CS354R Final Project Report

1. Introduction

This is the writeup for our CS354R capstone project. The goal of this assignment is to make a robust game that builds on our existing AI to create a more advanced AI.

2. Game Idea

We took inspiration for our game from two places: Wii Sports Resort Swordplay and Infinity Blade. The game consists of an infinite world where you fight against enemies through a series of attacks and defenses. To create a complex AI, we chose a total of three different attacks and three defenses.

The attacks are the following and performed with clicking [Q,W,E] respectively:

Chop - The player/enemy swings their sword down vertically. Players can dodge the chop to avoid damage completely, block the chop for lessened damage, or counter chop to parry the attack.

Slice - The player/enemy swings their sword across horizontally. Players can jump to avoid damage completely, block the slice for lessened damage, or counter slice to parry the attack.

Stab - The player/enemy stabs with their sword. Players can block to avoid damage completely and stun the enemy. This move does extra damage.

The defenses are the following and performed with clicking [I,O,P] respectively:

Dodge - The player/enemy moves to the right to avoid a chop attack.

Jump - The player/enemy jumps in the air to avoid a slice attack.

Block - The player/enemy draws their shield to block attacks and minimize damage. There is a limit to the shielding amount denoted with a blue bar in the top right that regenerates when not blocking.

We wanted to create a relatively complex AI but also wanted to focus on creating a complete game. This means that we also focused a lot on the game aspect. We spent a decent amount of time planning out the visuals regarding the map layout, as well as timing the animations to properly set off attacks. We also wanted the game to be relatively balanced, so we added indicators for enemy attacks and enemy hits that land. Overall, we believe our game is a flushed

out simple version of the game inspirations. Further advancements could be made by adding more complex moves for the player, such as making it so players can choose any angle to attack rather than the three presets we have.

3. World Generation

We opted to make this game an infinite game until the player dies. The enemies get progressively faster throughout the process and they learn from the player's actions, which make it increasingly harder to block their attacks. To simulate an infinite world. We create two instances of the arena, connected with a hallway. When the player defeats the current enemy and walks to the new arena, we create a new instance of the arena and move it to the end:

```
if(p.get_location().x < a.global_position.x-200) :
    var arena = ArenaScene.instantiate()
    arena.position = Vector3(a.global_position.x-600,a.global_position.y,a.global_position.z);
    arena.add_to_group("arena2")
    arena.get_node("Enemy").add_to_group("enemy2")
    a2.remove_from_group("arena2")
    a2.add_to_group("arena1")
    e2.remove_from_group("enemy2")
    e2.add_to_group("enemy1")
    e2.connect("enemy_chop", enemy_chop);
    e2.connect("enemy_slice", enemy_slice);
    e2.connect("enemy_stab", enemy_stab);
    e2.connect("enemy_death", enemy_death);
```

This also means we need to keep track of which arena we are currently in. We use groups "arena1" and "arena2" to differentiate between the arenas and "enemy1" and "enemy2" to differentiate between the enemies in the arena. One annoying thing is that we need to connect signals every time we create a new instance of the enemy.

We also delete our current arena. This helps keep the scene tree free while simulating an infinite world:

```
$Arena.add_child(arena,true)
a.queue_free();
```

4. Player Actions

5. AI

The AI we implemented is for the enemy that the player faces. We decided to have the AI decide their next moves based on the past moves of the player by storing them inside of an array and calculating each move's percent occurrence when the player initiates a move. We also decided to have enemies further in the game have the player's past moves against previous opponents to simulate the enemy learning as the player went along.

```
void Enemy::add_move_list(int m) {
    player_move_list[m] += 1;
    total_moves += 1;

    chop_probability = (double)player_move_list[1] / (double)total_moves;
    slice_probability = (double)player_move_list[2] / (double)total_moves;
    stab_probability = (double)player_move_list[3] / (double)total_moves;
    dodge_probability = (double)player_move_list[4] / (double)total_moves;
    jump_probability = (double)player_move_list[5] / (double)total_moves;
    block_probability = (double)player_move_list[6] / (double)total_moves;
    // for (int i = 0; i < 7; i++) {
    //     UtilityFunctions::print(player_move_list[i]);
    // }
}
```

