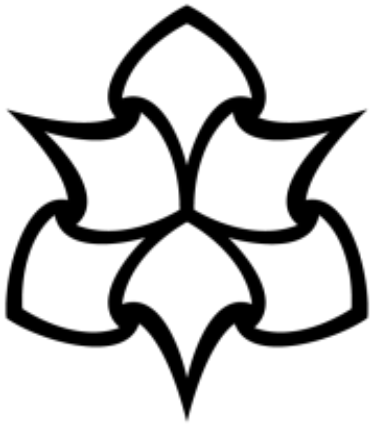
EXPLORING THE VIABILITY OF COMPUTER GAMES AS TOOLS IN INFECTIOUS DISEASE SIMULATION



Submitted to Manchester Metropolitan University for the degree of Bachelor of Science in the Faculty of Science and Engineering

By **James Grey**

School of Computing, Mathematics and Digital Technology

2020

Declaration

No part of this project has been submitted in support of an application for any other degree or qualification at this or any other institute of learning. Apart from those parts of the project containing citations to the work of others, this project is my own unaided work.

This work has been carried out in accordance with the Manchester Metropolitan University research ethics procedures and has received ethical approval number 13254.

Signed

Acknowledgements

I would like to thank my supervisor Dr Matthew Crossley for his support and guidance during this project.

I would also like to thank the participants that helped me with the testing and allowed me to gather data for use in this work.

Abstract

Computer-based disease simulation models are a cornerstone of modern epidemiology. Huge increases in the speed and power of computing, as well as the low cost of hardware, have made it possible to create and process computations that would have been impossible to run until a few years ago.

This project will look at whether computer games can be used to gain meaningful data on the spread of infection and whether iterative changes to a virtual environment and the properties of the playable agents within those environments affect the disease infection rates. By creating a robust infectious disease simulation with agents driven by human players, there is a wide scope for the observation of emergent behaviours based on natural human adaptation.

A playable agent-driven infectious disease simulation was created for this research, with environments that gradually increase in complexity. Using this simulation, data was recorded to allow for the comparison of infection transmission based on the changes to the environment.

Contents

List of Figures

Abbreviations

API: A set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service.

PLUGIN: A software component that adds a specific feature to an existing computer program.

UI: User interface.

STATE MACHINE: A finite-state machine, a device that can be in one of a set number of stable conditions depending on its previous condition and the present values of its inputs.

Section 1 – Introduction

1.1 Background

Computer-based disease simulation models are a cornerstone of modern epidemiology. Huge increases in the speed and power of computing, as well as the low cost of hardware, have made it possible to create and process computations that would have been impossible to run until a few years ago. (Burke, 2003)

The emergence of larger and more complex multiplayer videogames in recent years has provided an unexpected area of study for epidemiologists. In 2005 a virtual epidemic swept through the popular massively multiplayer online role-playing game World of Warcraft that affected more than a million people. This became known as "the corrupted blood incident", when a plague designed to stay in one part of the world was spread via in-game pets and non-playable characters. The virtual infection involved a slow, gradual draining of health. Higher-level players were mainly unaffected. Because of this, higher-level players became carriers, and although these players found the plague to be debilitating, due to it requiring constant healing, among lower-level players, the infection was often fatal (Viani-Walsh, 2016).

This event provided researchers with a model of how networks of people behave during outbreaks of infection and drew into focus how videogames can be used to simulate disease transmission. By having player-controlled agents, human driven emergent behaviours can be observed without dangerous or unethical experimentation.

1.2 Aim

The aim of this project is to create a multiplayer, agent-based, pedestrian borne disease simulation that looks at how changes to the player-controlled agent and the environment affect the rate of epidemic. This will be used to assess the validity of using videogames as a means of obtaining data disease simulation.

1.3 Objectives

The objectives needed to achieve this aim are to:

- Create a multiplayer disease simulation.
- Have testers play the disease simulation.
- Be able to record time between infection events in gradually more complex environments.
- Be able to extract data from the simulation tests.
- Analyse the data gathered to see if it shows any trends.

1.4 Contribution

This research will look at whether computer games can be used to gain useable data for the study of infectious disease by creating a playable simulation. The product will test how iterative changes to a virtual environments' complexity affects the disease infection rates and attempt to prove that game-based tools have potential use in epidemiology.

Section 2 – Literature Review

In order to understand the core elements needed in the creation of the product, research must be completed into computer usage in epidemiology, the links between videogames and epidemiology, and multiplayer game development in Unity engine.

2.1 Computer Usage in Epidemiology

Computers have played a huge and crucial role in transforming public health. Their technological advancement and usage have influenced every single field of medicine. This has enabled powerful tools in disease surveillance to be created, which rapidly transmit and analyse morbidity data within a populace and allow the monitoring of ongoing disease control programs at a national level, with the aim of being able to detect early warning signs of potential outbreaks. (Panth & Acharya, 2015)

Computation epidemiology has capitalized on the power of modern computers and has become a multidisciplinary area that incorporates techniques from epidemiology, molecular biology, applied mathematics, theoretical computer science, and high-performance computing. Synthetic Information environments (SIE) can now be created that take into account a statistical model of a population of interest, an activity-based model of the social contact network, models of disease progression, and models that represent and evaluate interventions, public policies and individual behavioral adaptations. This provides a much more detailed and dynamic way of assessing and predicting disease progression than through a traditional mathematical modeling system. (Marathe & Ramakrishnan, 2013)

Another area of technological progression is Big-Data driven real-time epidemiology. This is a rapidly growing area developing within public health that can utilize new technologies such as search engines and social media to track the progression of diseases. This provides a new and novel system of disease surveillance. An example of this is Google FluTrends, which used search terms

relating to flu in order to track potential progression through the populace. This program was shut down when its accuracy became uncertain, however. (Arthur, 2014)

2.2 Disease Transmission

Despite advancements in modern medicine, humans are still faced with emerging and re-emerging diseases and the challenges of pandemics. Though some of these diseases are human in origin, many originate from animals (zoonoses). Human population increase has led to increased contact between humans and animals (wild, and domestic). This has created an environment where pathogens have a greater chance of transmission. The equation for disease transmission is $\boldsymbol{\beta = \gamma * K}$. It is assumed that $\boldsymbol{\beta}$ is the transmission rate of infection $\boldsymbol{\gamma}$ is the probability of pathogen transmission at contact and **K** is the contact rate. (Craft, 2015)

However, it is hard to fully estimate the transmission rate of infection. The Global Health Security (GHS) index suggests that as low as 19% of countries are able to rapidly detect epidemics that would be of international concern. This low ability to detect infection allows potentially dangerous pathogens to easily travel through human population, as seen with the recent outbreak of COVID19. (Zhongwei & Zuhong, 2020)

There are many ways that pathogens can spread. Two main types of direct contact transmission exist. The most obvious of which is person-to-person contact. This occurs when an infected person touches or exchanges bodily fluids with someone else. Another method of direct contact transmission is droplet spread. This happens mainly when a person coughs or sneezes but can also occur from the droplets when you speak.

Indirect contact transmission happens via other mechanisms. Airborne agents can travel large distances suspended in the air even after an infected person has left the area. Some pathogens can be transmitted via contaminated objects. Other notable methods of transmission include contaminated food and drinking water, animal-to-person and insect bites (for example malaria). (Higuera & Pietrangelo, 2016)

2.3 Epidemiology & Computer Games

Computer games have become a source of interest in epidemiology in recent years. The 'corrupted blood incident' that saw a virtual virus spread throughout the massively multiplayer online game World of Warcraft served to highlight the power of games as tools that could be used to the study of infection.

Though meant as entertainment, games can unintentionally model the spread of disease in a realistic manner. In the case of World of Warcraft, the spread of the corrupted blood virus was dependent on ease of travel, interspecies transmission by player pets and asymptomatic higher-level characters. Though modern computer-based disease simulations have been modelled, they lack the huge number of variables and diversity of unexpected outcomes that are found in real-world diseases and epidemics. Given that player response to danger in games is approximate to real-world reactions, and the number of players in World of Warcraft sitting at around 10 million at the time, the corrupted blood event allowed for "an excellent opportunity to examine the consequences of human interactions within a statistically significant and controlled computer simulation". (Racaniello, 2008)

Pandemics and disease have also been a source of inspiration in games, both thematically and as core mechanics. Games like Left 4 Dead, Resident Evil and The Last of Us use fantastical pandemics as a backdrop to frame their horror narratives, having you fighting against infected enemies usually amid a post-apocalyptic world ravaged by pandemics. Games like Plague Inc however use the processes of infection and pandemic as the core of their gameplay.

Plague Inc, released in 2012 by Ndemic Creations, is a real-time strategy simulation that revolves around creating and evolving a pathogen, turning it into a deadly global pandemic to try and wipe out the human race. The game only lets players control the virus indirectly, upgrading it with mutations that allow more routes of infection and making it resistant to attempts to cure it. Though Plague Inc.'s mechanics do not simulate the transmission of infection scientifically, it has been hailed as a tool to help raise awareness about the spread of disease and how viral infections work. (Feder, 2020)

2.4 Multiplayer Game Development

2.4.1 Game Engines

There are two main game engines that offer comprehensive multiplayer solutions competing in the games industry currently. One is Unreal Engine (also known as UE4), the other is the Unity engine. Both engines have strengths and weaknesses and offer different methods for solving the problems faced when developing online games.

Unreal Engine is a cross-platform engine that is under continual development by Epic Games. The engine is currently on its 4th iteration, with the original Unreal Engine debuting in 1998. Though the majority of programming in Unreal is done in C++, it contains an incredibly powerful node-based visual scripting system known as Blueprints.

Access to Unreal was originally operated on a subscription model, however in 2015 Epic Games made access to the technology completely free. It instead shifted over to a model that charged 5% royalty on revenue after the first $3,000 per product and per quarter. (Gaudiosi, 2015)

There are a host of benefits to developing in Unreal. It is widely considered to be one of the best engines for studios developing high-end triple-A games due to its graphical fidelity and extensive range of post-processing abilities. Another core selling point is the Blueprint system, which simplifies complex code processes down to nodes that can be placed like building blocks to create surprisingly complex actions with very little need for code. (Malashniak, 2016)

Unreal includes its own multiplayer networking solution, however, due to its proprietary multiplayer system known as ‘Replication’, alternative API’s that replace the backend are not available to the same level as on some competing game engines.

Competing with Unreal, the Unity engine (also known as Unity3D) is another cross-platform game engine that is developed by Unity Technologies and that was first released in 2005, in a somewhat primitive state. (Brodkin, 2013) Programming in Unity can be done using C#, JavaScript. This makes it incredibly easy and open to program comprehensive low-level functionality.

The engine operates on a licensing model that offers both free and paid options. Developers using the game for personal use, or that generate less than $100,000 in revenue have access to the free license while users of the pro version pay subscription. (Unity Technologies, 2015)

One of the key selling points of the engine is how suitable the engine is its ability to scale down to smaller projects. Where Unreal is considered to be a high-end focused solution, Unity is considered to be more focused on indie development.

