Final Technical Report

Project Title: Randomly Generated Survival Simulation Game
Submitted By: Dennis Dao
Submitted To: Dr. Arunita Jaekel
Date Submitted: December 1, 2023
Demo Link: https://youtu.be/qxMqgqgrTtY
Built Version: https://dennis0802.github.io/files/The%20Western%20Trail.zip

Summary

This game development project was inspired by existing choose your own adventure games such as *Organ Trail* by *The Men Who Wear Many Hats* and Death Road to Canada by *Madgarden* and *RocketCat Games*. This game combines features from the two and puts it into a 3D environment. In particular, the features are ones that were interesting in one but missing from the other. The player's party must survive from their starting point at Montreal to their end point at Vancouver in a fictional post-apocalyptic scenario. Along the way, resources need to be managed wisely with the random nature of events on the road.

Design and Specifications

A design document was created to put my ideas for the game in writing during the winter semester. This allowed ideas to be tracked and potentially implemented later. While majority of the ideas were in place, some were discarded because of complexity and design conflicts. With the ideas in place, this allowed work to be done during the summer and fall semesters. Overall, the document lists the overview of the game, the target genre, technical specifications such as camera settings and game engine, what will be expected to be included in the project, details about the game world, and the user experience envisioned.

During implementation, Unity's built-in button system has been used as a type of player input, where the OnClick() is given the behaviour to perform. For example, selecting the New Game button calls for the main menu to be inactive, play a click sound, the file access component to be active, and to call the public AccessFiles method in the MainMenu script.

Database Management

To save the player's data, there were two options - the built-in PlayerPrefs or introduce a database into the system. A database was introduced since using PlayerPrefs requires key-value pairs, which becomes difficult to implement with save files. In addition, PlayerPrefs only allows strings, floats, and integer values.

For a database, SQLite was chosen since it is lightweight in setup complexity and resource usage. During an initial implementation of the database, precompiled binaries that came with Windows could be used to simplify the setup. In addition, it just needs to bind and works with its libraries. C# is an example of a language that can bind with it and work together with Unity. Initially, the queries were unparameterized. Noting the security issues, they were parameterized during the second version of the database queries. However, the build versions still exposed the database, requiring another revision to prevent users with basic knowledge of a DBMS/DB Browser to exploit. Figure 1 displays an example table of a Car ingame.

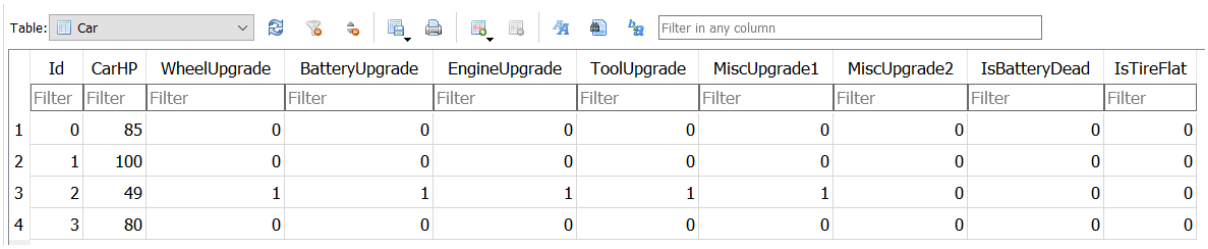

Table: Car

Filter in any column

	Id	CarHP	WheelUpgrade	BatteryUpgrade	EngineUpgrade	ToolUpgrade	MiscUpgrade1	MiscUpgrade2	IsBatteryDead	IsTireFlat
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	0	85	0	0	0	0	0	0	0	0
2	1	100	0	0	0	0	0	0	0	0
3	2	49	1	1	1	1	1	0	0	0
4	3	80	0	0	0	0	0	0	0	0

