |  |
| 3.5.2 | Does the simulation end when if no bacteria are present | If no bacteria are present for 5 seconds then the scene is changed to the endScene scene | If there are no bacterial cells then the simulation ends | As expected |  |
| 4 | End Scene |  |  |  |  |
| 4.1 | Main menu button functionality | Upon being pressed the loadScene function is run and the scene “MainMenu” is loaded | The button takes the user to the main menu | As expected |  |
| 4.2 | Quit to desktop button functionality | When pressed, runs the Application.quit() function closing the program | Quits out of the application | As expected |  |

## Screenshots

2.1.1 (1)

Graphical user interface

Description automatically generated with medium confidence

2.1.1 (2)

A picture containing text, device, screenshot, control panel

Description automatically generated

2.1.2 (1)

A screenshot of a computer

Description automatically generated with medium confidence

2.1.2 (2)



2.2.1 (1)

Background pattern

Description automatically generated with medium confidence

2.2.1 (2)



2.2.2 (1)

A picture containing background pattern

Description automatically generated

2.2.2 (2)



2.2.3

A screenshot of a computer

Description automatically generated with medium confidence

2.2.4 (1)

Graphical user interface, application

Description automatically generated

2.2.4 (2)

A screenshot of a computer

Description automatically generated with low confidence The entity count at the top says that in the highlighted region there are 332 objects, this accounts for the 327 mammalian cells and the 5 bacterial cells from the previous screenshot

A screenshot of a computer

Description automatically generated with low confidence

2.3.1 (1)

A picture containing background pattern

Description automatically generated

2.3.1 (2)



2.3.2 (1)

A picture containing text, device

Description automatically generated

2.3.3

Graphical user interface, icon

Description automatically generated

2.4.1 (1)

Background pattern

Description automatically generated

2.4.1 (2)



2.4.2 (1)

A picture containing shape

Description automatically generated

2.4.2 (2)



2.4.3

A screenshot of a computer

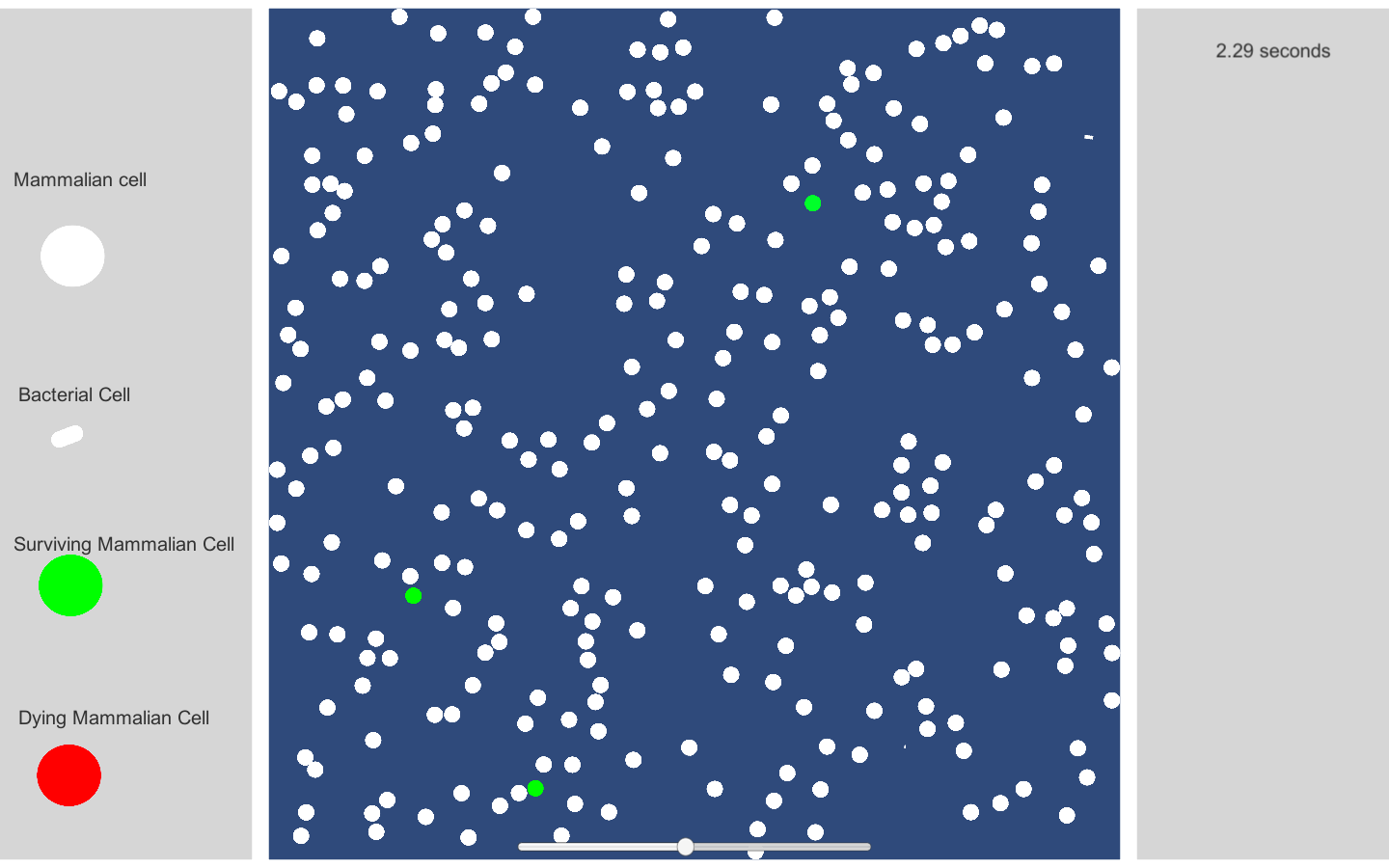
Description automatically generated with medium confidence

