OUTBREAK

User Manual

Table of Contents

What is Outbreak *3*

Starting the Game *3*

Controls *3*

Main Menu *4*

Game Characters *4*

Game Worlds and Overview Map *6*

Credits *7*

What is Outbreak

Outbreak is set in a postapocalyptic University of Bath setting, where humans have been infected with a new, zombie like variant of COVID-19 and are infesting the campus.

Your objective is to find a special item hidden within the Chancellor's building that contains the secret to ending this outbreak. Beware, however, that it is located in an area that is said to hold the world's longest infected people of this new variant. You may need some tools and help from other areas of the university to achieve your goal.

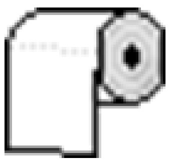
The journey begins…

Starting the Game

Controls

In this game, the primary controls are movement and shooting the gun. To move the player, simply use the arrow keys on the keyboard or WASD and the spacebar to jump. If the player has unlocked the double jump movement, press the spacebar twice to perform this. The faster you press the spacebar twice, the higher the jump will be.

To shoot the gun, press J



Collect the toilet rolls by walking into them to increase your final score.

Main Menu

Ready to start? Here we will talk about the menu system. When you start Outbreak, you will be presented a main menu that allows you to play the game, configure settings and quit the game.

Play

Press the Play button to boot up the game and it will automatically load up the tutorial, which is set in the Engineering building. When you complete this first mission, all other missions will automatically unlock and be available for you to play. There are three missions in total, including the tutorial. You may wish to complete all other levels before tackling the final level, but this level will be available to you upon completion of the tutorial.

Options

Here you are able to adjust the volume of the game according to your preference.

Quit

Press quit to end the game.

Game Characters

In this chapter we will introduce you to our main characters.

The Player Character

You will control a player wearing a hazmat suit in the game, designed to protect from the deadly viral spores. Our character is a student who managed to avoid being infected the initial outbreak while they were in janitor's room. They heard the screeches of infected students and quickly suited up to prevent any further risk of infection. Once the player left the room, they quickly realised they had left their all-important USB stick inside the Chancellor's building, which contains their dissertation and all other assignments.

Desperate to retrieve the stick, the player set out to find their way through campus to recover it but realised that they might need some extra gear to fight through the infected. As the player gets deeper into campus, they will find that everything is not as it seems….

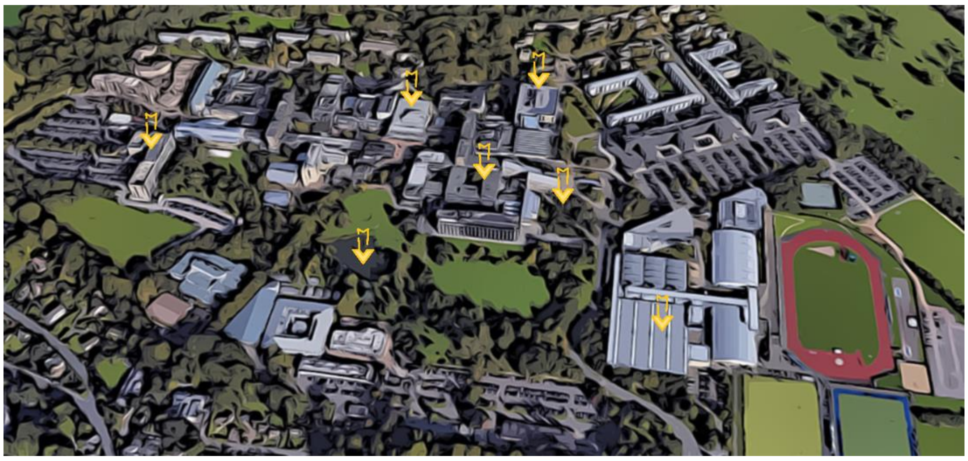
The Infected

There poor students were unfortunate enough to be caught in a massive viral explosion on campus recently. Now that their higher brain function has ceased to exist, they are determined to kill and infect anything that may cross their paths. Be very careful around these people as rumours have it that simply being within their vicinity can pierce through your hazmat suit and damage you.

Frank the Janitor

Finally, there is 29-year-old Frank. Being the longest serving janitor at the University, Frank is just starting to enter his midlife crisis. Sadly, he was also the first victim to be exposed to this dangerous new variant. Having been infected for a long time, he has developed into a special type of infected, where he is significantly stronger than someone who has been recently infected, like most students. You must figure out how to defeat this powerful infected as he is guarding the Chancellor's building, where your USB stick is located.

