## **Version 1: Developer Task List (User-Facing Step-by-Step Guide)**

### **Setup and Planning**

* **Initialize the Godot Project:** Create a new Godot project (2D). Set up a main scene (e.g. Main.tscn) that will serve as the game’s root. This scene can be a simple **Node2D** (for a 2D game) or a **Node** that holds everything.  
  + Create folders for organization: e.g. Scenes/, Scripts/, Scenes/Rooms/, Scenes/Characters/, etc., to keep content organized.
  + Enable **snap/incremental saves** if needed, so you can run the game quickly to test as you build.
* **Configure Input Map:** Define input actions in **Project Settings > Input Map** for character control and combat. For example:  
  + Movement keys: define actions like "move\_up", "move\_down", "move\_left", "move\_right" (e.g. W/A/S/D or arrow keys).
  + Attack keys: define at least one attack action. For a Binding-of-Isaac style shooter, you might use arrow keys for shooting directions (e.g. "shoot\_up", "shoot\_down", etc.), **or** use a single action (like "attack") that fires in the player’s facing direction. (Using multiple shoot directions is closer to the original game’s feel, but a single attack button is simpler – choose based on what’s comfortable for you.)
  + Jump, dash, or other actions are not needed for this minimal prototype (unless you plan a specific action). Keep it limited to movement and attack.
* **Plan a Component-Based Architecture:** Before coding, outline how you will use a modular, component-driven structure in Godot:  
  + **Godot’s Node System:** Remember that every game entity will be a Scene (Node) that can have child Nodes acting as components (Godot’s scene system naturally supports this kind of composition). For example, your player could be a CharacterBody2D (or KinematicBody2D in Godot 3.x) with child nodes for things like a sprite, collision shape, health, etc.
  + **Separation of Concerns:** Decide on the main components you need:  
    - *Movement Component:* Handles reading input (for player) or AI (for enemies) and moving the character.
    - *Attack Component or Weapon:* Handles firing projectiles or melee attacks.
    - *Health Component:* Manages hit points, taking damage, and death.
    - *Others:* You might not need more for the prototype, but know that later you could add components for things like **pickup collector** or **power-ups**. Keep each piece self-contained.
  + **Communication:** Plan how components will talk to each other or to the main game:  
    - Use **signals** (Godot’s event system) for loose coupling. For example, the Health component can emit a "died" signal when HP reaches 0, and the parent (or a manager) can listen to handle death logic.
    - Use node grouping or direct references for interactions. e.g., mark all enemies with an “Enemies” group to easily find them, or have a central game node that keeps track of active enemies.
  + This planning step ensures your code is organized so you can expand it later (adding new enemy types, items, etc., won’t require rewriting core systems).

### **Player Character & Controls**

* **Create the Player Scene:** Make a new scene for the player (e.g. Player.tscn). This will represent the player character.  
  + Set the root of the player scene as a **CharacterBody2D** (Godot 4.x) or **KinematicBody2D** (Godot 3.x) so you can use built-in physics for movement and collisions. Name it Player.
  + Add essential child nodes:  
    - **Sprite2D** (or AnimatedSprite2D) for the player’s visual. Use a simple placeholder graphic (e.g. a colored circle or square) if you don’t have art yet.
    - **CollisionShape2D** for the player's physical body. Attach it to the CharacterBody2D. Use a shape (circle or rectangle) roughly fitting the sprite. This will allow collision detection (with enemies, walls, etc.).
    - **Camera2D** (optional): If your rooms are larger than the viewport, add a Camera2D as a child of the player and enable it, so the camera follows the player. For small rooms that fit on screen, this isn’t necessary, but having it now can help if you later expand room sizes.
    - (Optional) **Light2D** or other visual effect nodes as needed (not crucial for prototype, only if you want some visual feedback later).
  + Organize the Player scene for components:  
    - You can attach scripts to the root and/or child nodes. For example, attach a Player.gd script to the root Player node for core player logic. This script might coordinate child components or handle high-level player events.
    - Alternatively, attach separate scripts to child nodes for specific behaviors (for instance, a Health.gd on a “Health” node). Decide which approach you’re comfortable with – having one script vs. multiple smaller scripts. A component approach leans toward multiple scripts (one per function), which can keep code modular and easier to manage.
* **Implement Player Movement:** Make the player responsive to input and move around smoothly.  
  + **Movement Script:** Create a script (GDScript) to handle movement. This could be part of the main Player script or a dedicated PlayerMovement component (attached to a child Node). For simplicity, you might do it in the Player’s main script first, but keep the logic separate so it could be moved to a component later if desired.
  + In the \_physics\_process(delta) function, read input and apply movement:  
    - Get the input direction vector. For example, var direction = Vector2(0,0) then add components:  
      * if Input.is\_action\_pressed("move\_right"): direction.x += 1 (and similarly for left, up, down).
      * This gives a vector like (1,0), (0,-1), etc. Normalize it if you're allowing diagonal movement so speed is consistent in all directions.
    - Multiply by a speed constant (e.g. speed = 200 pixels/sec) to get velocity.
    - Move the player: since we used a CharacterBody2D/KinematicBody2D, call its movement function. In Godot 4, use move\_and\_slide() or just velocity = direction \* speed combined with the built-in physics integration of CharacterBody2D. In Godot 3, use move\_and\_collide() or move\_and\_slide().
    - Ensure the player stops when no input (if using move\_and\_slide, passing a Vector2.ZERO velocity when nothing pressed will naturally stop).
  + **Diagonal Movement & Smoothing:** The above will allow 8-direction movement. If using a joystick or analog input, the approach is similar but using Input.get\_vector("move\_left","move\_right","move\_up","move\_down") which directly gives a normalized vector. This is a shorthand for handling diagonal input nicely.
  + **Test Movement:** Run the scene (or the whole game). Make sure the player moves and the collision shape is working (the player shouldn’t pass through walls once we add them). Right now, there are no walls, but you can add a temporary StaticBody2D with a CollisionShape as a floor boundary to test collision (e.g., a rectangle around the room). We will properly add room boundaries in the room scene later.
* **Implement Player Attacking (Basic Combat Action):** Allow the player to perform a basic attack. For our prototype, this will likely be a projectile (like shooting a magic bolt or arrow) since that’s straightforward and similar to Binding of Isaac’s shooting mechanics.  
  + **Bullet Scene:** Create a new scene (e.g. Bullet.tscn) for the projectile.  
    - Use an **Area2D** as the root for the bullet, with a **CollisionShape2D** (a small circle or box) to detect impacts.
    - Add a sprite or color rectangle so you can see the bullet (maybe a small circle or spark image).
    - Attach a script to the Bullet (e.g. Bullet.gd). This script will handle moving the bullet and detecting collisions:  
      * Give it a velocity: Vector2 property (set when the bullet is fired).
      * In \_physics\_process(delta), move the bullet: position += velocity \* delta (since we’re using Area2D, we manually update position). Alternatively, you could use a RigidBody2D and apply impulse, but manual movement is simpler and gives more control for a projectile.
      * Collision detection: One easy way is to use the Area2D’s built-in signal. **Connect** the bullet’s **body\_entered** signal to itself (or use \_on\_Bullet\_body\_entered(body) signal handler) to know when it hits a physics body (like an enemy). In the handler:  
        + Check if the body has a certain property or group (e.g., if body.is\_in\_group("Enemies")). If yes, that means an enemy was hit.
        + If an enemy is hit, tell the enemy to take damage (you might call a method on the enemy’s script, such as body.take\_damage(damage\_amount), or emit a signal that the enemy listens to). We will set up the enemy’s health logic soon.
        + Destroy the bullet after impact (queue\_free the bullet so it’s removed).
      * Also handle bullet lifetime: if the bullet goes off-screen or travels beyond the room bounds, you should remove it to free memory. You can do this by:  
        + Setting a **Lifetime** timer (queue\_free after a second or two), **or**
        + Check position each frame – if it's outside the room area (e.g., beyond certain distance from origin), queue\_free it.
    - **Bullet Speed & Damage:** Decide on a speed (e.g. 400 px/sec) and a damage value (maybe 1 or some unit). These can be exported variables on the Bullet script so you can tweak in the editor if needed.
  + **Player Attack Input:** Now back in the Player script (or a PlayerAttack component script attached to Player):  
    - Handle input for attacking. If you chose separate shoot directions (e.g. arrow keys for shooting), check those. For example:  
      * if Input.is\_action\_just\_pressed("shoot\_right") then spawn a bullet with velocity (bullet\_speed, 0).
      * Do this for each direction key (left, up, down with appropriate velocity vectors). This allows shooting independently of movement direction.
    - If you chose a single attack button (e.g. spacebar or click), you need to decide a direction. You could shoot in the direction the player is moving/facing. One way: store the last non-zero movement direction from the movement code, and use that as the firing direction when attack is pressed. If the player is stationary, you might choose not to fire or fire in a default direction (depending on design).
    - **Spawn Bullet:** To spawn, preload the Bullet scene at the top of the script (var BulletScene = preload("res://Scenes/Bullet.tscn")). When attack input is detected:  
      * Instantiate the bullet: var bullet = BulletScene.instantiate().
      * Position it at the player’s position (or at the player’s gun/muzzle if you have one; for now, maybe center of player).
      * Set the bullet’s velocity property based on the desired direction \* bullet\_speed.
      * Add the bullet to the scene tree. If the Player is in the scene tree, you could do get\_tree().get\_current\_scene().add\_child(bullet) to add it to the active scene (or simply get\_parent().add\_child(bullet) if the player’s parent is the room).
    - **Test Shooting:** Run the game and press the attack keys. Bullets should spawn and travel in the correct direction. Verify collisions: since we haven’t made enemies yet, temporarily you can place a dummy target (like a StaticBody2D with a CollisionShape) to see if bullets collide and get freed.
    - Ensure bullets don’t collide with the player or other unwanted things:  
      * Set up collision layers/masks: e.g. put player on one layer, enemies on another, bullets on another. Configure the bullet’s CollisionShape2D mask to collide only with the enemy layer (and maybe environment walls), not with the player. Similarly, set the player not to collide with bullets if you want the player’s own bullets to pass through them.
      * This prevents friendly-fire and weird collisions (like bullet hitting the player immediately). In Godot, you can adjust the collision\_layer and collision\_mask in the Inspector for the physics bodies/areas.
* **Add Player Health and Damage Handling:** The player needs a health pool and the ability to take damage (from enemies).  
  + **Health Component:** Create a child node on the Player (name it "Health" or similar, as a simple Node or Node2D) and attach a script (e.g. Health.gd). This script will handle health logic for any character (player or enemy, if reused). Key elements:  
    - Define export(int) var max\_health and var current\_health. When the player spawns, set current\_health = max\_health (e.g. max\_health could be 5 or 10 for testing).
    - Provide a method func take\_damage(amount) that subtracts from current\_health. If current\_health falls to 0 or below, trigger a death.
    - Emit a signal upon important events:  
      * Define signal died in the script, and emit it when health hits 0 (before freeing or whatever).
      * You might also emit a signal for "health\_changed" if you want to update UI (optional).
    - For the player, you might handle death by resetting the game or stopping input (since this is a prototype, a full game over screen isn’t needed, but you should at least log or print "Player died" or similar).
  + **Integrate Health with Player:** On the player’s script, connect the Health node’s signals:  
    - For example, in Player \_ready(): get\_node("Health").connect("died", self, "\_on\_PlayerDied") (assuming you have a method \_on\_PlayerDied in the player script).
    - The \_on\_PlayerDied could handle what happens when player dies (e.g., print a message or restart the scene).
    - If using separate scenes, alternatively a central Game node could listen for the player’s death signal to trigger a restart.
  + **Damage on Collision (Player):** We will make enemies deal damage when they touch the player. This can be done in a couple of ways (we will set it up when implementing enemies). For now, ensure the Health component has the take\_damage() ready to be called by an enemy or other source. For example, if an enemy “hit” signal is emitted, it will call player\_health.take\_damage(1).
  + **Health Pickup (for player):** Though optional, since we allowed health pickups, you might plan a simple pickup item:  
    - Create a HealthPickup.tscn (Area2D with CollisionShape and maybe a heart sprite). When the player overlaps it (area\_entered or body\_entered signal on the pickup), increase the player’s health (e.g. +1, but capped at max\_health) and remove the pickup.
    - You can place a couple of these in rooms for testing (they will be the only pickup type in this prototype). This will let you test that picking up health works and the health component updates.