2.5.1

Graphical user interface, application

Description automatically generated with medium confidence

3.1.3



3.2.2

Graphical user interface, qr code

Description automatically generated

3.3.3

Background pattern, qr code

Description automatically generated

3.3.4

Background pattern, qr code

Description automatically generated

3.4.3.1

Chart, bubble chart

Description automatically generated

3.4.3.2.1

Chart, bubble chart

Description automatically generated

3.4.3.2.2 (1)

Graphical user interface, timeline

Description automatically generated

3.4.3.2.2 (2)

Graphical user interface

Description automatically generated

3.4.3.3

Chart, bubble chart

Description automatically generated

The bacteria are all stacked on top of each other as they spawn from the same coordinate, from the entity count in the top right it can be seen that there are 11 bacteria in the bundle which is the same as on the slider value in the previous screenshot.

## Code

In unity scripts, the term gameObject is used to refer to objects in the program, on its own, it refers to the object that the script is acting on, ie, gameObject in a bacterial detection script is a bacterial cell. But gameObject can be used to refer to other objects if it is specified, ie, collider.gameObject, this refers to the gameObject attached to a collider which may not be the object the script is attached to.

The Time class in unity handles physics updates and time management, the deltaTime method gets the time between physics updates which can vary depending on the amount of collisions or actions per second whereas the time between frames must remain constant. The timescale attribute is used to set the speed at which the frame is updated.

To instantiate cells with precise configurations prefabs are used, these are used as templates to spawn new cells. The prefab is fed into the script and the script instantiates clones of that prefab rather than having to describe each object in code.

##### M cell spawning script

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using System.Linq;

public class mCellSpawner : MonoBehaviour

{

public GameObject mCellPrefab;

public Vector2 spawnPoint;

private Camera cam;

// List of GameObjects which will contain cells

public static List<GameObject> mCellList = new List<GameObject>();

// Number of cells to be spawned

private int numOfCells;

private float dist;