Figure 1. Sample SQLite database table in DB Browser for SQLite

During the third version of the database queries, the database was encrypted with a salted password using Python's bxcrypt module, C#'s Cryptography namespace, and SQLCipher. An object-relational model (ORM) was added to abstract majority of the raw SQL code in the scripts. Figure 2 shows a comparison between the two implementations – listed with '-' is the original query implementation while the ORM implementation is listed with '+'. The ORM model simplified the process and allowed the use of libraries such as LINQ to substitute as the SELECT query.

```
-            dbCommandReadValues = dbConnection.CreateCommand();
-            dbCommandReadValues.CommandText = "SELECT carHP, wheelUpgrade, batteryUpgrade, engineUpgrade, toolUpgrade, miscUpgrade1, miscUpgrade2 FROM
  CarsTable WHERE id = @id";
-            queryParameter = new QueryParameter<int>("@id", GameLoop.FileId);
-            queryParameter.SetParameter(dbCommandReadValues);
-            dataReader = dbCommandReadValues.ExecuteReader();
-            dataReader.Read();
-
-            wheelText.text = dataReader.GetInt32(1) == 1 ? "Durable Tires\nTires always last regardless of terrain." : "No wheel upgrade available to
  list.";
-            batteryText.text = dataReader.GetInt32(2) == 1 ? "Durable Battery\nBattery always has power." : "No battery upgrade available to list.";
-            engineText.text = dataReader.GetInt32(3) == 1 ? "Fuel-Efficent Engine\nEngine consumes less gas for more distance." : "No engine upgrade
  available to list.";
-            toolText.text = dataReader.GetInt32(4) == 1 ? "Secure Chest\nNo supplies will be forgotten again." : "No tool upgrade available to list.";
-            misc1Text.text = dataReader.GetInt32(5) == 1 ? "Travel Garden\nGenerate 1kg of food/hour." : "No misc upgrade available to list.";
-            misc2Text.text = dataReader.GetInt32(6) == 1 ? "Cushioned Seating\nParty takes less damage when driving." : "No misc upgrade available to
  list.";
-            int carHP = dataReader.GetInt32(0);
+            Car car = DataUser.dataManager.GetCarById(GameLoop.FileId);
+
+            wheelText.text = car.WheelUpgrade == 1 ? "Durable Tires\nTires always last regardless of terrain." : "No wheel upgrade available to list.";
+            batteryText.text = car.BatteryUpgrade == 1 ? "Durable Battery\nBattery always has power." : "No battery upgrade available to list.";
+            engineText.text = car.EngineUpgrade == 1 ? "Fuel-Efficent Engine\nEngine consumes less gas for more distance." : "No engine upgrade available
  to list.";
+            toolText.text = car.ToolUpgrade == 1 ? "Secure Chest\nNo supplies will be forgotten again." : "No tool upgrade available to list.";
+            misc1Text.text = car.MiscUpgrade1 == 1 ? "Travel Garden\nGenerate 1kg of food/hour." : "No misc upgrade available to list.";
+            misc2Text.text = car.MiscUpgrade2 == 1 ? "Cushioned Seating\nParty takes less damage when driving." : "No misc upgrade available to list.";
+            int carHP = car.CarHP;
```

Figure 2. Comparison of raw SQL and ORM code

Player-Generated Content

A feature of the game is the ability for players to create their own content – up to 45 of their own characters. These characters can be generated during gameplay when the event for a party member to join is rolled during the custom gamemode. If a custom party member dies during gameplay, they are placed into a separate table to keep track and cannot be re-rolled again.

Players can choose their name, outfit, color, perk, and trait. The name is restricted to 10 characters and must match the regex [A-Za-z0-9], allowing for most names to be input while avoiding potential injection attacks. Traits and perks can help or provide a challenge to the player to manage, such as conflict within the team or buffs during combat.

Graphics

While majority of the graphics have been imported from other sources (documented in the repository, mostly from *opengameart.org, pixabay.com, kenney.nl*), the skybox was created using shader graphs to create gradients for a midday, nighttime, and dimmed sky. The shader graphs allowed manipulating the shader as a new material to be created in the project, with normalizing and adding features to it. Figure 3 shows a sample of the skybox created and the shader graph to create it.