### **Enemy and Combat Elements**

* **Create an Enemy Scene:** Design a basic enemy that the player can fight.  
  + Make a new scene Enemy.tscn. Root it as **CharacterBody2D** or **KinematicBody2D** (similar to player, so it can collide with walls and the player). Name it Enemy.
  + Add child nodes under Enemy:  
    - **Sprite2D** for visuals (placeholder shape or sprite).
    - **CollisionShape2D** for the body collision (for colliding with walls and player). Size it appropriately around the enemy sprite.
    - **Area2D** (child) for the enemy’s attack range or hitbox. Name it "AttackArea". Give it a CollisionShape2D (maybe the same size as the enemy or slightly larger if you want the enemy to damage the player when close). This area will be used to detect the player for dealing contact damage.  
      * In the **AttackArea** script (or enemy script), connect the area\_entered or body\_entered signal. We want to detect when the player enters the area:  
        + If the body\_entered is the Player (or has group "Player"), then reduce the player’s health. For example: call a method on player’s health: player\_health.take\_damage(1). (You can get the player node via the body parameter, or if using groups: if body.is\_in\_group("Player"): ...).
        + Decide if the enemy should damage the player continuously on contact or just once per touch. For a simple approach, you might apply damage and then perhaps disable the area for a second (invincibility frames) or kill the enemy on impact. However, in Binding of Isaac style, touching an enemy usually hurts the player and maybe knocks them back, and can hurt again if they stay in contact after a short cooldown.
        + For the prototype, you can implement a simple rule: when collision is detected, reduce player health by 1 and maybe print "Player hit!". You could add a short Timer to prevent immediate repeated damage (optional).
    - **Health Node:** Instance the same Health.gd script on a child Node for the enemy’s health (or attach to Enemy node itself). Set a max\_health (e.g. 3). This will allow the enemy to die after enough hits.  
      * Connect the enemy Health’s died signal to the Enemy node or a central handler. On enemy death, you’ll want to remove the enemy and possibly notify the room manager (to count kills).
    - (Optional) **Particle2D** for death explosion or **AudioStreamPlayer2D** for enemy sounds, if you want to add effects later.
  + **Enemy AI (Movement):** Attach a script to the Enemy (e.g. Enemy.gd). This will handle enemy behavior:  
    - In \_physics\_process(delta), implement a simple AI that makes the enemy move towards the player. For example:  
      * Ensure the enemy can find the player node. One way: in the enemy script, get a reference to the player. If the player is always present, you could use get\_tree().get\_nodes\_in\_group("Player") or have the player added to a group. Or you could pass the player reference when spawning the enemy (if spawned by code). For now, possibly place enemies as children of the room scene (so you can set a NodePath to the player or find it on ready).
      * Compute direction to player: var dir = (player.global\_position - global\_position).normalized().
      * Set a speed for the enemy (maybe slower than player, e.g. 100 px/sec for testing).
      * Move the enemy: If using CharacterBody2D, do velocity = dir \* speed and call move\_and\_slide(). For KinematicBody2D, use move\_and\_collide(dir \* speed \* delta).
      * This will make the enemy constantly chase the player. It’s a very basic AI. You might also add a condition to only move if the player is within a certain range or if the enemy “sees” the player (for now, constant chase is fine).
    - **Collision Avoidance:** Because both player and enemy are physics bodies, they will collide and not pass through each other (due to the CollisionShapes). This can create physical pushing. That’s okay for now – it means the enemy will bump into the player rather than overlap completely. You may adjust collision layers if you want the enemy not to physically push the player (for example, allow overlap but still trigger damage).  
      * If you prefer overlap, you can remove the collision shape or set collision layers such that enemy and player do not collide as physics bodies, and rely solely on the Area2D for damage. However, having them collide gives a sense of solid bodies and might be fine.
    - **Enemy Attack on Contact:** We already set up AttackArea. Now ensure the logic:

In the enemy’s \_ready(), connect its AttackArea’s body\_entered signal to a function (e.g. \_on\_AttackArea\_body\_entered(body)). In that function, do something like:  
  
 gdscript  
CopyEdit  
if body.name == "Player": # or body.is\_in\_group("Player")

var player\_health = body.get\_node("Health")

if player\_health:

player\_health.take\_damage(1)

* + - * This will reduce the player’s health by 1 when the enemy touches the player.
      * Optionally, you could also emit a signal or call a method on the player to apply knockback or invincibility frames. For now, just reducing health will let us test the mechanic.
    - **Enemy Death:** In the enemy’s Health component, when health <= 0, you should handle death:  
      * The Health component will emit died. Connect this signal to a method on the Enemy (e.g. \_on\_EnemyDied) or to the Room manager. When triggered:  
        + Play a death effect (if any, like an animation or sound).
        + Remove the enemy from the scene (queue\_free() on the enemy node).
        + Inform the game that an enemy died – we will use this to check if the room is cleared. For instance, if using a Room script or manager, you could call something like room.on\_enemy\_died(self) or simply decrement a counter (if the room or manager is tracking enemy count).
      * Make sure to only call the room-cleared check after the enemy is fully removed or at least counted. We’ll detail room clear logic next.
* **Testing Combat in a Single Room:** At this stage, it’s wise to test the player and one enemy in a simple environment before adding room transitions.  
  + Create a quick test scene (could reuse Main or make a new scene like TestRoom.tscn) with:  
    - A floor (for visuals you could use a ColorRect or TileMap, or just leave blank for now).
    - Some walls or boundaries (StaticBody2D with rectangular CollisionShapes at the edges of the screen, so the player and enemies remain in view and can’t leave the area).
    - Instance the Player into this scene (drag the Player scene in as a child).
    - Instance one or two Enemies into the scene, placed a bit away from the player.
    - Hit play and try it out: The player should move and shoot; the enemy should chase. Check that:  
      * Bullets hit the enemy and the enemy loses health and dies after the correct number of hits.
      * The enemy touching the player causes the player’s health to drop.
      * When player or enemy dies, the nodes get removed properly (no errors).
    - Adjust any values for good feel: e.g. bullet speed, player speed vs enemy speed, health values. The combat should feel responsive: if it’s too hard to dodge the enemy, maybe slow it down. If shooting is too slow, maybe allow holding the key to fire continuously (could be an enhancement, like use is\_action\_pressed with a cooldown timer for rate of fire).
    - This is an **iterative tuning** step – since the prototype is about feel, don’t hesitate to tweak variables and re-run until movement and shooting feel snappy and fun.

### **Room Design and Progression**

* **Create Room Scenes:** Instead of one test room, now set up the structure for the infinite linear progression of rooms.  
  + Make a base room scene (e.g. Room.tscn) with Node2D as root (name it Room). This could act as a template for all rooms.  
    - Design the room’s layout: For simplicity, you can make each room the same size (e.g. 1024x768 or whatever fits your screen/camera). Use either a TileMap or simple static bodies:  
      * Add walls around the edges (StaticBody2D or TileMap collisions) so that the player and enemies can’t leave the room. If using StaticBody2D, four rectangular CollisionShape2D nodes (top, bottom, left, right boundaries) will do.
      * You can also design some obstacle inside (like a column or pit) to test navigating around, but keep it minimal for now.
    - Place an **Exit point** in the room: since progression is linear, we can imagine an exit door or staircase the player goes through to get to the next room.  
      * For example, add a **Marker2D** or Position2D node at the right edge (if we assume player goes left-to-right through rooms) or bottom edge (if top-to-bottom). This marker can denote where the next room will attach or where the player should appear when coming from previous room.
      * Alternatively, simply plan that when a room is cleared, we will load a new room and spawn the player at the center or some default location. (A visual door is optional in a prototype, but you could add a sprite to represent a door and place it at the edge as a goal.)
    - Save this Room.tscn. Now duplicate it to create a few variants:  
      * For instance, create Room\_Easy1.tscn, Room\_Easy2.tscn, etc., which have different enemy placements or slight layout differences. These will represent rooms in the “easy” difficulty pool.
      * Later rooms (higher difficulty) could have more enemies or tougher layouts. For now, you might just make two difficulty levels: “Easy” and “Medium”. Copy an easy room to a Room\_Medium1.tscn and increase enemy count or have a different enemy type (if you choose to add a second type).
      * Each room scene should contain the necessary nodes: the walls, maybe some decorative sprite for background, and spawn positions or the actual enemies.  
        1. You can either **place enemies manually** in each room scene (e.g. add an Enemy instance at design time where you want it), or use spawn points. For a few handcrafted rooms, placing manually is fine. It gives you full control per room.
        2. If placing manually, ensure the enemy nodes in the room have their health script attached and any necessary setup (like group assignment to "Enemies", etc.).
        3. If using spawn points, you’d have to write code to spawn enemies when the room loads. That’s more complex for now, so manual placement is recommended in this prototype scope.
      * Also add a **Position2D** for player spawn location in each room, especially if it’s not always at the exact same coordinate (e.g., in one room, the entry might be on the left, in another on top). Name it "PlayerSpawn". This will help position the player correctly when entering the room from the previous one.
      * Similarly, a "ExitSpawn" marker for where the next room’s player should appear might be useful (or simply use the same Position2D as the exit of the current room which corresponds to entry of next).
* **Room Manager (Progression Controller):** We need a system to load new rooms in sequence and handle transitions.  
  + One approach is to use the Main scene as the room manager:  
    - In Main.tscn, have a child Node (maybe just the root Node2D itself) with a script that keeps track of the current room index and difficulty.
    - Preload the room scenes to have them ready (for example, an array of PackedScenes for easy rooms and another for medium rooms).
    - Start by instancing the first room and adding it as a child of Main. Also instance the Player and add it to the room (or to Main). You might actually put the Player in Main and then move it into each room scene, but a simpler way: add the Player as a child of the current Room. That way, when we switch rooms, we can re-parent or reposition the player.