// Variables

GameObject newCell;

void Start() {

numOfCells = StaticVar.mNum;

cam = FindObjectOfType<Camera>();

float randomXPos;

float randomYPos;

float xV;

float yV;

for (int i = 0; i < numOfCells; ++i) { // repeats for however many cells need to be spawned

randomXPos = Random.Range(-cam.orthographicSize, cam.orthographicSize);

randomYPos = Random.Range(-cam.orthographicSize, cam.orthographicSize);

dist = Mathf.Sqrt(randomXPos \* randomXPos + randomYPos \* randomYPos);

xV = Random.Range(-4, 4);

yV = Random.Range(-4, 4);

spawnPoint = new Vector2(randomXPos, randomYPos);

// uses the random x and y coordinates to generate a spawn vector

newCell = Instantiate(mCellPrefab);

newCell.transform.position = spawnPoint;

var rb = newCell.GetComponent<Rigidbody2D>();

rb.velocity = new Vector2(xV, yV);

mCellList.Add(newCell);

}

}

}

This script is attached to the camera in the scene and spawns the mammalian cells in the map. It is run in the start function meaning that it is only run once when the scene is loaded, if it had been in the update function (ie the game loop) it would be spawning cells every frame.

This script is fed a value called mNum to know how many times to repeat the main for loop. In the for loop it produces a random position within the cameras range and a random vector between the values –4, and 4, for both x and y components of the cells velocity.

The cell is then instantiated, is added to the list of mCells, is moved to the random coordinate, is given a rigidbody component, and then has the random vector applied to its velocity.

##### Mammalian Collision handling script

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class tesat : MonoBehaviour {

Rigidbody2D rb;

Vector2 lastVelocity;

void Start()

{

rb = GetComponent<Rigidbody2D>();

}

// Do this in FixedUpdate, so you're caching the velocity in every

// physics step, rather than only on rendered frames.

void FixedUpdate()

{

lastVelocity = rb.velocity;

}

void OnCollisionEnter2D(Collision2D collision)

{

if (collision.gameObject.tag == "wall" || collision.gameObject.tag == "MammalianCell" || collision.gameObject.tag == "InfectedMammalianCell" || collision.gameObject.tag == "UnaffectedMammalianCell") {

var speed = lastVelocity.magnitude;

var direction = Vector2.Reflect(lastVelocity.normalized, collision.contacts[0].normal);

// .contact.normal draws a line at the normal of the collision

rb.velocity = direction \* speed;

}

}

}

This script runs on every mammalian cell in the scene.

In the start function it gets the data regarding the cell’s rigidbody component so that it can be used to change its velocity.

The fixedUpdate function is a gameLoop that updates every physics update which is always a fixed amount of time, in this loop it is caching the velocity of the cell so that it can be used to get the new direction of movement in a collision.

The OnCollisionEnter2D() function is a listening function which runs if two colliders interact. The parameter passed to the function is the collider of the incoming object, so the object must be accessed through the collider (eg collision.gameObject). The if statement here checks if the object the cell has just collided with is something it will reflect off of, like a wall or another mammalian cell, it does this by checking the objects tag

The speed of the cell should remain unchanged so it is assigned to a temp variable called speed.

To get the new direction of the cell the inbuilt reflect function (which reflects a vector off a normal. The parameters taken by Reflect are the vector which is being reflected (the vector the cell was moving in), and the normal of the collision. The normal of the collision is obtained by accessing the point of contact of the collision and using the normal function which is inbuilt in unity.

The speed and the direction obtained are then multiplied to create the new velocity given to the cell

##### Bacteria cell spawner

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class bCellSpawn : MonoBehaviour

{

public GameObject bCellPrefab;

private Vector2 spawnPoint;

private Camera cam;

// Number of cells to be spawned

private int numOfBCells;

// Variables

GameObject newCell;

void Start() {

numOfBCells = StaticVar.bNum;

cam = FindObjectOfType<Camera>();

float randomXPos;

float randomYPos;

float xV;

float yV;

for (int i = 0; i < numOfBCells; ++i) { // repeats for however many cells need to be spawned

randomXPos = Random.Range(-cam.orthographicSize, cam.orthographicSize);

randomYPos = Random.Range(-cam.orthographicSize, cam.orthographicSize);

xV = Random.Range(-5, 5);

yV = Random.Range(-5, 5);

spawnPoint = new Vector2(randomXPos, randomYPos); // uses the random x and y coordinates to generate a spawn vector

newCell = Instantiate(bCellPrefab);

newCell.transform.position = spawnPoint;

var rb = newCell.GetComponent<Rigidbody2D>();

rb.velocity = new Vector2(xV, yV);