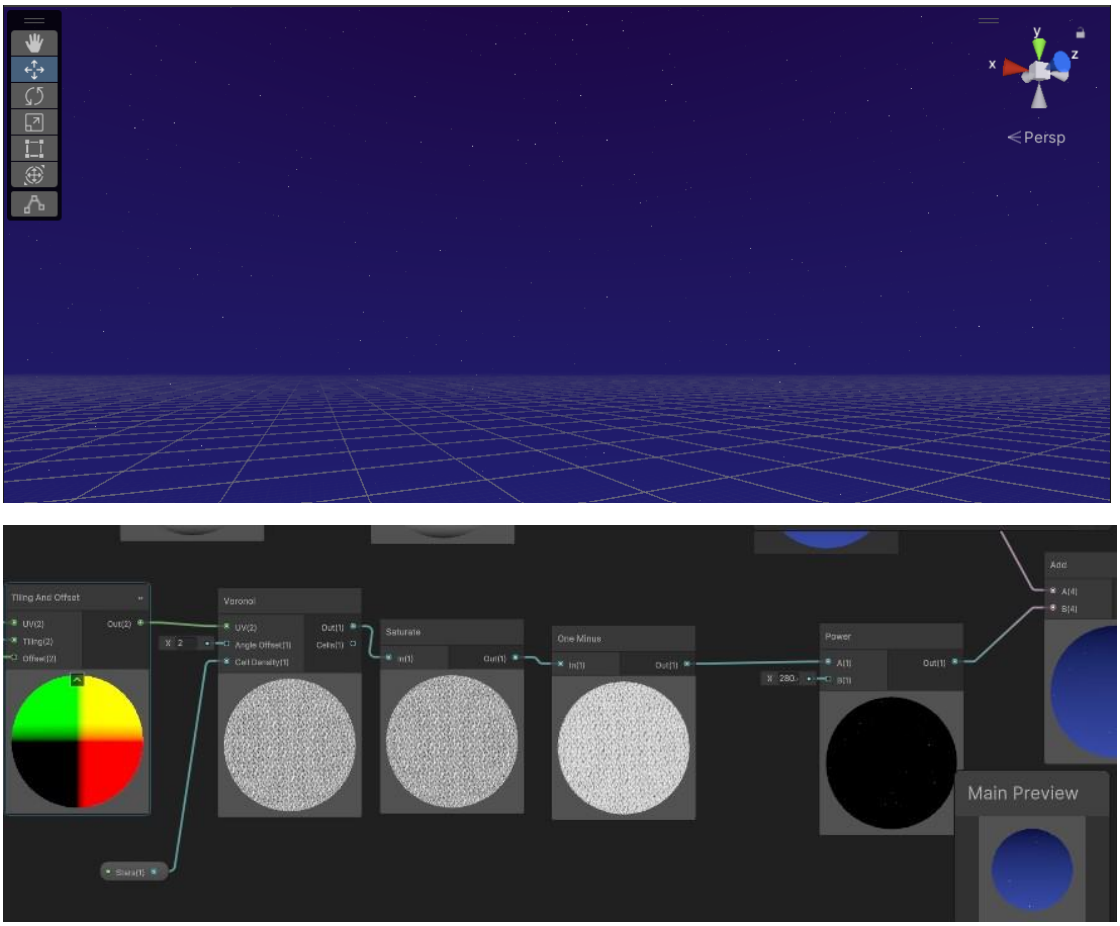

Figure 3. Sample skybox and a portion of the shader graph.

There have been attempts to fix the inconsistent lighting from switching scenes by changing the lighting settings between baked and realtime and others, though no permanent solution has been found yet. This lighting issue only occurs outside of combat – it is fine in the combat module.

Player Input

Majority of the player input takes place on a GUI with buttons during the travel, rest, and main menu modules. In combat, however, the player can move around with WASD, switch weapon types with Q, shoot/attack with left mouse click, reload ammo with R, and look left and right by moving the mouse. There are no plans on making this a game on other platforms than PC yet, so no other control schemes have been implemented.