Another one of Unity's other major features is the Unity Store. This allows developers to buy and sell a wide variety of Unity specific assets. These can range from 3D models to plugins that greatly expand the functionality of the engine and implement new features (Nicoll & Keogh, 2019).

There are many advantages to using Unity for development. The engine currently has support for 25 platforms. These range from mobile devices to next-generation consoles. It has comprehensive documentation and an active and helpful community that readily help out with advice on solving problems.

2.4.2 Multiplayer Networking

Multiplayer gaming has existed since the creation of some of the very first computer networks. Games were an excellent way to test these early systems. As networking technology has grown, the complexity of networked games has grown with them. (Rivenes, 2017)

Modern networking can trace its legacy back to ARPANET (Advanced Research Projects Agency Network), which pioneered data packet switching in 1965 and allowed two computers at the MIT Lincoln Lab to communicate with each other. (Zimmermann & Emspak, 2017)

The early games were mainly text adventures between university students; however, these early experiments would lay the groundwork for the advanced multiplayer networking of today's games.

Multiplayer games are for the most part reliant on a client/server method of communicating. This is a modular and performant form of networking that improves the usability, flexibility, system interoperability and that is heavily scalable dependent on requirements. (Yadav, 2009)

The server controls the logic of the game itself. Clients connect to it and update it with changes to their local state. The server then validates and implements these changes before distributing them to the other connected clients. This becomes especially important in the prevention of cheating, as if someone makes unauthorized changes to their local client, the server can detect these actions

and take steps to counter them. This may include forcibly disconnecting the client or banning the network address of the involved client.

Unity and Unreal have their own versions of client/server networking already implemented into their engines, with several built-in features such as matchmaking (matching players based on certain conditions and connecting to them) that make use of their servers. In both cases these services come at a rental cost that is dependent on usage.

There are other multiplayer client/server network solutions than Unity's stock version, however. Photon is one of the leading alternatives to the built-in network functionality and is available from the Unity store. Created by Exit Games, it provides low to high-level APIs and uses the company's own proprietary servers to achieve the same functionality as Unity's offering, but at more affordable prices. (Exit Games, 2020)

2.5 Conclusion

The simulation should use Unity engine for development and utilize a client/server networking model to facilitate the connection of multiple clients across the internet. Though Unity has it's own built-in methods for doing this, using an alternative API, such as Photon, with a wider range of functionality would be preferable.

When simulating disease transmission, direct-contact person-to-person transmission should be used as it will generate clearly trackable infection events with which to track infection times and generate data.

Section 3 – Design

3.1 Resources

The resources required to make this product will be:

- Unity game engine 2019
- Visual Studio Community
- Playmaker visual scripting plugin for Unity
- Photon2 multiplayer networking plugin for Unity
- Working internet connection

3.2 Overview

The product will be an infectious disease simulation built using Unity engine and presented to players in a game-like manner. The simulation will be made available in the form of a standalone client that testers will be able to download which will then allow them to connect over the internet to take part in the tests. The simulation will focus on direct contact transmission, specifically person-to-person transmission.

By breaking down the product into essential elements, four pillars can be used to guide development:

- **Accessibility**
- **Networking/Multiplayer**
- **Player Infection**
- **Data Tracking**

The product will be a 2D agent-driven game, where each player takes control of an avatar and connects to a test instance run by an observer that will oversee the test and extract relevant data. Accessibility is a core requirement in order to be able to gain data from a wide variety of individuals, who may not have a background or understanding of complex game mechanics.

At the beginning of each test one of the testers will be designated as infected. The infected tester will then have the task of chasing down other players and transmitting the infection via contact with the non-infected players. This will take place in environments of increasing complexity so that the rate of infection can be observed.

Data on infections will be tracked during each test session and then exported upon the session's completion. This data will detail the times of each infection, and the overall time from initial infection to the total infection of all testers.

3.3 Networking

In order to have multiple testers able to enter the same instance of the game for testing, a networking solution will be required to facilitate players connecting across the internet. The project will utilize a client/server system to handle connection, due to its potential for scalability and being the networking solution of choice within the games industry. This will allow for the use of the significant resources and learning materials that exist for this architecture.

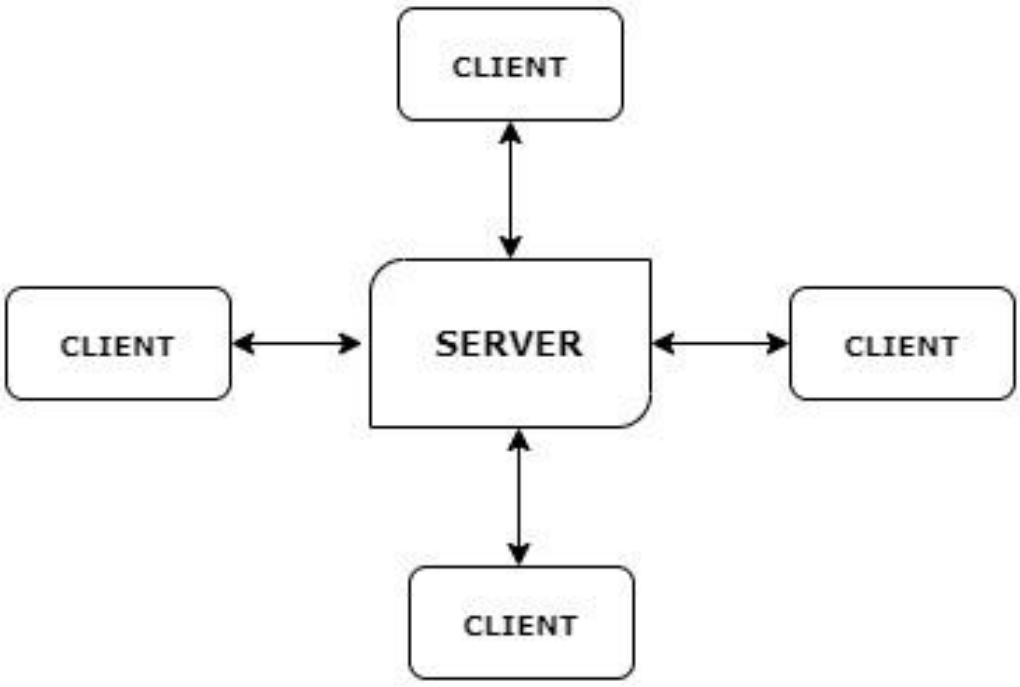


Figure 1 Client/Server Networking

Each tester will run their own client version of the simulation. Each client will then pass any changes in state to the server to validate and update the changes to all other clients.

Though Unity engine has its own high-level API for client/server networking, the product will instead use Photon2 plugin to handle the multiplayer aspect of the simulation. This has a large library of pre-built networking functionality that can be utilized to handle client connection, movement, state synchronization and a comprehensive server lobby system.

The solutions to basic yet complex networking actions will allow for the implementation of the core mechanics without the need to write complex network algorithms and without being limited to the standard networking functionality of Unity.

3.4 Player Controller

The player controller will be the primary method of interaction used to drive the simulation. It will be responsible for the locomotion of the player agent within the environment as well as transmitting and receiving infection states from players to the server. Syncing the player controllers to the server must also be handled robustly at runtime in order to minimize issues with latency.

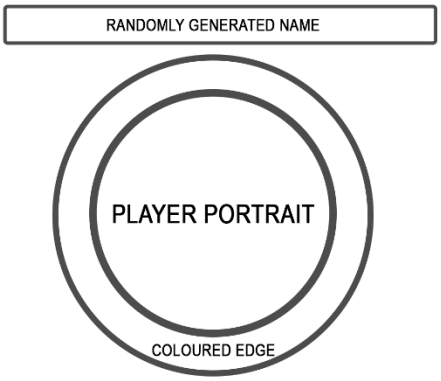
The player controller will be comprised of several parts:

- **The player controller component**
- **The networking components**
- **Visual components**

The player controller component will take the form of a script that handles all the core code used to drive the players' avatar locally and to push state changes across the network when they occur. It will drive the local locomotion and track whether the player is infected.

The networking components are majoritively found within the Photon2 networking solution. Photon has a library for handling commonplace functions such as tracking movement. They will be used to sync player controller location as well as the various low-level networking requirements like client identification and transferring and receiving data from the server. Bespoke networking

elements will be needed to handle tasks specific to this project, such as player infection. These will be handled via state synchronization via calls to the server.



Visual components will be used to present the controller to the player, and will include a randomly selected player portrait, as well as a randomly selected avatar portrait and name. There may also be the option to facilitate player chosen aliases to provide a higher level of player engagement, and to also aid in the distinction between players.

Figure 2 Player Controller Design

3.5 Observer Controller

The observer controller will allow one connected player complete overview of the entire map to observe the infection events and be able to record player behaviours. The controller must have the ability to freely zoom in and out, and also pan around the test environment without restriction. Players will be able to change to observers before the test starts from the lobby system interface.

3.6 Infection

Infection is one of the core parts of the simulation and has two key elements; player-to-player transmission in the game and the tracking of infection events to acquire usable data for further external analysis.

Infection between players in-game will be handled by the player controllers. By opting for a contact-based transmission system, the mechanics of spreading the infection amongst testers can be kept simple. When two of the player controllers' collider elements come into contact, a check will be performed. Both controllers will look to see if the other is in an infected state. If one controller is infected the non-infected controller will initiate a state change to become infected itself.

When a controller has its infection state is triggered, it will perform several tasks in succession. As the change in infection state will only be local to controller's client, the event client must push its change of state to the server. The server will validate this change on its own client and then update the infection state of each instance of the controller on all other connected clients.

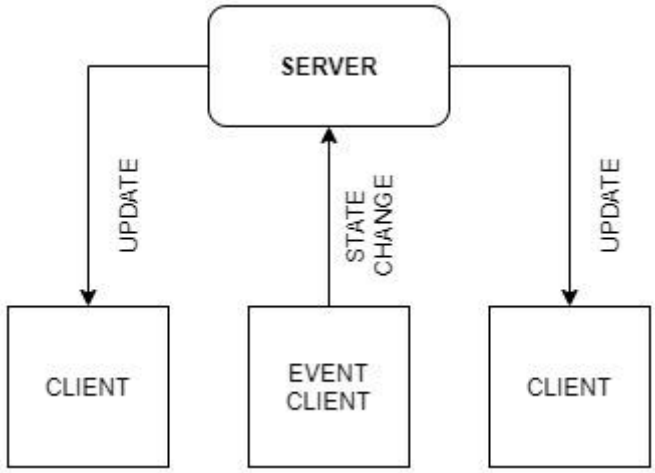


Figure 3 Client State Change Updates

Secondly the controller will update its portrait and colour to signify that it is infected. This visual cue will be needed to make it clear to the tester that they are now infected, and that they should now begin trying to infect others.