}

}

}

This script works extremely similar to the mammalian spawning script just using a different count variable and instantiating from a different prefab

##### Collision Handling script for bacteria

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class collision : MonoBehaviour {

public float DropSpeed;

Rigidbody2D rb;

Vector2 lastVelocity;

void Start()

{

rb = GetComponent<Rigidbody2D>();

}

void FixedUpdate()

{

lastVelocity = rb.velocity;

}

void OnCollisionEnter2D(Collision2D collision)

{

var randomNum = Random.Range(0, 1);

if (collision.gameObject.tag == "wall" || collision.gameObject.tag == "BacterialCell" || collision.gameObject.tag == "InfectedMammalianCell" || collision.gameObject.tag == "UnaffectedMammalianCell") {

var speed = lastVelocity.magnitude;

var direction = Vector2.Reflect(lastVelocity.normalized, collision.contacts[0].normal); // .contact.normal draws a line at the normal of the collision

rb.velocity = direction \* speed

}

if (collision.gameObject.tag == "MammalianCell")

{

rb.isKinematic = true;

gameObject.transform.SetParent(collision.gameObject.transform);

gameObject.GetComponent<Collider2D>().enabled = false;

}

}

}

The beginning of this script is almost exactly the same as the collision handling for mammalian cells, in the if statement it checks if the object it is colliding with is a wall, another bacterial cell, or an unaffected mammalian cell, in which case it is reflected using the same algorithm used to get the reflection for mammalian cells.

If the tag shows that the incoming object is a mammalian cell it runs a different algorithm which is what begins the infection process.

Once the bacterial cell comes into contact with the mammalian cell it is removed from the physics simulation by setting the isKinematic of its rigidbody component to false, after this the colliding mammalian cell is assigned to be the parent of the bacterial cell, meaning that its velocity is shared with the bacteria, in effect making it stick to the mammalian cell.

After this the collider component is disabled so that the bacterial cell can no longer interact with other cells in any way.

##### Infection script

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class infection : MonoBehaviour

{

private float redCount;

private float greenCount;

private float deathCount;

private float fadeCount;

private int infectionChance;

private int random;

private bool haveAdded;

// Start is called before the first frame update

void Start()

{

infectionChance = StaticVar.infectionPercentage;

redCount = 1;

greenCount = 1;

deathCount = 2;

fadeCount = 1;

}

void Awake()

{

random = Random.Range(1, 100);

}

// Update is called once per frame

void Update()

{

if (gameObject.GetComponent<Collider2D>().enabled == false)

{

if (random < infectionChance)

{

GameObject mammalianCell = gameObject.transform.parent.gameObject;

mammalianCell.tag = "InfectedMammalianCell";

mammalianCell.GetComponent<SpriteRenderer>().color = new Color(1, redCount, redCount, 1);

redCount -= Time.deltaTime;

if (redCount < 0)

{

redCount = 0;

}

// killing the Mammalian cell

deathCount -= Time.deltaTime;

if (redCount == 0)

{

mammalianCell.GetComponent<SpriteRenderer>().color = new Color(1, redCount, redCount, fadeCount);

fadeCount -= Time.deltaTime;

if (fadeCount < 0)

{

mammalianCell.GetComponent<Rigidbody2D>().simulated = false;

mammalianCell.GetComponent<Collider2D>().enabled = false;

}

}

}

else if (random > infectionChance)

{

GameObject mammalianCell = gameObject.transform.parent.gameObject;

mammalianCell.tag = "UnaffectedMammalianCell";

mammalianCell.GetComponent<SpriteRenderer>().color = new Color(greenCount, 1, greenCount, 1);

greenCount -= Time.deltaTime;

if (greenCount < 0)

{

greenCount = 0;

}

}

}

}

}

This script handles the infection and the outcomes of infection eg; color changes and fading.