Rest Module

During the rest module, players can check all their resources and manage their party's status. The screen is refreshed after any resource/health change, reading from the database for that file promptly after completing all requested update operations. During certain coroutines, pausing is disabled to avoid timing issues. Figure 4 shows a sample view of the rest menu and the RefreshScreen function.

```
/// <summary>
/// Refresh the screen upon loading the rest menu.
/// </summary>
public void RefreshScreen(){
    ManageRewards();

    // Main menus
    Save save = DataUser.dataManager.GetSaveById(GameLoop.FileId);
    IEnumerable<ActiveCharacter> characters = DataUser.dataManager.GetActiveCharacters().
    ActiveCharacter[] temp = characters.ToArray<ActiveCharacter>();
    int money = save.Money, food = save.Food, scrap = save.Scrap, medkit = save.Medkit,
        curDistance = save.Distance;
    phaseNum = save.PhaseNum;
    float gas = save.Gas;
```

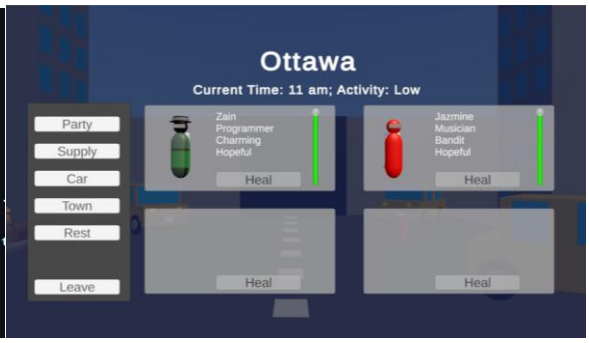

Figure 4. Sample rest menu and snippet of RefreshScreen function

Party Menu

Here, players can view their team members and their status read from the database Given that they have enough medkits and don't have full health, players can choose to heal a team member for 20HP at the cost of one medkit from their resources.

Players can choose to rest up to 12 hours to pass time. Resting will heal party members if they have food available – the amount healed with depend on the rationing of food and where they are resting. Otherwise, resting without food passes time but damages the team. Resting calls a coroutine to run in the background to delay the full operation, giving the player time to change their mind.

Supply Menu

Here, players can view their party's supplies read from the database. The type of rationing can be adjusted to change the amount of food consumed per party member. Normally, it is 2kg/person but can be changed 3kg or 1kg depending on the player's preference. Players can choose to scavenge for supplies, which will be mentioned in more detail in the combat module.

Players can choose to trade their supplies for other supplies. When choosing to wait for a trader, it costs 1 hour of in-game time and food consumption and there is no guarantee a trader will appear (lower if a member with the paranoid trait is present). When a trader does appear (40%), they will outline the resources demanded and offered for the player to decide if they want to accept it.

Car Menu

Here, players can view their car's status read from the database. Upgrades for the car can be found randomly during travel and can be viewed in this menu. The pace of the car can be set – normally it is 80km/h but can be set to 65km or 95km. The higher the pace, the more damage the car takes.

Players can fix the car with 1, 2, or 4 scrap if they have enough. A minigame starts where the player must line up bolts with circular outlines. The closer they are to that position (based on their RectTransform component), the more HP the car recovers (higher if a member with the mechanic trait is present). Missing the outlines result in a lower HP gain overall.

Town Menu

Here, players can view the town if they are currently in one. A shop is available to buy and sell supplies. The further the party is ingame, the higher the buying prices will be and the lower the selling prices will be, emphasizing the importance of preparing supplies as much as possible ahead of time.

A job board is available for players to do jobs to get more supplies as rewards. Currently, there are collection jobs and defence jobs. These will be detailed in the combat module section. On completion of a job, a ManageRewards() method during screen refreshing to check if a job has been completed and present the player with a popup of what they are rewarded with. Jobs are generated during town generation. The difficulty is a random number from 1-100. It is an easy job if the number is 1-20, a medium job if the number is 21-40, a hard job if the number is 41-60, or no job otherwise.

