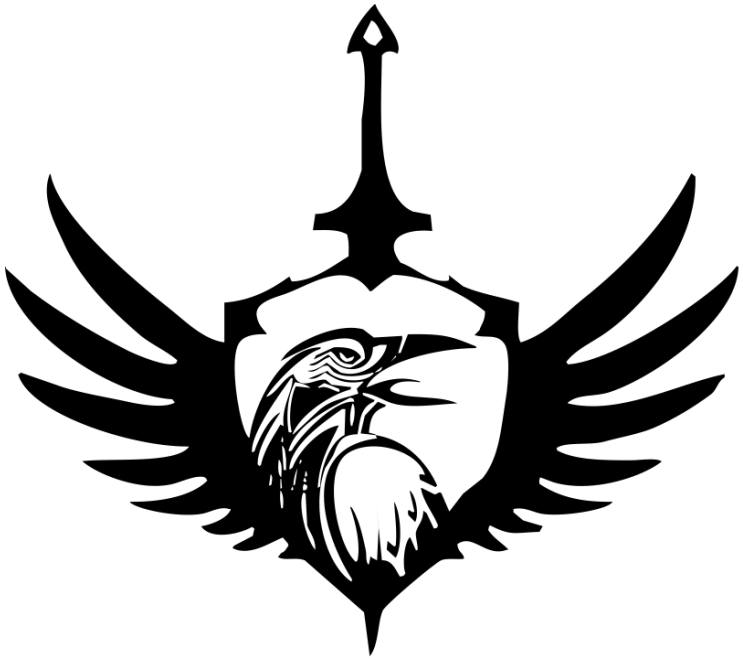


Technical Document and Risk Assessment

Authors: Christopher Foster & Aidan Murphy

Team Members: Leah Gimbutas, Gavin Chambers, Jake Seide, Anders Lindberg, Brittany Ollendieck, Jack Tenda

EGD220-06

Sprint 3

Table of Contents

Development Environment

Unreal

Our team will be using Unreal for this project for several different reasons. To start with, every member of the team is looking to expand their knowledge base to a new engine to increase future versatility in our careers. This does come with a few drawbacks with a lack of experience being at the forefront. This issue, however, is mitigated by all the resources (multiple professors, mentors, and strong documentation) we have available to us when questions do arise.

We also chose unreal for the flexibility it gives us in our development pipeline as a result of its blueprint system. The blueprint system has many premade modules that allow us to hit the ground running rather than starting from scratch in most cases as we would in Unity. Secondly, the visual scripting aspect of the blueprint system allows designers to implement smaller features on their own and adjust values for testing to lighten the load on the programmers.

One of the final reasons we chose Unreal was for its networking flexibility and its build system. Unreal has a multitude of built-in features to help with networking such as variable replication options between client and host. Additionally, the build system makes it incredibly easy to make different build for clients versus server builds.

As we are making a dedicated server for our game, the entire team will be required to use the source version of the engine. This can be obtained from the Epic Games GitHub page through git source control or through a download of the repo through Chrome. For more information on how that can be set up please refer to our repositories README file.

Difficulty: Medium

Using Unreal's Build System

Unreal provides a build system that allows for the game to be easily built on different platforms with easy to set up client and server configurations. Building should only be done by those using the source version of the engine.

Windows client build configuration:

1) Build
    a) Do you wish to build: Detect Automatically
    b) Configuration: Development
    c) No advanced setting changes
2) Cook
    a) How would you like to cook the content: By the book
    b) Cooked Platforms: Windows
    c) Cooked Cultures: none selected
    d) Cooked Maps: All maps that can be experienced by a player
    e) No DLC Patching settings or advanced settings required
3) Package
    a) How would you like to package the build: Package & store locally
    b) No other changes necessary
4) Archive
    a) Do you wish to archive: false