The script is attached to all bacterial cells, the infection chance is obtained from the staticVar script nd all instantiated variables are given values when the scene is loaded.

When the script is run, a random number is generated, this random number is used to determine the chance of a cell surviving infection or dying.

The way the script knows to run is by checking the state of the bacterial cell. If the collider of the bacterial cell is disabled the script knows it has infected a mammalian cell.

After it detects it has infected a cell it then decides if it will survive or if it will die. It uses the random number generated and if it is greater than the infection percentage it survives, if the random number is less than the infection percentage the mammalian cell dies.

If the mammalian cell dies, new temp gameObject is created and the mammalian cell is assigned to this for easy access, rather than refering to it by (gameObject.transform.parent.gameObject) I can refer to it by mammalianCell which is much easier to read.

(in unity gameObject is a way to refer to the object the script is acting upon, but also can be used to specify other gameobjects when it is specified ie, gameObject.transform.parent.gameObject is accessing the object that is the parent of the current gameobject the script is acting upon)

I then change the tag of the mammalian cell so that it cannot be infected by more bacterial cells.

After this I begin changing the mammalian cells color by accessing its sprite renderer, which allows me to alter color and transparency. The variable redCount is slowly decreased by subracting the deltaTime from it each frame and assigning all color components but red to this value, resulting in a slow shift to red as all but the red component move towards zero. The time taken to shift to red can be increased by increasing the redCount variables value.

A small if statement then prevents redCount from going into the negatives.

After this, the same process is used to slowly decrease the opacity of the mammalian cell so that it fades away.

Once it is completely transparent it is removed from the simulation so that the physics engine does not need to render it, saving a small amount of performance.

If the mammalian cell survives, it begins to fade to green implying that it is alive still, it does this using the same process described, but instead of altering all but the red component of the color, it alters all but green.

The tag is then changed to unaffectedMammalianCell so that other bacteria know to reflect off of it rather than try and infect it.

##### Bacteria Fade out script

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class fadeOut : MonoBehaviour

{

private float count;

void Start()

{

count = 1;

}

void Update()

{

if (gameObject.GetComponent<Collider2D>().enabled == false)

{

gameObject.GetComponent<SpriteRenderer>().color = new Color(1, 1, 1, count);

count -= (Time.deltaTime \* 2);

}

}

}

This script is extremely similar to the code used to make the mammalian cells disappear above, except that it first checks if the bacterial cell still has an active collider or not, signifying when it has infected a mammalian cell. And then it multiplies the deltaTime value so that the bacterial cell disappears faster.

##### Duplication script

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class duplication : MonoBehaviour

{

public GameObject bCellPrefab;

private Vector2 refVector;

private GameObject newCell;

public bool haveSpawned;

// Start is called before the first frame update

void Start()

{

haveSpawned = false;

}

// Update is called once per frame

// if a mammalian cell instantaneously comes into contact with multiple bacterial cells then they all get turned into children of the mammalian cell

//causing many bacteria to be spawned from the same co-ordinates which causes a lag spike and is not accurate to the model

void Update()

{

if (haveSpawned == false) {

if (gameObject.GetComponent<Collider2D>().enabled == false) {

GameObject mammalianCell = gameObject.transform.parent.gameObject;

refVector = mammalianCell.GetComponent<Rigidbody2D>().velocity;

int infectionCount = mammalianCell.transform.childCount;

if (infectionCount == 1) {

if (mammalianCell.GetComponent<Rigidbody2D>().simulated == false && mammalianCell.tag != "UnaffectedMammalianCell") {

mammalianCell.GetComponent<Collider2D>().enabled = false;

for (int i = 0; i < StaticVar.dupeNum; ++i) {

newCell = Instantiate(bCellPrefab);

newCell.transform.position = mammalianCell.transform.position;

var rb = newCell.GetComponent<Rigidbody2D>();

newCell.GetComponent<SpriteRenderer>().color = new Color(1, 1, 1, 1);

newCell.GetComponent<Collider2D>().enabled = true;

rb.velocity = refVector;

rb.isKinematic = false;

}

haveSpawned = true;

}

}

}

}

}

}

This script handles the duplication of bacterial cells after a mammalian cell has been infected and died.