Travel Module

During the travel module, the party is travelling from a destination to another sub-destination on their way to the goal. One successful timestep/hour of driving on the road in-game is approximately 8 seconds. When driving, regular decrements of food and health and reading/updating the database is done. There is a 44% chance of a random event occurring during each hour that doesn't involve a party member perishing or breaking the car. In this section, some queries are still written as the raw SQL code with prepared statements to avoid players reloading the file and still make progress when a bad event is rolled. These queries and their parameters are stored in a List to be executed at a later point. Figure 5 shows a sample view of the travel module.

```
if(Timer >= 8.0f){
    if(Drive()){
        int eventChance = Random.Range(1,101);

        // 44/100 chance of generating an event
        if(eventChance <= 44){
            string msg = eventGenerator.GenerateEvent(eventChance);
            if(!msg.Equals("")){
                LaunchPopup(msg);
                if(msg.Equals("You suddenly find yourself surrounded by mutants.")){
                    GoingToCombat = true;
                }
            }
        }
    }
    ChangeGameData();
    HasCharacterDied();
    Timer = 0.0f;
```

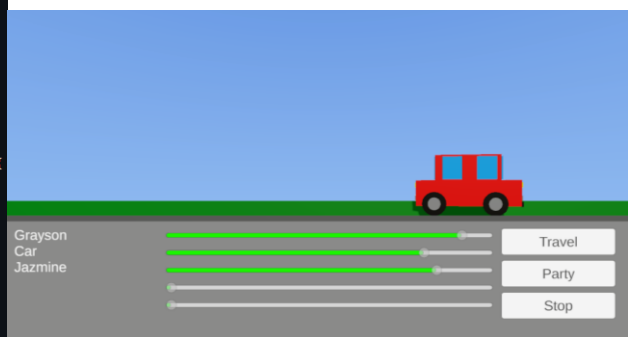

Figure 5. Sample travel module and snippet of code for generating events and performing queries

Upon entering the travel module, towns are generated in a utility class to randomize the price, stock, and available jobs when players are selecting a town to go to. At certain destinations, players can pick to go to one of two towns with differing distances and available supplies (summed up from all resources generated by the Town class). To track the names of towns and the distances required, two Dictionary structures stores the town number and respectively, the list of names and distances.

Random Event Generation

During travel, random events can occur. An event chance is rolled and if a 44 or less is rolled, an event occurs. From there, the possibility is further divided based on what event chance was rolled (ie. 33, 43, 21, 2…). Events are influenced by available resources, team status, and in-game time. Some events include party-wide damage, car upgrades being found, resources being gained/lost, and gaining/losing party members. During all events, SQL queries and their parameters are stored in a list to be executed at a later point. Figure 6 shows a sample event.

```
// 4/44 possibility for a random player to take extra damage (Ex. Bob breaks a rib/leg)
if(eventChance <= 4){
    int cushioned = car.MiscUpgrade2, rand = 0;

    // Pick a player
    rand = Random.Range(0,tempCharacters.Count());
    ActiveCharacter selected = tempCharacters[rand];
    string name = selected.CharacterName;
    string[] temp = {" breaks a rib.", " breaks a leg.", " breaks an arm.", " sits down wrong."};
    int hpLoss = diff % 2 == 0 ? Random.Range(13,20) : Random.Range(5,13), curHealth = selected.Health;
    // Lose less HP if cushion upgrade found
    hpLoss -= cushioned == 1 ? 5 : 0;
    curHealth = curHealth - hpLoss > 0 ? curHealth - hpLoss : 0;

    string commandText = "UPDATE ActiveCharacter SET Health = ? WHERE Id = ?";
    TravelLoop.queriesToPerform.Add(commandText);
    List<object> parameters = new List<object>(){curHealth, selected.Id};
    TravelLoop.parametersForQueries.Add(parameters);
    msg = name + temp[rand];
}
```

Figure 6. Sample event

Combat Module

Players can choose to go into combat from jobs in towns, scavenging, a rolled travel event, or reaching Vancouver. When entering, a ranged and a physical weapon must be chosen, each with their own damage outputs and drawbacks. Players go into combat with the total amount of ammo stored in the database divided by the number of team members.