5) Deploy
    a) Do not deploy
6) Launch
    a) No changes necessary

Windows server configuration:
7) Build
    a) Do you wish to build: Detect Automatically
    b) Configuration: DebugGame
    c) No advanced setting changes
8) Cook
    a) How would you like to cook the content: By the book
    b) Cooked Platforms: WindowsServer
    c) Cooked Cultures: none selected
    d) Cooked Maps: Only maps where two or more players can interact
    e) No DLC Patching settings or advanced settings required
9) Package
    a) How would you like to package the build: Package & store locally
    b) No other changes necessary
10) Archive
    a) Do you wish to archive: false
11) Deploy
    a) Do not deploy
12) Launch
    a) No changes necessary

For information regarding deploying a server build to the Google cloud Services platform, please review the Server Workflow document

Google Cloud Services

We will be using Google Cloud Services to host our dedicated server. One of the major reasons for choosing GCS over other services like AWS was that first time sign ups get $300 free credit as well as the relatively low pricing of the VM instances. GCLS is very flexible in how we can set up our server regarding hardware and software making it easy choose something that fits our needs. More Information on how these fits into the development pipeline can be found in out repositories README file.

Difficulty: Medium

Git

Our team will be using Git for version control with our repository hosted on RedMine. Since several members of our team are familiar with Git, we have been able to implement branching as we add in new features. This allows for a release build of the game to be available while we work on the development build.

Difficulty: Medium

Overview of Branching

- "main" Branch
  - Branch meant for stable builds and the finished/polished product.
- "dev" Branch
  - This is the work in progress branch where all the unstable code and unfinished assets go.
  - Code for specific features should not be altered in this branch.
- "feature" Branches
  - Feature Branches are created off the dev branch to work on a specific feature.
  - These branches are mostly for the programmers or designers to work on specific game features (UI, Character Selection, Movement, etc.).
  - The programmers/designers should push their work to the feature branch that they are working on until that feature is complete.
  - Once a feature is complete it should be merged into the dev branch.

Basic Git Workflow

1. git status (Checks for any changes on your local machine)
2. git fetch (Checks for any changes in remote repository)
3. git pull (If fetch shows changes, pull the changes to your local machine)
4. git status (Checks for any changes on your local machine)
5. git add <file/folder name> (Stages files for commit)
6. git commit -m 'message' (Commits files to be pushed, add a commit message so you know what was worked on)

   Example commit messages:

   "Adding README.txt"
   "Adding move function to players
   "Adding playerSprite.png to Assets folder"

7. git push (Pushes files to the remote repository ON YOUR CURRENT BRANCH)

Further git workflow information can be found in the repository's README.txt. Workflow information contains different steps for the different disciplines (aka. Artist, Programmer, Designer)

Delivery Platform

Main Delivery Platform

Bodysnatchers will be developed for PCs using Unreal's build system that has preset options for windows platforms. We will also need to make a windows server build with a similar preset option. While it will be mainly developed using WASD controls, the game would ideally support a controller with at least one joystick. This will be relatively easy using Unreal's input system.