First, to prevent the cell from infinitely duplicating, I create a new boolean variable called has spawned which is set to false when the scene is loaded.

In the update function checks if the cell has already spawned cells, if not it continues. After this it gathers information on the bacterial cell’s parent mammalian cell. First creating a gameObject off of the reference gameObject.transform.parent.gameObject to increase readability. Then creating a temp vector to hold the velocity of the mammalian cell as this will later be applied to the duplicated bacteria.

Then, the number of children which the mammalian cell has is checked by accessing a list of children objects, as if it is greater than one, that means that it has been infected with multiple bacteria which causes a massive influx of bacterial cells and performance issues.

If the mammalian cell has only one child then, the script checks if the mammalian cell is being simulated, as if it is not being simulated that means it has died, and the bacteria can be spawned. It also checks if the mammalian cells tag is not unaffected mammalian cell (this is to solve a bug I have, where a surviving cell is removed from the physics simulation and the reason is not yet apparent, without this conditional statement a surviving cell may infinitely spawn bacteria).

If it passes all of the conditional statements that means that the cell will duplicate. Then it uses a variable called dupeNum from the staticVar script to know how many times to go through the for loop.

The instantiating process is similar to the spawning algorithm, instantiating new cells with each iteration, mapping their position to the vector position of the mammalian cell, giving the new cell a rigidbody component and a collider2D component, passing the color white to the cells sprite renderer, applying the reference vector to the new cells velocity, and finally, setting the isKinematic setting in the new cells rigidbody component to false to ensure it is included in the unity physics updates.

After all of this the boolean hasSpawned is set to true so that it only runs once

##### Duplication bug solution

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class dupeFix : MonoBehaviour

{

private bool hasSpawned;

private bool hasSurvived;

private int dupeNum;

private GameObject newCell;

public GameObject bCellPrefab;

// Start is called before the first frame update

void Start()

{

hasSpawned = false;

dupeNum = StaticVar.dupeNum;

}

// Update is called once per frame

void Update()

{

var childCount = gameObject.transform.childCount;

var refVector = gameObject.GetComponent<Rigidbody2D>().velocity;

if (gameObject.GetComponent<Rigidbody2D>().simulated == false) {

if (childCount > 1 && hasSpawned == false) {

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

newCell = Instantiate(bCellPrefab);

newCell.transform.position = gameObject.transform.position;

var rb = newCell.GetComponent<Rigidbody2D>();

newCell.GetComponent<SpriteRenderer>().color = new Color(1, 1, 1, 1);

newCell.GetComponent<Collider2D>().enabled = true;

rb.velocity = refVector;

rb.isKinematic = false;

}

hasSpawned = true;

}

}

if (hasSpawned == false && gameObject.transform.childCount >= 1) {

StaticVar.infectionCount += 1;

hasSpawned = true;

}

}

}

This script is run when a mammalian cell has more than one child object.

Similarly it has a boolean variable hasSpawned which is set to false on instantiation in the start function.

In the update function is similar to that of the duplication script above except that it only runs If the number of children of the mammalian cell is greater than one.

##### End scene Transition

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

public class EndGame : MonoBehaviour

{

private Scene currentScene;

private int infectedCellCount;

private bool gameEnd;

private string scene;

private float count;

// Start is called before the first frame update

void Start()

{

StaticVar.infectionCount = 0;

currentScene = SceneManager.GetActiveScene();

infectedCellCount = 0;

scene = currentScene.name;

infectedCellCount = StaticVar.infectionCount;

count = 4f;

}

// Update is called once per frame

void Update()

{

infectedCellCount = StaticVar.infectionCount;

if (infectedCellCount == StaticVar.mNum) {

count -= Time.deltaTime;

if (count < 0) {

Application.LoadLevel("EndScene");

}

}

}

}

This script runs in the background while the simulation is taking place.

When the scene is first loaded the variable infection count is set to 0, whenever a mammalian cell is infected, the infectionCount it is increased by one.

