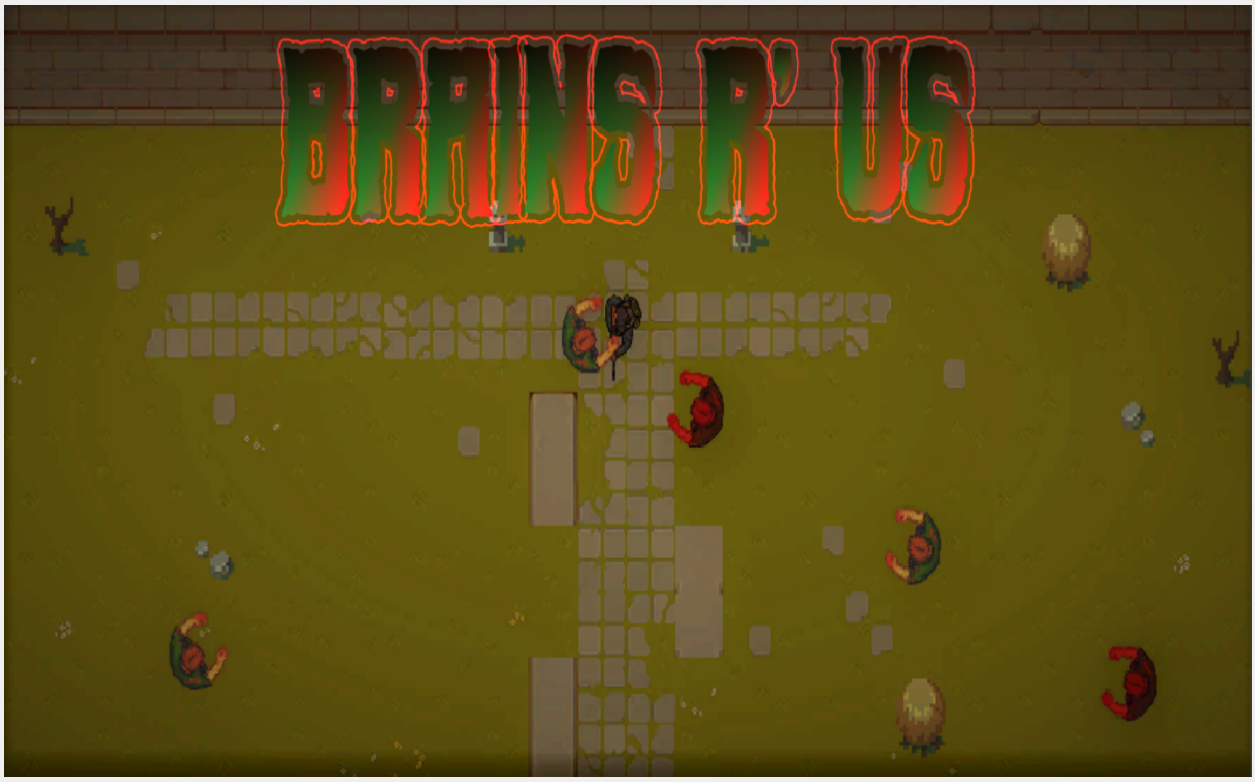
FINAL VIDEO GAME REPORT
BRAINS R'US

By: Saurjya Mukhopadhyay, Vikhyath Avantsa, Blake Berry, Lucas Song, and Nish Patel