Game Worlds and Overview Map



The Outbreak Overview Map allows the player to select a level to play. Hover the mouse over the yellow arrows to see a description of each level, click on the arrow to start playing the level. Press E to go back to the start menu.

Engineering Building

The Engineering students have managed to invent a gun that sprays hand sanitiser, to cure infected academics. In order to fight back against the new virus variants, the player must enter the Engineering building, dodging infected, traversing the strange landscape of the Engineering world, and recovering the hand sanitiser gun. Upon completion, there is a magic blue door to teleport the player back into the map. Press E to interact with the door.

The Engineering Building is the tutorial level of Outbreak. This level is where the player can get used to jumping, moving, avoiding enemies, and collecting toilet paper, before finally collecting the hand sanitiser gun.

Sports Village

The Sports Village is full of trampolines that enable you to reach new heights. There is possibly a way to mimic these movements, maybe with the help of a special pair of shoes? Navigate your way through an area full of dangerous obstacles, such as deep water, moving platforms, and hordes of infected students to complete this level and a double jump will be available to you.

Chancellor's Building

Frank the Janitor is holed up in the Chancellor's Building, having taken the USB from the Lost & Found. He's been infected for a long time with the new variant, so he's developed into a highly dangerous foe. He'll shoot bubbles of infection at the player, as well as summoning other infected to assist him, so beware! Make sure you're prepared for a tough fight by collecting as many upgrades as you can from other areas of campus.

Frank is the final boss of Outbreak, and he must be defeated to retrieve a special item and complete the game. He will shoot projectiles at the player that deal damage, and which must be avoided. Be aware that there are breakable platforms so you must be on your toes throughout the whole battle!

Credits

Written By

Jack Tilney

Directed By

Lionel Ng
Declan Kelly

Software Developers

Jun Xiu Low
An Zhang
Timothy Gray
Timothée Keller
Jordan Layton

UX & Level Design

Lionel Ng
Declan Kelly
Timothée Keller
Jun Xiu Low
An Zhang
William Record

Animators

William Record
Timothy Gray
Timothée Keller

Project Management

William Record
Jack Tilney
Jordan Layton

Executive Unit Tester

Timothy Gray

Special Thanks To

Julian Padget
Andreasa Morris Martin
Tory Frame
Fahid Mohammed

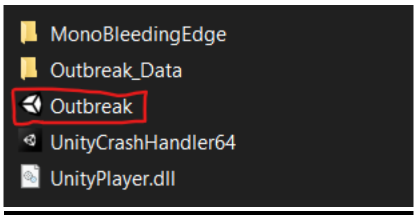
Outbreak Installation Guide

In order to install the game, the user should download the "DeliveryCompiled" branch from BitBucket at this link: https://bitbucket.org/tsg41/blue-two_u/src/delivery/ and open the folder DesktopApplication. They can also clone the branch instead if they prefer.

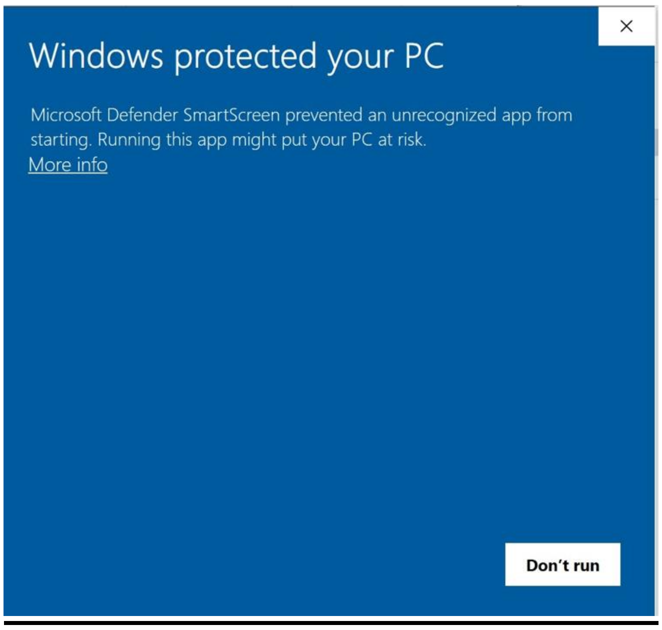
If this does not work please use the Delivery Branch and do the following:

Windows Installation

Inside this folder, run the "Outbreak.exe" file by double clicking, or right clicking and selecting "Open".