In the update loop, the conditional checks if the number of infected mammalian cells is equal to the number of mammalian cells in total.

If it is, a counter is slowly decreased until it reaches zero at which point the end scene is loaded. The time taken to load the scene can be increased by increasing the value of the count.

##### Static variable script

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

using UnityEngine.SceneManagement;

public class StaticVar : MonoBehaviour

{

public static int infectionPercentage;

public static int bNum;

public static int mNum;

public static int dupeNum;

public static int infectionCount;

public Slider \_percentageInput;

public Slider \_mInput;

public Slider \_bInput;

public Slider \_dupeNumInput;

// Update is called once per frame

void Update()

{

bNum = (int)\_bInput.value;

mNum = (int)\_mInput.value;

infectionPercentage = (int)\_percentageInput.value;

dupeNum = (int)\_dupeNumInput.value;

}

}

This script is used to store variables between scenes, and as the variables instantiated in it are static an public, they can be accessed by any other script.

The update loop is used to feed the values from the sliders into the values stored in the script using the inbuilt .value function. For this to work the script needs to have access to the slider objects, this is done in the unity editor by dragging the slider objects into their respective input fields.

##### Wall spawning algorithm

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class wallSpawn : MonoBehaviour

{

public GameObject sideWallPrefab;

public GameObject bottomWallPrefab;

public Camera cam;

public Vector2 spawnPoint;

GameObject newWall;

void Start()

{

cam = FindObjectOfType<Camera>();

// top wall

newWall = Instantiate(bottomWallPrefab);

spawnPoint = (new Vector2(0, (1/2 + cam.orthographicSize)));

newWall.transform.position = spawnPoint;

newWall.tag = "wall";

// bottom wall

newWall = Instantiate(bottomWallPrefab);

spawnPoint = (new Vector2(0, (-1/2-cam.orthographicSize)));

newWall.transform.position = spawnPoint;

newWall.tag = "wall";

// right wall

newWall = Instantiate(sideWallPrefab);

spawnPoint = (new Vector2((1 / 2 + cam.orthographicSize), 0));

newWall.transform.position = spawnPoint;

newWall.tag = "wall";

// left wall

newWall = Instantiate(sideWallPrefab);

spawnPoint = (new Vector2((-1 / 2 - cam.orthographicSize), 0));

newWall.transform.position = spawnPoint;

newWall.tag = "wall";

}

}

This algorithm spawns all four walls of the map from their corresponding prefabs these have to be fed into the script.

This script only has to be run once as the walls do not change in the simulation.

First it gets the object of type camera so that it can access the camera size

There are two prefabs available for the script to use to instantiate the walls, one called sideWallPrefab which is vertical, and one called bottomWallPrefab which is horizontal. This could have been done with one prefab but I would have needed to have access to change the rotation of the prefab which I had difficulty accessing.

Each chunk is the same except for the spawning co-ordinates applied to the object, I use the size of the camera to determine the spawning point of the walls so that they can work with any screen size. I include the + or – ½ so that the walls spawn half way off the screen

I give each wall segment the tag “wall” so that the cells know to reflect off of it.

#### UI handling scripts

##### main menu UI script

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class MainUIManager : MonoBehaviour

{

public void LoadLevel(string level) {

Application.LoadLevel(level);

}

public void Quit() {

Application.Quit();

}

}

As is previously mentioned each function is to be applied to a different action.

The loadlevel function is what loads the next preSimulation scene and the string “preSimulation" is fed to this function in the unity editor.

The quit function closes the program

##### Pre simulation UI script

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class PreUIManager : MonoBehaviour

{

public Slider percentageSlider;

public Slider mammalianSlider;

public Slider bacterialSlider;

public Slider dupeNumSlider;

public Text percentageSlideText;

public Text mammalianText;

public Text bacterialText;

public Text dupeNumText;

// Update is called once per frame

void Update()

{

percentageSlideText.text = "" + percentageSlider.value + "%";

mammalianText.text = "" + mammalianSlider.value + " mammalian cells";

bacterialText.text = "" + bacterialSlider.value + " bacterial cells";

dupeNumText.text = "" + dupeNumSlider.value + " bacterial cells per infection";

}

public void LoadLevel(string level) {

Application.LoadLevel(level);

}

}

This script takes input from objects in the scene and these must be fed into the script through the scenes UI manager object.

