Functionality:
We implemented a variety of functions and features. First, we created 2 different enemy nodes that will have various behaviors. We created a purple eater node as well as a red attacker node. We added multiple of each agent type throughout the map in order and scattered them throughout at random. In order to implement the behavior switching, we implemented a finite state machine that handles switching between different behavior states when triggered by a signal. We chose to use a finite state machine since our states were very distinct and had no overlap that a finite state machine couldn't account for.

The eater node has 3 behaviors that its finite state machine switches between. It starts off with the initial chase behavior of finding the nearest food node and chasing it by moving towards it. Once within a certain range, it switches to the eat behavior and actually eats the food. After eating the food, it switches back to its chase behavior and begins looking for the next nearest food node. While chasing food, it can detect the location of the player and switches to the retreat behavior of running away when the player gets within a certain distance of the eater node. Once a safe distance away, it switches back to the chase behavior.

The attacker node also has 3 behaviors that its finite state machine switches between. It starts off with the initial chase behavior of locating the player and chasing it. Once within a certain distance, it switches to the attack behavior and actually takes a life (shown in the GUI and indicated by the player turning red) by attacking the player. After attacking the player, it switches to a dodge behavior where it runs away from the player in order to recharge its attack. Once a safe distance away, it recharges its attack and switches once again to the chase behavior.

The player is able to attack both of these enemies at any time by jumping on top of the agent. Doing so will send the enemy back to a respawn point.

When either agent type chases/dodges, it uses our custom navigation. Our navigation calculates the target position and moves our agent towards that target position through various calculations. Throughout this movement, it detects any walls or obstacles in its way. When it detects an obstacle, it is able to teleport up to get past the obstacle.

Code Layout:
For the general system, we have a main node that has a field node with the main bulk of the content. The field node has the ground/walls and lighting as the basics. It also has the player node as well as multiple of both agents (3 eater enemies and 3 attacker enemies). With each node, we have further scene trees.

For each AI agent, we have the basic collision nodes and mesh nodes in addition to the new AI content. For the AI system, we have a generic finite state machine class that is used by both agents. This finite state machine sets an initial state and receives signals to switch between various states. There are also 6 different behavior classes (3 for each agent) that are children of the finite state machine nodes. They are the actual brains of the AI system and make decisions about what to do and when to switch to other states (the finite state machine simply does the switching). We also created a navigation class that takes care of all navigation and moves the nodes (the behavior states make the decisions about where to go). In order to keep everything scalable, we made all our code very modular so that adding multiple agents is as easy as adding the agent scenes to the scene tree in the Godot GUI.

Code Segments:

```
// transitions states modularly
// takes in an old state and changes it to a new state, doesn't matter what they are
void FiniteStateMachine::on_child_transition(String old_state_name, String new_state_name) {
    if (old_state_name.to_lower() != current_state->get_name().to_lower()) {
        return;
    }
    Variant new_state = states.get(new_state_name.to_lower(), nullptr);
    if (!new_state) {
        return;
    }
    if (current_state) {
        current_state->exit();
    }
    current_state = Object::cast_to<State>(new_state);
    Object::cast_to<State>(current_state)->enter();
}

// generic state class that all states inherit
// makes it easier to work with them modularly
void State::_bind_methods() {
    ADD_SIGNAL(MethodInfo("transitioned", PropertyInfo(Variant::STRING, "old_state"),
PropertyInfo(Variant::STRING, "new_state")));
}

State::State() {}

State::~State() {}

void State::_ready() {}

void State::enter() {
    UtilityFunctions::print("enter generic state");
}

void State::exit() {
    UtilityFunctions::print("exit generic state");
}

void State::update(double delta) {
    UtilityFunctions::print("generic update");
}

void State::physics_update(double delta) {
    UtilityFunctions::print("generic physics update");
}



// chase method that gets points and paths from A* and navigates using them
```

```
// takes in a node and a destination, can be used for both entities
void Navigation::chase(CharacterBody3D *source, Vector3 dest) {
    astar->add_point(1, source->get_position());
    astar->add_point(2, dest);
    astar->connect_points(1, 2, false);
    PackedVector3Array path = astar->get_point_path(1, 2);
    Vector3 dir = Vector3(0, 0, 0);
    if (path.size() > 0) {
        dir = path[1] - source->get_position();
    }
    dir.normalize();
    if (!source->is_on_floor()) {
        dir.y = source->get_velocity().y - 1400;
    } else {
        dir.y = 0;
    }
    source->set_velocity(dir * 10);
    source->move_and_slide();
    source->set_position(source->get_position());
    astar->disconnect_points(1, 2);
    astar->remove_point(1);
    astar->remove_point(2);
}
```