Using these probabilities, we then use a Mean of Max algorithm to decide on the most probable next move of the player.

```
int Enemy::predict() {
    UtilityFunctions::print("Probabilities:");
    UtilityFunctions::print((double)(block_probability + jump_probability + dodge_probability +
            stab_probability + slice_probability + chop_probability));

    // Predict player's next move
    // Weighted range of a move determined by the number of times a move has been done
    // The range where the float lands determines the predicted next move
    double prediction = rand.randf_range(0, 1);
    if (prediction <= chop_probability) {
        return move_response(Moves::CHOP);
    } else if (prediction <= slice_probability + chop_probability) {
        return move_response(Moves::SLICE);
    } else if (prediction <= stab_probability + slice_probability + chop_probability) {
        return move_response(Moves::STAB);
    } else if (prediction <= dodge_probability + stab_probability + slice_probability +
            chop_probability) {
        return move_response(Moves::DODGE);
    } else if (prediction <= jump_probability + dodge_probability + stab_probability +
            slice_probability + chop_probability) {
        return move_response(Moves::JUMP);
    } else if (prediction <= block_probability + jump_probability + dodge_probability +
            stab_probability + slice_probability + chop_probability) {
        return move_response(Moves::BLOCK);
    } else {
        // May want to change default behavior
        return move_response(-1);
    }
}
```

This algorithm calls move_response(), which will populate an array of probabilities of the enemy's moves which respond correctly to the predicted next move (Jump and Block not shown).

```
int Enemy::move_response(int m) {
    double decision[7] = {0, 0, 0, 0, 0, 0, 0};
    // m being the predicted next move of player
    // Populate probabilities of enemy's next move
    int num_nonzero_a;
    int num_nonzero_d;
    switch(m) {
        case Moves::CHOP:
            decision[Moves::CHOP] = 0;
            decision[Moves::SLICE] = 0.1;
            decision[Moves::STAB] = 0.6;
            decision[Moves::DODGE] = 0.3;
            decision[Moves::JUMP] = 0;
            decision[Moves::BLOCK] = 0;
            num_nonzero_a = 3;
            num_nonzero_d = 2;
            break;
        case Moves::SLICE:
            decision[Moves::CHOP] = 0.1;
            decision[Moves::SLICE] = 0;
            decision[Moves::STAB] = 0.6;
            decision[Moves::DODGE] = 0;
            decision[Moves::JUMP] = 0.3;
            decision[Moves::BLOCK] = 0;
            num_nonzero_a = 3;
            num_nonzero_d = 2;
            break;
        case Moves::STAB:
            decision[Moves::CHOP] = 0;
            decision[Moves::SLICE] = 0;
            decision[Moves::STAB] = 0.1;
            decision[Moves::DODGE] = 0;
            decision[Moves::JUMP] = 0;
            decision[Moves::BLOCK] = 0.9;
            num_nonzero_a = 1;
            num_nonzero_d = 1;
            break;
        case Moves::DODGE:
            decision[Moves::CHOP] = 0;
            decision[Moves::SLICE] = 0.85;
            decision[Moves::STAB] = 0;
            decision[Moves::DODGE] = 0;
```

We also decided to add an aggression variable in order to have more and less aggressive enemies. So, after the array is populated, we modify the default values based on the aggression fuzzy variable, which ranges from 0.0 to 2.0 where 1.0 is default or medium. Based on the difference between the enemy's aggression value and the default, we adjust the move probabilities inside of decision[].

```
double temp = aggressiveness - default_agg;
double aggression_factor = abs(temp);
if (temp > 0) {
    // If more aggressive than default: take from defending, give to attacking
    double total_d = decision[Moves::DODGE] + decision[Moves::JUMP] + decision[Moves::BLOCK];
    double modifier_a;
    double modifier_d;
    if (num_nonzero_a == 0 || num_nonzero_d == 0) {
        modifier_a = 0.0;
        modifier_d = 0.0;
    } else {
        modifier_a = (aggression_factor * total_d) / num_nonzero_a;
        modifier_d = (aggression_factor * total_d) / num_nonzero_d;
    }

    for (int i = 1; i < 4; i++) {
        if (decision[i] > 0.0) {
            decision[i] += modifier_a;
        }
    }
    for (int i = 4; i < 7; i++) {
        if (decision[i] > 0.0) {
            decision[i] -= modifier_d;
        }
    }
} else if (temp < 0) {
    // If less aggressive than default: take from attacking, give to defending
    double total_a = decision[Moves::CHOP] + decision[Moves::SLICE] + decision[Moves::STAB];
    double modifier_a;
    double modifier_d;
    if (num_nonzero_a == 0 || num_nonzero_d == 0) {
        modifier_a = 0.0;
        modifier_d = 0.0;
    } else {
        modifier_a = (aggression_factor * total_a) / num_nonzero_a;
        modifier_d = (aggression_factor * total_a) / num_nonzero_d;
    }

    for (int i = 1; i < 4; i++) {
        decision[i] -= modifier_a;
    }
    for (int i = 4; i < 7; i++) {
        decision[i] += modifier_d;
```