Overview

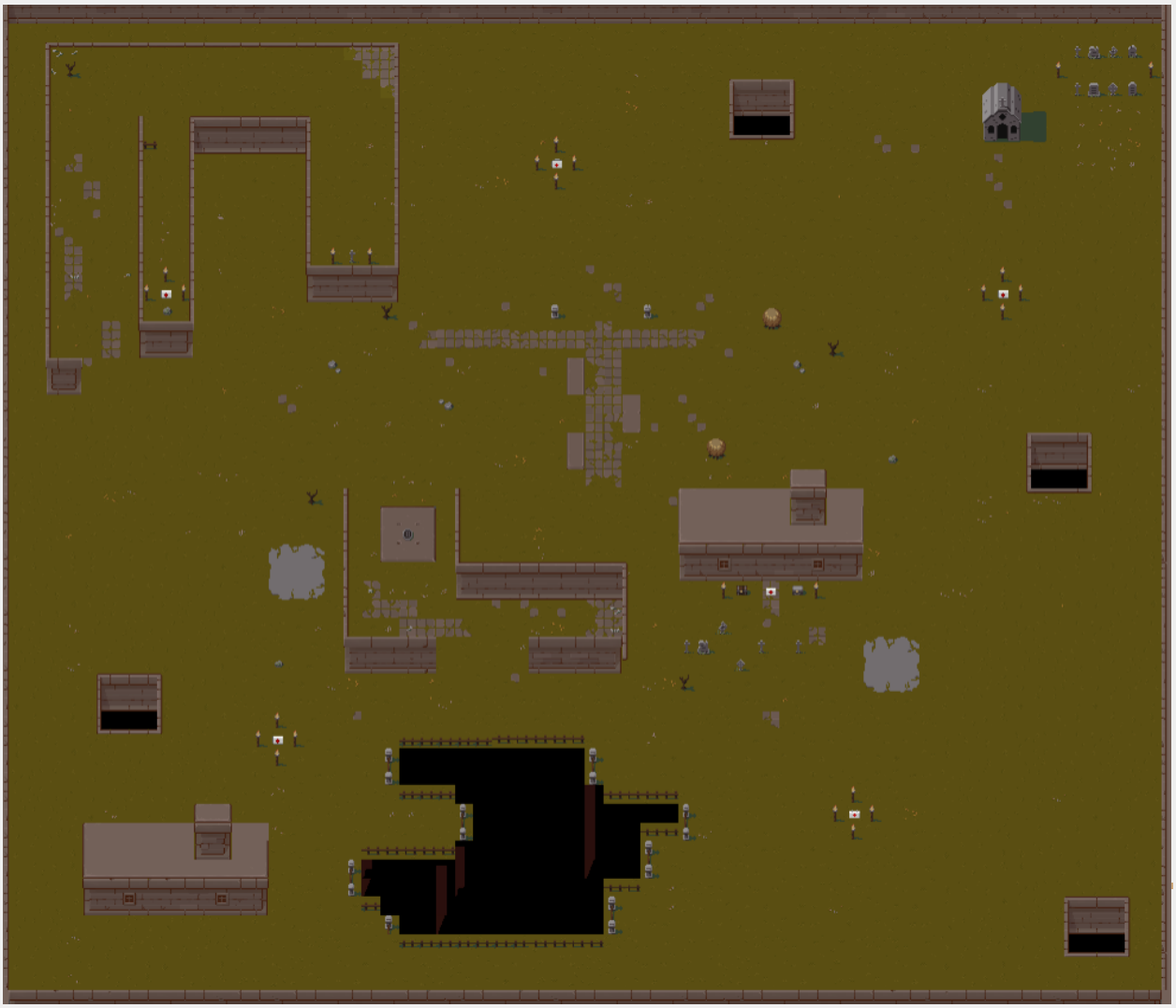
Brains R' Us is a 2D top-down zombie survival game created for all ages. In our game, the player will want to survive wave after wave of zombies by shooting bullets at them. The more zombies you kill, the higher your score!

Game Description

Levels

Our map is a desolate, ruined village. Nature has overtaken much of the land, and many buildings are nothing but crumbling walls. As a result of a supposed biohazard, the area was closed off, with large walls constructed around the vicinity to keep *something* from getting out. Our player is left inside these walls to survive whatever lurks here...

The game is separated into three waves. The first two waves deal with basic zombies, while the last wave has fast zombies (faster than the player!). For demo purposes, the difficulty and time taken to complete these waves has been reduced. Our wave system is fully extendable though, using arrays to manage the different values like enemy type and spawning frequency, so we can seamlessly add more waves to adjust gameplay difficulty.



Characters

The characters involved in the game are the zombies that spawn and chase down the character, and the character is controlled by the player.

Gameplay

The game begins with the player being spawned into a map with zombies. The player is equipped with a gun and can use the gun to target the zombies and kill them. The player can run around and try to evade the zombies while also trying to shoot them. If the zombies get to the player before they are killed, the zombies can attack the player. If the player kills all the zombies, the player proceeds to the next wave, where more zombies are spawned. Throughout the game, the player can pick up medpack powerups to refill their health. When the player has reached 0 health, the game ends and the player's score is shown. Progress is saved between waves, so the player can pick back up on the wave they lost in.

Implementation

This section covers details on the internal designs and implementation of the game: code, algorithms, game engine and other miscellaneous tools used.

Game Engine

The game runs on the Unity Engine. This engine was chosen for its affinity toward 2D games, as *Brains R' Us* is a top-down 2D game. All assets (sprites, tilemaps, animations) were fully handled in Unity without the use of external applications/tools.