A combat manager script manages the initialization, AI and their stats, mid-combat events such as generating entities (items/enemies), and cleanup. During cleanup, the party's status is always updated to the database.

Procedural Generation

Following a tutorial by Sebastian Lague, Perlin noise is a type of coherent noise with changes occurring gradually. Taking a section of this noise results in something that looks like a section of terrain. Multiple levels of noise are layered with octaves/other noise maps. Each subsequent octave should increase in detail (frequency) by using a lacunarity variable. With more detail, the influence should diminish by using a persistence variable to affect the amplitude of each octave. This creates a more natural looking outline for terrain. These graphs can be seen in Figure 7 where the red graph is the final output, and the blue graphs are the octaves. Using a height map generated from the noise, a height multiplier, and a height curve, the height of this terrain is set. Using a color map, the color of the mesh at a particular height is set to that color. A scriptable object stores all the data for the noise that the script accesses.

The mesh for a chunk is generated with triangles. The number of vertices is determined by the area and the number of triangles (listed as their vertices) in an array as listed in Figure 7.

Each square of the mesh is composed of two halves of the generated triangles. In general, for a square with width w, the first half has vertices at i, i+1, and i+w+1 and the second half has vertices at i, i+w, and i+w+1 as seen in Figure 7. Each mesh has uvs as a percentage between 0 and 1 to tell each vertex where it is in relation to the rest of the map. The normal are recalculated to adjust the lighting of the mesh. Threading with locks has been implemented to attempt to speed up the terrain generation. A scriptable object stores all the data for the mesh/terrain type.

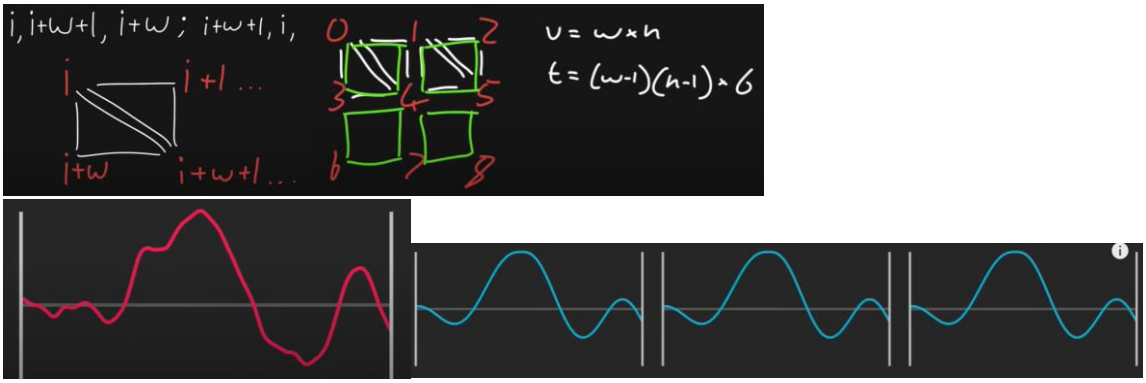

Figure 7. Diagrams for the height map, generating the mesh, and noise from Sebastian Lague's work

The mesh is also given a mesh collider to allow players and AI to interact with the terrain and a flat-shaded style (triangles with the same color all around) is implemented with the triangle's vertices whose normals is pointing in the same direction. Figure 8 shows a sample output. While obstacles have been omitted from the figure, these are placed randomly by using a Raycast pointing below the object while it is temporarily in the air to determine if a collision with the ground can be detected to place the object.