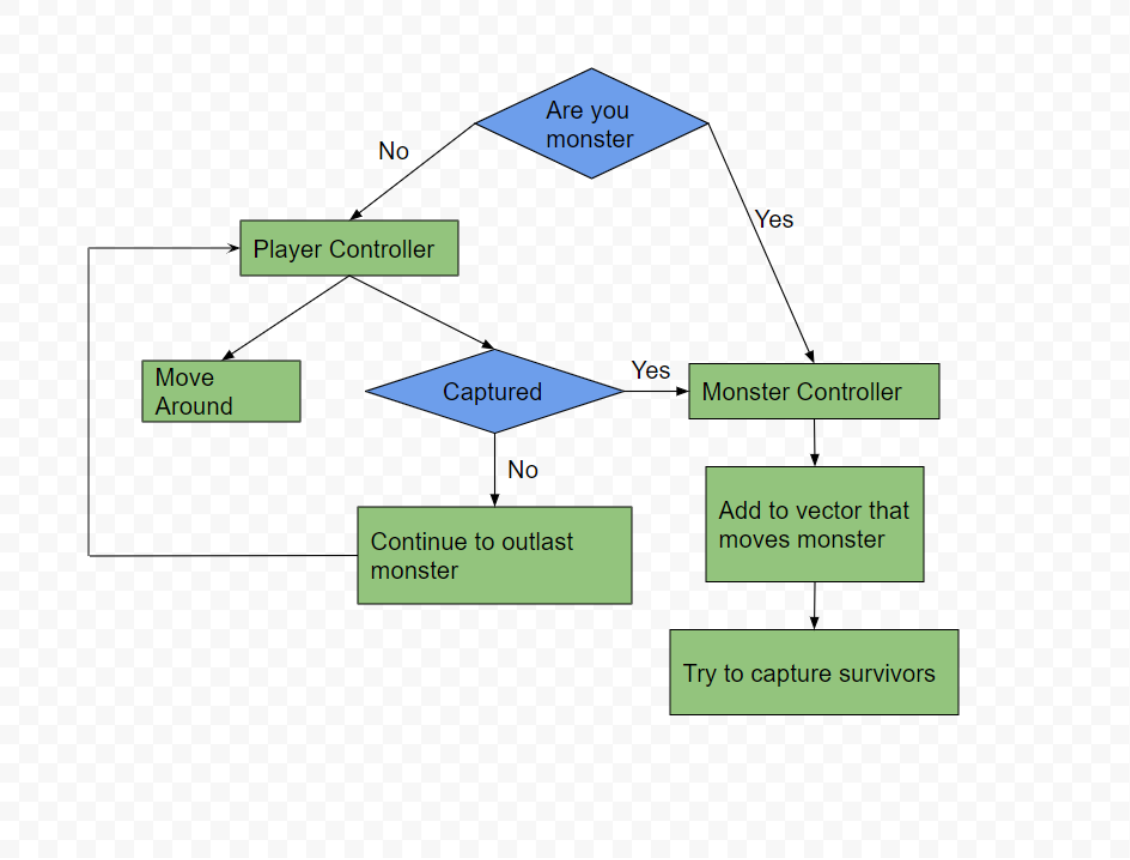
Difficulty for Main Delivery Platform: Low

Other Delivery Platforms

While we be developing for PCs mainly, Unreal also supports different platforms like android, iOS, and other platforms this will be relatively easy to build with the build system and will only require minor UI adjustments and input adjustments.

Difficulty for Other Delivery Platforms: Low - Medium

Logical Flow Diagram



Game Mechanics and Systems

UI

All UI in the game will be done using UE4's widget system. The widget System is easy to use and is very easy to mold into what we want it to do. Additionally, it allows UI to be implemented in a modular manner meaning things can be added and removed easily without changing large amounts of code or widgets.

Mechanics

Camera & Controls

Bodysnatchers is, by design, able to be played with just a joystick, meaning that the controls are very simple and only require 2 axis motion. This will be implemented using WASD controls. This will be extremely easy to implement in Unreal using its default character controller blueprints. This will be the same for the both the monster and the survivors. Both players controlling a survivor and players controlling the monster will have an action button.

The camera for this game will be an Isometric view fixed relative to the player's position. This means the player does not need any further controls to manage their view. The only difference between the monster and survivors is that the monster will have a wider field of view.

Difficulty: Low

Monster Movement

The monster in this game will ultimately be controlled by multiple players at the same time. This is a stark contrast to the survivors who are controlled on an individual basis (1 player per survivor). When one player tries to move north and one player tries move south, the monster will not move at all. In other words, the players movement vectors are added and normalized. The math portion of this is relatively simple, however, Unreal does not allow multiple player controllers to control a single object. We plan to achieve this affect by hiding the players in game bodies while sending their movement vectors to the Monster to interpret.

Difficulty: medium

Systems

Voice Channels

One key aspect of this game will be the voice communication between players, specifically those controlling the monster. The built in Voice chat subsystem for UE4 relies on Using the Steam Advanced Sessions subsystem. Because we will not be using steam, we will not be able to use this and will need to find another option. We plan to do this by using a plugin called VivoxCore. We are choosing to usie this plugin to simplify the already built in Vivox tools which can only be accessed in C++;

Difficulty: medium

Game Clock

There will be a count down clock at the top of every players UI showing them how much time is left in the game. This will be relatively easy to implement as we can simply take the time elapsed each frame and add it to a running value which is then subtracted from the time remaining. UE4 has many built in features that make the UI implementation very easy to achieve.