Scripts

Start Menu Scripts ***Functions in the scripts are written in sub-bullets***

Main Menu:

- Manages the options in the start menu
- Buttons for options to either “Play” or “Quit”
    - PlayGame(): starts the game -- called when pressing “Play” button
    - QuitGame(): closes the game -- called when pressing “Quit” button

---

Level Scripts ***Functions in the scripts are written in sub-bullets***

Ammo:

- Handles the player’s ammo count
- Decrements total ammo count when player shoots
- If the player has no ammo, pressing Mouse1 does not shoot bullets.
- Displays ammo count to UI under the player’s health bar

Bullet:

- Destroys bullets upon collision with other colliders
- Instantiates explosion animation when bullet hits colliders

Camera Follow:

- Controls camera to follow the player and their mouse
- Uses CameraPos object that follows the player’s mouse
- Camera will overall follow the player, but will move a little bit towards the mouse (CameraPos)
    - More dynamic view than directly following player’s head, creating more cinematic gameplay and helping the player view their surroundings more
- Uses map’s collider to keep camera from looking past the map (‘clamping’ the position to not go past the map boundaries)

Enemy Health:

- Manages enemy health (which is different depending on the enemy type)
- Handles destroying the enemies when health reaches 0

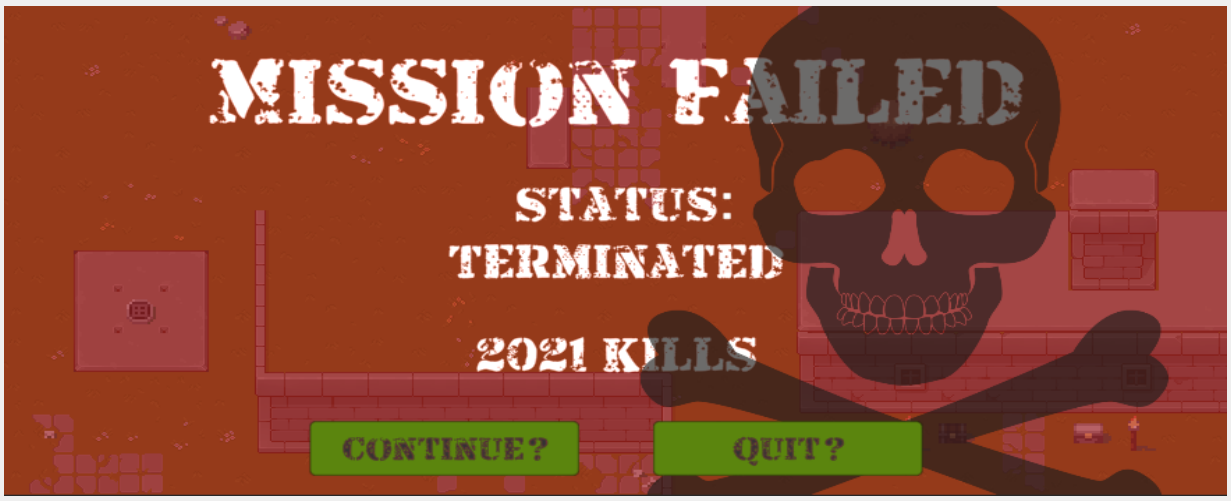
- When health reaches 0:
    - Plays sound indicating enemy is dying
    - Instantiates blood explosion
    - Destroys enemy and blood explosion objects
    - Increments player's kill count

Enemy Movement:

- Adjusts enemy AI rotation to face direction of movement (velocity)
- Configures AI by interacting with AstarPathfinding Project (attributions) scripts
    - faceVelocity(): sets rotation to face direction of path as new paths are calculated