E.g. Main script:  
  
 gdscript  
CopyEdit  
var easy\_rooms = [preload("res://Scenes/Rooms/Room\_Easy1.tscn"), preload("res://Scenes/Rooms/Room\_Easy2.tscn")]

var med\_rooms = [preload("res://Scenes/Rooms/Room\_Medium1.tscn")]

var current\_difficulty = 0 # 0 for easy, 1 for medium, etc.

var current\_room\_index = 0

var current\_room\_node = null

* + - * When the game starts, you might call a function load\_next\_room() in \_ready().
  + **Loading a Room:** Implement a function (in Main or a dedicated autoload script called RoomManager) to transition to the next room:  
    - If current\_room\_node is not null (means an existing room is loaded), remove it:  
      * You can current\_room\_node.queue\_free() to delete the old room (and it will free all its child nodes, like enemies, automatically). Alternatively, you could keep previous room if you wanted backtracking, but in linear progression we don’t go back, so free it to save memory.
    - Determine which room to load next:  
      * Decide on difficulty progression. For example, after a certain number of rooms, increase the difficulty level (pull from medium pool instead of easy). Since it’s infinite progression, you could do something like: every 5 rooms, increase difficulty level by 1 (capped at max difficulty available).
      * For now, maybe start with all easy rooms until, say, room 10, then start medium. Or simply randomize from the easy pool for the first few, then include medium ones.
    - Pick a room scene from the appropriate pool (randomly or sequentially). Instance it: var room\_instance = room\_scene.instantiate().
    - Add it as a child of Main (or a specific container node under Main meant to hold the current room).
    - If the Player is not instanced yet (only at start), instance the Player and add as child of the room (or Main, but position relative to room).  
      * If the Player already exists from a previous room, you have two options:  
        1. **Keep Player persistent:** Don’t free the player between rooms. Instead, remove the player from the old room and add as a child of the new room. For example: player\_node.get\_parent().remove\_child(player\_node) then room\_instance.add\_child(player\_node).

Then update player’s position to the new room’s spawn point: player\_node.global\_position = room\_instance.get\_node("PlayerSpawn").global\_position (assuming you named the spawn marker).

* + - * 1. Alternatively, you could free and instance a fresh player each time, but that would reset health unless you carry it over manually. It’s usually better to keep the same player node so health and other states persist from room to room.
      * Since we have no permanent upgrades to carry, it’s simpler to keep the same player instance and just move it.
    - Position the player at the new room’s spawn:  
      * Use the "PlayerSpawn" Position2D in the new room. If you placed it in the room scene, do: var spawn\_point = room\_instance.get\_node("PlayerSpawn") then set player position to spawn\_point.global\_position. (Make sure to do this **after** adding the player as child of the room, so that the global\_position is relevant, or use room\_instance.get\_node("PlayerSpawn").position and add to room’s global transform).
    - Transfer the camera as well if it’s child of player (it will move automatically since it’s a child).
    - Keep track of current\_room\_node = room\_instance for reference.
  + **Room Clear Detection:** Each room should be cleared of enemies before moving on. We need to know when all enemies in a room are defeated:  
    - One simple way: When you instance a room, count its enemies. For example, in the Main script after adding the room, you can get the list of enemy nodes: var enemies = room\_instance.get\_nodes\_in\_group("Enemies") (if you assign enemies to an "Enemies" group). Or, just count how many Enemy nodes are children of the room (if you know all direct children that are Enemy, or mark them with a group).
    - Alternatively, connect each enemy’s death signal to a handler:  
      * For each enemy in the room, do something like: enemy.get\_node("Health").connect("died", self, "\_on\_EnemyDied") in the Main or room script. In \_on\_EnemyDied, increment a counter or simply call check\_room\_cleared().
    - Use a counter for alive enemies. Initialize it to the number of enemies in the room when loaded. Each time an enemy dies, decrement it. When the count reaches 0, the room is clear.
    - **Transition to Next Room:** When room clear is detected (enemy count 0):  
      * Possibly give the player a brief pause or a reward (e.g., spawn a health pickup or play a sound). For now, you can just print "Room cleared!".
      * Then allow moving to next room. You could require the player to do something (like press a key to continue or walk through a door). But since we have linear auto progression, you might just auto-transition after 1-2 seconds.
      * Maybe show a message or screen fade as feedback for transitioning. For a prototype, even a quick fade-out/fade-in using a ColorRect or ScreenFade node can make it feel smoother.
      * Call the load\_next\_room() function to switch scenes as described above. This will unload the current and bring the next.
    - **Persistent Data:** Ensure that player’s current health carries over to the next room (since we’re keeping the same player node, it will). If you were respawning the player anew, you’d have to pass the health value manually.
    - Increase difficulty as needed: before instancing the next room, you can increment a room counter. For example, rooms\_completed += 1. If rooms\_completed == 5 then set current\_difficulty = 1 to switch to medium rooms pool.
    - Randomness: Choose a random room from the pool of the current difficulty, so the test feels varied. If you have only one or two rooms in a pool, it might repeat often, which is fine for testing.
    - **Memory management:** Since this is infinite, it’s important we always remove the old room. The player and any persistent nodes should ideally not be duplicated endlessly. By freeing old rooms and reusing the player, we avoid memory leaks. (If you keep previous rooms around, the game would eventually slow down or crash.)
* **Health Pickups and Other Drops:** To keep the scope minimal, we only include health pickups:  
  + If you haven’t already, design a simple health pickup scene (HealthPickup.tscn). For example, an Area2D with a CollisionShape2D and maybe a heart icon sprite. Attach a script that on body\_entered with the player, increases the player’s health:  
    - e.g. player\_health.current\_health = min(player\_health.max\_health, player\_health.current\_health + 1).
    - Then remove the pickup (queue\_free).
  + You can place a health pickup in some rooms to test it. For example, maybe always drop one when a room is cleared. You can do that in the room manager logic: after clearing, instance a HealthPickup at the center of the room or at enemy's death position. Or simply pre-place one in some room scenes to simulate a drop.
  + This ensures we test the interaction of picking up items and the modularity (the player’s health component should update, and UI should reflect the change if we have a UI).

### **Testing and Polish**

* **Playtest the Full Loop:** Now run the game from the Main scene. Try clearing a few rooms in a row:  
  + Verify that transitions work: after you defeat all enemies, the next room loads, and the player appears in the correct spot with the same health.
  + Check that difficulty progression triggers (if implemented). For instance, if after 5 rooms you switch to medium, see that a “medium” room appears (with maybe more enemies).
  + Ensure there are no errors or crashes when loading/unloading rooms repeatedly. If something isn’t freeing properly, Godot’s output will warn you. Common issues could be forgetting to disconnect signals or trying to access freed nodes. If an enemy dies and the room is freed immediately, make sure any delayed calls are managed (you might want to delay the room unload slightly to ensure all enemy death logic finished).
  + Game Over: Let the player die to test this as well. When player health hits 0, whatever logic you implemented should trigger (like game restart or stopping the game). Since this is just a prototype, a full restart or a message is okay. If the game should restart, you can simply reload the current scene or send the player back to a main menu (if you have one). At minimum, print “Game Over” and perhaps reset the game after a couple seconds for convenience.
  + As you test, note how the movement and combat feel. Is the player’s speed good? Is shooting responsive? Adjust the tuning parameters (speed, fire rate, enemy health, etc.) to hit that satisfying feel. This might involve a lot of small changes and re-tests, which is normal at this stage.
* **Add Basic UI:** Even for an internal prototype, a simple UI can help visualize what’s happening (and will be needed to verify things like health).  
  + Create a simple HUD: For example, add a CanvasLayer node in the Main scene for UI. Inside it:  
    - Add a Label or TextureRect for the player’s health. This could be hearts icons or just text like "Health: 5". You can update it every frame or via a signal when health changes.  
      * E.g. connect the player Health component’s health\_changed (if you made one) to a UI script that updates the Label’s text.
      * Or poll the player’s health in \_process and update the label. Simpler: after each damage or heal, update the text.
    - (Optional) Show the current room number or score. A Label like "Room: 3" can be updated each time you load a new room. This helps ensure your progression logic is working (you see it counting up).
  + The UI should be minimal and not distract. It’s mainly for you during testing.
* **Polish Feedback (Optional but encouraged):** To make combat feel satisfying, add some quick feedback effects:  
  + **Screen Shake:** When the player takes damage or an enemy dies, you can add a small screen shake. In Godot, you might do this by briefly altering the Camera2D position. For instance, Camera2D has a method to apply impulse or you can script a shake by random offset for a few frames. Even a tiny shake on heavy actions can add juice.
  + **Flash Sprite on Hit:** When the player or enemy is hit, flash their sprite (e.g., modulate it to white or red, then back to normal). This gives a visual cue of damage. You can do this in the Health component’s take\_damage by, say, calling a method on the node’s sprite if it exists (you can pass a reference of the Sprite to the Health component, or have the owner handle it).
  + **Sound Effects:** If possible, add a couple of placeholder sounds (Godot can use WAV/OGG). Example: a shoot sound when firing, an enemy grunt or explosion on death, a player hurt sound. This greatly enhances perceived responsiveness. Keep volumes reasonable.
  + **Particles:** Add a particle effect for bullet impact or enemy death (like a small burst). Godot’s CPUParticles2D can be used with a default particle texture for sparks or smoke. This is optional eye-candy.
  + These polish items are not strictly required for functionality, but since the goal is to ensure the combat “feels good,” they can make a big difference in the perceived quality of the prototype. Even simple placeholders are fine (e.g., a quick color flash can stand in for a fancy animation).
  + Keep any polish toggled or easy to remove, so you can disable them if they cause issues or when you move to more advanced development.
* **Iterate based on Feel:** With these systems in place, play repeatedly and iterate:  
  + Maybe the rooms feel too empty or repetitive – you could add more enemy types (perhaps a stationary turret that shoots bullets, to test dodging). If you do, reuse components: e.g., make a new scene TurretEnemy.tscn that also has a Health component, but instead of moving, it periodically spawns bullets towards the player. This would test projectile interactions from enemies.
  + If the player’s shooting isn’t fun, tweak it: try faster fire rate, or allow holding the key for continuous fire (you’d need to implement a cooldown timer in the attack component).
  + Test edge cases: what if the player tries to leave before killing all enemies (our walls should prevent that)? What if two enemies overlap? What if the player dies exactly as the last enemy dies? Ensuring the game handles these (does the room still transition if player is dead?) will make the prototype robust.
  + **Document findings:** As you test, note what feels good and what doesn’t. The purpose is to ensure the core loop (enter room -> fight -> clear -> progress) is enjoyable. Any observation will guide what to focus on when expanding the prototype (e.g., “Combat feels good when there are 3-5 enemies, but gets tedious with 1 – so in the full game, spawn multiple enemies to keep it exciting”).

This completes the minimal combat prototype task list. Following these tasks will result in a playable prototype where you can move between rooms infinitely, fight basic enemies, and test the fundamental gameplay. Keep the code modular (each piece in its own script/node) so that adding new features (new enemy types, items, etc.) is straightforward in the future. Good luck, and have fun testing and tweaking the combat!