Finally, the controller will update the infection logging system to give the time the infection event has taken place and which controllers were involved.

3.7 Infection Logging System

This system will be responsible for monitoring a recording infection information within each test and outputting it in a way that can be used for further analysis. The logger will record the time and agents involved in each infection event, which will be passed into it by a player controller upon becoming infected. The time and date of the test-taking place should also be recorded for convenience purposes when the data is exported.

At the end of each test, the logger must be able to output the data into a readable format that can easily be accessed for data analysis. Ideally this will be in the form of a text document with the

datapoints clearly separated out for input into spreadsheets or data analysis software. One other consideration is that in order to keep everything organized and prevent large amounts of file generation, all output data should be added to one output file.

3.8 Environments

The in-game environments that each test will be conducted in will change in size and complexity with each consecutive test. To keep the tests consistent, the scaling of the environments must also be consistent.

Each level will scale in a complexity of *2(n)* with *n* being the number of rooms in the environment. By doubling the complexity in each test, results that are gained should be repeatable and should enable multiple sessions of testing to provide further data.

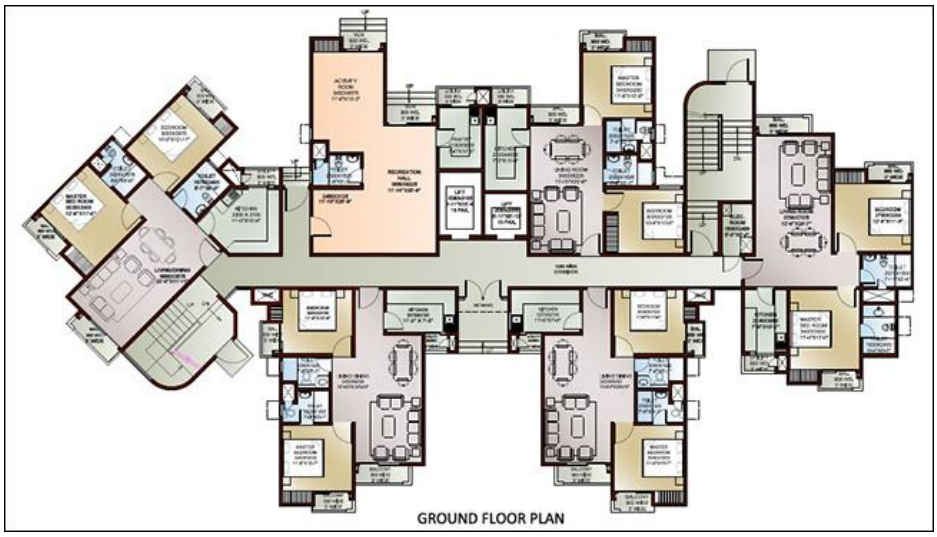


Figure 4 Example Floorplan

Levels will be constructed as 2D floor plans, similar to building schematics. This should be easily recognizable to testers and convey the game perspective in its visual language. Combined with the design of the player controller, it should allow for straight forward navigation throughout environments.

3.9 Visual Design & User Interface

The design of the user interface is of specific importance, due to the fact testers will not have demonstrative instruction on how to set up the application and connect to tests. The product must convey all the relevant information as simplistically as possible to the testers in an understandable manner. This approach will account for anyone testing who has very little understanding of traditional computer game mechanics and help to increase accessibility and readability of the processes required in connection.

As the focus is on driving a data gathering simulation and not a commercial gaming product, there is less requirement for visually striking aesthetic and will instead focus on functionality and ease of use over form.

One area that needs to be easily understandable and streamlined is the process of connecting to the tests themselves. If the process is too convoluted, then accessing the rooms might be incredibly difficult without increased levels of guidance.

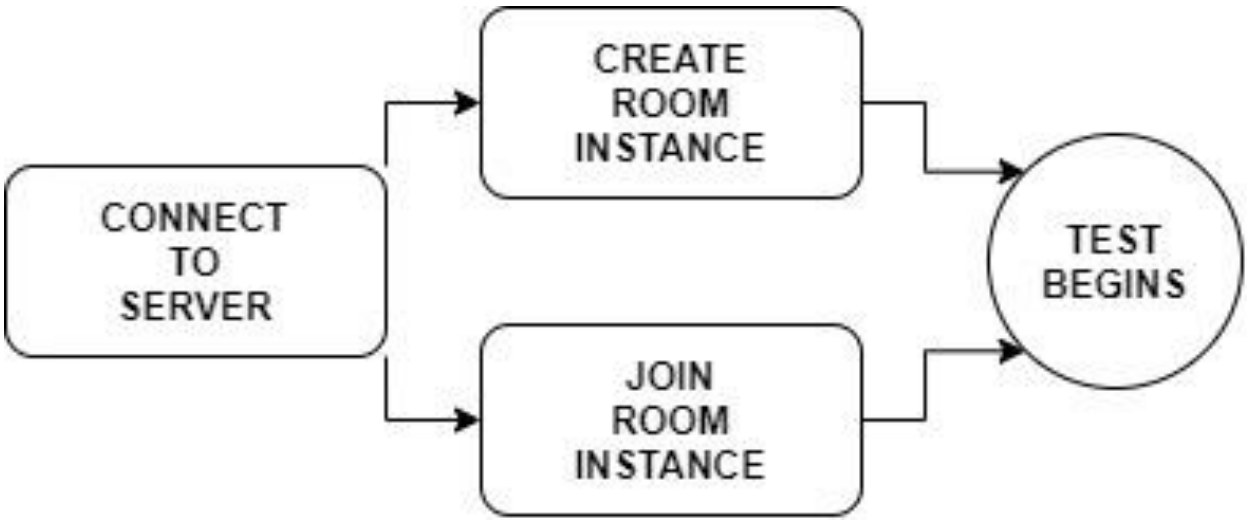


Figure 5 User Interface Flow

To make the user experience as uncomplicated as possible, the menus will be as streamlined and have as few steps between starting the application and entering a test as can be achieved. As pictured in the diagram above, the process must have a simplified flow to it, with any complex actions of the networking components concealed behind these simple interfaces. By keeping the focus on reducing the complexity of network actions at the front end to binary UI choices, the number of steps users must go through will be minimal.

The second area that will require attention is the ease of extracting data. This needs to be available at the end of every test and will be controlled by a button in the UI menu within the game scene that will be linked to a function in the data logger that will handle data exporting.

Section 4 – Implementation

4.1 Engine Setup

The first step in creating the product was to start an empty Unity 2D project, as the project had no need for 3D specific functionality. The latest version of Unity was used, which is 2019.3.5f1. The Playmaker and Photon2 plugins were added to the project via the Unity store in order. These were implemented at the start so that any conflicts between them could be resolved at the earliest possible opportunity.

4.2 Player Controller

As the primary method of user interaction and driving force of the simulation, the player controller was the first item to be developed. The initial step in doing this was to create the temporary graphics and text elements as well as a master control script to contain most of the controller's functionality. Circle collider and Rigidbody2D components were added to handle both collision events and drive movement.

Figure 7 Player Controller Visual Components

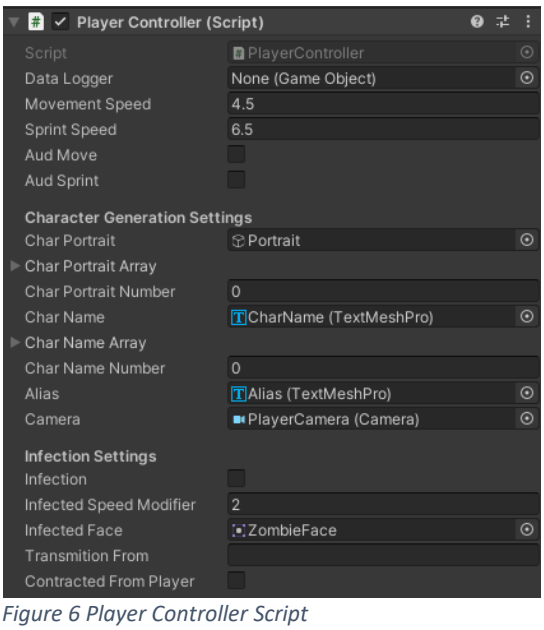


Figure 6 Player Controller Script

To get more natural feeling movement, a method was created to drive the controllers Rigidbody by applying force based on user input. This helped to give a degree of physicality to movement with players having momentum, which meant that the acceleration and deceleration and the prevention of sudden directional change to the player controllers felt more consistent with human movement.

```
void Movement()
{
    float posX = gameObject.transform.position.x;
    float posY = gameObject.transform.position.y;

    if (!Infection)
    {
        if (!sprinting)
        {
          Vector2 movement = new Vector2(Input.GetAxis("Horizontal"),
          Input.GetAxis("Vertical")).normalized;
          rigid.AddForce((movement * movementSpeed * Time.deltaTime) * 1000);

        }
        if (sprinting)
        {
          Vector2 movement = new Vector2(Input.GetAxis("Horizontal"),
          Input.GetAxis("Vertical")).normalized;
          rigid.AddForce((movement * sprintSpeed * Time.deltaTime) * 1000);
        }
    }

    if (Infection)
    {
        if (!sprinting)
        {
          Vector2 movement = new Vector2(Input.GetAxis("Horizontal"),
          Input.GetAxis("Vertical")).normalized;
          rigid.AddForce((movement * movementSpeed * Time.deltaTime) * 1000);
        }
        if (sprinting)
        {
          Vector2 movement = new Vector2(Input.GetAxis("Horizontal"),
          Input.GetAxis("Vertical")).normalized;
          rigid.AddForce((movement * (sprintSpeed + InfectedSpeedModifier) *
          Time.deltaTime) * 1000);
        }
    }
}
```

The *Movement* method contains two states, one to govern non-infected player movement and one to control player movement once a player becomes infected. The character infection state decides upon which one is active and in use at any given time.

In order to track/get movement synchronizing across the network, the player character uses several components from Photon2. Each controller needed to have a Photon View component which is used as an interface to track local Photon functionality and pass its data to the server.

The Photon Rigidbody2D View component synchronizes the forces applied locally to the controllers Rigidbody. As locomotion using a Rigidbody utilizes physics, this component attempts to keep all physics computations consistent across all instances of the controller. This prevents situations in which two local clients calculate two separate results. If for some reason this does not work, and an instance of the Rigidbody is located outside of a certain distance range, the instance is teleported to the location of its originator on the affected local client.