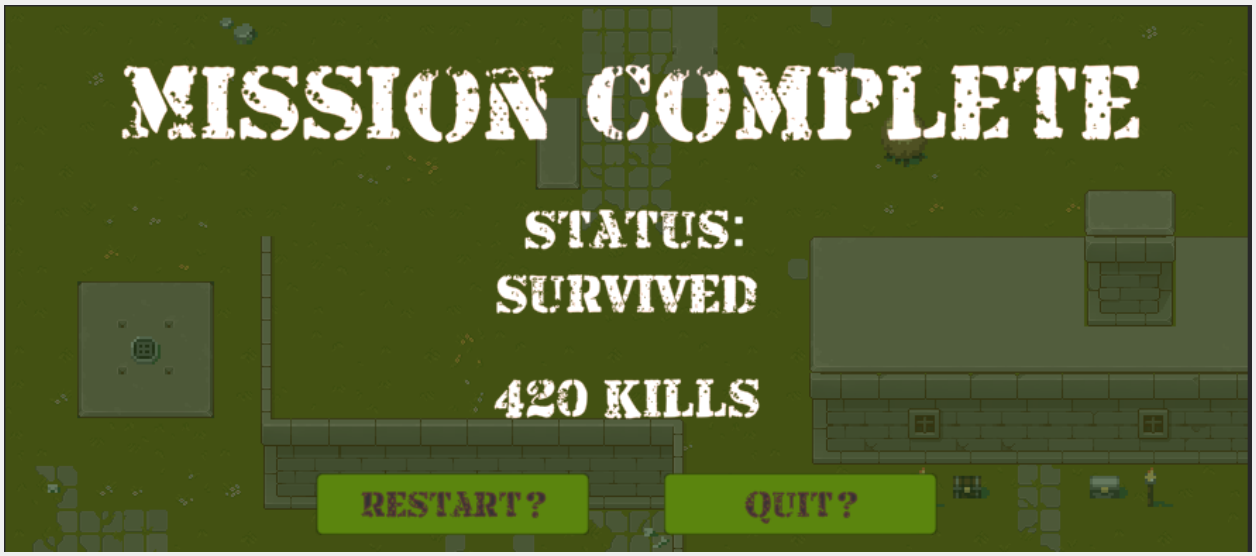
Game Fail:

- Handles displaying the Game Over screen when the player's health reaches 0
- Shows the player's kill count
- Buttons for options to either "Continue?" or "Quit"
    - EndGame(): Shows Game Over screen
        - Disables in-game UI (health bar and ammo count)
        - Enables game over screen, displaying final player kill count
    - Restart(): restarts the game from last wave-- called when pressing "Continue?" button
    - QuitGame(): closes the game -- called when pressing "Quit" button



Game Win:

- Handles displaying the Victory screen when the player kills all the enemies in the last wave
- Shows the player's kill count
- Buttons for options to either "Restart?" or "Quit"
    - EndGame(): Shows Game Over screen
        - Disables in-game UI (health bar and ammo count)
        - Enables game over screen, displaying final player kill count
    - Restart(): restarts the game -- called when pressing "Restart?" button
    - QuitGame(): closes the game -- called when pressing "Quit" button



Health Bar:

- Manages the health bar gauge in the UI to display the player's current health
- Color gradient to indicate player's health range
    - White: *practically full health!* (90-100%)
    - Green: *pretty healthy!* (65-90%)
    - Yellow: *little low...* (25-65%)
    - Red: *DANGER* (0-25%)

- Gauge slowly depletes as health goes down and fills up when regaining health
    - SetMaxHealth(int maxHealth): sets maximum possible health
        - Planned to add upgrades/armor to raise max health
    - SetHealth(int health): sets player's current health and updates UI health bar accordingly (as percentage of maxHealth)

Med Collect:

- Destroys medkits upon collision with the player
- Heals the player for 20 HP using PlayerHealth.Heal(20)

Player Controller:

- Controls player movement, aiming, shooting, and taking damage
- Player moves using WASD or arrow keys, aims with mouse and fires with left-click (fire1)
- Bullets colliding with enemies will reduce their health
- Cross hairs used to lock onto the target (zombie)
- Laser implemented for down sight firing
    - OnTriggerEnter2D(Collider2D other): if player enters enemy collider (trigger), take initial damage
    - OnTriggerStay2D(Collider2D other): player receives additional small damage each second they stay collided with enemies
    - Shoot(): shoots bullet when player inputs fire1 (assuming they have ammo)
        - Plays rifle sound
        - Instantiates bullet with rigidbody
        - Applies force to rigidbody based on player's aim

Player Health:

- Manages player's kill count across attempts (resets when surviving the last wave), health received from medkits and damage received by enemies
- Ends game if the health reaches 0
    - Heal(int health): adds additional health to the player's current health (limited to the player's max health)
    - TakeDamage(int damage): reduces health by specified damage number
        - Calls GameFail.EndGame() if damage results in 0 health

Spawn Meds:

- Manages spawning medkits on the map
- There are 6 designated medkit locations, marked by a diamond formation of torches
- When the total number of medkits on the map is 0, randomly chooses how many new medkits to spawn in (maximum of 6)

- Randomly decides which locations to spawn the medkits and waits for 3 seconds before spawning the next medkit (assuming there are more medkits to be spawned, based on randomly chosen number from previous bullet point)
    - IEnumerator SpawnKits(): handles spawning a random number (0,6] of medkits to spawn and spawns each at one of the 6 designated medkit locations (also chosen at random)