## **Version 2: Internal Design Notes (Detailed Architecture & Rationale)**

**Intent of the Prototype:** This internal document expands on the above task list with technical notes and design reasoning. The goal is to create a foundation for *Binding of Magical Persona* that can be easily extended. The prototype focuses on core gameplay feel: responsive controls and satisfying combat in a series of linear rooms. We deliberately exclude complex systems (procedural map generation, inventory, meta-progression, bosses) to concentrate on polish of basics. We are using a **component-based architecture** in Godot to keep the code modular. This means each game entity (player, enemy, etc.) will be composed of reusable pieces (movement, health, attack, etc.), rather than one monolithic script. This design will make it easier to scale up the project with additional features later. Below is a detailed breakdown of tasks, including how systems will interact, names of nodes/scripts, signals, and future expansion hooks.

### **Setup and Architecture Planning**

1. **Project Initialization & Scene Structure** We start a new Godot project (the engine version can be 4.x for latest features, or 3.x if needed – but we’ll assume 4.x). The project will be structured to promote clarity and reuse:  
   * **Filesystem Layout:** Create directories like Scenes/Characters/, Scenes/Rooms/, Scenes/Objects/ (for pickups, bullets), and Scripts/Components/ for reusable component scripts. This separation follows a component approach as recommended in Godot communities, where everything is a scene (Node) and can serve as a component in a larger scene[forum.godotengine.org](https://forum.godotengine.org/t/gentle-introduction-to-godot-component-paradigm/85556#:~:text=Unreal%2C%20Unity%20and%20Godot%20just,can%20represent%20a%20player%20component). For example, a Health.gd script in Scripts/Components can be attached to both player and enemy nodes.
   * **Main Scene:** Set up a Main.tscn (or call it Game.tscn) which will act as the entry point and hold the current room and maybe UI. This scene might just contain a root Node (e.g. Node2D) and possibly a UI CanvasLayer. We’ll attach a script (Game.gd) here to manage room transitions and overall game state (player’s persistent data like health across rooms, current room count, etc.).
   * **Autoloads (Singletons):** Consider if we need any autoloads. For a prototype:  
     + A Globals or RoomManager autoload could manage room loading logic globally. However, since our Main scene is doing that, an autoload might not be necessary. If the project grows, moving room management into a singleton might help separate game logic from scene.
     + An InputManager is likely unnecessary, as Godot’s InputMap and direct polling in player script suffice for now.
     + We might use an autoload for a global signal bus or a utility (for example, a AudioManager to play sounds, but again not strictly needed yet).
   * **Version Control:** If possible, set up version control (git) since we’ll be editing multiple scripts/scenes. This isn’t about code architecture per se, but it’s a good practice even for a solo dev to track changes as we iterate on “feel.”
2. **Input Configuration and Constants** In Project Settings, define input actions. Having them centralized in Godot’s Input Map allows easy rebinding later and makes code cleaner (using action names instead of hardcoded keys). We define:  
   * Movement: "move\_up", "move\_down", "move\_left", "move\_right".
   * Shooting/Attack: We have a design choice:  
     + **Option A:** Four actions for shooting (one in each direction, like Binding of Isaac with arrow keys). This gives full directional control independent of movement.
     + **Option B:** One action (e.g. "attack" or left mouse button) where the shoot direction is determined by another method (mouse position or last movement direction).
   * For this prototype, we lean towards Option A (cardinal shooting) because it’s easier for a beginner to implement without introducing mouse aiming or complex state. So we add "shoot\_up", "shoot\_down", "shoot\_left", "shoot\_right". This doubles the input count but keeps logic straightforward (each corresponds to a direction vector).
   * Also consider adding an "interact" or "confirm" action in case we want the player to press something to go to the next room or pick up items (though pickups can be auto-collected on overlap).
   * **Constants/Tuning:** It’s helpful to declare constants for things like speed or damage in scripts or a global config:  
     + Player speed, bullet speed, enemy speed, etc. We will expose many of these as exported variables in scripts for easy tweaking in the Inspector during testing.
     + For instance, in the Player movement script: export var move\_speed = 200.0.
     + We might create a Constants.gd autoload or just use script exports; given scope, exports are fine.
3. **Component-Based Design Approach** Godot does not have a formal Entity-Component-System built-in, but it encourages composition through Nodes. Each Node can be thought of as a component of a larger entity[forum.godotengine.org](https://forum.godotengine.org/t/gentle-introduction-to-godot-component-paradigm/85556#:~:text=Unreal%2C%20Unity%20and%20Godot%20just,can%20represent%20a%20player%20component). Our strategy:  
   * **Player as an Entity:** The player will be a scene composed of multiple nodes:  
     + A physics body for movement (so we leverage engine collision).
     + Child nodes for visual, collision, and logical components (health, maybe an attack point, etc.).
   * **Enemy as an Entity:** Similar structure to player, possibly with a different movement component (AI instead of player input).
   * **Reusable Components:** Scripts like Health, which can be attached to any node that needs health. This avoids duplicate code for player and enemy health mechanics.
   * We’ll use **signals** extensively. For example:  
     + Health.gd will emit died and possibly damaged signals. This decouples health changes from the rest of the game. The entity (player or enemy) can listen for its own Health’s died signal to handle death (e.g., an enemy plays death animation and frees itself, the player triggers game over routine).
     + The Room manager can also listen to enemy died signals (by connecting after instancing enemies) to know when to count down enemy count.
     + We may also use Godot’s built-in signals (like body\_entered on Area2D for collision events) rather than manual polling. This event-driven approach makes for clean, modular interactions.
   * **Groups for Organization:**
     + We’ll assign nodes to groups such as "Enemies", "Player", "Pickups" for quick identification. For example, the player node will be in group "Player" (even though there’s just one, it’s convenient for group-based checks or signals). All enemies in "Enemies" so that the room manager can do get\_tree().get\_nodes\_in\_group("Enemies") to retrieve them, or bullets can detect if a body is in group "Enemies" to apply damage.
     + Godot’s group system is a simple way to broadcast signals too (we could do get\_tree().call\_group("Enemies", "some\_method") to e.g. tell all enemies something, though we might not need that now).
   * **Inheritance vs. Composition:**
     + We will use composition (nodes) more than inheritance. Inheritance can still play a role (e.g., a base class for all enemies if we had variations, or a base class for all characters). For now, since we have just one player class and one enemy class, inheritance isn't necessary. But we might set up the structure so that adding another enemy type is easy (perhaps by inheriting from Enemy.gd or by composing different components).
     + For example, if we later add a ranged enemy, we could either create a new scene for it that shares the Health component and maybe a similar AI component but also has a shooting component. We can reuse the Bullet scene for enemy bullets too, which is nice because our bullet logic will be generic.
   * **Memory aid (class & node structure):**
     + **Player** (CharacterBody2D)  
       - Sprite2D (child)
       - CollisionShape2D (child, for body collision)
       - Health (child Node with Health.gd script)
       - [Maybe] Weapon or Attack node (could also just be handled in Player script)
       - [Optional] Camera2D (child, to follow player)
     + **Enemy** (CharacterBody2D)  
       - Sprite2D
       - CollisionShape2D (for body)
       - Health (child Node with Health.gd)
       - AttackArea (Area2D with CollisionShape for detecting player contact)
       - [Optional] other component nodes as needed (if we had separate AI node, but likely the Enemy script itself will handle movement AI).
     + **Bullet** (Area2D)  
       - CollisionShape2D
       - Sprite2D or Particles2D
       - (Script moves it and handles collision signals)
     + **Room** (Node2D)  
       - TileMap or StaticBody2D walls
       - Maybe some decorative sprites
       - (Enemies could be pre-placed here as child nodes, or spawned by script)
       - Position2D "PlayerSpawn" (where player should appear)
       - Position2D "ExitSpawn" (where next room’s entry is, if needed for positioning next room or just for reference)
       - [Optional] could have a child Node2D as a container for enemies/objects to keep the scene tree tidy.
     + **Game (Main)** (Node or Node2D)  
       - Script manages current Room
       - Possibly child node for UI (CanvasLayer)
       - Could hold a persistent Player node as child (or we dynamically add the player into rooms; design decision discussed later)
       - Could also contain a background node if we want a persistent backdrop (probably not, each room has its own environment).
   * Note: We ensure each component (Health, etc.) uses signals or method calls in a way that the main entity doesn’t need to know internal details. E.g., the Player doesn’t need to constantly check its Health; the Health will inform when critical events happen. This makes it easier to replace or modify components (say, swap out a different health system or add armor later) without rewriting the Player logic.
4. **Gameplay Variables & Balance Setup** Before implementation, define some initial values for gameplay:  
   * Player: speed ~200, health maybe 5, damage per shot 1.
   * Enemy: speed ~100 (so slower than player, giving player ability to kite), health maybe 3 (so a few shots to kill).
   * Bullet: speed ~400, lifetime ~1.5 seconds, damage 1.
   * Room progression: maybe after 5 rooms, difficulty goes up (spawn more enemies or tougher ones). For initial prototype, we’ll implement one basic enemy type; if we have time, a second type can represent “higher difficulty”.
   * These values will likely change during playtesting, but having a starting point is useful.
   * We’ll store these in scripts as exported constants so they can be tuned live in the Godot editor or tweaked quickly in code.

### **Implementation Tasks**

1. **Player Scene & Movement**
   * Create Player.tscn with root **CharacterBody2D** named "Player". (Using CharacterBody2D if on Godot 4 because it provides a built-in move\_and\_slide integration with customizable physics. If on Godot 3, use KinematicBody2D with similar methods. Either way, we get collision response.)
   * Add a Sprite for the player. If no art, a simple colored shape is fine. This is just to visualize the character.
   * Add a CollisionShape2D to Player. Shape: probably CircleShape2D or Capsule (for smooth movement around corners). This will collide with walls and enemies (if we allow physical collision).  
     + Adjust the collision layer/mask: We might set Player on layer 1, and have it collide with layer 2 (enemies) and layer 3 (walls). We’ll refine this when we also set enemy’s layers.
   * Attach a script Player.gd to the Player node. This will handle input and possibly coordinate other components:  
     + In Player.gd, define variables for speed, etc.: export var speed = 200.0.