Additionally, an issue can be encountered where Windows will provide an error when attempting to run the game, displaying the following window.



Windows protected your PC

Microsoft Defender SmartScreen prevented an unrecognized app from starting. Running this app might put your PC at risk.
More info

In this case, to run the game, click "More info", then "Run anyway".

Dealing with Known Errors:

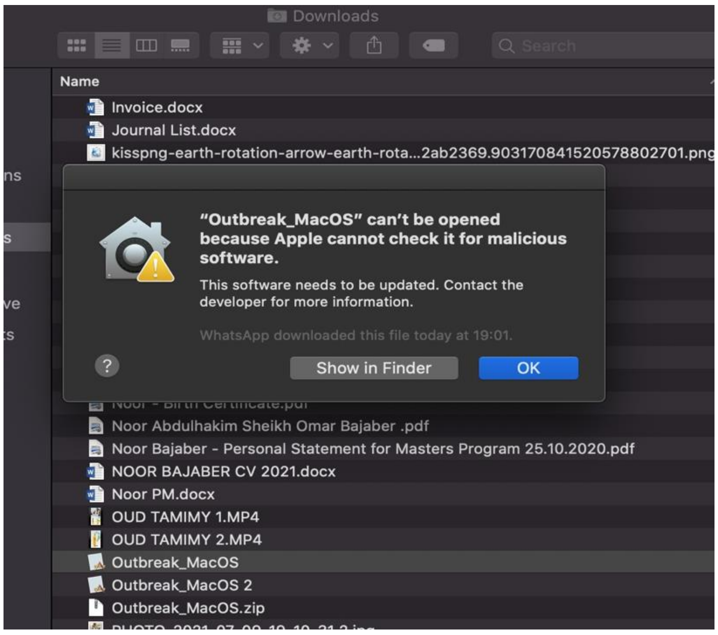
There is a known bug within the executable file for Windows which means that the graphic for the overview map does not load properly on occasion. If this occurs, you can ignore it as it does not affect game play. However, if you would prefer to see the game in its full working state, you can do so by building within Unity. This is done by completing the same steps to download the delivery branch from BitBucket, then open this project in Unity, click Assets > Reimport All Assets. Then select File > Build and Run. You will then be able to play the game with the fixed map.

MacOS Installation

Open the MacOSApplication folder and there should be an executable file to run.

If this does not work, open the Delivery branch and do the following:

In order to install the game on a MacBook or iMac, the user should create a build of the game from Unity because MacOS does not support exe files but you can still access it by cloning delivery and opening this in Unity and selecting Build and Run, as detailed above. The game should load up straight away from the main menu, however, a message may appear saying that “Outbreak_MacOS can’t be opened because Apple cannot check it for malicious software.”



Invoice.docx
Journal List.docx
kisspng-earth-rotation-arrow-earth-rota...2ab2369.9031708415205788027O1.png

“Outbreak_MacOS” can’t be opened because Apple cannot check it for malicious software.

This software needs to be updated. Contact the developer for more information.

WhatsApp downloaded this file today at 19:01.

Noor Abdulhakim Sheikh Omar Bajaber .pdf
Noor Bajaber - Personal Statement for Masters Program 25.10.2020.pdf
NOOR BAJABER CV 2021.docx
Noor PM.docx
OUD TAMIMY 1.MP4
OUD TAMIMY 2.MP4
Outbreak_MacOS
Outbreak_MacOS 2
Outbreak_MacOS.zip

If this message appears on the screen, please follow the guidance on https://macpaw.com/how-to/unidentified-developer-mac to configure your settings to allow the game to open.

Maintenance Guide

Folder structure:

Unity, the engine used to develop our game, offers a Project window under which all game objects can be organised. Our approach has been to separate these elements according to a few conditions:

- What *kind* of asset is it? An animation, background art, scripts, etc…
- Is it unique to a specific level? Some items are ubiquitous in the game
- What is its *function* in the part(s) it interacts with?