Wave Notifier:

- Blinks screen with a message when a new wave is beginning
- Outputs text "WAVE #" ~ ex. "WAVE 1"
- Sets UI to display text for current wave at the top of the screen (UI text appears after the initial blinking to avoid redundancy)
    - Display(int num): calls coroutine Disp
        - Used coroutine to access the WaitForSeconds function for creating the blinking effect
    - IEnumerator Disp(int num): blinks "WAVE #" text on screen and then displays the text in the UI
        - Manages UI and blinking text by setting different objects active and inactive

Wave Spawner:

- Fully manages the entire wave system
- Transitions between different waves, each with specific values
    - Programs the four spawners to randomly spawn *count* enemies at a specific *rate*
    - Instantiates enemies of specific type (we chose basic enemies for the first waves and faster enemies for the later waves)
    - Moves to next wave when conditions are met: *count* # of enemies are killed and wave intermission period has passed
        - Grace period between waves is 5 seconds
- SpawnWave(Wave wave): begins wave
    - Spawns specified number of enemies at specified rate
    - Finishes once all enemies are killed
- SpawnEnemy(Transform enemy): spawns enemies

    - Randomly picks one of the four spawn points and instantiates new enemy
- EnemyIsAlive(): checks if enemies are alive to signal next wave
    - Searches for enemies at constant rate (once per second)
- WaveCompleted(): determines what to do once the current wave is over (which is signaled when no more enemies are alive)
    - If not the last wave, move to the next
    - If last wave, signal victory by calling GameWin.EndGame(), giving the player the option to either restart the game or quit

Algorithms

A* Pathfinding:

A* is a graph-traversal and pathfinding algorithm that is useful for configuring enemy AI to follow a specified target (the player). This method uses nodes to create the map and applies qualities to nodes based on the terrain of the map (i.e. unblocked, blocked, source, target). Given the algorithm is often visualized using grids AND the useful "smartness" of the pathfinding, the solution was perfect for our 2D top-down grid-based map. We used a popular A* implementation for Unity called the *A* Pathfinding Project* by Aron Granberg (*amazing* project!). For our project, we had to adjust many settings -- including the node count and enemy movement -- to work with our map.

Tools

No additional tools were used for developing this game, aside from the native Unity Engine and imported assets (and also researching the Internet!)

Project Post Mortem

This section covers the groups' thoughts as a whole after the project has been finished (including what we were not able to get done and what we think we could have done better).

What tasks were accomplished?

The majority of the tasks that we initially assigned to ourselves were accomplished by the day of the final demonstration. The first tasks that we needed to finish were player character movement and player character shoot as well as creating a map. Once these tasks were done, we were able to add the smaller detail tasks, such as ambience sound and weapon sounds. The next big task we accomplished was the zombie movement, and after this had been done, the base of our final game was essentially built. Excluding the tasks mentioned below, all the tasks after (like the start menu and score system) were add-ons to make the game flow better and seem like an actual video game that someone would purchase.

What planned tasks were not done?

Certain animations like enemy attacks and movement were not done/incomplete. While it was mostly just potential ideas, aspects like an inventory system and armor/weapons were not realized (though we did not fully agree/commit to these features). We also did not implement limited ammo and pickable ammo, so pressing Spacebar will just refill your ammo; this was partially a balancing decision, though, to blend with the style of countless enemies vs. 1 player.

A personal idea (from the map creator) was having the player go down to the depths of the mysterious black hole at the bottom of the map. The player would descend deeper, both physically and metaphorically, to uncover what is going on in this closed-off region. Due to time constraints, this idea was scrapped, but the hole remains.

How did scrum work for you?

The scrum worked really well for our group after a slow start. For the first few weeks, we were unsure exactly on how to use the spreadsheet and we were also learning how to use the game engine as well. However after those initial weeks, the pace of our progress picked up and assigning tasks via the scrumsheet became easier. Overall, more of an introduction or some more documentation on how to use the scrumsheets would have been helpful, but once we figured out how to use it, our work flow improved drastically. We now understand why companies use scrum meetings and scrum sheets for big projects; it is a very organized way to assign work equally throughout the team and monitor the team's progress.

What would you do differently?

There are not many things that our group would have done differently; we feel as if we worked very hard on this project and we are proud of what we accomplished. However, as mentioned before, we did have a slow start in the first few weeks so if we had to do something differently, it would be to expedite our learning process in the beginning so we could start implementing the actual game a little earlier. This would have saved us from doing more of the work in the final few weeks, but we accomplished what we wanted to and are proud of the game we made.