After adjusting the values for aggression, we then choose a move based on weighted ranges.

```
// Choose move based on random float
// The range where the float lands determines the move
double choice = rand.randf_range(0, 1);
if (choice <= decision[1]) {
    return Moves::CHOP;
} else if (choice <= decision[1] + decision[2]) {
    return Moves::SLICE;
} else if (choice <= decision[1] + decision[2] + decision[3]) {
    return Moves::STAB;
} else if (choice <= decision[1] + decision[2] + decision[3] + decision[4]) {
    return Moves::DODGE;
} else if (choice <= decision[1] + decision[2] + decision[3] + decision[4] + decision[5]) {
    return Moves::JUMP;
} else if (choice <= decision[1] + decision[2] + decision[3] + decision[4] + decision[5] +
        decision[6]) {
    return Moves::BLOCK;
} else {
    return Moves::IDLE;
}
```

A problem arose where the AI would sometimes act too random, even after tuning the default probability values in move_response(). The most obvious example was that if the enemy successfully blocked a stab and knocked the player back, it would sometimes not attack (and if it would, it wouldn't stab, which is not ideal). In other words, it ran the fuzzy logic algorithm at a time when it did not need to. To combat this, we added a conditional, which made its actions more believable.

```
if (is_fighting && next_move == -1) {
    if (successful_block) {
        next_move = Moves::STAB;
        successful_block = false;
    } else {
        next_move = predict();
    }
```

6. GUI

We have three pieces of the GUI that exist throughout the game state: player health, shield, and score.

These update through signals that we connect when an enemy attack lands:

```
func enemy_chop() :
    var p = get_tree().get_first_node_in_group("player")
    var move = p.get_move()
    # dodge
    if move == 4 :
        return
    # block
    elif move == 6 :
        p.set_health(p.get_health() - 10)
        update_health(p.get_health(),true)
    else :
        p.set_health(p.get_health() - 20)
        update_health(p.get_health(),true)
    pass
```

If the player lands an attack, we grab the health node and update the health bar. The same goes for shield.

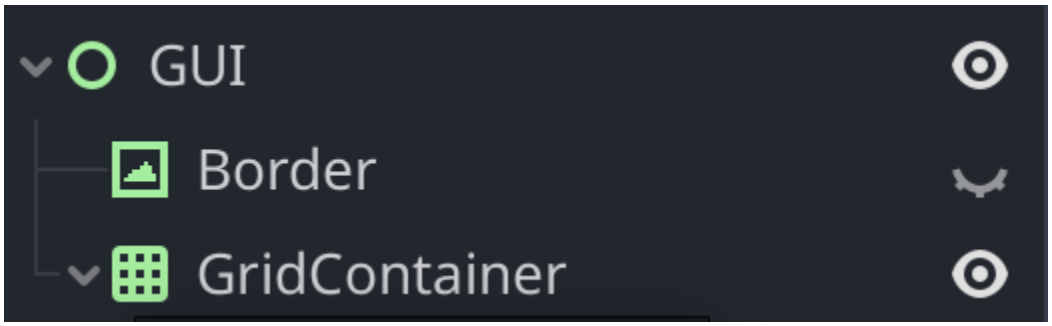
When an enemy dies, we also update the score. One thing we wanted to consider was incentivizing players to focus on defending instead of attacking as fast as possible. Thus, if the player goes an entire round with full health, they gain three score instead of just one:

```
func enemy_death() :
    var p = get_tree().get_first_node_in_group("player")
    p.set_running(true);
    p.set_fighting(false);
    if(p.get_health() == 100):
        kills += 3

    else:
        kills += 1
    p.set_health(p.get_health() + 40);
    update_health(p.get_health(), false);
```

There are other pieces of the UI that we believe enhance the game experience. First, we realized it is rather difficult for the player to guess when and what attack the enemy is going to choose. Thus we added an attack indicator for when the enemy is about to attack, as well as adding a delay so the player can react in time:

We achieved this by taking a transparent image and adding it to the GUI:



Then when the enemy chooses and attack, we emit a signal to update this visibility to show the attack indicator:

```
if (is_fighting && next_move == -1) {
    next_move = predict();
    // if we are defending, dont wait
    if(next_move >= 4) {
        timer = 2;
    } else {
        if(get_node_or_null("../../../GUI/Border"))
            get_node<TextureRect>("../../../GUI/Border")->set_visible(true);
    }
}
else if (timer > 1) {
    if(get_node_or_null("../../../GUI/Border"))
        get_node<TextureRect>("../../../GUI/Border")->set_visible(false);
```

Furthermore, when the player gets hit, we want some indicator that is different from the attack indicator. We opted to make the screen shake a bit when the enemy attacks. To do so, we have two variables: RANDOM_SHAKE_STRENGTH and SHAKE_DECAY_RATE:

```
@export var RANDOM_SHAKE_STRENGTH = 2.0;
@export var SHAKE_DECAY_RATE = 5.0;
```