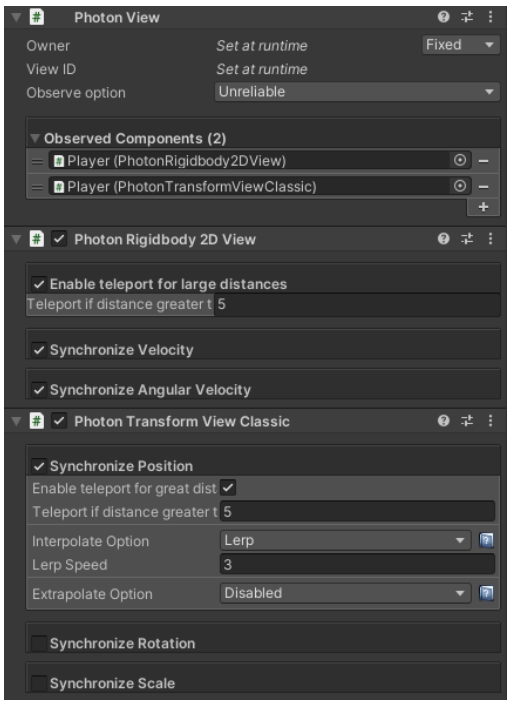


Figure 8 Photon Components

The Photon Transform View component tracks the player transform for synchronization across the network and then linearly interpolates the position on other clients to adjust for latency. It can also synchronize rotation and scale, but these options were not needed or used.

Figure 9 Randomly Generated Player

After the implementation of movement, the next step was to create a way of distinguishing between players. A random character generator was added that is initiated at the start of a level and used to generate random characters based on a pool of names and portraits. Each character also generates a random coloured outline to reduce the chances of identical characters being created.

Once the character is generated, the character name, portrait, and colour are pushed to the server for synchronization. The server implements the changes, and then pushes the variable details to the other connected clients.

```
void InitiateCharacter()
    {
        int random = Random.Range(0, charPortraitArray.Length);
        charPortraitNumber = random;
        charPortrait.GetComponent<SpriteRenderer>().sprite =
charPortraitArray[charPortraitNumber];

        if (charPortraitNumber <= (charPortraitArray.Length / 2))
        {
            random = Random.Range(0, charPortraitArray.Length / 2);
            charNameNumber = random;
        }
        if (charPortraitNumber >= (charPortraitArray.Length / 2))
        {
            random = Random.Range(charPortraitArray.Length / 2,
charPortraitArray.Length);
            charNameNumber = random;
        }

        charName.text = charNameArray[charNameNumber];

        Color customColor = new Color(Random.Range(0.0f, 1.0f), Random.Range(0.0f, 1.0f),
Random.Range(0.0f, 1.0f));
        this.gameObject.GetComponent<SpriteRenderer>().color = customColor;

        string alias = PhotonNetwork.LocalPlayer.NickName;
        Alias.text = alias;

        PV.RPC("RPC_PushCharacterInitiate", RpcTarget.AllBuffered, charPortraitNumber,
charNameNumber, alias, customColor.r, customColor.g, customColor.b);
    }
```

Though premade functions were available for tracking movement, custom functions that synchronize whether the player is infected or running Booleans were required. These were created using Photon2 RPC calls, which send a value through the Photon Proxy and then triggers the server to push the value change to all clients.

```
#region Network Functions
    [PunRPC]
    void RPC_PushCharacterInitiate(int portNum, int nameNum, string alias, float red,
float green, float blue)
    {
        charPortrait.GetComponent<SpriteRenderer>().sprite = charPortraitArray[portNum];
        charName.text = charNameArray[nameNum];
        Color custCol = new Color(red, green, blue);
        this.gameObject.GetComponent<SpriteRenderer>().color = custCol;
        Alias.text = alias;
    }
    [PunRPC]
    void RPC_InfectPlayerSync(bool infection)
    {
        Debug.Log("Pushing infection state update to network");
        Infection = infection;
        gameObject.tag = "Infected";
        charPortrait.GetComponent<SpriteRenderer>().sprite = InfectedFace;
        this.gameObject.GetComponent<SpriteRenderer>().color = new Color(0, 1, 0);
    }

    [PunRPC]
    void RPC_SyncMoving(bool moving, bool sprinting)
    {
        //Debug.Log("Pushing movement to network");
        audMove = moving;
        audSprint = sprinting;
    }
    #endregion
```

4.3 Observer Controller

The observer controller is utilized by the person overseeing the test to be able to gain complete oversight of the test environment to monitor all the test participants, the progress of infection, any errors that may be occurring within the simulation, and to provide a platform with which to record the tests for future reference when used in conjunction with an external video recording tool.

It is different from the player controller in the fact that it has a completely free range of movement and the ability to zoom in and out to focus individual players or areas or get a complete aerial view of the scene.

Figure 10 Observer Controller Selection Box

The tick box that controls the observer mode that utilizes the observer controller is found before a test is started inside the lobby room. By ticking the box, its controlling state machine alters a value in the applications player preferences, so that when the test begins, it will spawn the observer controller instead of the regular player controller.

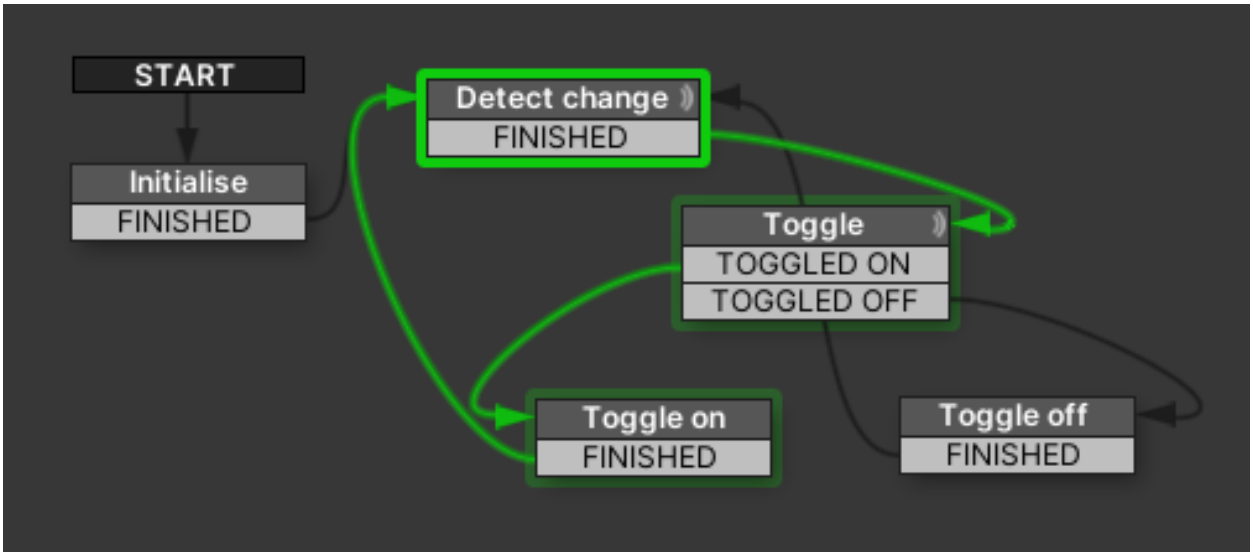


Figure 11 Observer Controller State Machine

The controller itself uses a completely different set of methods to control its movement and systems. As it does not have to simulate physics-based locomotion like the player controller, a much simpler method of translating the camera's position could be employed.

```
void MoveCamera()
    {
       transform.position = new Vector2(transform.position.x
       + (CameraSpeed * Input.GetAxis("Horizontal") * Time.deltaTime),
       transform.position.y + (CameraSpeed * Input.GetAxis("Vertical") *
       Time.deltaTime));
    }
```

The camera zoom is achieved by simply adjusting the size of the orthographic camera.

```
  void ZoomCamera()
    {
        if(Input.GetKey(KeyCode.Q))
        {
            mainCam.orthographicSize -= CameraZoom;
        }
        if(Input.GetKey(KeyCode.E))
        {
            mainCam.orthographicSize += CameraZoom;
        }
    }
```

4.4 Network

As the product uses Photon2 for its networking, the plugin needed to be configured and implemented correctly both within the project and externally on the Photon2 website.

A new application was set up for the product, and an application identification code generated. This code, when implemented into the project via the setup wizard, links the game to the application on the server.

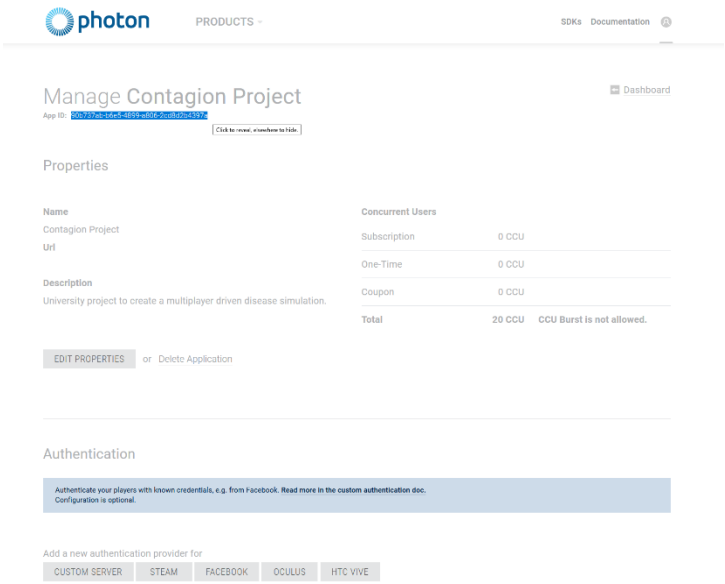


Figure 12 Photon Backend Setup

Getting the game client connecting to the server involved using Playmaker visual scripting plugin to harness Photon's pre-built network setup functionality.

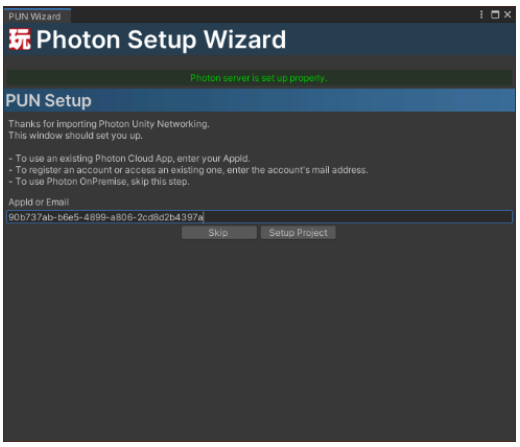


Figure 13 Photon Setup in Engine

In order to get multiple character controllers to connect to together to specific instances of levels required the use of the Photon2 lobby system. This system works by establishing a connection to the master server hosted by Photon. Once authentication has been completed and the client is connected to the server, players can create or search for rooms.

When players create a room, a new lobby is instanced on the master server where the player can wait for others to join their session. If the player decides to join a room instead, the master server will search for any available room instances and then connect the player to any that have open slots.