Difficulty: low

Game States

Player States

During the gameplay there are two different states the player can be in. On is controlling the monster while the other is controlling a survivor. At the start of the gameplay, one player (of eight) will be chosen at random and made to control the monster and the others start as a survivor. At this point, the players can use the controls previously outlined for their respective roles.

Start Menu

Once opened, the game will load directly into a cutscene that ahs been rendered into an mp4 file. On this screen, there will be a button for:

- Play Game
  - Brings the player to the *Lobby Screen*
- How to Play
  - Brings player to *How to Play Screen*
- Credits
  - Brings player to *Credits Screen*
- Exit
  - Closes application

These Buttons will appear after a preset amount of time that has been synced with the animation as well as a song that will be played looping in the background. Both the How to Play and Credits screens will be made as widgets that are overlayed on the Menu screen while the lobby screen will have its own game state and screen.

Lobby Screen

The lobby screen will require a connection to the internet and will connect to a predetermined IP address, that of our dedicated server. If the player does not have an internet connection, they will not be able to get to the lobby and will boot back to the main menu. All players that join the lobby will join the same voice channel so everyone can hear each other. There will be a ready button that toggles their player to ready and will change color accordingly (red/not ready, green/ready). Once every player in the lobby is ready, they will all be sent into the game.

Game State

As the players enter the game, they will be given their roles at random (with only one person as the monster) as well as their respective voice channels(monster/survivor). After the game has begun, the clock will start ticking and can be seen on the player HUD. The speed of this clock can be increased when a player puts a battery in one of five generators around the map.

Generators
Of State – In this state the generator will have no effect on game clock speed. Additionally, it will produce no sound and have minimal light slight on its model. Needs a battery to be turned on.

On State – In this is state the generator will cause time count down 15% faster. Additionally, it will have the lights on top turned on as well as the front facing light. It will also make a sound as it transitions to this state and will play a constant "low hum" sound that can only be heard from a distance (effect achieved using UE4 sound attenuation settings).

Design Pipeline

The ideal pipeline for the designers is for them to be able to work in engine, implementing features where they can, testing different values when necessary for testing, and documenting features for the programmers to implement. This will be supported by the multitude of features in Unreal, like blueprints, that reduce the amount of coding required to implement features. Additionally, designers will have access to the Google Cloud Services allowing them to make server builds for testing different features.

More specifically the process will start with designers documenting the various features they want in the game. If a designer feels like they can implement it themselves, they will start by communicating to programmers what the feature is and that they will implement it. Once that is done, they will make a branch off "dev" with "git checkout -b feature/[featureName]" and push it with "git push –set-upstream origin feature/[featureName]. Any work on this branch will stay on this branch until the feature has been implemented. Once it is in a functional state, they will commit their changes with "git add <filename>" and "git commit". They will then switch to the dev branch with "git checkout dev" and pull the most recent work with "git pull". Next the will checkout the previously made with "git checkout feature/[featureName]". Then they will need to merge dev into their branch with "git merge dev", if any problems or "conflicts "they should notify the programmers. If no conflicts arise, then they can push their changes to their branch. After that they can create a pull request on GitHub so everyone is on the same page before merging. After merging, they can create a server build if necessary.

Art Pipeline

Like the designer pipeline, the artists will also use the branching pipeline to implement features where they need to (see designer pipeline for more information). Prior to implementing an art asset artist should connect with programmers to see if there are any special technical requirements for the asset to function in engine. Once both disciplines are on the same page artists can implement the feature, making sure to follow the branching pipeline, eventually creating a pull request.

Milestones

Sprint 1

- Repository was made and set up with a README.txt file
- Physical Prototype created and tested
- Unreal Project Created
- Networking research
- Basic Networking Prototype
- Engine pipeline

Sprint 2

- Dedicated server research and prototype
- Concept art for direction/theme and Art Bible
- Body Snatchers concept iteration

Sprint 3

- Implement basic dedicated server
- Iterate on Art Bible and art direction
- Monster movement with basic visuals
- Player game test environment