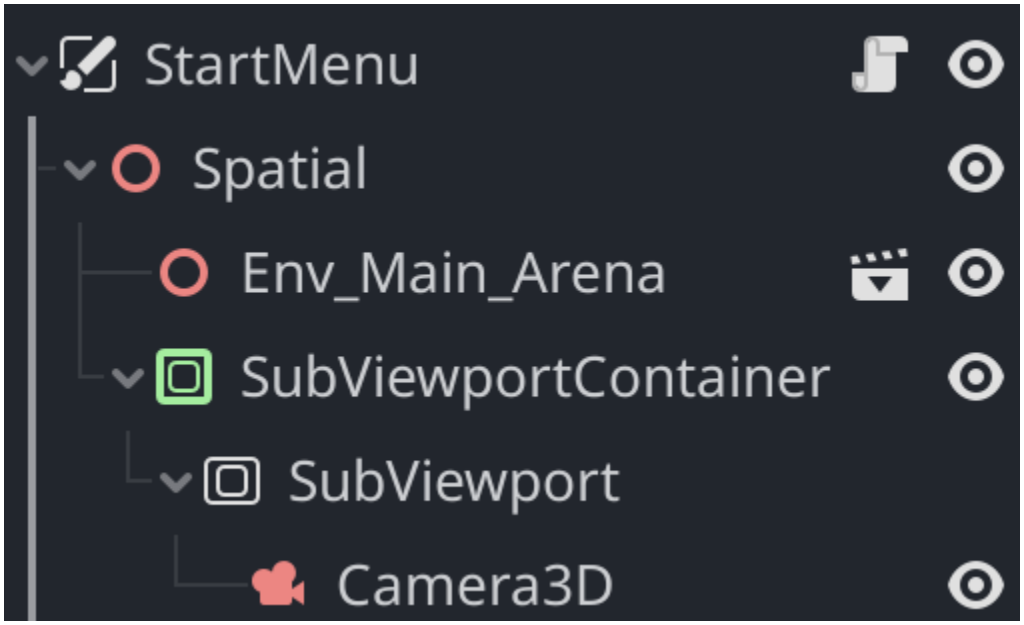
These determine how far the camera shakes the screen as well as how long it should shake. When the player gets hit, We set the shake strength to RANDOM_SHAKE_STRENGTH. Then we randomize the location of the camera within this strength while decaying it over time:

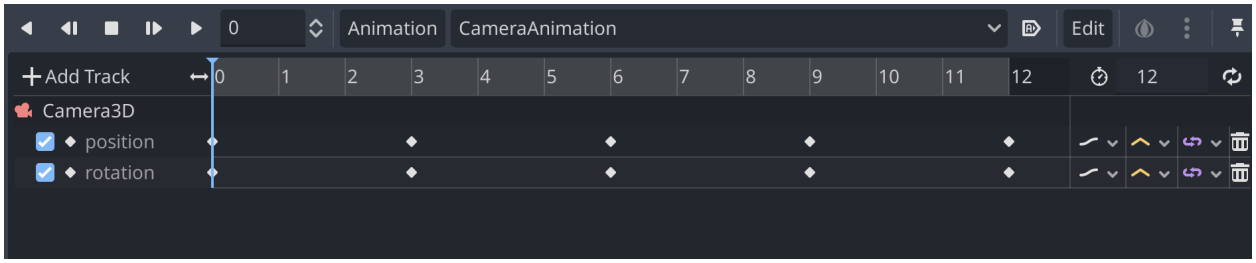
```
shake_strength = lerp(shake_strength, 0.0, SHAKE_DECAY_RATE * _delta);
var camera = get_node("Body/Camera3D");
var offset = get_random_offset();
camera.h_offset = offset.x;
camera.v_offset = offset.y;
```

While this is not a bad addition, one thing to consider is that this approach creates a disconnect between the camera and the scene because the camera can move anywhere within the random shake strength. One improvement could be to use a noise function to better simulate the shaking.

Finally, my previous projects had very simple start menus. Instead of doing just a simple 2D screen, I opted to create 3d scene with a 2d menu:

This required me adding a camera to a 3d scene and animating it to move throughout the scene:





Overall, I believe these small details helped teach game design beyond what was taught in class. Having good design is crucial for a good player experience, so finding ways to implement features like the flashing indicator and screen shake helps make the game more immersive.

7. Problems

When making the executable, we ran into a problem where the input map didn't work for the Linux version of the project. We weren't able to fix this problem, so the executable does not work properly, although it has everything else other than the input problem.

8. Division of Labor

a. **Zach:**
   i. Advanced AI Algorithm with Fuzzy Logic
   ii. Level Design, Addition of Props

b. **Sam**
   i. Infinite World Generation
   ii. Player Attacks
   iii. GUI (In game and start screen)
   iv. QoL Updates (Screen shake, attack indicator)

c. **Josh**
   i. Initial Map Creation & Player Setup
   ii. Enemy Attacks
   iii. Basic AI
   iv. Attack Details (attack timing, attack flash)