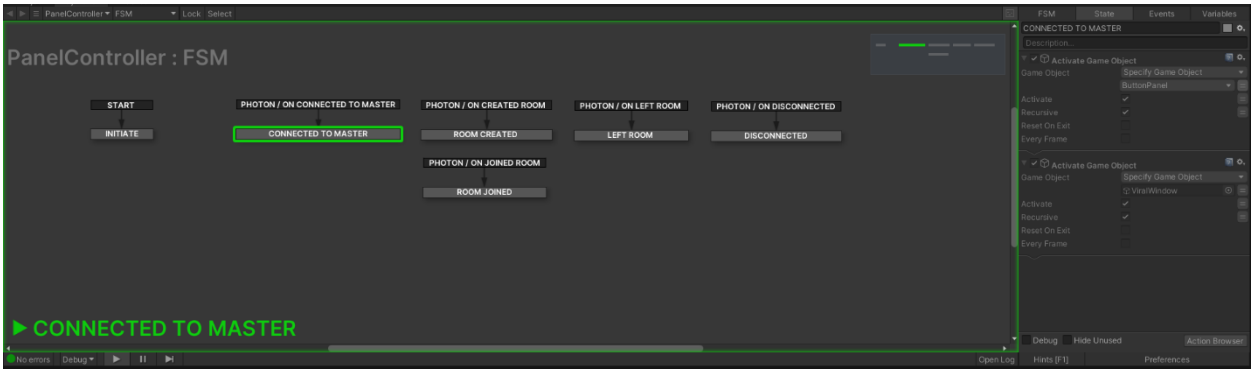


Figure 14 Master Connection State Machine

By using Playmaker, state machines were utilized to handle all aspects of the networking process and to connect the low-level functionality to the UI front end. This was achieved by giving individual elements of the UI their own state machines to handle the entirety of their individual functions.

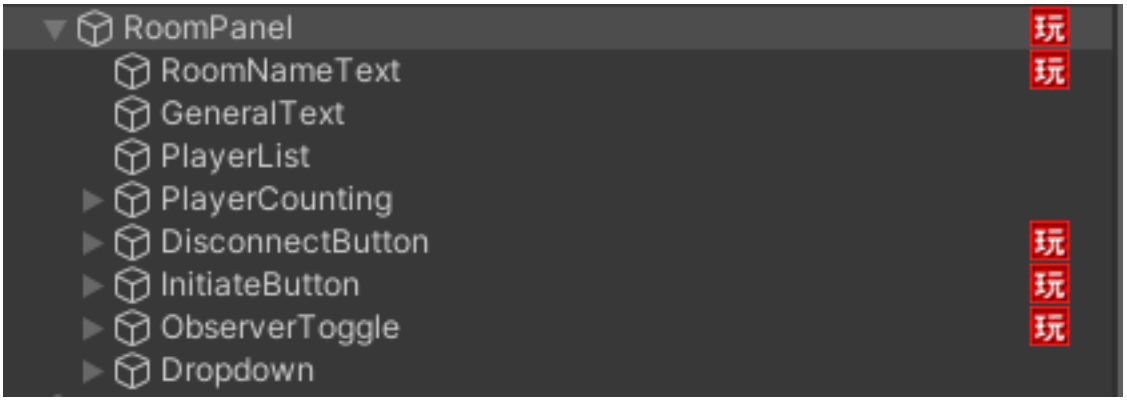


Figure 15 UI PlayMaker Component Hierarchy

Using this approach was especially beneficial for the visualization and creation of the lobby system, where individual UI components had their own state machines running individual networking tasks. For instance, the *RoomPanel* components state machine was built to handle changes to the room instances' stored variables, such as player count and room name, both of which it retrieves from the Photon API's standard functions. Using simple logic, it is able to cycle through and update all the corresponding UI components efficiently without any need for time-consuming low-level code implementation.

This approach of setting up the networking allowed for the smooth and rapid implementation of a very clean main menu that has a very efficient flow to joining a test, and a level of simplicity that should be easily navigable by users.

4.5 Game Manager

The game manager is responsible for initializing the game scene and setting up all of the network components at the start of each test. The state machine in charge of handling this looks at the type of controller that is required, whether that be a standard player controller or an observer controller, and will then instantiate that component inside the scene.

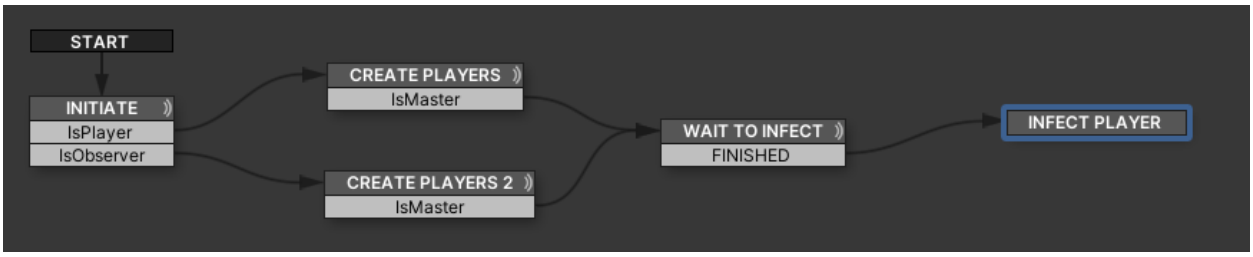


Figure 16 Game Manager Initialization State Machine

The game manager is set to wait a certain length of time after being initiated so as to make sure that all clients have sufficient time to connect to the test. After this wait period has finished, the game manager then selects a player at random to infect. It infects the player by triggering the *InfectPlayer* method. This is then logged as the initial infection event.

4.6 Infection

Two steps to handling infection were created. The first to handle infection locally on the client and the second to then change the controller infection state across the network.

The client-side infection event is triggered when an infected controller's collider overlaps a non-infected controller's collider, creating a contact event.

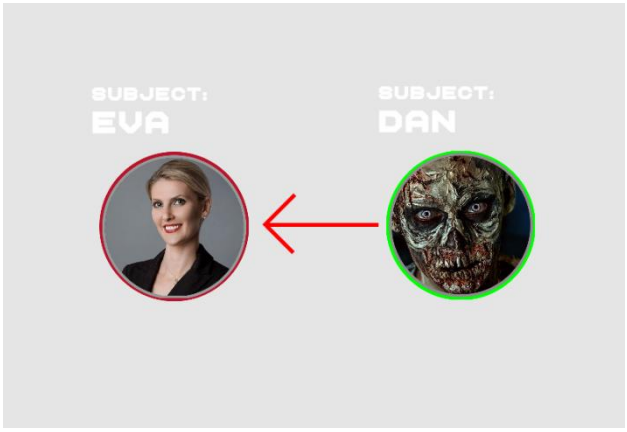


Figure 18 Before Infection Transmission

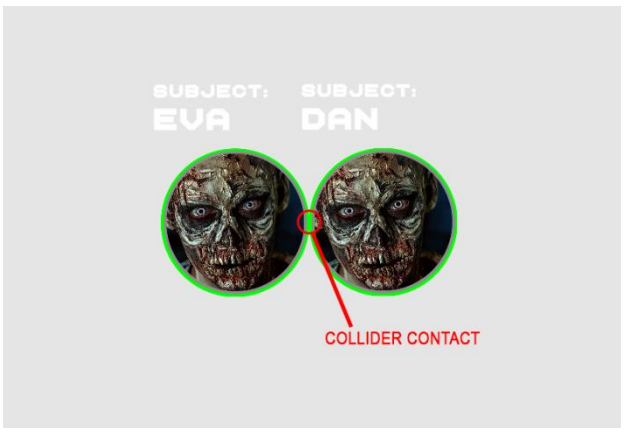


Figure 17 Contact Event

The process of the infection itself occurs within the player character being infected. This is done by doing a check to see if the colliding player is tagged as infected, which when true goes through two steps.

If the colliding player is infected, the controller changes its infection Boolean to true. This then changes some of the core functionality such as increasing the movement speed. It also logs the identification of the player that collided with it for passing to the infection logger.

```
private void OnCollisionEnter2D(Collision2D collision)
    {
        if(collision.gameObject.tag == "Infected" && !closedToInfection)
        {
            Infection = true;
            ContractedFromPlayer = true;
            TransmitionFrom = collision.gameObject.name;

            if (PV.IsMine)
            {
                InfectPlayer();
            }
        }
```

Secondly it triggers the infect player method, which in turn calls the network method *RPC_InfectPlayerSync* on the server. The purpose of this network method is to pass the change in infection state to the server for validation. This method performs the changes to the portrait and game object tag and outline colour. In order to keep the process synchronized between all versions of the controller on all clients, the changes are made on the server, then pushed from the server so that both the local player controller and all instances update at the same time.

```
[PunRPC]
    void RPC_InfectPlayerSync(bool infection)
    {
        Debug.Log("Pushing infection state update to network");
        Infection = infection;
        gameObject.tag = "Infected";
        charPortrait.GetComponent<SpriteRenderer>().sprite = InfectedFace;
        this.gameObject.GetComponent<SpriteRenderer>().color = new Color(0, 1, 0);
    }
```

4.7 Infection Logging

Logging was created to be handled locally based on infection events. Infection events are stored in a list of strings within the logger, and the data relating to these events are passed in by player controllers as they become infected. This is done via two versions of the *LogInfectionStates* method.

```
public void LogInfectionStates()
    {
        string eventDetails = "Infection at " + Time.time;
        InfectionEvents.Add(eventDetails);
        Debug.Log(eventDetails);
    }

    public void LogInfectionStates(string agent1, string agent2)
    {
        string eventDetails = "Infection at " + Time.time +", Between " + agent2 + " and
" + agent1 + ".";
        InfectionEvents.Add(eventDetails);
        Debug.Log(eventDetails);
    }
```

The first version of the method is solely to handle ‘infection 0’, which receives its infection from the server, and does not have another agent passing it the infection. In this scenario the time of the event is logged only.

The second version is for player to player transmission. When this method is called, the controller becoming infected has already recorded the player identification of the controller that is infecting it. Both are passed in and stored along with the infection time.

4.8 Data Output

The output of data is controlled by the OutputData.cs script, which is responsible for taking all the event data stored within the infection logging script and converting it into an external format readable once the product is closed.

The script initiates at the start of each level and records the scene name and start time to make test data easily readable when being evaluated externally. It then sets the path the data will export to.

```
void Start()
    {
        LevelName = SceneManager.GetActiveScene().name;
        StartTime = ""+ System.DateTime.Now;
        path = "C:/ContagionData/InfectionData.txt";
    }
```

Players can click on the floppy disk icon to export in the in-game menu to export the data at any time.