\_physics\_process(delta): read input and move. Use Input.get\_vector("move\_left","move\_right","move\_up","move\_down") which returns a normalized vector from those four actions. Multiply by speed to get velocity.  
  
 gdscript  
CopyEdit  
var input\_dir = Input.get\_vector("move\_left","move\_right","move\_up","move\_down")

velocity = input\_dir \* speed

move\_and\_slide()

* + - This uses CharacterBody2D’s internal velocity property (which is a feature of Godot 4). In Godot 3, we’d do move\_and\_slide(velocity).
    - This single line handles stopping too (when input\_dir is (0,0), velocity becomes zero).
    - If we want to restrict diagonal speed to the same as cardinal (which get\_vector already normalizes), we’re good. Otherwise, normalization can be done manually.
    - We might also implement facing direction logic: e.g., if input\_dir is not zero, store last\_dir = input\_dir. This last\_dir can be used for shooting direction if needed.
    - No gravity here since it’s top-down (ensure if using CharacterBody2D to set gravity scale 0 or use move\_and\_slide with infinite inertial).
  + **Component integration in Player:**
    - Add a child Node "Health" (Node type is fine). Attach Health.gd (we will create this script soon) to it. Set its max\_health in the inspector, say 5.
    - Also add any other children if splitting scripts: For instance, we could add a Node "Weapon" or "Attack" to handle shooting. Or we can handle shooting in Player.gd itself for now. A modular approach: create an Attack.gd component that you attach to "Weapon" node.  
      * If we do that: The Weapon node might need to know input too. We could forward input to it or let it poll input as well. Alternatively, it could simply handle firing logic (cooldown, spawning bullet) triggered by Player.
      * To keep it simple: implement shooting in Player.gd, but in a way that could be refactored. We’ll not create a separate node for Attack at this moment to avoid too many moving parts, but we keep the code isolated in its own section for easy migration to a component.
    - Connect the Health node’s signals: In Player.gd \_ready(), get the Health node (health = get\_node("Health")) and connect:  
      * health.connect("died", self, "\_on\_PlayerDied").
      * Possibly connect a "health\_changed" if we want (or we can just access health.current\_health when updating UI).
      * The \_on\_PlayerDied function in Player.gd might, for now, just print "Player died" and disable player controls. In a full game, it would trigger a game over sequence. For our testing, we might call get\_tree().reload\_current\_scene() to restart, or simply stop movement input by not processing it further.
    - If using groups, add the player to group "Player": add\_to\_group("Player") in \_ready or via the Node tab in editor. This helps enemies/pickups identify the player node easily.
    - Summarizing, the Player entity now has:  
      * Movement logic (in Player.gd).
      * Health logic (in Health.gd, attached to child).
      * Attack logic (to be added next).
      * Proper signals and group set up.
  + **Camera:** If we use a Camera2D as child:  
    - Enable current on it so it follows the player. We might set camera.drag\_margin or smoothing if needed, but since rooms are small, an instant camera is fine.
    - Ensure it’s limited to room bounds if the room is larger than screen, but we can skip that if each room fits the view or if the camera isn’t strictly necessary (some designers prefer fixed camera per room like classic Zelda/Isaac).
    - We can revisit camera settings once rooms are built (e.g., we might want it to not show outside the room—Godot Camera2D has limit settings that we can set based on room size).

1. **Player Shooting (Attack Mechanics)**
   * **Bullet Setup:** We create Bullet.tscn:  
     + Root: Area2D (name it "Bullet").
     + CollisionShape2D (child of Bullet). Use a small CircleShape2D for collision. Set its collision layer to "Bullet" layer (perhaps layer 4) and mask to collide with "Enemy" and "Wall" layers but not with "Player".  
       - For example, assign: Bullet layer = 4; mask = 2 (enemy) and 3 (walls). Enemy layer = 2, mask = 1 (player) and 4 (bullet), etc. We’ll define enemy’s layer soon accordingly.
     + Sprite: optionally add a Sprite for visibility (maybe a 4x4 white dot or something). You can also just rely on collision debug visuals initially.

Attach script Bullet.gd. Key elements in Bullet.gd:  
  
 gdscript  
CopyEdit  
@export var speed = 400.0

@export var damage = 1

var velocity = Vector2.ZERO

* + - * We export speed and damage for easy tweaking.
      * velocity will be set by whoever spawns the bullet (the Player or possibly an enemy if reused).
      * In \_physics\_process(delta): we move the bullet. We could use position += velocity \* delta. This doesn’t use physics for movement (which is okay because we rely on Area2D overlaps for collision, not physical response).
      * We need to detect collisions manually if moving this way. Alternatively, we could use move\_and\_collide() by making Bullet a KinematicBody2D, but then detecting collisions with specific bodies is a bit different.
      * Another approach: keep it Area2D and use its detection:  
        + Set Bullet’s CollisionShape to be a **sensor** (so it won’t stop on collision but will detect overlaps). Or keep as non-sensor if we want it to collide and stop; but since it’s not a physics body, it won’t automatically stop anyway.
        + Connect Bullet’s body\_entered signal to a function in Bullet.gd \_on\_Bullet\_body\_entered(body).
        + In that function, we check if body is in group "Enemies": if yes, we tell that body to take damage.

Assuming enemy has a method or we can access its Health node: we can do if body.has\_method("apply\_damage") call it. Or simply var enemy\_health = body.get\_node("Health"); enemy\_health.take\_damage(damage).

Actually, we need to design how enemies take damage. We decided to have Health component with take\_damage. So doing the above is fine. We just have to ensure the enemy’s CollisionShape or body is the one passed. Because if Bullet collides with an enemy’s physics body (CharacterBody2D), body will be the CharacterBody2D node (which doesn’t directly have Health script, but its child does).

So, body.get\_node("Health") should find the child. That will work if the child is named "Health" (which we plan). Alternatively, we could put the Health script on the root of Enemy too (we could attach Health.gd directly to Enemy node rather than a child; either way works. Child keeps it more modular, but direct attach is slightly simpler to get via body reference. We’ll do child for modularity and just fetch it).

* + - * + After dealing damage, call queue\_free() on the bullet to remove it.
        + Also, handle if bullet hits a wall (we might want bullets to disappear on walls). If we set collision to detect walls, then in body\_entered if the body is in group "Walls" or a StaticBody2D, just free the bullet without damage event.
        + If bullet hits the player (shouldn’t if layers set right), we can ignore or ensure mask prevents that.
      * Also, add a lifespan: in \_ready(), we could start a Timer node or simply use yield(get\_tree().create\_timer(1.5), "timeout") then queue\_free(). This ensures the bullet is removed after 1.5s even if it never hit anything (so it doesn’t fly forever).
      * Note: Another method is using move\_and\_collide():  
        + If bullet were a KinematicBody2D, we could do var collision = move\_and\_collide(velocity \* delta) and if collision, check collision.collider is enemy etc. That’s a fine approach too. However, using Area2D’s signal might be simpler for a beginner to grasp since it avoids manual collision logic. We’ll stick with Area2D approach.
  + **Player Shooting Implementation:** In Player.gd, add code to handle the attack input:  
    - In \_physics\_process (or \_process since input checking can be fine in either if not frame-perfect critical):

Check for shoot inputs. If using discrete actions:  
  
 gdscript  
CopyEdit  
if Input.is\_action\_just\_pressed("shoot\_up"):

shoot(Vector2.UP)

if Input.is\_action\_just\_pressed("shoot\_down"):

shoot(Vector2.DOWN)

...

If using one action "attack":  
  
 gdscript  
CopyEdit  
if Input.is\_action\_just\_pressed("attack"):

if last\_dir != Vector2.ZERO:

shoot(last\_dir)

* + - * (We would have maintained last\_dir from movement as discussed.)
      * The shoot(dir: Vector2) function (we write this in Player.gd) will:  
        + Instantiate a Bullet: var bullet = BulletScene.instantiate(), where BulletScene is a class member preloaded at top: const BulletScene = preload("res://Scenes/Objects/Bullet.tscn").
        + Set bullet position to player’s position (or maybe a bit in front of the player in the direction of shooting, so it doesn’t start inside the player’s collision shape to avoid immediate collision).

For example: bullet.position = global\_position + dir \* 10 (assuming 10 pixels in front).

* + - * + Set bullet’s velocity: bullet.velocity = dir \* bullet.speed (since bullet script has an exported speed, we can use it, or we could define bullet\_speed in Player and set velocity directly).
        + Add the bullet to the scene tree. Ideally, add it to the same parent as the player (which will be the Room currently). If the player is a child of Room, then get\_parent().add\_child(bullet) suffices (that adds bullet to Room).
        + Optionally, play a shooting sound.
        + If we want a continuous fire (holding key), we would need a cooldown timer to throttle bullets. For now, is\_action\_just\_pressed ensures one bullet per key press, which is fine.
      * Collision layers: We ensure the bullet doesn’t collide with the player by setting masks appropriately, but also spawning it slightly in front helps avoid immediate intersection.
      * We should also mark bullet as belonging to "Player" side so we don’t accidentally consider it an enemy projectile. The collision layers already handle that logically.
    - Confirm that in the bullet’s body\_entered, when it calls enemy\_health.take\_damage(1), the Health script for enemy will handle reducing health and possibly emitting died.  
      * We might also consider if multiple bullets hit at same time, but that’s rare. The health logic should handle going negative gracefully and only emitting died once (maybe ignore further damage after death).
    - Also, define what happens to an enemy when health <= 0: likely in Health.gd we’ll emit died and maybe call queue\_free() on the parent (the Enemy). But auto-freeing the parent from Health might not be ideal if we want to do something in enemy script first (like play animation). Instead, Health will emit died, and the Enemy script will listen and handle the free.  
      * We should ensure bullet doesn’t free itself before applying damage and doesn’t try to damage a freed enemy. But if we do bullet -> damage -> then enemy might queue\_free -> then bullet queue\_free, the order is fine. The signal emission and processing order might matter a bit, but as long as we queue\_free enemy, it stays in tree until end of frame, so bullet’s collision detection is fine.
  + **Summary of Attack Design:** By implementing shooting this way, the player’s attack logic is decoupled from the enemy’s response:  
    - The bullet knows how to apply damage to whatever it hits.
    - The enemy’s health component knows how to respond to damage (reduce HP and die if 0).
    - The player just spawns bullets and doesn’t need to know about enemy specifics. This is good modular design – e.g., later if we give enemies bullets, we can reuse the Bullet scene and just spawn them from enemy side (maybe on a different collision layer for player detection).
    - This architecture also allows adjusting bullet speed/damage per source if needed (e.g., different bullet scene or variable bullet damage).

1. **Health Component Implementation**
   * Create Health.gd script (no specific node type needed, can attach to Node or Node2D). It will serve as a general health manager for any character.

Properties:  
  
 gdscript  
CopyEdit  
@export var max\_health: int = 3

var current\_health: int

signal died

signal health\_changed(current\_health)

* + - We give a default max\_health which can be overridden per instance (player might set 5, enemies 3, etc.).
    - current\_health will be set in \_ready(): current\_health = max\_health.
  + Methods:

func take\_damage(amount: int):  
  
 gdscript  
CopyEdit  
if amount <= 0:

return

current\_health -= amount

emit\_signal("health\_changed", current\_health)

if current\_health <= 0:

current\_health = 0

emit\_signal("died")

* + - * We emit health\_changed for UI or other uses. We emit died when health drops to 0 or below. (We ensure to clamp at 0 to avoid negatives.)
      * We do *not* free the node or its owner here, because we want the owning scene (player/enemy) to decide what to do on death. This is important – e.g., the enemy might play an animation or drop loot.
      * The player’s case: we might not free the player immediately on death (we might trigger game over sequence).
    - We could also add a heal(amount) method to increase health (used for pickups), with similar signal emission for health\_changed.
  + Usage:  
    - Attach Health.gd to the Player’s Health node and to the Enemy’s Health node. Set their max\_health via inspector (player maybe 5, enemy maybe 3 as planned).
    - In Player.gd, we connected the died signal already. We might also connect health\_changed to update a health UI label if desired.
    - In Enemy.gd (to be created), we will connect the enemy’s Health died signal to the enemy’s own death handler.
  + The Health component design ensures any object (player, enemy, boss, etc.) can use it. In future, if we add an item that gives the player more max health, we can simply adjust max\_health or current and perhaps emit health\_changed to update UI. If we add shields or armor, we might extend this system or make a separate component, but the basic health can remain.