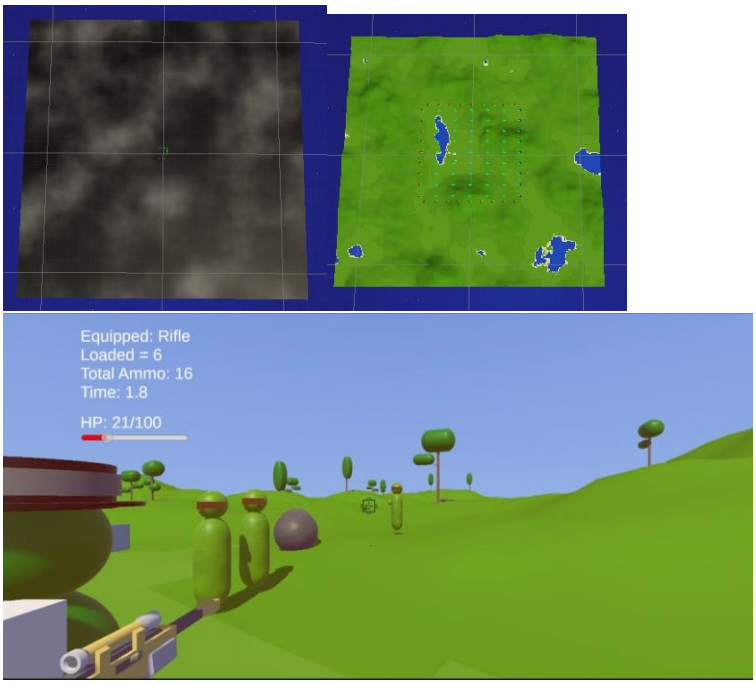

Figure 8. Sample Perlin noise, sample generated terrain, and sample with obstacles

While it has been implemented so that the terrain is endless with a Dictionary when the player approaches the end of the center chunk, the game's combat only takes place in the centre chunk for now. This may be changed in the future.

Scavenging

Players have a set amount of time to collect supplies that randomly appear on the map. The spawn points are initially in the air and then a raycast is sent down to detect the ground and move the spawn point down. Supplies that can be picked up are food, scrap, gas, money, and medkits, though some are more common than others. Enemies will occasionally spawn at the edges of the map. During cleanup, the summed resources is updated to the database in addition to the party status.

Collection Job

Players are set into the map to look for a chest object which spawns like a scavenging pickup. The player or a teammate must collide with the object to complete the job. Enemies will occasionally spawn at the edges of the map.

Defence Job/Travel Event/Endgame Combat

Players are set into the map to defeat a set amount of enemies. Enemies will not automatically spawn at the edges. During endgame, a boss mutant is spawned in addition to the regular amount. As a job, a set amount will always spawn depending on the difficulty. As a travel event, the amount spawned depends on the current in-game time. As endgame combat, a set amount will spawn, regardless of difficulty or job.

Game AI

Borrowing elements of an AI manager class during a previous coursework, this has been implemented in the CombatManager class. During each Update call, if the game hasn't been paused, it attempts to perform the AI agent's current action chosen for that frame. Figure 9 show the finite-state machines for the teammates and mutants and their actions based on detected objects and data it knows about itself.

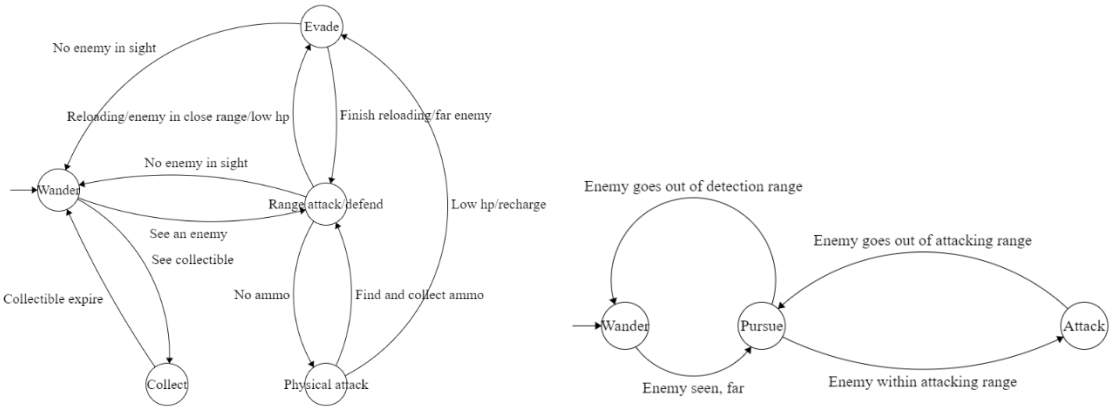

Figure 9. Finite-state machine for the teammate AI (left) and mutant AI (right)

CombatMind handles the above FSMs, where it handles each agent on a case-by-case basis. Each agent has sensors to help decide what it will do in that frame, such as the nearest collectible, defensive point, enemy, or party member.