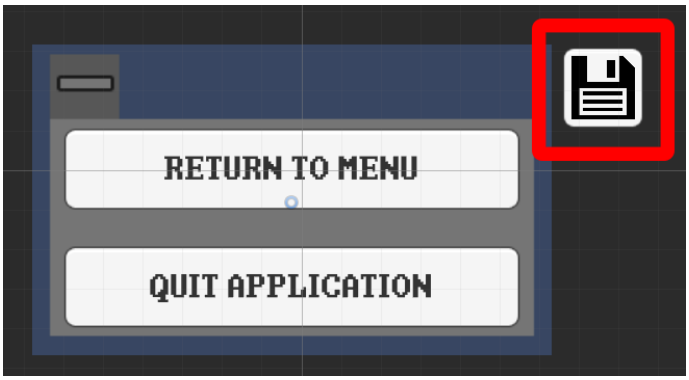


Figure 19 Data Output Button

This will call the *WriteDataToFile* method. This method handles the entire exportation process. The first step of this is to check if the directory the product stores its data to exists, and if it doesn't exist it creates it. It next converts the start time, the end time and the scene name into a string ready in the *ToData* variable to be exported as readable text. When this is completed, it goes into the infection logger and each item in the infection event list is appended to *ToData*. Finally, it checks to see if a file containing results is already present. If there is already a file the contents of *ToData* is appended to the text document that already exists, and if no file exists, it creates an empty document to write to.

```
public void WriteDataToFile()
    {
        if (!Directory.Exists("C:/ContagionData"))
        {
            Directory.CreateDirectory("C:/ContagionData");
        }

        ToData = "Test Initiated: " + StartTime + "\nTest Ended:" + System.DateTime.Now +
"\nTest Environment: " + LevelName + "\n \n";

        foreach(string events in tracker.InfectionEvents)
        {
            ToData += events + "\n";
        }

        ToData += "\n-------------------------------\n\n";

        if (File.Exists(path))
            File.AppendAllText(path, ToData);

        if (!File.Exists(path))
            File.WriteAllText(path, ToData);
    }
```

4.9 Audio Controller

The audio controller was created to enable localized sound in each playable character. This required a networked solution as though the localized player controller's momentum is handled by applied force to the Rigidbody, the client-side instances of other controllers have their location translated by their Photon Transform components. This was solved by having a network method in the player controller update Booleans on the server when the local controller is moving.

```
[PunRPC]
    void RPC_SyncMoving(bool moving, bool sprinting)
    {
        //Debug.Log("Pushing movement to network");
        audMove = moving;
        audSprint = sprinting;
    }
```

The audio controller has a coroutine constantly running in the background ready to play audio based on the results of these Booleans when they are distributed back to the instances of the character control script.

```
IEnumerator PlayFootsteps()
    {
        while(true)
        {
            yield return new WaitForSeconds(currentTimeBetween);
            if (PlayerMoving)
            {
                int rand = Random.Range(0, Footsteps.Length);
                if(Footsteps[rand] == FootstepEmitter.clip)
                    rand = Random.Range(0, Footsteps.Length);
                FootstepEmitter.clip = Footsteps[rand];
                FootstepEmitter.Play();
            }
        }
    }
```

A similar system controls the noise of infected characters. However, it works based on the infection state in the character control script, which is already solved in the infection process.

```
IEnumerator PlayInfected()
    {
        while (true)
        {
            yield return new WaitForSeconds(Random.Range(2.5f, 5.0f));
            if (controller.Infection)
            {
                int rand = Random.Range(0, InfectedNoise.Length);
                if (InfectedNoise[rand] == InfectedNoiseEmitter.clip)
                    rand = Random.Range(0, InfectedNoise.Length);
                InfectedNoiseEmitter.clip = InfectedNoise[rand];
                InfectedNoiseEmitter.Play();
            }
        }
    }
```

The audio files themselves are stored in arrays and selected at random to provide variance to the audio played and prevent repetitive sounds.

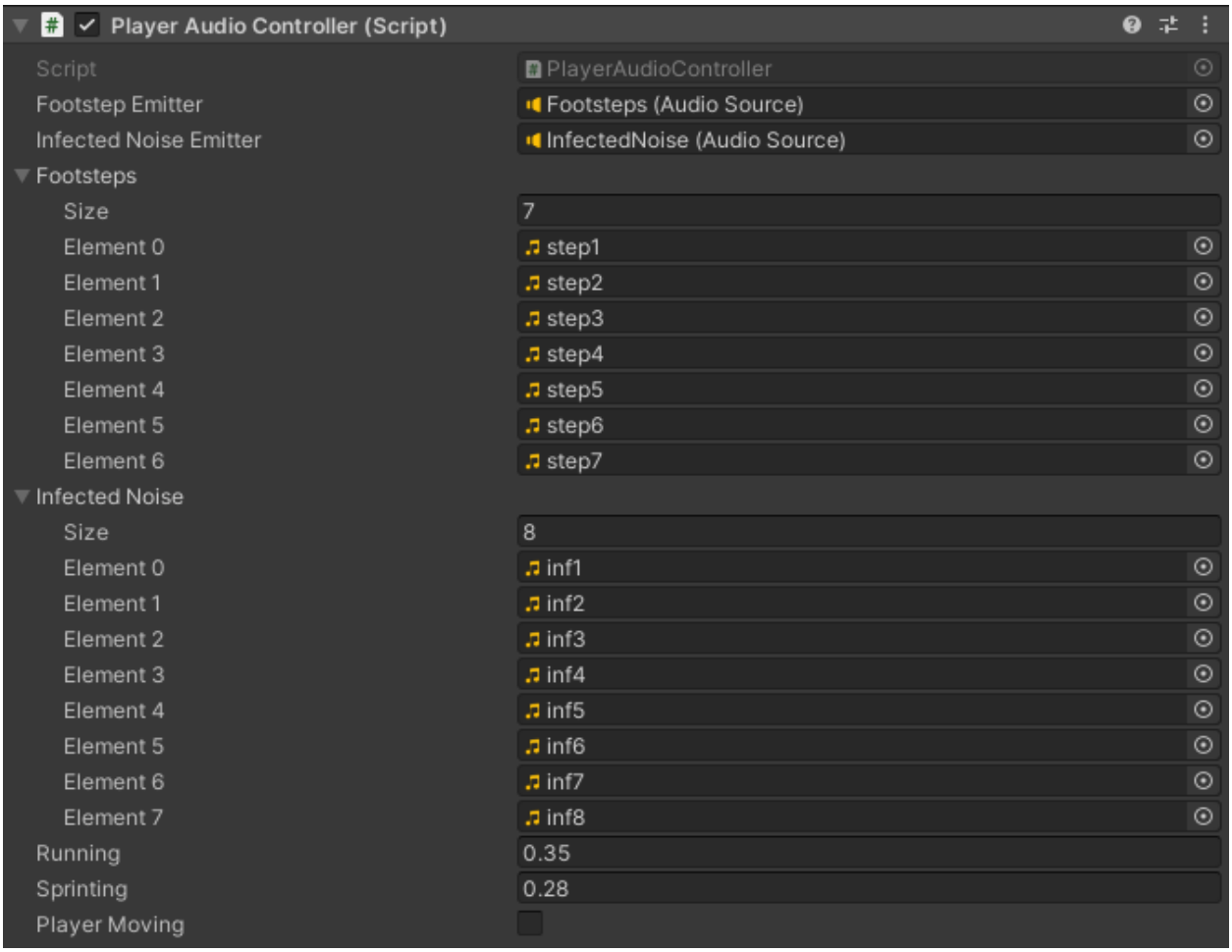


Figure 20 Audio Controller Script

4.10 User Interface

The UI was created to be as simple as possible to read and have a clear and concise visual language. Visually designed to mimic old windows operating systems, each of the UI elements can be dragged around and moved like windows in an operating system would usually.

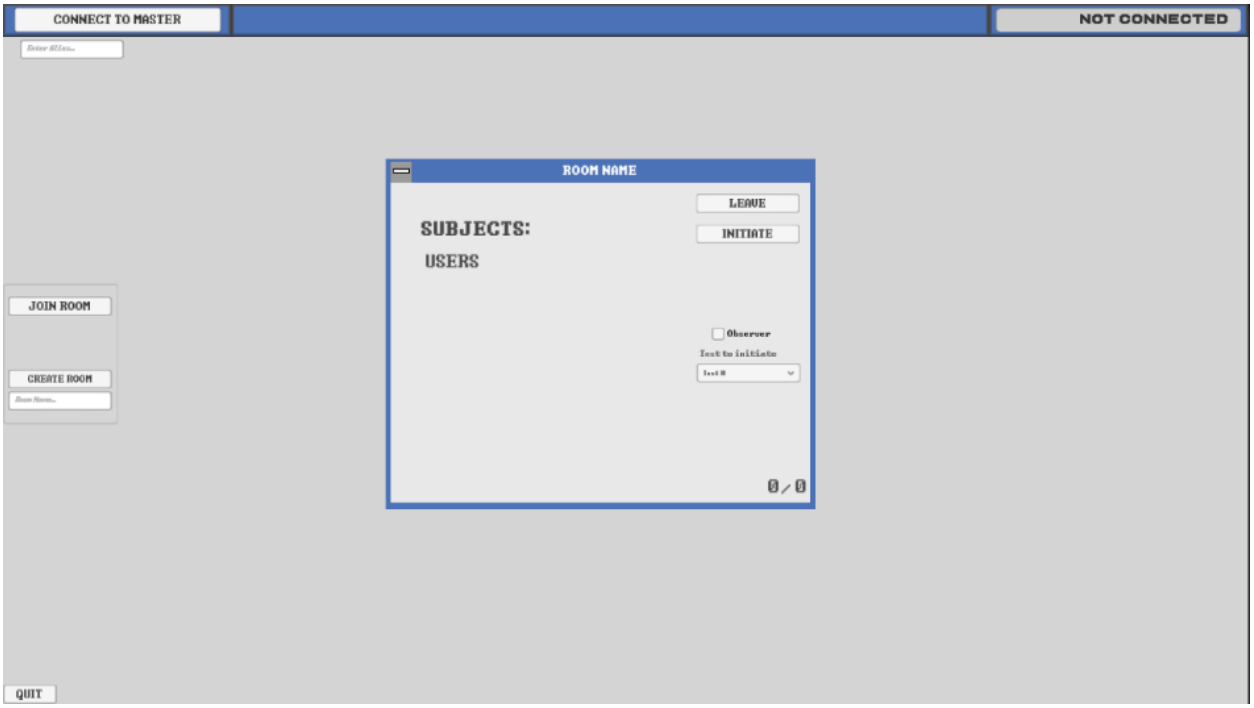


Figure 21 UI Overview

Each UI component is driven by Playmaker state machines, as they contain specific network functionality that they need to drive. By creating self-autonomous state machines, components can update themselves when changes in the network occur. An example of this is the room panel, where the state machine is constantly re-evaluating changes to the number of players currently connected, the maximum player counts the room is allowing and names of players currently active within that room.