To answer these questions in a well-ordered manner, we have come up with a file structure that is as follows (inside Assets/_Blue_Two):

- *_Prefabs*
- *Fonts*
- *Physics Materials*
- *Scenes*
- *Scripts*
- *Temp*
- *Tilemaps*
- *UI Assets*

The first folder, *_Prefabs*, contains everything related to objects that span multiple levels, such as the Player_Model, trampolines, moving/destructible platforms, and interactable doors. The folder shares an internal classification with most of the other large files, which is that it contains subfolders named *Animation, Enemies, Environment,* and *Friendlies*. Some of these also contain subfolders, but their names generally are quite self-explanatory in their naming (*Idle animation* or *Walking*). Any assets that are to be added to the game, that will be used across multiple scenes are recommended to be made into the Unity class of prefabs, and added to the *_Prefabs* folder.
The folders *Fonts, Physics Materials, Temp, Tilemaps,* and *UI Assets* are all either small or serve a simple function so we will describe all of their functions here. *Fonts* keeps track of any different fonts used in the game, which currently only represents a single file. If any wished to be added to the UI or to other levels, their corresponding asset should be saved there. Similarly, *Physics Material*, only contains two objects as of now. Materials in Unity generally define the way certain items look, in particular how their texture or colour appear. This can be expanded to handle special cases of assets that are desired to stand out from the rest of a scene for example. The *UI Assets* folder is designed to hold anything concerning the User Interface, but has not as of yet been filled, since a robust UI is still yet to be implemented in the game. The team, however, decided that it would make sense for the category to warrant its own separate file hierarchy for any eventual features to be added. Because all of our levels have been made with the Tilemap builder, it also seemed appropriate to organise the multiple scene parts that are used by the Tilemap constructor into their own file. It is recommended for any extensions to level to continue using the same way of building, namely the tiles and terrain that are already a part of the *Tilemaps* folder. If any new tiles are to be added, a separation should probably be made, by creating new subfolders inside of *Tilemaps*. Finally, the *Temp* folder is used purely for ease of cooperation between different team members. Some assets can be partly created by one or two developers, but might still require a proper art or animation to be considered a functional object (usually a prefab). Conversely, the art team might be outpacing the production of new code, and the

Temp folder acts as a place to store *temporary* and *unfinished* assets. Since our situation generally saw the same kinds of team pacing, no subfolders have been made as of yet, but if *Temp* becomes a host to a large number of files, subfolders are also recommended.
The *Scenes* folder contains subfolders for every level currently in the game, and any additional ones should be added there. Currently, each scene has subfolders for anything that is used inside it, and that is not a prefab. Examples include animations that are unique to a level, art for certain platforms, or Non-Player Characters (NPC's). To this end, the subfolders always follow the same process; each level has inside folders for:

- *Animation*
- *Art*
- *Audio*
- *Characters*
- *Objects*

While the first levels do not yet contain certain features, such as any form of audio, the team thought it better to have a unified way of organising assets, so as to simplify future implementation. Whether for new scenes, or for expanding existing ones, it is advised that additional development should utilise this file structure to maintain linearity.
Finally, *Scripts*, manages every script that is in the game, and any additions should be added to the folder. It also has several subfolders for organisational purposes: *Animation, Camera, Enemies, Environment,* and *Friendlies*. While they could have been attached to their corresponding prefabs or various level objects, the team found that having a specific folder for scripts made finding different components simpler. This in turn allows for faster tweaking and implementation with both new and existing assets.

The code: Unity Scripts

The way this part is structured, is in the form of a list, in alphabetical order, detailing every script that is currently inside of the game, and explaining how it works and which Unity functionalities it makes use of. We would emphasise here, however, that features that derive from the game engine itself will not be discussed in much detail, because it can be accessed in greater depth from the official documentation online. Something that is also worth mentioning is the encapsulation of variables in unity. Public fields allow the developer to adjust some values inside the editor without modifying the script, and as such, in many cases variables have been created with this in mind. Common examples include health, speed, and damage.

- *BossAttack*: All current scripts that are named Boss_ only apply to the first boss Frank, but are named this way in case multiple bosses need to be implemented with similar base functions, and perhaps some specificities that will have more particular names. Frank has two kinds of "attacks" currently; he can shoot projectiles at set intervals, through the Update() and CheckIfTimeToFire() functions, but he can also spawn regular zombie enemies with SpawnZombie(). The latter attack is made to occur at fixed times, namely when the boss's health drops to predefined levels (50% and 25% at the moment).
- *BossHealth*: Most variables that are used inside the different boss scripts are defined here, such as health, attackTime, and invincibileTime. The Start() method links the slider healthbar in the scene to Frank's total health pool, currently fixed at 100. The

two functions Vincible() and Invincible() handle whether or not the boss can take damage at any given time, which is indicated to the player by changing the model's overall colour to red.