1. **Enemy Scene & Behavior**
   * Create Enemy.tscn with root CharacterBody2D named "Enemy". Add:  
     + Sprite2D (placeholder graphic).
     + CollisionShape2D for body collision (circle or rectangle). Assign to collision layer 2 ("Enemies") and mask to collide with layer1 (player) and layer3 (walls) and maybe bullets (layer4) if we want bullets to register as collisions. Actually, bullet detection uses Area2D signals, not body-body collision, so the enemy body’s mask doesn’t need to include bullets – the bullet’s mask includes enemy layer and that triggers area signal. So for enemy body, mask can just include walls and player for physical interactions.  
       - We do want the enemy to stop when hitting walls, so colliding with walls (layer3) is important. Colliding with player (layer1) if we want them to bump each other (which likely we do).
     + Instance a "Health" Node (child) and attach Health.gd. Set max\_health = 3 (or any difficulty-scaling value).
     + Add an Area2D "AttackArea" (child of Enemy). Give it a CollisionShape2D (slightly larger than the body or same size). This area will be used to detect the player for contact damage.  
       - Set AttackArea’s collision layer perhaps same as enemy’s or separate? If AttackArea is just for detection, we can put it on layer 5 ("EnemyAttack") and set its mask to collide with Player (layer1). It’s a sensor essentially.
       - We will connect AttackArea’s body\_entered signal in the enemy script to handle when the player enters the area.
     + Add to group "Enemies" (via the Node->Group editor or code in \_ready).
   * Attach script Enemy.gd to the Enemy root. This script will manage movement and coordinate with components:

Variables:  
  
 gdscript  
CopyEdit  
@export var move\_speed = 100.0

var player\_ref = null

* + - We’ll need a reference to the player to chase it. How to get it:  
      * If the player is always present as a unique Node, we might get it by name or group. E.g., player\_ref = get\_tree().get\_nodes\_in\_group("Player")[0] in \_ready (assuming we added player to "Player" group).
      * Or if the room adds enemies after adding player, perhaps we can pass the player reference to the enemy (like if we spawn enemies via script, we can do enemy.player\_ref = player).
      * Since we might place enemies in the scene ahead of time, they need to find the player at run-time. Using the group method is straightforward as long as the player is added to group "Player".
    - In \_physics\_process(delta):  
      * If player\_ref is not null and player\_ref is still in scene (we might check if player\_ref and player\_ref.is\_inside\_tree():):  
        + Compute direction: var dir = (player\_ref.global\_position - global\_position).normalized().
        + Move: for CharacterBody2D, we set velocity = dir \* move\_speed and call move\_and\_slide().

Actually, CharacterBody2D expects us to call its built-in physics integration. There’s a nuance: CharacterBody2D in Godot 4 has an internal velocity we set and then we call move\_and\_slide() (which uses that velocity and updates it for collisions). So something like:  
  
 gdscript  
CopyEdit  
velocity = dir \* move\_speed

move\_and\_slide()

And ensure to set velocity as a member of CharacterBody2D (which might be the case because CharacterBody2D extends PhysicsBody2D with velocity property).

In Godot 3 KinematicBody2D: we’d do move\_and\_slide(dir \* move\_speed).

* + - * + This will push the enemy toward the player. Collisions with walls or player will automatically happen because of physics:

If enemy collides with a wall, move\_and\_slide prevents moving through it.

If enemy collides with player’s physics body, by default move\_and\_slide might treat the player as obstacle and slide along or stop. This might make the enemy stop at the player boundary, effectively “touching” the player which is what we want for contact damage.

We might consider using move\_and\_slide with no gravity and infinite inertia so it doesn’t bounce off.

* + - * + We could implement a simple state machine (idle, chase, attack) but for minimal we can consider the enemy always chasing. If we wanted an idle until player close, we could check distance and only move if within some range.
      * We won’t implement pathfinding or complex AI. The rooms are small and obstacles minimal, so direct line movement is fine. If later obstacles become an issue, we could add a Navigation2D node and use pathfinding, but that’s beyond prototype need.
    - Connect the signals:  
      * In \_ready(), after setting player\_ref, connect AttackArea:  
         get\_node("AttackArea").connect("body\_entered", self, "\_on\_AttackArea\_body\_entered").
      * Also connect Health:  
         get\_node("Health").connect("died", self, "\_on\_EnemyDied").
      * (If using health\_changed for something like flashing sprite on hit, we could connect that too, but we can also handle flash in take\_damage by accessing the owner. It might be cleaner to handle it here: e.g., connect health\_changed to a method that does: if current\_health < old\_health then flash.)

Define \_on\_AttackArea\_body\_entered(body):  
  
 gdscript  
CopyEdit  
if body.is\_in\_group("Player"):

# Player entered attack range

var player\_health = body.get\_node("Health")

if player\_health:

player\_health.take\_damage(1)

# Optionally, we can implement a cooldown to not immediately hit again.

# Perhaps disable the AttackArea for 0.5s:

get\_node("AttackArea").monitoring = false

await get\_tree().create\_timer(0.5).timeout

get\_node("AttackArea").monitoring = true

* + - * This means when the enemy touches the player, the player takes 1 damage. We temporarily turn off monitoring so the signal doesn't fire continuously every physics frame while overlapping, then re-enable after 0.5 seconds to allow damage again (simulating invincibility frames).
      * The await (yield) usage is a GDScript way to wait on a timer. Alternatively, use a Timer node.
      * This is a simplistic approach. It doesn’t push the player or enemy away. If we want knockback, that’s more advanced (we could apply an impulse to player or enemy’s velocity opposite direction). For now, we skip knockback to avoid complexity.

Define \_on\_EnemyDied():  
  
 gdscript  
CopyEdit  
# Called when this enemy's Health signals "died"

# We can play a death animation or effect here:

if $Sprite: # if a sprite exists

$Sprite.modulate = Color(1,1,1,0) # make invisible or you could start an anim

# If we had a sound, play it

# Queue free the enemy node:

queue\_free()

# Additionally, notify room manager:

# We could emit a signal from Enemy.gd like enemy\_dead (but easier: the Room or Main will detect the freed node or was counting via signals)

* + - * Actually, if Main is tracking via signals, we might not need to notify explicitly here. But it could be helpful to emit something. Another method: the Room can connect to enemy's tree\_exited signal to know it was removed. But using the health died is straightforward.
      * We may prefer not to free immediately if we want an animation. In that case, we’d start an animation and then free after a short delay. But for now, instantaneous removal is okay (especially if we flash or make sprite transparent to show it's dead).
      * Important: If we free here, the died signal was emitted from Health, which we caught. We should ensure the Room manager also knows. If we connected the Health died to Enemy’s \_on\_EnemyDied, we should also connect Health died to the Room script or Main (which will be done at room load time). Or we can rely on the fact that queue\_free will remove the node and then possibly the Main could detect that the count of enemies changed. But more robust is connecting the signal.
      * So plan: When spawning or when room enters tree, Main will connect each enemy’s Health.died to a Main function that decrements the alive count (or calls a method on the room script).
      * We should avoid double freeing: here Enemy does queue\_free, if Main also tries to free maybe, but Main likely will only track count and not free itself. So it’s fine.
    - With this, each enemy autonomously handles its chase and attack and death, while signaling outward. The player doesn’t directly control or know about enemy internals; the main game loop just monitors enemy count via signals or group queries.
  + **Enemy Placement and Prefabs:** We can design multiple enemy scenes if needed:  
    - For now, one enemy type is enough. If we wanted a slightly tougher enemy for later rooms, we could make a variant scene perhaps with higher health or different color. Or spawn two at once to simulate higher difficulty.
    - Since they are all instances of the same scene, adjusting their properties (like speed or health) individually can be done either by making distinct scenes or by modifying after instancing (like setting enemy.health.max\_health after creation).
    - We'll implement difficulty by quantity rather than new type to keep it simple.

1. **Room Scenes and Progression System**
   * **Room Scene Template:** Create a scene RoomTemplate.tscn as a blueprint:  
     + Node2D root named "Room".
     + Add a layer for visual floor (optional). Could be a ColorRect or TileMap. E.g., place a TileMap with a simple floor tile covering some area, just for visual reference of room size.
     + Add boundary walls: Four StaticBody2D nodes, each with CollisionShape2D (RectangleShape2D). Position them at edges of desired room area.  
       - For example, if the camera viewport is 800x600, you might define walls at x=0, x=800, y=0, y=600 as boundaries (with some thickness).
       - Or use a TileMap for walls. But static bodies are quick.
       - Put walls on a separate layer (layer3 "Walls") and have them collide with player and enemy bodies and bullets. (Our bullet mask includes walls, so it will detect collision and we should program bullet to destroy on walls.)
     + Mark a **PlayerSpawn** (Position2D) where the player will start in this room. For linear, likely on the left side if the exit is right, or in center. Actually, since it’s linear progression, if we imagine the player always comes in from one side and leaves from the other:  
       - For example, player enters from left, exits right. So spawn at left-middle.
       - Place PlayerSpawn accordingly (say at (50, 300) if 800x600 room).
     + Mark an **ExitSpawn** (Position2D) where the next room’s entrance would align. Possibly at the right-middle (e.g., (750,300)). This could be used if we wanted to position the next room relative to current (for instance, physically adjacent). However, in our approach, we’re unloading and loading rather than physically stitching rooms in one world, so ExitSpawn might not be needed. Instead, we may just teleport player to next room’s PlayerSpawn.  
       - Alternatively, we could use ExitSpawn to place a door sprite and also as a trigger: when player overlaps it and room is clear, then load next room. This is a more interactive approach rather than auto-transition.
       - For now, we can skip having an actual door trigger, and just do auto transition via code. So ExitSpawn is just informational or can be omitted.
     + Save RoomTemplate. Then make a couple of instances for variety:  
       - e.g., Room\_Easy1.tscn, Room\_Easy2.tscn. Each can be based on template or copy-paste. Populate:  
         * In each, instance a few enemies (drag the Enemy scene in as child of Room). Position them differently in each room.
         * E.g., Room\_Easy1: 2 enemies, placed relatively far from spawn. Room\_Easy2: 3 enemies, maybe different arrangement.
         * Ensure to adjust their properties if needed (the instances can override exported values like move\_speed or health if desired in Inspector).
         * You could also place a HealthPickup somewhere in one of these as a test. Or we can spawn it by code after clearing.
         * Make sure to double-check each placed enemy has its Health node and such. When you instance the scene, those should come along with it.
       - If doing difficulty escalation:  
         * Keep “easy” rooms with fewer enemies. Then make a “medium” room (Room\_Medium1.tscn) perhaps with more enemies (or stronger enemies if you decided to differentiate by a property).
         * Could simulate a stronger enemy by simply placing more or by giving one enemy a higher health via inspector override (since max\_health is exported).
       - This manual design of rooms meets the requirement "each room is a hand-crafted scene chosen from difficulty pools".
     + All room scenes should share general structure (so our code can handle them generically). They all have a PlayerSpawn etc.
     + If needed, attach a script to Room scenes (e.g. Room.gd) to manage room-specific things (like spawning enemies at runtime or handling events). But since we pre-placed everything and the Main script will oversee transitions, we might not need a Room.gd. We can manage enemy signals from Main. However, having a Room.gd could be useful to encapsulate room logic (like “when room start, open door, etc.”). For prototype, it might be overkill, so we skip attaching a script to each room and manage externally.
   * **Room Loading & Manager Logic:**
     + We will handle room transitions in the Main scene’s script (Game.gd or similar).
     + In Main.tscn, ensure there is a Node to hold the room. This could simply be the Main’s root if we always add the room as a child. Or create a child Node (e.g., Node2D named "RoomContainer") to add rooms into, which helps isolate them from UI nodes.