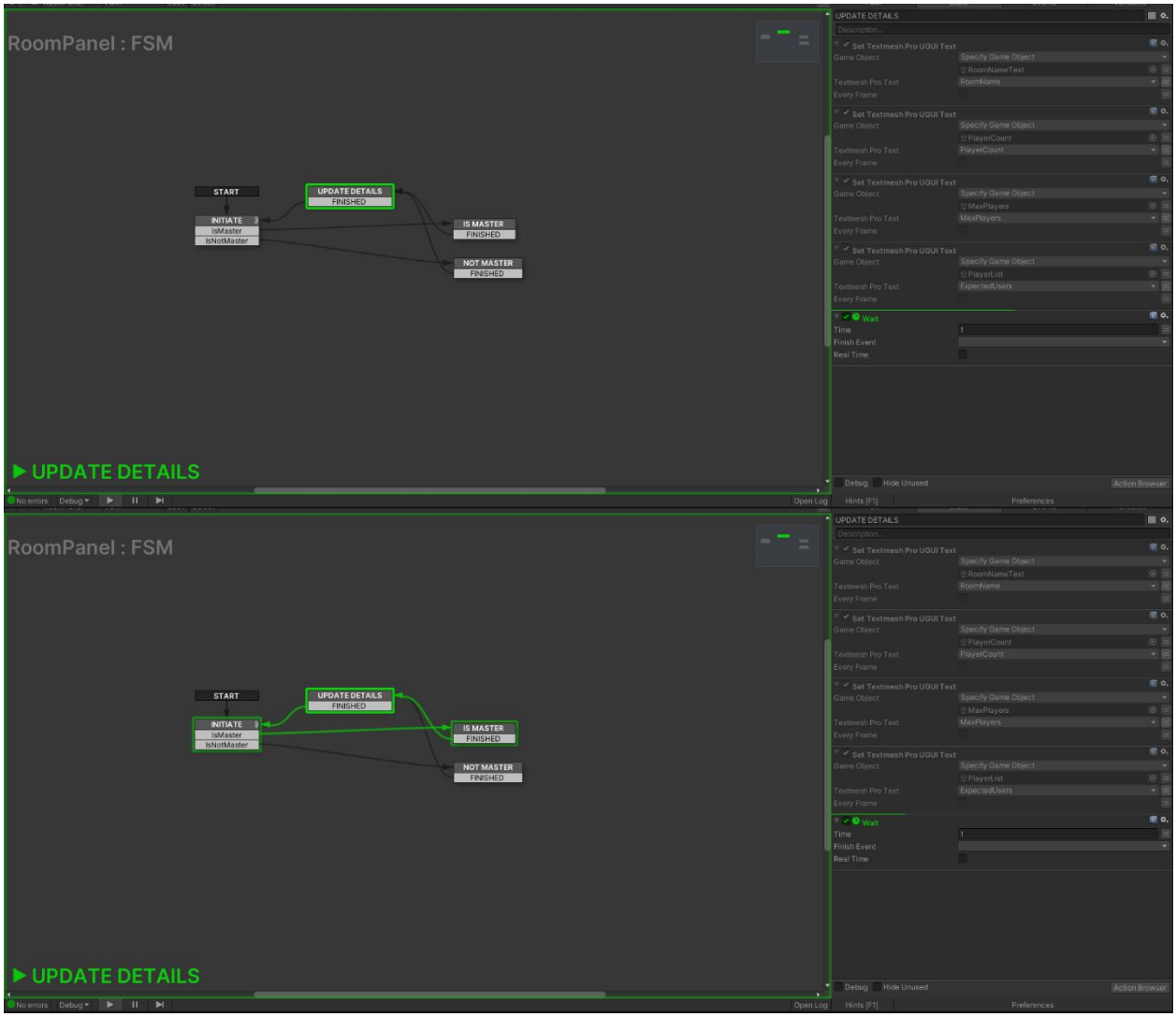


Figure 22 PoomPanel State Machine

Section 5 – Testing

5.1 Introduction

Testing the product required a group of testers to run through each test. Each test had them trying to avoid infection for as long as possible, and then once infected, trying to infect anyone left uninfected.

5.2 Local Testing

As constantly testing each iterative build with testers was not logistically viable, making sure components worked over the network had to be done locally on one computer. This process involved running a primary instance from inside the Unity editor and also building a test client to be run separately on the same computer.

Errors in the test builds were only visible within Unity editor, which meant that tests on specific functions had to be replicated in both the editor instance and the standalone instance. This method of testing was incredibly time-consuming as sometimes bugs were only present during abstract conditions that were incredibly hard to replicate consistently using only two running instances of the product.

When bugs were located and changes to the code made, in some cases these were only fixed for situations with a low level of connected clients and the bugs would go on to later reappear in the first testing phase requiring further work to fix.

5.3 Testing Phase 1

In the first phase of testing, testers were asked to download the client build of the product and run through each of the seven tests. This also served as an initial stress test, as previously there had been a lack of availability of testers to verify connection and functionality would work under full load. A total of 7 people took part in this part of testing, and the majority of the testers managed to take part in all of the tests.

The first phase of testing was incredibly difficult to manage, as gathering all the testers together and effectively communicating instructions proved to be an entirely unforeseen difficulty. The phase also highlighted a host of problem areas in the network functionality that needed further work and fine-tuning.

Though the tests were hampered by issues, data was able to be extracted that was in line with the product's objectives.

5.4 Problems

During the first testing phase a host of problems presented themselves that hadn’t shown in development where the ability to completely test functionality had been limited by a lack of testers. Tracking the network identification of players as the infected each other proved to only work in specific instances. Connection issues seriously affected a majority of the testers and varied in the most severe instances actually prevented the testers from connecting to rooms before the tests had even begun.

When players did manage to enter into the tests, they in some instances had issues with latency spikes that caused unforeseen problems with the early triggering of infection in some character controllers before any form of contact had been made and invalidating test results due to the inaccuracies in the data being record.

Crashes in the application also caused some testers to drop out at inconvenient points during the test. This subsequently became a huge problem, as there would be instances where the initial infected player would crash out of the game before infecting others. This prevented any further infections occurring and meant that tests needed to be restarted repeatedly. A lot of these problems had initially been identified in local testing and thought fixed, however, the changes implemented had not taken into account the expanded number of connections.

Because of all these issues, the data gathered could not be considered accurately gained and a further phase of testing would be required after fixes were put in place.

5.5 Fixes

Even though the first round of testing resulted in a failure to export reliable data the data recording method worked.

As problems in the stability of the networking had meant that testing could not be achieved in any effective way, resolving the connection issues was the core priority. This meant going over the network solutions and backend setup to make sure there were no glaring errors, as well as tidying up the existing state machines that handled connections to the Photon server.

One specific bug caused one of the testers to constantly crash out of the application on tests being initiated. Isolating the cause of this proved to be near impossible and would have required a huge amount of time from the tester, so as this issue was solely occurring on their computer they were not included in later tests.

5.6 Testing Phase 2

Having learnt lessons from organizing the first phase of testing, the second phase used a smaller group of five testers. This change streamlined the process of getting the tests set up and the time required to get all the testers ready between tests.

In this second phase, all of the testers managed to connect to without any of the prior connection issues. Tests were also able to be completed without disconnections or severe latency issues. Outputting the data worked as expected and consistent data was able to be extracted.