- *BossMove*: Very much in the same way as EnemyMove is described just lower, this piece of code simply relies on the Update() function and a Flip() one. Frank is made to roam the room from left to right, and in the case of proximity detection, done via a Unity Raycast, to one of the smaller zombies or a wall, he invokes Flip() which changes his direction.
- *Bullet*: This script is used for the bullet projectile inside the game. It uses two public variables (speed and damage) and a Rigidbody. The Start() function gives the bullet its initial velocity, launching the projectile, and then the OnTriggerEnter2D function, which is a Unity method, causes any enemies to take damage upon contact with the bullet, which is consequently destroyed.
- *EnemyHealth*: Herein lies the value of the enemies' health, currently set as a public float at 100. The script also has two other functions, the first of which, TakeDamage(), takes an int *damage*, and inflicts the value of *damage* to the enemy health. If it's health decreases to 0, the Die() function is called to destroy the enemy.
- *EnemyMove*: Any movement made by common enemies is handled by this script which gives them their speed, and movement axis (in this case the x-axis). While the enemy continually moves in one direction, if it detects any game object that is not the player at a certain distance, it will invoke Flip() and turn around. This script is used to have enemies move between two blocks on a level, and can be expanded to handle other axes, and more complex movement for enemies.
- *FallingPlatform*: The bulk of this code once again uses the Unity OnCollisionEnter2D method, which allows the developer to detect collision between rigidbodies. The function uses a coroutine (another Unity feature that allows one to spread out tasks through multiple frames instead of them happening every update) to spawn a new platform while it also destroys the current one after a delay, set in the scripts field variables. This happens as soon as the *Player* gameobject collides with the object using this code. The PlatformManager class is responsible for the platform spawning and it is another team-made script which is detailed further on.
- *GunPickupScript*: This simply handles the *Player* transition from the no-gun model to the with-gun one, also adding the relevant animation to the new object. This is done with the usual collision trigger detection, which destroys the gun asset when the player comes into contact with it, effectively making it seem like it has been picked up.
- *HoverOver*: The particular nature of this script lies in its inheritance, because it not only uses the MonoBehaviour superclass, but also the IPointerEnterHandler, and the IPointerExitHandler. These Unity features allows pointer position feedback to be used inside functions, as is the case with the two OnPointerEnter and OnPointerExit. The variable Text is displayed when the user's pointer enters a certain portion of the screen, and is hidden when the pointer leaves the area.
- *InteractDoor*: This is attached to the Player prefab and allows the object to use the doors that are currently used in the Engineering Building level. It uses a Boolean variable as well as input to double check entry conditions. Input is set to work when

the user presses the ‘e’ key, and the Boolean is dependent on the OnTriggerEnter2D collision detection function.

- *ItemCollector*: Any non-unique item collection should be handled by this code. At present, this is only used for toilet paper rolls, but the collecting is done with the usual OnTriggerEnter2D Unity function, incrementing the script’s counter variable every time the user encounters and collects the object.
- *MainMenu*: Here, we define functions for accessing the OverviewMap scene from the main menu, as well as making a QuitGame() function that lets the user close the application.
- *OverviewMap*: This script is very simple and only handles the clickable buttons on the OverviewMap scene. It uses Unity’s SceneManager to direct the user to the corresponding scene when a selection is made in the Map.
- *PlatformManager*: As mentioned in the FallingPlatform script, this class handles the respawning of new platforms. The IEnumerator function is needed for the coroutine, and signals the engine to Instantiate a new designated object at the chose location after a certain amount of time. All of these parameters can be changed and/or adapted if modifications are desired.
- *PlatformMover*: The speed of the platform is determined by a public variable, and the range of the platform movement is defined in the two conditional statements inside of the Update() function. We found 7 to be an adequate distance, but any desired modifications can be made here. The two other conditionals signal the object to change the direction once the two previously set coordinates have been reached.
- *PlayerHealth*: The Player model’s health is represented in this script by two public variables: a float and a slider. The latter is used in conjunction with UI art to make the green healthbar that is currently in the game. The Start() function gives the slider its base attributes, with it’s value being assigned to the health variable. The function also causes the EnemySearch method to repeat, making any proximity to enemies decrease the player’s health every set time interval. The collision detection function uses the Die() function to restart the current scene if the player comes into direct contact with an enemy. Finally, the Update() method handles death in case of health decreasing to 0, as well as falling.
- *playerMove*: This script should contain anything related to the Player’s movement, and as such it holds many variables, all of which are public. The Start() and Update() functions take care of the model’s animation, as well as starting the PlayerMove() function, which handles user input, and calls other methods accordingly. Jump() and FlipPlayer() are very simple functions that do what their name suggests, using Unity’s prebuilt AddForce and Rotate functions. Finally, OnCollisionEnter2D allows the developers to centralise any interactions that the player has when colliding with different objects. So far, Trampolines utilise this function, and Jump() also disallows mid-air jumps by detecting collision (or lack thereof) with the ground. In addition, we add conditionals that let the Player move with the platform when on top of a moving one.
- *Projectile*: Projectile works in a similar way to Bullet, but with inverted targets. Essentially, it is there for the first boss NPC Frank to be able to shoot the player, so the trigger function detects collisions with the player, and when one happens, the player’s