In the update function, all that is being done is changing the text of the text objects percentageSlideText, mammalianText, bacterialText, and dupeNumText to their corresponding sliders values. As the values of these sliders are by default floats I have added an empty string before concatenating the slider values as this Is a simple way to convert floats to strings.

The loadLevel function is used by the button in the scene and is fed the string “SimulationScene” to transition to the next scene

##### Simulation UI script

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class UIManager : MonoBehaviour

{

// Start is called before the first frame update

GameObject[] pauseObjects;

public Slider timeSlider;

public Text timeText;

public Text totalTime;

public float time;

void Start()

{

Time.timeScale = 1;

pauseObjects = GameObject.FindGameObjectsWithTag("ShowOnPause");

hidePaused();

time = 0f;

}

// Update is called once per frame

void Update()

{

if (Input.GetKeyDown("escape"))

{

if (Time.timeScale > 0)

{

Time.timeScale = 0;

showPaused();

}

else if (Time.timeScale == 0)

{

Time.timeScale = 1;

hidePaused();

}

}

if (Time.timeScale != 0) {

var roundedSliderValue = Mathf.Round(timeSlider.value \* 10.0f) \* 0.1f;

timeText.text = "" + roundedSliderValue + "x";

Time.timeScale = roundedSliderValue;

}

time += Time.deltaTime;

totalTime.text = "" + (Mathf.Round(time \* 100.0f) \* 0.01f) + " seconds";

}

public void Reload()

{

Application.LoadLevel(Application.loadedLevel);

}

public void pauseControl()

{

if (Time.timeScale == 1)

{

Time.timeScale = 0;

showPaused();

}

else if (Time.timeScale == 0)

{

Time.timeScale = 1;

hidePaused();

}

}

//shows objects with ShowOnPause tag

public void showPaused()

{

foreach (GameObject g in pauseObjects)

{

g.SetActive(true);

}

}

//hides objects with ShowOnPause tag

public void hidePaused()

{

foreach (GameObject g in pauseObjects)

{

g.SetActive(false);

}

}

//loads inputted level

public void LoadLevel(string level)

{

Application.LoadLevel(level);

}

}

This script also takes input from a slider in the scene, but more importantly it handles paused objects.

All object with the showOnPause tag are added to a list called pauseObjects.

The function update listens for a button click of the escape key. If this is detected, the time scale is set to zero and the objects in the pauseObjects list are set to active. If the escape key is detected again, the time scale is set back to one and the pauseObjects are disabled again, unpausing the game.

the next if statement is what handles the timeScale slider at the bottom of the screen. The value of the slider is rounded to 1 decimal place and the time scale is set to that rounded value. but this can only happen if the game is unpaused, as the time scale must be 0 and nothing else If the game is paused.

The variable time is used to keep track of the number of seconds that have elapsed since the simulation scene was loaded. It is rounded to two decimal places in the same fashion as the rounded slider value

The showPaused and hidePaused functions both loop over the list of paused objects and activate the (showPaused) if the game is paused, or deactivate them (hidePaused) if the game is unpaused

The reload function is used by the reset button and resets the current scene

The pause control function used by the unpause button to resume the game, deactivating the paused objects and setting the time scale back to one (if the slider is not on one after unpausing, the timeScale is set to one for a single frame and then set back to the slider value).

The loadLevel script is exactly the same as before, being fed the string “MainMenuScene”.

##### End Scene UI script

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class EndUIManager : MonoBehaviour

{

// Start is called before the first frame update

void Start()

{

}

// Update is called once per frame

void Update()

{

}

public void loadNext(string level)

{

Application.LoadLevel(level);

}

public void Quit()

{

Application.Quit();

}

}

This script is very similar to the main menu as it only contains a load level function and a quit function.

# Evaluation

## Review of Objectives

## Client Feedback on Client testing

## Evaluation