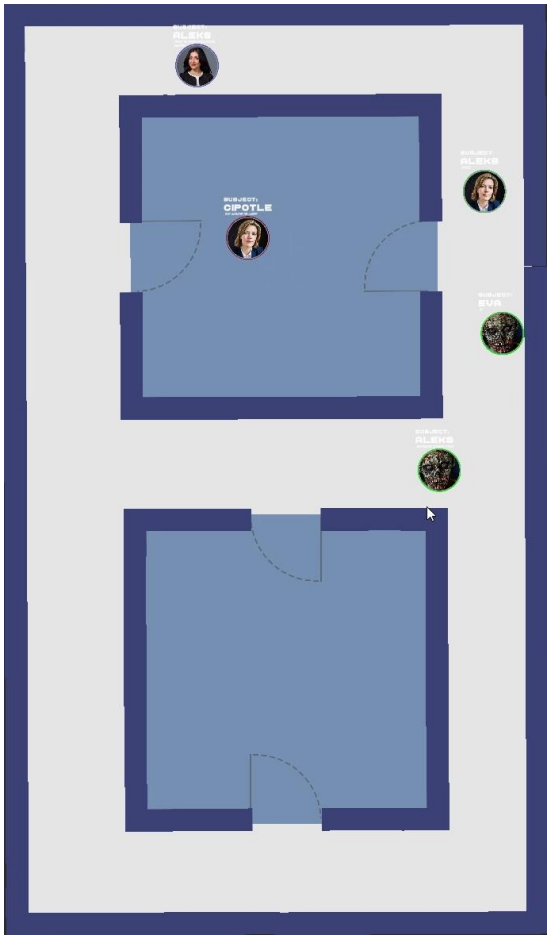


Figure 23 Test in Progress

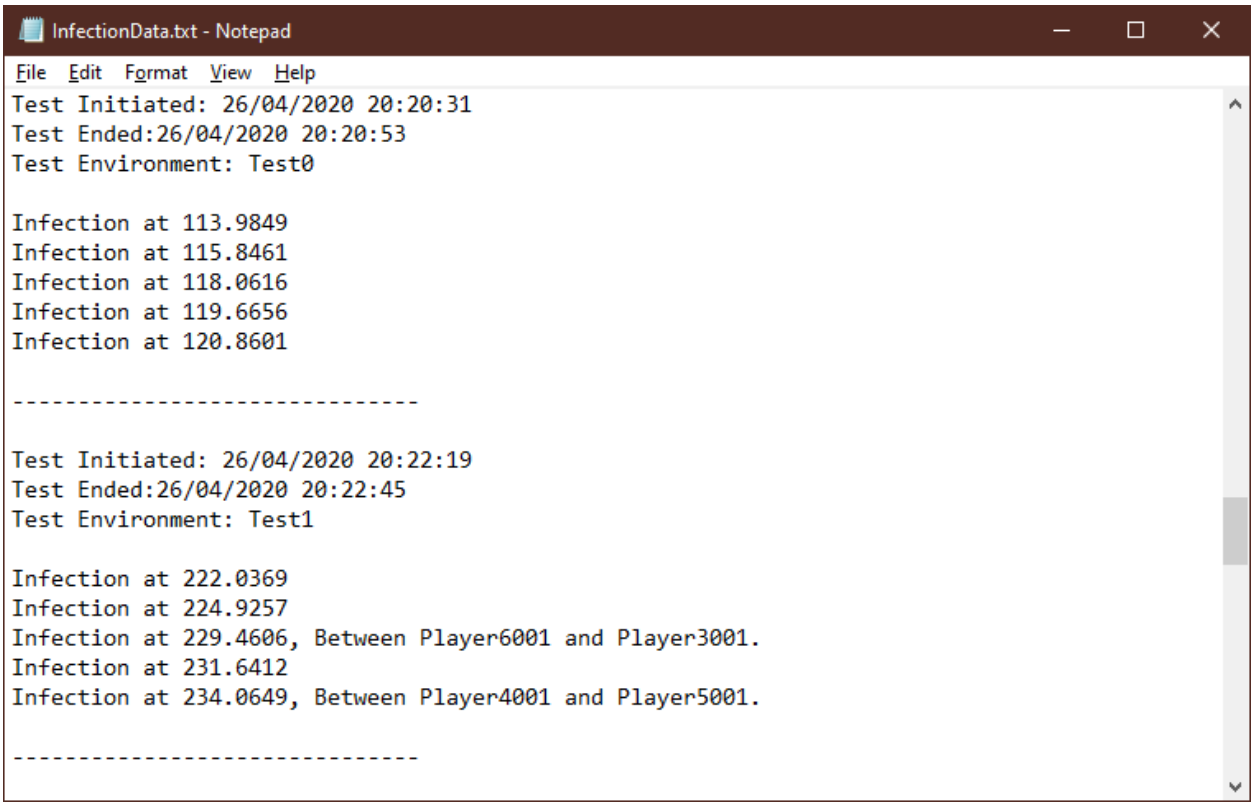


Figure 24 Example Data Output

Section 6 – Evaluation

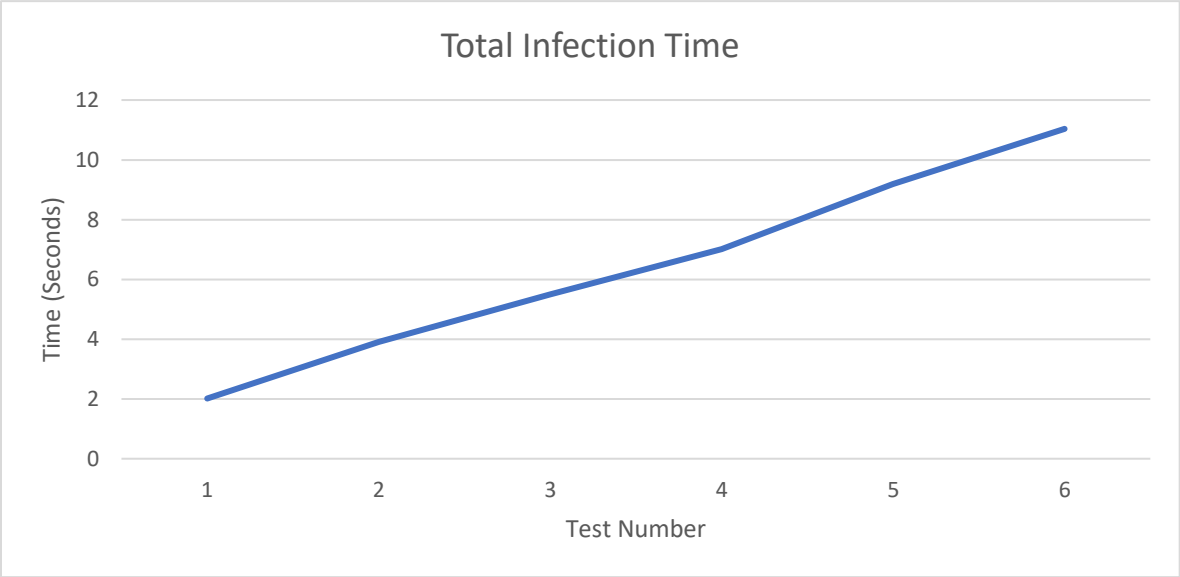
The aim of this research was to create a product that was able to log data on the transmission of infection between players within environments of increasing complexity. This was successfully carried out.

There were two forms of data gathered from the tests carried out. The main focus was to record data on the time between player infections and the time between initial infection and final infection events. The secondary data acquired was a video of the player actions during the test. By analyzing both, there is sufficient material to not only look at trends in infection times but also to look at how the behaviour of testers affected those infection time.

Due to restrictions on the availability of testers and the limited number of test phases that could be accomplished, only one data set per test could be achieved. This means that the data acquired is qualitative instead of quantitative, however, it demonstrates the capability of the product to be able to produce quantitative data with more test phases and a wider number of testers.

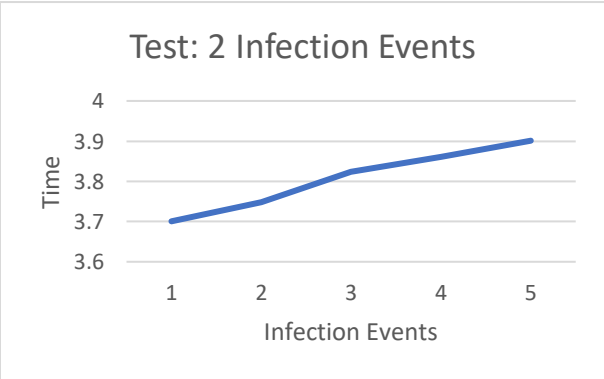
The product managed to consistently log the timings of infection events from the initial random player infection to the final event of player-to-player transmission. The first six tests were designed to gather data on environments that become more complex by adding rooms at *2(n)*, with

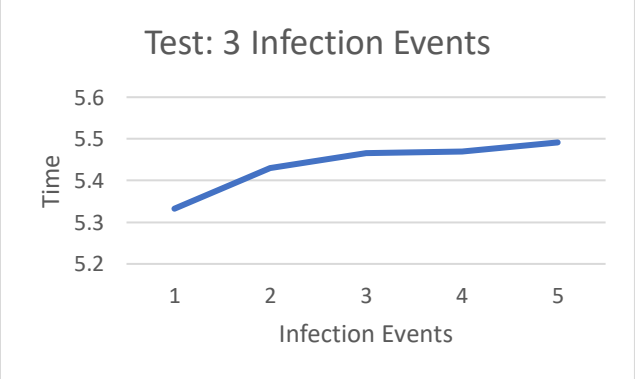
n being the number of rooms. The first test having no rooms to act as a baseline to look at how long it would take to achieve total infection with no environmental complexity.



As the graph shows, the time it took to reach total infection, for the most part, moved consistently with the scaling of the complexity of the environment in the first three tests, but in the final three tests the infection times were faster than expected. Given that the environment was scaling at a constant rate, total infection occurred faster than expected. This indicates that player infection was not significantly tied to the complexity of the environment, but more controlled by the behaviours within those environments.

The product also allowed for these to be recorded as a further avenue for research. The individual infection events within each test were also recorded. By comparing the events graphs for tests 2 and 3, we can see that infection transmissions can occur consistently over time but also in clusters of multiple infections in a very short amount of time.





Section 7 – Conclusion

The overall aim of this research was to look at the validity of using computer games as a tool in epidemiology. To prove this a multiplayer infectious disease simulation was created that was able to simulate contact-based disease transmission and that successfully demonstrates that computer games can be created and used as simulative tools. The final product was not only able to log infection events as they happened, but also able to output that data in a readable way for further data analysis, meeting the objectives set out in the project. By proving that meaningful metrics could be gathered from a game-based product, the research has successfully met its aims.

7.1 Future Works

During the test phase, it became apparent some parts of the product could be further developed and expanded upon:

- The complexity of disease transmission could be increased, with more methods of infection such as simulating airborne pathogen transmission. Elements of immunity could also be added.
- An increase in the number of test phases so that significant levels of quantitative data can be collected.
- More comprehensive metrics could be recorded. Specifically, those to do with distances.
- Larger tests could be conducted with an increased number of people.
- Procedurally generating test environments based on configurable variables that increase or decrease the complexity of the environment.

References

Arthur, C., 2014. *The Guardian.* [Online]
Available at: https://www.theguardian.com/technology/2014/mar/27/google-flu-trends-predicting-flu
[Accessed 27 April 2020].

Brodkin, J., 2013. *https://www.dice.com/.* [Online]
Available at: https://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/
[Accessed 23 April 2020].

Burke, D., 2003. In: M. Smolinski, M. Hamburg & J. Lederberg, eds. *Institute of Medicine (US) Committee on Emerging Microbial Threats to Health in the 21st Century..* s.l.:s.n., p. Appendix E.

Craft, M. E., 2015. Infectious disease transmission and contact networks in wildlife and livestock. *Philosophical Transactions B,* 370(1669).

Exit Games, 2020. *Photon Engine.* [Online]
Available at: https://doc.photonengine.com/en-us/pun/current/getting-started/pun-intro
[Accessed 25 April 2020].

Feder, S., 2020. *Business Insider.* [Online]
Available at: https://www.businessinsider.com/plague-inc-creates-new-version-lets-people-save-planet-coronavirus-2020-3?r=US&IR=T
[Accessed 30 April 2020].

Gaudiosi, J., 2015. *Fortune.* [Online]
Available at: https://fortune.com/2015/03/03/epic-games-unreal-tech-free/
[Accessed 1 May 2020].

Higuera, V. & Pietrangelo, A., 2016. *Healthline.* [Online]
Available at: https://www.healthline.com/health/disease-transmission
[Accessed 25 April 2020].

Institute of Medicine (US) Committee on Emerging Microbial Threats to Health in the 21st Century, 2003. Microbial Threats to Health: Emergence, Detection, and Response.. In: Washington (DC): National Academies Press (US), p. Appendix E.

Malashniak, M., 2016. *N-iX.* [Online]
Available at: https://www.n-ix.com/unity-vs-unreal-choose-best-game-engine/
[Accessed 1 May 2020].

Marathe, M. & Ramakrishnan, N., 2013. Recent Advances in Computational Epidemiology. *US National Library of Medicine Nation,* 28(4), pp. 96-101.

Nicoll, B. & Keogh, B., 2019. The Unity Game Engine and the Circuits of Cultural Software. In: s.l.:Springer Nature, p. 35.

Panth, M. & Acharya, A. S., 2015. The Unprecedented Role of Computers in Improvement and Transformation of Public Health: An Emerging Priority. *Indian Journal of Community Medicine.*

Racaniello, V., 2008. *virology blog.* [Online]
Available at: https://www.virology.ws/2008/11/07/twiv-7-viruses-in-video-games/
[Accessed 30 April 2020].

Rivenes, L., 2017. *Datapath.io.* [Online]
Available at: https://datapath.io/resources/blog/the-history-of-online-gaming/
[Accessed 25 04 2020].

Unity Technologies, 2015. *Unity3D - Licenses.* [Online]
Available at: https://support.unity3d.com/hc/en-us/categories/201268913-Licenses
[Accessed 23 April 2020].

Viani-Walsh, D., 2016. [Online]
Available at: https://peoplepeasandpathogens.wordpress.com/

Yadav, S., 2009. *An Introduction to Client/Server Computing.* s.l.:New Age International Pvt Ltd Publishers.

Zhongwei, J. & Zuhong, L., 2020. Modelling COVID-19 transmission: from data to intervention. *The Lancet,* 20(4).

Zimmermann, K. A. & Emspak, J., 2017. *Live Science.* [Online]
Available at: https://www.livescience.com/20727-internet-history.html
[Accessed 25 April 2020].