health is decreased by a set amount of 15. After collision happens, the object is destroyed in the same way Bullet is when hitting an enemy.
- *ReturnOverviewMap*: Another simple map script that is attached to a UI button, and that loads the Menu scene.
- *Weapon*: The Weapon class contains references to the bulletPrefab, and a position that sets the bullet spawn point. Update() waits for the detection of user input, and when the right button is pressed, it calls Shoot(), which instantiates a bulletPrefab. For details on the object, look at the *Bullet* script.
- *ZombieScore*: This very simple script keeps count of zombies killed as a temporary measure of scoring. In future implementations where score might be judged differently, this code could be adjusted and renamed to fit the new way of calculating play performance.

Game Objects:

Now that all scripts have been listed, we will go through the major game objects and how they interact with the code, as well as with each other. For the most part, we will be talking about the prefabs here, since most other objects are simply built through Tilemap and do not have any associated C# components.
- *Bullet*: The base projectile used by the player to inflict damage to enemies. It uses the Bullet script, as well as a circle collider for collision detection inside the code. It works as a standalone object, but is instantiated and called by the player functions in the *Weapon* script. So far, the Player model gathers user input, and when the fire button is pressed, it calls the Bullet prefab.
- *Zombies*: This is the default object when creating enemies for the game. So far, they have an attached animator for effects, such as the green particles that radiate around Frank, the first boss (although this is not used for all zombies). Also, the two scripts *EnemyHealth* and *EnemyMove* are, as expected, attached to the object and are where most of the changes to enemies are made. If healthbars are to be implemented for zombies for example, they should be added to the existing health script inside the enemy prefab. In the case of new action features (shooting, vomiting, etc…) it is advised to create a new script and connect it to the prefab. This way, if different kinds of zombies are to be implemented, they can be fabricated by the base asset that already contains the code common to all enemies.
- *Entrance Door*: Navigating from the OverviewMap scene to the different levels functions through these doors, that are as of the writing of this guide all different instances of the same object. It relies on the *InteractDoor* script that is attached not to the door but to the player. It uses a Unity tag to recognise the asset when detecting collision. Any modifications that are wanted should take into account both the door prefab and the code inside the player one.
- *Falling platform*: Most of the object's functionality is dependent on its corresponding script, which is explained in more detail earlier on, but this was made into a prefab to be used throughout level two as well as the boss level, and any future modifications should most likely be addressed inside the object's script.
- *Player*: There are currently two versions of the player prefab, which represent the base model with and without the gun that is acquired at the end of the first level. The

difference between the two is most of all in the animations used, but there is the weapon script that also is not shared between the two of them. Other than that, the prefab has attached to it the most amount of scripts out of any asset currently in the game. Not only does it contain all of the code named after the player model, but also the *Weapon*, *ItemCollector, ZombieScore,* and *InteractDoor* for example are all used by the object. In addition, a lot of the used scripts have direct consequences on other objects inside the game. *Weapon* invokes Bullets that in turn affects enemies, but the *playerMove* also dictates how the player interacts with moving platforms, and trampolines. We recommend most of the future code to be tied to the player model, if possible, because it makes code management, and particularly variables, easier.

- *Trampoline*: Most of what this prefab does is actually located inside of the *playerMove* script, sending the playing flying on collision detection. At the moment, these are used in second level only, but they can be extended to other levels if appropriate.
- *Frank*: The first and, as of the writing of this guide only, boss in the game. Important to mention is that as soon as further bosses are implemented, the team might want to reflect on which of them should be considered the "final" one. Since the game was always intended to be played in any order the player decides, one of the bosses must be deemed as the correct one to drop the final *USB stick* item. This mention aside, Frank's prefab works very similarly to the Player object but with enemy scripts. He has his code for health, movement, and interactions with other zombies, but he also has *Projectile* which invokes bubbles that can damage the player.
- *USB stick*: The current end objective of the game's storyline, this object simply drops when the player kills the last boss (currently Frank) and launches the end game screen.