Teammates have a goal of surviving and helping the player. If no mutants are visible, they Wander, looking for collectibles and moving around. Otherwise, they look for a defensive point to move to and attack with their ranged weapon. If low on HP, they keep a safe distance away and evade the mutant to try to stay alive. Figure 10 visualizes the process – they use the velocity of the mutants to predict where they will be within the next timestep and move away from it.

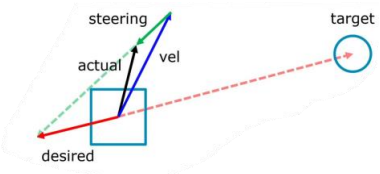

Figure 10. Flee steering behaviour visualized in previous coursework, COMP4770-2023W. Evasion is an extension of fleeing, noting the previous velocity of the agent to evade.

Given that they have ammo, teammates will prefer to use ranged attacks over physical attacks.

Mutants only have a goal of killing the player and their teammates. They are initially in a Wander state but if they see either, they go into the Attack state and pursue them until they are close enough to attack. Depending on their type, they can either attack physically or shoot from a distance. Once the teammate/player is dead, they go back to a Wander state.

Sensors are used for the AI to detect what's around them. These are implemented using FindObjectsOfType and filtering for specific aspects, such as a tag making the objects distinct from other objects. For example, the NearestCollectibleSensor. In addition, teammates can use the evasion steering behaviour. They use the velocity of the mutants to predict where they will be within the next timestep and move away from it, visualized in Figure 10.

Endgame

Upon reaching Vancouver, the player is forced into a defence combat with an extra boss mutant spawned in. This mutant has more HP than other mutants but essentially has the same behaviour as the FSM.

The game is over whether your party perishes before reaching Vancouver or successfully reaching the end. The score is calculated on several items from the database with this formula:

$$ResourceScore = food + gas + scrap + money + 2(medkit + \ tire + battery) + ammo$$

$$TeamScore = \ ResourceScore + \frac{dist}{10} + 500 * allies$$

$$TimeScore = TeamScore + (1000 - time) \ if \ time < 1000, else \ TeamScore$$

$$FinalScore = TimeScore * 2 \ if \ hard \ difficulty, else \ TimeScore$$

This formula incorporates how well you've managed your resources and team, the distance you've travelled, the mode you played, and the leader's name will be recorded to the database. The score is recorded, and the file is promptly deleted after triggering the game over components to be active.

Testing

Overall, testing used a combination of unit and integration testing. At the start, unit testing ensured each module worked well on its own before moving to integrating with more modules. Upon discovery of a bug, Debug.Log was used to attempt to pinpoint the cause if the root cause wasn't obvious. This testing strategy ensured a seamless flow between the modules of the game and a good experience in each.

Next Steps

Some next steps include further expanding the front-end graphics and models of the game. While the game is in a completed and working state by giving emphasis to the back end, majority of the models and graphics are simplistic. Further expanding the graphics and models would allow further game immersion for end-users.

Additionally, time could be better managed for any future projects, since I did only design during the winter and moved slowly during the summer semester on weekends, resulting in the bulk of the coding being done during the fall semester. With better time management, the workload could have been more evenly distributed throughout the timeline.

References

http://www.deathroadtocanada.com/

https://github.com/SebLague/Procedural-Landmass-Generation

https://hatsproductions.com/organtraildc/

https://www.youtube.com/playlist?list=PLFt_AvWsXl0eBW2EiBtl_sxmDtSgZBxB3

https://www.youtube.com/watch?v=yw2J9NWRdow