Using a container node might be clean: e.g.,  
  
 scss  
CopyEdit  
Main (Node)

- RoomContainer (Node2D)

- CanvasLayer (for UI)

- maybe Player (but we might instance player separately)

* + - * Actually, deciding where the Player lives is important:  
        + We could have Player as a persistent child of Main (outside RoomContainer), and move its position when changing rooms. But if we want to confine physics interactions inside the room scene, it’s easier if Player is a child of the room scene.
        + If Player is outside, collisions with room walls might not register because it’s not in the same scene branch? Actually, collisions would still register if layers/masks overlap globally. But moving the player between rooms becomes just position change vs reparent.
        + Another reason to keep player as a child of room: if we free the room, it would free the player too (not good). So if we want to free room without freeing player, player should not be a child of it.
        + Possibly best: keep Player as a child of Main (persist across rooms). Then every time we load a new room, we reposition the player to the new room's spawn point. We must ensure collisions still work (since the walls/enemies are in RoomContainer and player in Main, but since they are all in the scene tree and collision is by physics engine, it should be fine).
        + We have to also ensure the camera follows properly; if camera is a child of player, and player is now not in the room node, camera might need adjusting to limit to room.
        + Simpler: we can reparent the player to the RoomContainer each time. That is, we treat the player as moving from one scene to another physically.

That could be: remove player from old room (or Main) and add as child of new room node. But the downside is when we free the room, we would also free the player unless we take it out first.

So the safe sequence would be: remove player from old room, add to Main (or keep in Main temporarily), free old room, instance new room, add player into new room.

* + - * + This is a bit juggling. It might actually be easiest to simply **not free the old room until after moving player**. Or maintain player outside.
        + Another approach: do not child the player under room at all. Keep it under Main always, and just change its position. The collisions will still work as long as layers match because physics in Godot doesn’t require common parent (it’s world-based).

The only complexity: camera limiting to room boundaries won't be automatic because camera doesn’t know room edges. But we can set camera limits when room changes (like camera limit rect = room size).

To keep it straightforward: we’ll opt to keep Player under Main. It persists. We won’t reparent it each time, just reposition.

We will free and instance room scenes as needed, but not containing player, so it’s fine.

We must be careful that if the player was a child of room originally (in our test we did that), we should change that now. Perhaps we spawn the player in Main at game start and not as part of the room scene. We might need to change how we tested earlier, but that’s okay for final structure.

* + - Implement Main.gd:

Preload room scenes arrays:  
  
 gdscript  
CopyEdit  
const EasyRooms = [ preload("res://Scenes/Rooms/Room\_Easy1.tscn"), preload("res://Scenes/Rooms/Room\_Easy2.tscn") ]

const MediumRooms = [ preload("res://Scenes/Rooms/Room\_Medium1.tscn") ]

var current\_difficulty = 0

var room\_count = 0

var current\_room = null

* + - * We preload for performance (Godot loads them at scene load time rather than during gameplay, reducing lag when instantiating[docs.godotengine.org](https://docs.godotengine.org/en/stable/tutorials/scripting/nodes_and_scene_instances.html#:~:text=Preloading%20the%20scene%20can%20improve,is%20only%20available%20with%20GDScript)).

In \_ready(), instance the Player:  
  
 gdscript  
CopyEdit  
var PlayerScene = preload("res://Scenes/Characters/Player.tscn")

player = PlayerScene.instantiate()

add\_child(player)

player.position = Vector2(...) # we will set proper position after room loads, but maybe put it temporarily somewhere off-screen

* + - * Add to Main, not inside a room yet.  
         Add player to group "Player" if not already in its scene (we did in Player.gd likely).

Load the first room:  
  
 gdscript  
CopyEdit  
func load\_room(room\_packed):

# If a room is currently loaded, remove it:

if current\_room:

# If player is child of current\_room, remove it first (not the case with our design now, but just in case)

if player.get\_parent() == current\_room:

current\_room.remove\_child(player)

add\_child(player) # put back to Main

current\_room.queue\_free()

current\_room = null

# Instance new room

var new\_room = room\_packed.instantiate()

# If we have a designated container, add to it, otherwise to Main:

add\_child(new\_room) # as child of Main

current\_room = new\_room

# Position the player at the new room’s spawn point:

if current\_room.has\_node("PlayerSpawn"):

player.global\_position = current\_room.get\_node("PlayerSpawn").global\_position

else:

player.position = Vector2(100,100) # fallback, ideally every room has a spawn

# If player was outside, it's already in Main which is fine. If we wanted player as child of room for collision, do:

# current\_room.add\_child(player) (and remove from Main). But we'll keep player in Main.

# Now connect enemy signals in the new room:

var enemies = current\_room.get\_children() # or better, get\_nodes\_in\_group("Enemies")

enemy\_count = 0

for e in enemies:

if e.is\_in\_group("Enemies"):

enemy\_count += 1

# Connect that enemy's Health.died to a handler:

var health = e.get\_node("Health")

if health:

health.connect("died", self, "\_on\_EnemyDied")

```

We maintain a variable `enemy\_count` in Main to track how many enemies are alive in the current room.

- Actually, using `current\_room.get\_children()` assumes all enemies are direct children. If some are inside a subgroup node, we should either flatten or use get\_nodes\_in\_group which is easier:

`for e in current\_room.get\_nodes\_in\_group("Enemies"):` yields all enemies anywhere in that room's scene tree (provided we added them to group).

- Connect signals for each enemy’s death. We connect to Main’s `\_on\_EnemyDied`. We might also connect the player's died if we want to handle game over in Main, but we already did player died in player script (that might just queue reload). We could handle it centrally too.

- If the room has pickups or other things that might need connections (like maybe a door Area if we had one), connect those similarly.

Implement \_on\_EnemyDied() in Main:  
  
 gdscript  
CopyEdit  
func \_on\_EnemyDied():

enemy\_count -= 1

# Optionally disconnect the signal if needed (Godot auto-disconnects on emitter freed, so not strictly required)

if enemy\_count <= 0:

on\_room\_cleared()

* + - * This assumes each connection calls the same method. Actually, better to connect with deferred or each connection separate? It's fine, multiple enemies can connect to same handler.  
         We don’t get which enemy died here, because we didn’t pass a bound parameter. But we don’t need it, just decrement count. Alternatively, we could connect to a lambda that calls \_on\_EnemyDied(e) by binding the enemy or health node if needed to drop specific loot at enemy position. For now, not needed.

on\_room\_cleared() (or \_on\_RoomCleared):  
  
 gdscript  
CopyEdit  
func on\_room\_cleared():

print("Room clear!")

room\_count += 1

# Optionally, drop health pickup:

var pickupScene = preload("res://Scenes/Objects/HealthPickup.tscn")

var pickup = pickupScene.instantiate()

# Place it at some location, e.g., center of room or player's position

pickup.global\_position = player.global\_position

current\_room.add\_child(pickup)

# The pickup itself will handle giving health on collision.

# Increase difficulty if needed:

if room\_count == 5:

current\_difficulty = 1 # move to medium after 5 rooms

# Schedule next room load:

# We can either auto-load after a delay or wait for player input.

yield(get\_tree().create\_timer(1.0), "timeout") # 1 second delay

# Choose next room from pool:

var next\_room\_packed

if current\_difficulty == 0:

next\_room\_packed = EasyRooms[randi() % EasyRooms.size()]

elif current\_difficulty == 1:

next\_room\_packed = MediumRooms[randi() % MediumRooms.size()]

else:

next\_room\_packed = MediumRooms[randi() % MediumRooms.size()] # if difficulty beyond defined, just reuse highest

load\_room(next\_room\_packed)

* + - * + This will remove the old room (in load\_room) after the timeout and proceed.
        + The delay gives a brief respite and ensures signals from dying enemies are processed before we kill the scene. (If we immediately free room on enemy death when count hits 0, there might still be some overlap in signals; a short delay is safer.)
        + Instead of yield, we could also set a boolean and in process check after some time. But yield with timer is straightforward.
        + We drop a health pickup at player’s position (or wherever). In a real scenario, maybe drop at center or random, but this is fine to test picking up.
        + Increase difficulty: after 5 rooms, use medium pool. If we had multiple difficulty tiers, continue logic similarly (like if we had hard, at 10 or so).
        + We randomly select the next room from the pool to avoid repetition. If you want to avoid repeating the same room back-to-back, you could track last used index. But not important in prototype.
        + Note: Because we keep player persistent, we also keep their health from room to room. If they took damage, they carry that forward, which adds challenge. The health pickups allow some recovery. This mimics roguelike progression where you carry your health through levels.
        + If we wanted to reset health each room (not typical in roguelike), we could, but we won’t.
        + If player dies, currently our player script will queue reload or something. We may refine that:

Possibly override the player’s death handling to call a game\_over() in Main. We can connect player Health died to Main similarly:  
 player.get\_node("Health").connect("died", self, "\_on\_PlayerDeath").

In \_on\_PlayerDeath: maybe print "Game Over - rooms cleared: X" and reset:  
 yield(get\_tree().create\_timer(1.0), "timeout"); get\_tree().reload\_current\_scene().  
 This would reset to the initial state. Or we could call load\_room(EasyRooms[0]) after resetting some values.

For now, it’s enough that the prototype ends or resets on death.

* + - * Memory management:  
        + We ensure to queue\_free rooms when done. We keep one room loaded at a time. This prevents memory buildup in the infinite loop.
        + We reused the player and didn’t free it, which is good because freeing and re-instancing could cause hiccups and lost state. The player persists.
        + Enemies are freed on death, and then the room is freed freeing any remaining (if something glitch left). We should avoid double freeing or accessing freed objects:

When we free the room, all its child enemies (if any still alive) get freed automatically. But we connected their signals to Main. Godot will auto-disconnect signals on object free to avoid dangling connections, so that’s fine.

The player remains since it’s not a child of room at that moment.

* + - * + It’s important the player not be freed with the room. Our approach of keeping player outside ensures that.
        + The health pickups we drop are added as child of room. If the player doesn’t pick them up before we leave the room, and we free the room, those get freed too (so no leak). That’s fine.
        + But what if player is in process of picking up as we transition? Possibly negligible scenario; but to be safe, maybe avoid immediately transitioning if pickup exists. We won’t complicate it for now.
      * Testing note: We need to test all this logic thoroughly because room transitions involve asynchronous signal handling:  
        + We should test that \_on\_EnemyDied fires correctly for each enemy. Because we connected to each Health, we should get as many calls as enemies. Once count is 0, we call on\_room\_cleared which yields then calls load\_room, which disconnects signals etc. If an enemy died just at transition time, signals might try to emit to Main after room freed. But since we yield 1s, likely by the time we free, all died signals done.
        + Possibly an enemy might die, trigger signal, and we call load\_room freeing everything including maybe another enemy that was still alive (if bullet kills two at almost same time). But our logic only triggers transition when count goes 0, meaning all died signals have been accounted for. It should be consistent.
        + The yield ensures any additional died signals in that 1s would still call \_on\_EnemyDied but enemy\_count would go negative if mis-counted. But since count triggers at exactly 0 (meaning all accounted), additional died signals shouldn’t happen because there were no more enemies alive.
        + It’s fairly robust as designed.

1. **HUD and User Feedback**
   * We add a CanvasLayer in Main for UI so that it stays on top and doesn’t move with camera.
   * Add a Label to CanvasLayer to show player health, e.g., "Health: X/Y". Or icons for hearts if we prefer. Text is simpler now.
   * Add another Label for room count or some score if desired.
   * We’ll update these via script:  
     + In Main.gd or a separate HUD.gd attached to CanvasLayer.

Connect signals: player’s health\_changed to a function that updates the health Label text.  
 Example: player.get\_node("Health").connect("health\_changed", self, "\_update\_health\_ui").  
 Then:  
  
 gdscript  
CopyEdit  
func \_update\_health\_ui(current):

$CanvasLayer/HealthLabel.text = "Health: %d/%d" % [current, player.get\_node("Health").max\_health]

* + - We might also call this on room start to initialize.
    - For room count: after each room transition or at start, update: $CanvasLayer/RoomLabel.text = "Room: %d" % room\_count.
    - If we do game over, we could display a "Game Over" label.
  + These UI elements help debugging (ensure health is actually dropping, etc.).
  + They are not part of core gameplay feel, but they prevent confusion when testing damage.

1. **Polish (Game Feel Enhancements)** Now that the basic loop is done, we want to add some effects to make it feel better. These are internal notes for improvements that are not strictly necessary but highly recommended:  
   * **Screen Shake:**
     + Implement a small camera shake when certain events happen:  
       - When player takes damage, or an enemy dies (especially if it’s a big enemy or maybe all enemies cleared).
       - How: since Camera2D is a child of player (which moves with player), shaking the camera could be done by a Camera2D script or by temporarily offsetting the player or an intermediate node.

A simple approach: in Main, have a CameraShake node (could be a script on Camera2D) that on call, applies a short random offset animation. E.g.,  
  
 gdscript  
CopyEdit  
func shake(intensity=5, duration=0.2):

var time = 0.0

while time < duration:

var offset = Vector2(randf\_range(-intensity,intensity), randf\_range(-intensity,intensity))

$Player/Camera2D.offset = offset

await get\_tree().create\_timer(0.05).timeout

time += 0.05

$Player/Camera2D.offset = Vector2.ZERO

css  
CopyEdit  
- For a prototype, even a single frame jolt could suffice.

* + - * Trigger it:  
        + Connect Player Health’s damaged (we didn’t explicitly have a damaged signal, but we could call shake in take\_damage of player’s health or connect health\_changed to if damage event).
        + Or in \_on\_AttackArea\_body\_entered when player is hit, call a shake.
        + When enemy dies, maybe call a smaller shake or none (depending on taste).
      * This gives immediate feedback for impacts.
  + **Flash on Damage:**
    - For Player: When health\_changed with a lower value (damage), flash the sprite. We can do this by modulating the sprite color to red or white and back:

e.g., in Player.gd, on take\_damage (we can override or catch via signal):  
  
 gdscript  
CopyEdit  
var sprite = get\_node("Sprite2D")

sprite.modulate = Color(1,0.5,0.5) # tint red

await get\_tree().create\_timer(0.1).timeout

sprite.modulate = Color(1,1,1) # back to normal

* + - * This is a quick red flash.
    - For Enemy: Do similarly. Perhaps in Enemy.gd, connect its Health.health\_changed to a method that flashes its sprite if current\_health > 0 (meaning not dead yet).
    - These visual cues make hits feel noticeable.
  + **Sound Effects:**
    - Even placeholders from free sources: e.g., a "pew" for shooting, a "hit" sound, and a "death pop".
    - Add AudioStreamPlayer2D nodes to relevant scenes or use an autoload AudioManager to play sounds.  
      * For simplicity, in Player.gd when shooting, do $AudioStreamPlayer.play() (with a shooting sound loaded).
      * In AttackArea body\_entered (player hit), play a hurt sound on player.
      * In \_on\_EnemyDied, play an enemy death sound.
    - Place these AudioStreamPlayers as children of the objects for positional audio (though in 2D, stereo pan might not be crucial).
  + **Particles:**
    - Add a small CPUParticles2D for bullet impact:  
      * Could be a child of bullet that triggers on collision, or simply spawn a one-shot particle node at collision point.
      * Alternatively, when enemy dies, spawn an explosion particle.
      * For example, design a Particles2D node (with a few particles, short lifetime) and instantiate it on enemy death at enemy’s position.
      * Or simpler, use an AnimatedSprite that plays an "explosion" animation and free it.
    - Not essential, but adds juice.
  + These polish items should be done after core is working to avoid confusion while debugging core mechanics. They are isolated enough that adding them doesn’t change the game logic, only the presentation/feedback.
  + We should keep track of these in code with maybe toggles (like a boolean "enable\_screen\_shake") to turn off if needed when debugging logic issues.

### **Future Expansion Hooks**

*(These notes serve as reminders for how the current architecture will accommodate future features without major refactoring.)*

* **New Enemy Types:** Thanks to our modular approach, adding a new enemy is straightforward. We can create a new scene (e.g. EnemyRanged.tscn) that also has a Health node, etc., but with a different attack pattern (like shooting bullets at intervals instead of moving directly to player).  
  + We could reuse the Bullet scene for enemy shots. The enemy’s AI script could have a timer to spawn bullets aimed at the player (similar to player’s shoot function, but possibly slower).
  + Because the player’s health system already listens for take\_damage and our collision layers are set, the same bullet can damage the player by just adjusting collision masks (the enemy bullet would be in a layer that collides with Player).
  + The Room manager wouldn’t need changes except to recognize maybe these enemies as part of "Enemies" group as well. Our current group system covers that as long as we add them to "Enemies".
  + Difficulty progression could involve mixing types (which our design can handle).
* **Items & Pickups:** We have one pickup type (health). Adding others (keys, bombs, power-ups) can follow a similar pattern:  
  + Each pickup as an Area2D with a script. On player body\_entered, it triggers an effect (e.g., increase damage, or grant a temporary buff).
  + The modular design means the player might need new components to handle certain power-ups (e.g., a component for tracking a new ability). But nothing in our current code prevents extending it.
  + For instance, if we add "coins" pickup for score, we can just have a Score system, perhaps an autoload that keeps track, or a player component for inventory.
* **Weapons and Persona Powers:** The game title suggests some Persona-like mechanics (maybe summoning personas or using spells). While not in prototype, our structure could support:  
  + Different attack modes by swapping the attack component or bullet type. If we implement the player attack as a separate node, we could potentially swap that node for a different one (e.g., a spread shot component) when the player changes weapon.
  + The player scene could have a child "Weapon" node that can be an instance of various weapon scenes (each with their own shoot logic and bullet type). Then picking up a new weapon would free the old Weapon node and instance a new one. Since the input to shoot is generic, it would call whatever weapon’s shoot method.
  + Signals can help here too (e.g., a weapon could emit a "shot\_fired" signal if needed to trigger some global effect).
* **Meta-Progression and Save Data:** Not relevant now, but if later implementing runs, our separation of Main (game manager) and player state means we could have a separate scene or autoload for storing persistent data. The prototype’s focus is on feel, so not much here.
* **UI Expansion:**
  + We would later add a start screen, pause menu, etc. None of our current design hampers that. The Main could load a UI scene, or we use separate Scenes for menus and a state manager. Since we modularized things, adding a menu won’t interfere with the game loop.
  + For in-game UI, adding a mini-map or boss health bars would tie into signals (just like health\_changed for boss HP).
* **Performance Considerations:**
  + The prototype likely runs fine, but if we scale up:  
    - Many enemies: our approach of connecting signals for each is okay up to moderate counts. If hundreds of enemies, a different pooling strategy might be needed, but that’s beyond scope.
    - We should monitor if adding and removing scenes frequently (in infinite rooms) causes stutters. Preloading helps with instantiation[docs.godotengine.org](https://docs.godotengine.org/en/stable/tutorials/scripting/nodes_and_scene_instances.html#:~:text=GDScript). We might also consider pooling room instances if reusing them, but since rooms are hand-crafted, better to instance fresh to reset their state.
    - Physics usage: using CharacterBody2D for enemies and player is fine for a few, but if performance dips, we could switch simpler physics for enemies (like moving them by position if we don’t need complex collision response between enemies and environment beyond boundaries).
    - For now, the approach is sound for a prototype and even a small game.

### **Conclusion and Next Steps**

With the tasks outlined and their rationale, implementing the above will produce a minimal combat prototype where:

* The player can move and shoot in a responsive manner.
* Enemies chase and can harm the player on contact.
* Combat outcomes (hits, deaths) are clearly communicated via health system and signals.
* The game progresses linearly through rooms, allowing continuous play and testing of how movement & combat feels over multiple encounters.
* The architecture is intentionally designed to be **modular**. Each system (movement, health, attack, room management) is somewhat independent. This reduces bugs (changes in one system less likely to break others) and makes it easier to extend. For example, to add a new enemy, we don’t have to rewrite player or room code – just plug it in and connect signals accordingly.
* For a beginner solo developer, this structure might seem like extra work initially, but it pays off as the project grows. It avoids the “messy Godot project” pitfall by enforcing a clean separation now.

**Memory Aids for Follow-ups:** (In case of further questions or iterations)

* **Classes & Nodes Recap:** Player.gd, Enemy.gd, Health.gd, Bullet.gd, potentially Room (no script), Main.gd, and maybe Pickup.gd for health pickups.
* **Signals Recap:**
  + Health: died, health\_changed (emitted by any entity’s Health).
  + Bullet: uses Area2D’s body\_entered to detect hits.
  + Enemy AttackArea: uses body\_entered to detect player contact.
  + We connected signals:  
    - Enemy Health.died -> Main.\_on\_EnemyDied (for room clear logic) and also -> Enemy.\_on\_EnemyDied (for self cleanup).
    - Player Health.died -> Player.\_on\_PlayerDied (and/or Main for game over).
    - We could also directly use queue\_free and count nodes, but signals give clarity on events.
* **Main game loop:** Essentially resides in Main.gd’s load\_room and \_on\_EnemyDied logic. That’s where progression is handled.
* **Room Pools:** Defined in Main.gd as EasyRooms, MediumRooms. We can easily add HardRooms if needed and increment difficulty beyond.
* **Potential Debug Flags:** It could be useful to have a debug mode to spawn a particular room or skip to later difficulty. For testing, one might add a key input (like pressing "N") to force next room, etc., if needed.

Finally, ensure to test each part in isolation if possible (unit testing by running individual scenes):

* Run Player.tscn alone – test movement and shooting (bullets won’t hit anything but you can see if they spawn).
* Run Enemy.tscn alone – it has no player, so it might throw errors on player\_ref. We might guard that (if no player\_ref, do nothing). Or test enemy in context of a room.
* Run a Room scene – ensure walls block movement, etc.
* Then run Main (the full game) and see how it all integrates. Use output console to print debug info (like enemy\_count, etc.) to verify the flow of logic.